How Programmers Comment When They Think Nobody's Watching

by

Simon Benjamin Orion Parent

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

Abstract

Documentation is essential to software development. Experienced programmers know this well from having worked with poorly documented code. They wish to improve their documentation techniques and habits, but there is little consensus for them to follow. Somehow, the many different standards must be compared objectively. This desire motivates my work, which aims to better understand existing documentation practices.

This work focuses exclusively on comments within the program code. Programming is a complex human activity, despite a widespread misconception among programmers that writing code is a mechanical process. This is especially true of comments, where programmers express themselves freely. My work fills a gap in research on software documentation by systematically investigating the comments in a unique database of code written by programmers under natural conditions.

The true variety of programming behaviour is surprising. But this variety does *not* mean that the output of programmers is completely arbitrary; there are patterns in this data, which my research aims to understand.

This work makes three contributions:

- A novel taxonomy of comments developed from the data, which to date is the most thorough description of commenting behaviour actually exhibited by programmers.
- Empirical hypotheses regarding large scale commenting behaviour, which were validated on separate test data. These hypotheses describe underlying regularities in programming which appear to transcend individual differences.
- The database of code I collected, which has unique opportunities for further research on software development, and is thus available for use by other researchers.

Acknowledgements

Thank you to my supervisor Bill, for imparting ways of thinking to me which were essential for me to succeed at my research. Learning from you about proper scientific method was essential for me to understand what I had actually accomplished so that I could even begin to write this thesis.

Thank you to the students who participated in my study. I truly appreciate your enthusiasm for my research; it is definitely rare to see such enthusiasm in the participants of an experiment. I hope that you enjoy what I was able to achieve thanks to your generous contributions.

Thank you to my wife Heather, for loving me, and for supporting me throughout this difficult journey. Also for your insistence on reading the entire thesis, and for always pointing out where the writing only made sense in my head.

Thank you to my family: my sisters Hillary and Miranda, my mother Judith, and my father Andrew. I know that I can always depend on your love and support.

Thank you to everyone who gave me a shoulder to lean on when I needed it most.

Finally, thank you to my friends and labmates, for providing an endless series of distractions to finishing this thesis which made graduate school so enjoyable.

Dedication

This thesis is dedicated to my wife Heather.

Table of Contents

List of Figures

Chapter 1

Introduction

Documentation is essential to software development. Experienced programmers know well the value of documentation from having worked with poorly documented code. All the same, experienced programmers are dissatisfied with their own documentation. Indeed, when pressed, a surprising number confess that they rarely document their own work. [15, 20, 21, 25, 29]

Thus, many programmers desire to improve their documentation techniques and habits, but there is little consensus on how to write better documentation [15,21,23]. Somehow, the many different standards for documenting code must be compared objectively. This desire motivates my work, which aims to better understand existing documentation practices.

This work focuses exclusively on one form of documentation: comments within the program code. Commenting is a feature of all programming languages which allows programmers to embed unrestricted text in their code. Comments exist solely for humans to read, since they have no effect on the executed code.

Programming is a complex human activity. Despite a widespread misconception among programmers, writing code is not a mechanical process. This is especially true of comments, where programmers express themselves freely. [4, 20, 21, 29]

My study was purely observational. I investigated the comments present in a unique database of code. My work fills a gap in research on how programmers comment, because I systematically investigated code written by programmers working under natural conditions.

In code freely written by programmers, the true variety of programming behaviour is surprising. But this variety does *not* mean that the output of programmers is completely arbitrary; there are patterns in this data, which my research aims to understand.

1.1 Scope

My study is limited to comments within the code. In doing so I had to consider the meaning of the code, but only as needed to understand the context of the comments. The detailed execution behaviour of the code was largely irrelevant.

I do not advance a particular theory of programmer behaviour. However, the empirical results I describe should emerge naturally from any successful theory. Nonetheless, I situate my observations with respect to existing knowledge from the literature on programmer behaviour.

1.2 Contributions

My work makes three main contributions. First, the data suggests a novel taxonomy of comments, allowing future researchers to catalogue the breadth of commenting behaviour in a systematic way. To date, this taxonomy is the most thorough attempt to describe the full range of commenting behaviour actually exhibited by programmers.

Secondly, from the *pilot data set*, I created a set of empirical hypotheses concerning large scale commenting behaviour. They were validated using the *test data set*, which was concealed while the hypotheses were being developed. Thus, even though programming is a complex human activity, it has some underlying regularities. Furthermore, many of these observed regularities appear to transcend the differences between individual programmers.

The third contribution may yet prove to be the most valuable. The database of code that I collected has unique properties which can be exploited for research on software development. For some research programs, this data represents an opportunity that is difficult to find elsewhere. To this end, the data was collected under terms of consent which allow it to be retained indefinitely and shared with other researchers (see Section 2.1.4 for more information).

1.3 Structure of this Work

This work is built around a case study of fifty-two code bases which I have collected. Chapter 2 gives an overview of this database, including relevant background information describing the context in which the code was written.

Chapter 3 contains qualitative observations on the twelve code bases which form the *pilot data set*. Thorough observation solves two serious problems with much existing research: ignoring phenomena which legitimately occur, and assuming the existence of phenomena which do not actually occur. Using this data to ground speculation and examining it comprehensively avoids both of these problems. The existing literature is referenced as needed to better understand the observations.

Chapter 4 provides quantitative observations of aggregate commenting behaviour in the pilot data set. These observations are formalized as hypotheses, which makes it possible to objectively test the validity of my findings using the forty code bases which comprise the *test data set*. This chapter is the most technical part of this work.

Chapter 5 contains speculation on the deeper meaning of what was observed. I also discuss the role that commenting plays in software development, and provide some directions for future work.

Chapter 2

Overview

Models of programming originate in unique personal experiences, but these experiences are extremely heterogeneous. As a result, disagreement is endemic in discussions on the nature of programming [10,23,29]. To be more objective, this work is based on the study of a fixed set of code bases.

This chapter describes this data set and its origin. To appreciate the examples in this work, which come directly from the data, this chapter provides the complete context in which the code was written. Finally, the specific procedures used to process the data are discussed.

2.1 The Data Set

The database of code comprises assignment submissions from three offerings of the course CS 452: Real-Time Programming. This course is offered in the School of Computer Science at the University of Waterloo [32], and is normally taken by Computer Science or Software Engineering undergraduate students in their final year. This course is widely known to have the most challenging programming assignments of any course offered by the university.

The code written in this course cannot be dismissed as "student code". A typical student in this course already has almost two years of industry experience, and most join the software industry shortly after taking the course. In other words, the students in this course are already writing code at the level expected of programmers in the industry.

The students did not expect their code or comments to be read in detail. Their code is evaluated by the course staff at a functional level: for the purposes of the course, the quality of their code is determined completely by how well it runs. They are also not given any specific coding style guidelines to follow. So, this data set exhibits programmers working in a natural environment, where they are free to do as they please.

2.1.1 Summary of the Course

After a preliminary assignment, which is a qualifying test for prospective students, the remaining students form groups of two (or three when there is an odd number of students). In the first part of the course, the students write an operating system to run on an embedded system which has an Advanced RISC Machine™ (ARM™) processor. This operating system has a microkernel architecture with message passing primitives for inter-process communication [6,14].

The operating system runs directly on the hardware; the students are provided only code for a sample program which displays the message: "Hello, World!". This is a traditional first program written in an unfamiliar environment to ensure that the toolchain is working as expected.

In the second part of the course, they use the attached serial port to control a model train set. They create a program which routes multiple trains to destinations selected by the user. The program manipulates switches on the track to direct the trains, using only intermittent input from contact sensors.

The course ends with a final project in which the students use their system to create an application of their choosing. In good projects, several trains move simultaneously to achieve multiple goals. Interesting behaviour occurs when the goals come into conflict.

For example, a popular type of project is a game of "tag", where one train tries to evade capture by another train. Another popular type of project is a simulation of cargo transport, where the trains strive to meet an artificial schedule. The students set their own goals for the final project; the philosophy of the course is that interesting goals should be neither too easy nor too difficult.

2.1.2 Characteristics of the Data Set

The *data set* consists of programming assignments collected from twenty-six *groups* of student programmers. They submit code for a number of assignment *milestones*. Each combination of group and milestone has an associated *code base*: the group's submission for that milestone. Each code base consists of many submitted *files*, some of which contain the *source code* which is the subject of this study.

Their code is primarily written in the C programming language, and is compiled with a version of the GNU Compiler (GCC) that emits code for the ARM™ architecture. This combination of language and compiler is recommended by the course staff, but the students are permitted to use other tools. Some ARM™ assembly language programming is required to implement critical functionality of their operating system. Each code base also includes support code, such as shell scripts and Makefiles, written in other programming languages.

The data was collected from three offerings of the course: Spring 2011, Fall 2011, and Spring 2012. These three portions of the data set are referred to as the A-, B-, and C-series respectively.

For the B- and C-series, every student gave consent for their code to be studied. Unfortunately, I obtained consent from only 6 of the 10 groups for the A-series. It was collected well after that offering of the course had ended, and contacting the course alumni by email was less reliable than asking in person.

Narrowing Focus

Each group submits seven assignments in total: four operating system milestones and three train project milestones. These assignments are cumulative;

each group progresses through the seven milestones by building on work completed for the previous milestones. I decided to focus exclusively on the final operating system milestone (coded as k4) and the final train project milestone (coded as p3).

The operating system and train project tasks are very different. Building the operating system is a well-defined problem: they are given a specification to implement, so the distinction between success and failure is clear. The train project is undefined; students must work out messy details by themselves, and success is evaluated subjectively. Therefore it is worth comparing the finished operating systems to the train projects.

The milestones leading up to these completion milestones are mostly redundant with the completed versions. They could be used to study the development history of the code. Owing to the difficulty of accurately tracking code changes [13], I leave this to future work.

Figure 2.1 gives an overview of the data. Each group is referred to by a code such as B07, consisting of the series letter together with a unique number, which was randomly assigned.

	Milestone k4		Milestone p3	
Group	Lines of Code	Commented Lines*	Lines of Code	Commented Lines*
A01	6 109	674 (11%)	14 388	1 206 (8%)
A02	4 255	519 (12%)	7 106	722 (10%)
A03	5 845	1 255 (21%)	11 594	2 405 (21%)
A04	5 996	988 (16%)	15 547	2 452 (16%)
A05	4 985	587 (12%)	10 811	1 419 (13%)
A06	8 110	707 (9%)	14 609	1 164 (8%)
A Total	35 300	4 730 (13%)	74 055	9 368 (13%)
B01	4 578	435 (10%)	7 784	630 (8%)
B02	4 636	717 (15%)	10 125	2 013 (20%)
B03	6 411	536 (8%)	9 153	678 (7%)
B04	2 528	166 (7%)	6 429	266 (4%)
B05	4 321	534 (12%)	10 498	1 176 (11%)
B06	3 801	658 (17%)	6 508	982 (15%)
B07	1 573	176 (11%)	3 060	231 (8%)
B08	3 886	289 (7%)	12 665	755 (6%)
B09	4 832	407 (8%)	11 824	1 100 (9%)
B10	4 613	316 (7%)	7 196	462 (6%)
B Total	41 179	4 234 (10%)	85 242	8 293 (10%)
C01	3 961	783 (20%)	8 576	1 510 (18%)
C02	6 345	1 290 (20%)	13 170	1 997 (15%)
C03	2 998	116 (4%)	6 684	275 (4%)
C04	3 331	186 (6%)	8 140	556 (7%)
C05	3 348	185 (6%)	6 978	922 (13%)
C06	4 686	375 (8%)	6 538	499 (8%)
C07	3 283	319 (10%)	10 840	898 (8%)
C08	3 676	427 (12%)	8 283	650 (8%)
C09	2 738	280 (10%)	5 017	530 (11%)
C10	7 574	802 (11%)	14 473	1 285 (9%)
C Total	41 940	4 763 (11%)	88 699	9 122 (10%)
Grand Total	118 419	13 727 (12%)	247 996	26 783 (11%)

* *Commented lines* counts the lines of source code which contain part of a comment. The bracketed quantity expresses this number as a proportion of the total number of lines.

Figure 2.1: Overview of the data set

2.1.3 Merits of the Data Set

Although the code in the data set is written in an academic environment, the course has aspects which give the code a higher level of quality than is normally expected from this setting. The code is also more natural in being written freely, without constraints imposed by specific development practices. Studying code written in an academic environment also provides valuable control over the programming task and programming environment.

Quality of the Data

This course is unusual in that the assignments are completely cumulative. In particular, students who fail to complete a given milestone are *not* given a solution after the deadline has passed (some courses provide this service to the students as a "safety net"). So, writing defective code has real consequences, especially during the operating system portion of the course.

Indeed, many groups experience setbacks requiring as much as forty hours of debugging time when they encounter the particularly nasty bugs typical of operating system development. Such bugs have no discernible effect until the system is put under stress; they evade detection until the operating system is used to support a complex application.

Since the students are not given elaborate starter code, they provide the design themselves, although they are given suggestions in the class lectures. They decide which data structures and algorithms to use, and how to decompose their program into separate modules. They make their own design decisions, and this variety is present in the data.

For the train project, they write programs which interact with entities in the physical world. This is different from most programming assignments given in an academic setting, where instructors isolate the key part of the problem from the burdensome details. So, the train project features programmers solving realistic problems.

Scientific Control

For the operating system task, the students implement to a common specification. So, there are twenty-six code bases all solving the same task, and written in the same environment. This much duplication of effort would never occur naturally within a single industrial organization. Also, the cost to obtain this level of duplication by hiring programmers is prohibitive (an informal estimate of the effort expended by the students to create the data set exceeds five person-years of full time employment).

This level of control has one major downside: results obtained from this data set may fail to generalize to other programming tasks and environments. This work accepts this limitation, and studies the universe of code bases written in the course CS 452. Since the B- and C-series have a perfect response rate, the data has the entire population of this smaller universe for those two terms. Future work will establish which results generalize to other environments.

2.1.4 How to Obtain the Data

This data set is a rich repository of programming in the wild. To focus properly on the study of commenting, I have ignored all other aspects of the code, many of which merit examination in their own right. There remains much to learn from this data set. Therefore, I am making the data available for further research.

The data set was collected from three offerings of the course: Spring 2011, Fall 2011, and Spring 2012. Spring 2011 was collected after the fact and is incomplete, with data from six out of ten groups. Fall 2011 and Spring 2012 are both complete with ten groups each. Each group submitted seven assignments in total: four operating system milestones and three train project milestones. The redacted versions of these code submissions are available for further study.

However, due to oversights during data collection, some submissions from the early project milestones are missing and could not be recovered.

- Fall 2011, p1: Missing B01, B02, B06
- Fall 2011, p2: Missing B06
- Spring 2012, p1: Missing C06
- Spring 2012, p2: Missing C06, C10

Contact information for obtaining the data is maintained at

`http://www.cgl.uwaterloo.ca/~commenting/`

Also available there for download is the list of files and comments studied in Chapters 3 and 4, with all textual information removed. This allows others to scrutinize or extend the statistical analysis presented in this work.

2.2 Methodology

Funny how when you're writing stuff like this your life is always as hard as possible because you need to deal with all cases.

From the comments of group A01

This is a study of naturalistic observation; no variables were manipulated. The students were informed about this study only after completing the final assignment. They could then choose to participate by providing their consent. In this way, their commenting behaviour is free of influence from awareness that their code would be read by others.

In discussing the observations, comments drawn from the data are presented as examples. Consider the code excerpt shown in Figure 2.2.

```
72  // Set interrupt entry points.
73  *SYS_VEC_DATA_ABT = (void *)asm_data_abort_entry;
74  *SYS_VEC_UNDEFINED_INST =
        (void *)asm_undefined_instruction_entry;
75  *SYS_VEC_SWI  = (void *)asm_swi_entry;
    ...
```

Figure 2.2: Excerpt from `a01/k4/src/system.c`

This excerpt comes from a source file named "`system.c`" in group A01's submission of milestone k4. Line numbers referring to the excerpt's location in the source file are printed in the left column. Leading whitespace common to all of the lines in the excerpt is omitted. For example, each line of the excerpt in Figure 2.2 begins with two spaces in the original source.

Regions of interest are highlighted; usually this is the comment itself. Lines of code surrounding the comment are also printed when they supply relevant context. Ellipsis represent part of a logical unit of code which has been elided. After line 75 in Figure 2.2, there are two more lines of code that also set interrupt entry points. The ellipsis is not part of their code, indicated by the absence of a printed line number.

Programmers often write lines of code which are too long to fit within the margins of a printed page, and must be broken, as in line 74 of Figure 2.2. This is not ideal, because the formatting is changed from the author's intention, but there is no satisfactory alternative which preserves the original code [24]. I broke the lines myself, and tried to preserve the visual aesthetics of the code. Broken lines are indicated by giving no line number to the continuation.

2.2.1 Anonymity Considerations

To maintain the integrity of each code base, files written by the same group must be kept together. This is also important because individual variation between programmers may obscure the differences being studied [2,9]. Therefore, each group is assigned an identification code which cannot be linked to their real identities. However, this identification code indicates the term in which they took the course, since each class shares an environment, and the differing environments between terms may affect the data.

The raw files contain identifiers such as names, emails, and usernames. For example, some build scripts reference personal directories by name. There are also student numbers (sometimes included in file headers), and public group numbers, provided so that group members can share files in the computing environment. Some groups gave names to their operating system or project, which I considered to be personal identifiers. Finally, they submit MD5 hash values of their files as checksums. For many files, personal identifiers from the original version can be reconstructed if this hash value is known, so these hash values are also sensitive information.

Redacting the Files

All personal identifiers were redacted. Because of the volume of data, the redaction process was largely automated with two specially written programs. The first program replaces personal identifiers with an appropriate variant of "redacted". This program is conservative; it matches the input against templates derived from the class list. The second program complements the first; it searches for personal identifiers left over in the redacted output. This program is lenient and generates many false positives. Alternating use of the two programs, I improved the first until it redacted every personal identifier detected by the second program.

Binary files (files not consisting entirely of human-readable text) are discarded, because personal identifiers in binary data easily evade detection. The remaining files are thus ensured to be strictly plain text for the redaction program. For a few files, special characters are changed to their ordinary ASCII equivalents to be intelligible to the redaction script. After this conversion, each file consists only of the printable characters in the ASCII code range 32–126, plus newlines (ASCII code 10), and tabs (ASCII code 9).

Some personal identifiers are less straightforward and were thus handled as special cases. For example, some files bear the name of their operating system or project; these file names are redacted. To preserve anonymity, further details about these special redactions are withheld. However, the redaction process has the following property: a line of code was altered from the original version if and only if it contains the string "redacted" (in any case).

Therefore, lines modified from the original are marked, so they can be excluded from the analysis when appropriate. For example, the data does not imply that the word "redacted" is commonly written in comments! There are also more subtle problems: failing to exclude redacted text skews the distribution of word lengths in favour of eight-letter words.

I manually reviewed all redaction changes to verify that the programmers' identities were obscured, and that every redaction change was necessary. The complex structure of the code, even as text documents, prevents absolute certainty that every link to identity has been removed. For this reason, only excerpts are published here, since they can be checked thoroughly.

2.2.2 Gathering the Comments

Their files were processed to compile a list of comments which is the main subject of this study. This requires more structure than is explicit in the actual data, so this section describes how this extra structure was recovered.

Sorting the Files

The files were first sorted to select files containing code, because their submissions also included other files. Presumably these other files resided in the directory of submitted source code. Some of the extraneous files are documentation, such as `README` files or "to do" lists. Others are simply junk, such as metadata files created by development tools.

Source code, the program text maintained by programmers, is distinguished from *generated code*, which is created by compiling source code. Generated code is redundant, since it can be uniquely derived from the combination of source code and compiler. It is also not directly read or modified by programmers.

I discarded the generated code, but this was complicated by a unique aspect of the course. The course-provided Makefile compiles C source code (`.c` files) first into assembly language code (`.s` files), and then compiles the assembly language files into binary object files (`.o` files). The intermediate assembly language files remain after the build finishes; students are encouraged to examine them to acquaint themselves with the assembly language used by ARM™ processors.

The students also write a small amount of assembly language code by hand to implement low level features of the operating system. This code may be embedded in C source files, or compiled directly from assembly language files. They often use their compiler's output to create the initial versions of these assembly language files.

Therefore, sorting assembly language files is difficult, since compiler output is present even in legitimate source code. Here I encountered my first big surprise about their commenting behaviour. Biased by my own habit of heavily commenting assembly language code, I assumed that any hand written assembly language code would have at least one comment. However, fifteen of the twenty-six groups have an assembly language source file devoid of comments! So, such a simple rule failed to correctly sort all of the assembly language files.

Indeed, no mechanical sorting procedure was found. In the end, I sorted the files manually. To mitigate human error, my manual judgements were compared with naïve heuristics; files for which these differed – 3% of 4301 files – were double checked.

The students are provided with a script to generate code which initializes a data structure with the topology and measurements of the train track. Some groups modified the generated code in "`track_data.c`", instead of modifying the script which creates it. So, this file began as generated code, and then became source code. The nature of this file poses problems: the original version is extremely long at 2357 lines, and the code itself is quite abnormal, having thousands of consecutive assignment statements with no comments. Therefore, files derived from "`track_data.c`" are unconditionally excluded from the analysis.

Extracting the Comments

Comments are defined by the formal syntax of the programming language. For the programming languages used in the data, a comment is either:

- a *line comment*, which begins with one of the *indicators* "`//`", "`#`", or "`@`", and runs until the end of the line,
- or a *bracketed comment*, which begins with the indicator "`/*`", and ends with the first occurrence of "`*/`" found thereafter.

The data includes code written in many programming languages, including C, C++, ARM™ assembly language, Make, GNU linker language, Python, and the shell scripting languages sh and bash. Therefore, I wrote a simple parser to extract the comments, since writing a parser which accepts comments from all of these programming languages is easier than obtaining an official parser for each language.

The parser ignores all syntax except comments and string constants. It must understand string syntax since a comment indicator within a string constant is part of the string. For example, the highlighted region in Figure 2.3 is *not* a comment.

```
void dump_calibration( struct train_data* DATA ) {
        int i;

        Printf( COM2, "//Train %d\n", DATA->calibration.number );
        for( i = 0; i < NUM_LOGICAL_SPEEDS; i++ ) {
                ...
```

Figure 2.3: Excerpt from `a04/p3/src/tasks/user_train.c`

Merging the Comments

When a line comment is too long, programmers often split it into multiple line comments. For example, the comment in Figure 2.4 is intended to be read as a single comment.

```
// Remove item from head of queue. If queue is empty,
// then item is set to null. Returns non-zero only when
// successfully retrieved last item and queue becomes empty.
int RemoveQueueFront(Queue* queue, TaskDescriptor** item) {
  ...
```

Figure 2.4: Excerpt from `a02/k4/src/kernel.c`

For this particular example, a single comment is surely intended because the line breaks occur in the middle of sentences. It is more difficult to be this certain for comments which are not fully punctuated.

Each line is a separate comment according to the programming language syntax, but I treat adjacent line comments as a single comment when they belong together. Adjacent line comments are automatically merged when all of the following conditions hold.

- They begin with the same comment indicator.
- They begin in the same column of the file. That is, they have the same level of indentation.
- There are only whitespace characters between them.

This heuristic makes mistakes, so some merges were undone by hand. For example, in Figure 2.5, the highlighted comment was manually separated from the comment on the next line.

```
// Create the user and guards (we don't trash these)
// game.user_tid = create(16, &dumb_user_main);
game.user_tid = who_is("ui"); // Commands now come from the UI
...
```

Figure 2.5: Excerpt from `a03/p3/src/redacted.c`

The strictness of the above rules prevents some correct merges, usually owing to inconsistent indentation. Therefore, I corrected the division of comments as mistakes were discovered during the classification of comments in Chapter 3. Interpreting comments is not an exact science, because it requires correctly inferring the intent of the programmer. Indeed, Van de Vanter states that "the notion of a single comment is itself ill-defined" [29]. Manual intervention occurred for 3% of the comments in the pilot data set.

Filtering the Comments

Some comments in the data set should be excluded from the analysis, because they are not documentation. These *undesirable comments* are described below.

The code provided to the students contains a few comments. As they do not originate from the students, these *provided comments* are excluded. In some cases, multiple identical copies of these comments would bias the analysis.

Many of their assembly language source files are modified copies of files generated by the compiler. Some of the generated code includes comments, such as the highlighted comment in Figure 2.6.

```
baseaddr:
        @ args = 0, pretend = 0, frame = 4
        @ frame_needed = 1, uses_anonymous_args = 0
        mov     ip, sp
        stmfd   sp!, {sl, fp, ip, lr, pc}
        ...
```

Figure 2.6: Excerpt from `a04/k4/src/kern/baseaddr.S`

These *compiler-generated comments* were not written by the programmers, so they are excluded. Fortunately these comments are easy to pattern match against a small set of templates, since they are created deterministically.

Finally, many comments are code that is *commented out*. Since comments are not executed, this is an easy way to "turn off" a piece of code. These comments are excluded since I am interested only in comments that programmers write as documentation.

Nested Comments

The data required me to consider *nested comments*: comments entirely contained within other comments. Consider the comment in Figure 2.7.

```
/*ap_init_buff( &pbuff );
ap_putstr( &pbuff, "\x1B[?25l");        //hide cursor
ap_putstr( &pbuff, "\x1B[1;75f");       //move to clock position
ap_putstr( &pbuff, "\x1B[1;32m");       //set attributes
ap_printf( &pbuff, "%d %", ( (100 * idle_time) / total_time) );
ap_putstr( &pbuff, "\x1B[0m");  //clear attributes
ap_putstr( &pbuff, "\x1B[u");   //return to CL
ap_putstr( &pbuff, "\x1B[?25h");        //show cursor
Putbuff( COM2, &pbuff );

if(i%10 == 0) {
        CommandOutput("total_time: %d idle_time: %d current time %d",
                total_time, idle_time, current_time);
}
*/
```

Figure 2.7: Excerpt from `a04/p3/src/tasks/system_idle.c`

This commented-out code contains six legitimate line comments. I separated these comments from the large comment, so that they can be retained while the highlighted section is discarded.

The notion of nested comments is absent in the programming language syntax, which sees Figure 2.7 as one comment. The extent of a comment, as defined by any of the programming languages in the data, is always contiguous as the file is read line by line, from left to right. This is because the compiler sees each file as a one-dimensional sequence of characters. This is depicted in Figure 2.8 for part of Figure 2.7, beginning on line 125. The end of a line is a character like any other (shown in Figure 2.8 as "↵").

```
··· ap_putstr( &pbuff, "\x1B[?25h");        //show cursor ↵ Putbuff( ···
```

Figure 2.8: Excerpt from `a04/p3/src/tasks/system_idle.c`

The highlighted region in Figure 2.7 looks contiguous on the page, but is clearly disconnected in Figure 2.8. This shows how programmers think about their code in terms of its *two-dimensional layout*. This layout is essential for correctly interpreting comments, and cannot be expressed in terms of the formal syntax of the programming language [29].

Finalizing the Comment List

Aside from the compiler-generated comments, the undesirable comments could not be identified automatically. Therefore, they were excluded manually, but with some assistance from automated heuristics. In the end, just 0.7% of the comments were classified incorrectly by the heuristics. Figure 2.9 shows the proportions of excluded comments.

Source	Proportion
compiler-generated	0.3%
provided	5.4%
commented-out code	6.7%

Figure 2.9: Proportions of undesirable comments in the A-series

The remaining 87.6% of the comments are the comments I studied; these are the comments that I believe were purposely written by the programmers as documentation.

There is an important trend, evident even during this preliminary data processing. Mechanical procedures can handle most of the data, but there are always exceptional cases requiring special treatment. This theme continues throughout the study: there are definite trends in the bulk of the data, but every rule has exceptions.

Chapter 3

Qualitative Observations

This chapter reports my observations of the twelve code bases comprising the pilot data set. In formulating hypotheses about commenting behaviour, I allowed myself to look freely at this data.

The examples in this chapter all come from this data set. In particular, I make a point of avoiding artificial examples. In this way, I only discuss aspects of commenting behaviour which actually occur. Relevant background material is introduced as needed to understand the observations.

This chapter focuses on the meaning and purpose of individual comments. The comments in the data set are very heterogeneous, so a first step to understanding is to separate them into categories. That is, I would like to know:

What kinds of comments are there?

By systematically cataloguing their comments, I developed a taxonomy of commenting as an attempt to answer this question.

3.1 Previous Work

Little previous work analyzes commenting at the granularity of individual comments. I found only two examples in the literature of attempts to describe what kinds of comments there are.

3.1.1 McConnell's Kinds of Comments

McConnell gives a classification of comments in his book on software construction [21]. This book is unique among books on serious programming in that it discusses the low level details involved in writing individual lines of code. In the chapter on commenting, McConnell divides comments into six categories. Figure 3.1 gives summarized versions of his definitions, with some of his commentary.

- *Repeat of the code*: States what the code does in different words. Just more to read.
- *Explanation of the code*: Explains complicated, tricky, or sensitive code. Make the code clearer instead.
- *Marker in the code*: Identifies unfinished work. Not intended to be left in the completed code.
- *Summary of the code*: Distills a block of code into one or two sentences. Such comments are useful for quick scanning.
- *Description of the code's intent*: Explains the purpose of a section of code, more at the level of the problem than at the level of the solution.
- *Information that cannot possibly be expressed by the code itself*: Copyright notices, confidentiality notices, pointers to external documentation, etc.

Figure 3.1: McConnell's kinds of comments [21, with modifications]

This classification scheme is *normative*; it is designed for writing code, and specifically for deciding what kinds of comments should be written. These categories are about the value of comments, and McConnell presents them from worst to best, excluding the last category which is a catch-all. Indeed, McConnell says that only summary, intent, and the last category are acceptable in completed code [21].

Many comments fall outside of this classification scheme, such as the conversation recorded in Figure 3.2.

```
/* FIXME: Can I assume all registers are 32 bits? */
/* Answer: no.  WDT uses 8 bit register */
```

Figure 3.2: Excerpt from `a06/p3/src/regopts.h`

These are not marker comments: deleting them from the finished version would discard valuable information gained during the development of the software.

Furthermore, the distinction between intent and summary comments as described by McConnell is too fuzzy for classifying individual comments. When I tried to use this scheme to classify the comments in the data, I felt that I was just guessing. McConnell does admit that it is difficult to distinguish intent and summary comments, but that this is irrelevant for the pursuit of writing better comments [21]. This scheme is simply not designed to classify comments which have been freely written.

3.1.2 Baecker and Marcus's Communication Objectives

Baecker and Marcus are concerned with typesetting programs, and recognize that different kinds of comments deserve to be formatted differently. This is their motivation for a preliminary taxonomy of comments [1]. In this taxonomy they provide a list of *communication objectives*; their definitions of these communication objectives are summarized in Figure 3.3.

This is a more complete set of categories to cover the variety of comments in the data. However, these categories are not distinct enough to classify individual comments. Furthermore, the Simulation and Indexing categories are impossible to justify both from the data set and from my personal experience.

- *Identification*: Calls attention to the existence of a section of code.
- *Emphasis*: Calls attention to some aspect of additional significance about the code.
- *Description*: Makes explicit intuitable attributes of the code.
- *Explanation*: Clarifies some aspect of the code.
- *Amusement*: Secondary text to help the reader through long or difficult code (jokes, anecdotes, epigrams, illustrations, etc.)
- *Summary and Review*: Reflects upon the reader's progress.
- *Announcements or Warnings*: Informs of recent changes, or provides cautionary remarks.
- *Testing, Gaming, or Simulation*: Quizzes may be useful for long code documents to test the reader's understanding.
- *Measurement or Indexing*: Metrics of the code which may be useful.
- *Analogies, Metaphors, Parables*: Aids in understanding otherwise impenetrable concepts.
- *Informal Remarks*: Spontaneous graffiti from past programmers.

Figure 3.3: Baecker and Marcus's communication objectives [1, with modifications]

3.2 My Taxonomy of Commenting

Both of these works fail to describe existing commenting practice because they focus on writing comments. Therefore I concentrate on *reading* comments, and on reading *all* of the comments, so that I cannot overlook any aspect of commenting behaviour.

Ideally a classification scheme would be *complete* (every comment belongs to a category) and *unambiguous* (each comment belongs to only one category). But not all comments can be classified in this way. Writing comments is a creative activity, so some comments will always fall outside pre-defined categories. Also, there is always a grey area between categories that comments can inhabit. Finally, some comments show features of several categories.

Therefore I am honest about these failures. Comments which fall outside the defined categories are not forced to fit into one of them. Counting these unclassified comments measures the completeness of a classification scheme. Also, the coherence of the categories is improved when they can reject comments whose membership is questionable.

I made multiple passes over the data, attempting to classify each comment according to different taxonomies. First I tried to develop a scheme from my own preconceptions. This failed because the number of categories grew unreasonably large, and the boundaries between them became so unclear that I had no confidence that my judgements were meaningful. Then, I tried to use the categories from McConnell, and also from Baecker and Marcus, which failed for similar reasons. This prompted me to concentrate my search on finding more robust distinctions. Six *basic categories* were found to be sufficiently robust, shown in Figure 3.4.

I performed the classification manually by looking at each individual comment. A small number of comments are left unclassified. I separated the data by milestone because many files were unmodified, or barely modified, between the two milestones, and it seems inappropriate to give these files twice as much weight.

The following sections investigate these categories one by one. Each category is separated into subcategories to understand the commenting trends in more detail.

	Milestone k4		Milestone p3	
Category or Subcategory	Overall %	Category %	Overall %	Category %
Execution Narrative	**54.1%**		**52.5%**	
....................action	44.7%	83%	43.1%	82%
.....................state	5.3%	10%	5.4%	10%
..........state-and-action	2.7%	5%	2.6%	5%
.....................other	1.4%	2%	1.4%	3%
Clarification	**3.3%**		**5.8%**	
.................constant	0.8%	25%	2.8%	49%
....................name	0.4%	11%	0.4%	6%
................end block	1.7%	52%	1.3%	23%
................positional	0.1%	4%	1.0%	18%
...............arithmetic	0.1%	3%	0.1%	1%
.....................other	0.2%	5%	0.2%	3%
Data Definition	**12.6%**		**12.9%**	
..........name expansion	8.3%	66%	8.3%	64%
..n. e. with other content	0.8%	6%	0.8%	6%
.....................other	3.5%	28%	3.8%	30%
Sectioning	**7.8%**		**7.5%**	
..................heading	6.8%	87%	6.6%	88%
..............end marker	0.5%	7%	0.5%	7%
...................divider	0.5%	6%	0.4%	5%
Development Narrative	**3.0%**		**3.2%**	
.....................to do	1.1%	38%	1.5%	46%
..................warning	0.4%	14%	0.6%	19%
...............instruction	1.0%	33%	0.7%	22%
...............attribution	0.2%	6%	0.1%	3%
.....................other	0.3%	9%	0.3%	10%
Prologue	**15.7%**		**14.5%**	
........function summary	8.9%	57%	9.3%	64%
.......................file	3.6%	23%	2.3%	16%
.........complex function	2.4%	15%	2.2%	16%
.....................other	0.8%	5%	0.7%	4%
Unclassified	**3.5%**		**3.6%**	

Figure 3.4: Proportions of all comment subcategories in the A-series

3.2.1 Execution Narrative Comments

First are execution narrative comments, the most common category, comprising just over half the comments in the data set. Figure 3.5 shows that this category dominates for every group.

Code Base	Comment Count	Execution Narrative	Clarification	Data Definition	Sectioning	Development Narrative	Prologue	Unclassified
Overall k4	3019	54%	3%	13%	8%	3%	16%	3%
Overall p3	5964	53%	6%	13%	7%	3%	14%	4%
A01/k4	328	58%	5%	7%	2%	7%	12%	9%
A01/p3	587	44%	13%	11%	2%	9%	13%	8%
A02/k4	373	68%	6%	14%	2%	1%	8%	1%
A02/p3	493	63%	7%	15%	4%	3%	7%	1%
A03/k4	955	52%	$< 1\%$	19%	3%	$< 1\%$	25%	1%
A03/p3	1981	57%	1%	17%	3%	1%	21%	$< 1\%$
A04/k4	701	63%	1%	6%	11%	4%	11%	4%
A04/p3	1726	56%	8%	7%	11%	3%	11%	4%
A05/k4	127	42%	9%	9%	2%	17%	20%	1%
A05/p3	300	40%	6%	19%	2%	14%	11%	8%
A06/k4	535	37%	7%	13%	21%	3%	12%	7%
A06/p3	877	40%	7%	14%	19%	3%	12%	5%

Figure 3.5: Proportions of basic comment categories by code base in the A-series

Execution narrative comments describe the execution of the program. The typical execution narrative comment summarizes a block of code by describing its effect in an English sentence, such as the comment in Figure 3.6.

```
//build the reply message
rpl.message_type = INPUT_SERVER_GETC_REPLY_MESSAGE;
rpl.c = cbuffer_pop( &cbuff );
```

Figure 3.6: Excerpt from `a04/k4/src/tasks/system_serial.c`

This comment describes the following two lines, which populate the structure `rpl` with message data. Comments that summarize a block of code belong to the subcategory of action comments, which account for almost all execution narrative comments: Figure 3.7 gives the breakdown of execution narrative comments into subcategories.

Code Base	Count	action	state	state-and-action	other
Overall k4	1632	83%	10%	5%	2%
Overall p3	3132	82%	10%	5%	3%
A01/k4	191	83%	9%	2%	6%
A01/p3	256	75%	13%	2%	10%
A02/k4	252	82%	9%	3%	6%
A02/p3	312	83%	9%	3%	5%
A03/k4	497	73%	20%	6%	1%
A03/p3	1130	79%	16%	4%	1%
A04/k4	442	86%	4%	10%	< 1%
A04/p3	965	83%	6%	10%	1%
A05/k4	53	96%	–	–	4%
A05/p3	121	79%	10%	2%	9%
A06/k4	197	93%	4%	–	3%
A06/p3	348	93%	3%	1%	3%

Figure 3.7: Proportions of execution narrative comment subcategories by code base in the A-series

Action Comments

Action comments summarize the execution of a block of code, as in Figure 3.6. These comments are written even for a single statement, as in Figure 3.8.

```
// Let the parent know we're done.
verify(0 == Send(parentTid, NULL, 0, NULL, 0));
```

Figure 3.8: Excerpt from `a01/k4/src/testprogs.c`

This suggests that programmers find action comments useful for more than summarizing a long section of code. To conclude this, there is a critical assumption of competence: every comment was written by the programmer for a reason. This can also be viewed as an assumption of efficient communication, since writing a comment takes effort.

Assembly language code is difficult to write, which is why other programming languages were invented in the first place. In the data, assembly language code is often accompanied by action comments written in an ad lib programming language, as in Figure 3.9.

```
asm_fiq_uart1_receive:
  LDR %r13, [%r8, #4]          @ %r13 <- size
  CMP %r13, #512
  BEQ uart1_receive_full
  LDR %r10, [%r8]              @ %r10 <- start
  ADD %r10, %r10, %r13         @ %r10 <- size + start
  ADD %r13, %r13, #1           @ increment size
  STR %r13, [%r8, #4]          @ store size
  ...
```

Figure 3.9: Excerpt from `a01/k4/src/asm_fiq.s`

The text of these comments is more like code than like English, although its meaning is not formally defined. The assembly language code might have been "compiled" from the comments, but this compilation is performed manually by the programmers. For example, the instruction "`ADD %r13, %r13, #1`" which adds one to register 13 is the compiled version of the comment "`@ increment size`".

Referent of a Comment

Each comment *refers* to a specific region of code, which is called the *referent* of the comment [5, 17, 29]. In the general sense of the word, comments can refer to things other than code, but I will use the words "refer" and "referent" to mean what *code* a comment refers to. In Figure 3.10, a box is drawn around the referent of the highlighted comment.

```
/* Now we should either be hitting a null or a char */
parse->token = parse->str;

/* If we hit null, then return nothing */
if( ! parse->token[0] ){
        parse->token = 0;
}

/* Set the next delimiter to null */
while( parse->str[0] && parse->str[0] != parse->delim ){
        ...
```

Figure 3.10: Excerpt from `a06/p3/src/userland/lib/parser.c`

An action comment describes the effect of executing its referent. Even when the comment is too vague to reconstruct the referent, it describes the complete effect of its referent as opposed to highlighting a single detail.

A comment is attached to its referent. If the code is rearranged, the comment should follow its referent since that is where the comment belongs. Unfortunately, the referent of a comment cannot be determined automatically, because it is often necessary to understand the meaning of a comment to determine its referent [5, 29]. For example, the highlighted comment in Figure 3.11 only refers to the next line.

```
486   case CONSOLE_OUT:
487     // Enable UART2 transmit interrupt
488     SetFlag(Uart2Ctrl, TIEN_MASK);
489     next->request.arg3 = 0;  // arg3 is
          used to keep track of data in buffer
490     break;
```

Figure 3.11: Excerpt from `a02/k4/src/kernel.c`

The referent of a comment in such a position is often the entire block (which would include everything up to the break statement on line 490). Some programmers use blank lines to communicate where the referent ends, as in Figure 3.10. Here only by understanding the text of the comment can line 489 be excluded.

Kaelbling suggests that each comment have explicitly marked scope to solve this problem [17]. This would certainly help, but the referent of a comment can have more complex structure than a contiguous block of code. Consider the comment in Figure 3.12.

```
case PLAY: // Both of the following are errors
case QUIT:
        player_response.response = BAD_REQUEST;
        ...
```

Figure 3.12: Excerpt from `a03/k4/src/rps.c`

According to the linear structure of the text, the referent in Figure 3.12 is disconnected in the way seen previously in the discussion of nested comments. Indeed, the referent surrounds the comment on both sides! That is, this comment neither precedes nor follows its referent, rather it is *beside* its referent. This is more evidence that programmers reason about the program text in terms of its two-dimensional structure.

The comment in Figure 3.13 has two referents, each with a distinct role.

```
/* previous values at last sensor read */
int last_sensor;  /* below only valid when this is not null */
double last_velocity;
double distance_since_last_sensor;
int last_sensor_time;
```

Figure 3.13: Excerpt from `a02/p3/src/model.h`

This comment says that the members `last_velocity`, `distance_since_last_sensor`, and `last_sensor_time` are only valid if `last_sensor` is not null, that is, the information about the last sensor is only valid if there actually was a last sensor. Comments with referents that are this complex rarely occur in the data, but they show that thinking of the referent as a contiguous block of code is not the whole story.

State Comments

Some execution narrative comments describe the current *state* of the program's execution. They make up just 15% of the execution narrative comments. However, they are 8% of all comments, and are as common as other basic categories. These *state comments* often describe if statements. In the data set, there are occasionally comments like the one shown in Figure 3.14, which describe the effect of executing an if statement using active language.

```
//check if the event id is invalid
if(req->eventid >= EVENTS_NUM_EVENTS || req->eventid < 0){
        ...
```

Figure 3.14: Excerpt from `a04/k4/src/kern/syscall_handlers.c`

However, it is more common for a comment to describe the outcome after entering the if statement, as in Figure 3.15.

```
482   if(next->request.arg0 < 0 ||
483      next->request.arg0 >= NUM_INTERRUPTS) {
484     // Invalid interrupt id
485     next->status = READY;
486     next->rv = -1;
487   } else {
        ...
```

Figure 3.15: Excerpt from `a02/p3/src/kernel.c`

The referent of such a comment is a *location* in the code, not a particular block of code. The comment in Figure 3.15 says that the interrupt id is invalid if execution has reached line 484. A state comment describes the program state at a specific point in time; the example in Figure 3.16 uses language that makes this explicit.

```
// At this point, we can assume that we're on the same segment as the
// destination.
const int distToDest = distances[lmLast.LandmarkNum()] -
        location.OffsetPos();
...
```

Figure 3.16: Excerpt from `a01/p3/src/track.c`

State-and-action Comments

Many comments that describe the program's state also describe an action, as in Figure 3.17.

```
if (i < size - 1 && cmd[i] == 'r' && cmd[i + 1] == ' ')
{
        cur_state = PARSE_TR;
}
else // Bad command, die
{
        return PARSE_ERROR;
}
```

Figure 3.17: Excerpt from `a03/k4/src/ui.c`

The prevalence of these *state-and-action comments* is my main motivation for combining action and state comments to form the category of execution narrative comments.

These comments imply that the program is performing the action *because* it is in such a state. The comment in Figure 3.17 says that entering the else block means that the command is bad, and so the program will die. It is rare for an explicit linking word such as "because", "since", or "therefore" to be used, but Figure 3.18 shows such an example.

```
/* If path is empty, then we are at the end of the journey.
               Therefore respect required stop distance. */
dprintf( "Planner forward path executing for %d\n", train->id );
          ...
```

Figure 3.18: Excerpt from `a06/p3/src/userland/apps/train/planner.c`

Connections to Programming Techniques

Execution narrative comments are related to some well-known programming techniques. Indeed, the relationship is so strong that it is easily to confuse them with the techniques they resemble.

Action comments resemble *pseudocode*, which has the logical structure of code in terms of control flow and indentation, but the statements of which are written mostly in English. Figure 3.19 shows a comment written in pseudocode, which exemplifies the style of traditional pseudocode.

```
// if redacted can be reached without reverse
//   go to redacted
// else
//   go to "nearest" end without reverse
```

Figure 3.19: Excerpt from `a02/p3/src/automatic_mode.c`

Many programmers are encouraged to write first in pseudocode, and then to implement each statement of pseudocode with a block of actual code. If the pseudocode is written as comments, it can be retained in the final version as documentation [21].

With a flexible enough definition of pseudocode, all action comments are written in pseudocode. However, it is curious that there is no counterpart to state comments in any description of pseudocode that I have ever seen. The data suggests that these state comments are part of the storytelling naturally used by programmers.

State comments resemble assertions. An *assertion* is a program statement which checks a condition assumed by the programmer to be true. If the condition is found to be false while the program is running, then the program is usually halted with an error message, since proceeding further is dangerous because the programmer's assumptions have been violated. For example, the state comment in Figure 3.20 is implemented by the assertion statement on the next line.

```
// This should mean that there is no path from here to the destination.
assert(distances[lmLast.LandmarkNum()] == INT_MAX);
```

Figure 3.20: Excerpt from `a01/p3/src/track.c`

Here the unreachability of the destination has been encoded by a very large distance. The assertion implements the meaning of the comment since it verifies that there is no path. Without the assertion, if such a violation occurred, it would not be detected. Many state comments are too ill-defined to become assertions, however, as in Figure 3.21.

```
// Another kind of interrupt might have provoked this.
```

Figure 3.21: Excerpt from `a01/p3/src/userevents.c`

But assertions cannot completely replace state comments even when the meaning is well-defined. Many well-defined properties are too expensive to compute with current methods [15]. For example, the comment in Figure 3.22 could be captured by an assertion, but this would involve too much calculation, likely duplicating work already done by the program.

```
if(dir)
{
  // Forward is optimal at this exact location.
  ...
```

Figure 3.22: Excerpt from `a01/p3/src/track.c`

Therefore state comments cannot always be replaced by assertions. However, the goal of theorem proving research is essentially to create a better kind of assertion which *can* replace all state comments, or at least those which are well-defined.

Summary

Comments that describe the execution of the program largely fall into two types: those which describe the current state of the program, and those which describe actions (changes in state). These comments refer to specific executable statements in the code, although the referent of a comment cannot be determined automatically. Execution narrative comments have connections to the programming techniques of assertions and pseudocode.

3.2.2 Clarification Comments

Clarification comments help the reader understand the meaning of a tricky piece of code. A typical example is a comment which explains a magic number in the source code, as in Figure 3.23.

```
// 0x4: Transmit interrupt
asm("tst r2, #0x04");
asm("bne uart1_xmit_ready");
```

Figure 3.23: Excerpt from `a02/k4/src/kernel_asm.c`

The referent of a clarification comment is normally smaller than a complete line of code. The comment in Figure 3.23 refers to the literal constant “`#0x04`” on the next line. Sometimes the referent consists of many disconnected pieces if there is repetition of what is being clarified, as in Figure 3.24.

```
// Note: 0x1B is ESC

#define ERASE_SCREEN    "\x1B[2J"
#define ERASE_LINE      "\x1B[K"

#define CURSOR_HOME     "\x1B[H"
#define CURSOR_TIME     "\x1B[1;1H"
#define CURSOR_SWITCH   "\x1B[2;1H"
#define CURSOR_SENSOR   "\x1B[3;1H"
#define CURSOR_COMMAND  "\x1B[15;1H"
```

Figure 3.24: Excerpt from `a05/k4/src/terminal.h`

Interestingly, in both of these examples the string which occurs in the comment does not literally match the referent: “`0x4`” versus “`0x04`”, and “`0x1B`” versus “`\x1B`”.

The breakdown of subcategories is shown in Figure 3.25. This table shows that the use of clarification comments varies greatly by group. None of the subcategories that I have identified are universal; each is absent from at least one group. So, whether a group uses a certain kind of clarification comment depends on their personal tastes.

Code Base	Count	constant	name	end block	positional	arithmetic	other
Overall k4	100	25%	11%	52%	4%	3%	5%
Overall p3	345	49%	6%	23%	18%	1%	3%
A01/k4	15	53%	7%	7%	7%	7%	19%
A01/p3	79	19%	1%	1%	67%	1%	11%
A02/k4	24	38%	42%	–	12%	4%	4%
A02/p3	34	35%	29%	–	24%	6%	6%
A03/k4	2	–	–	100%	–	–	–
A03/p3	13	54%	–	46%	–	–	–
A04/k4	7	100%	–	–	–	–	–
A04/p3	144	92%	7%	1%	–	–	–
A05/k4	12	8%	–	75%	–	8%	9%
A05/p3	17	12%	–	76%	–	6%	6%
A06/k4	40	–	–	100%	–	–	–
A06/p3	58	–	–	98%	–	2%	–

Figure 3.25: Proportions of clarification comment subcategories by code base in the A-series

Constant Clarification Comments

Constant clarification comments give a name to a constant expression in the code, as in Figure 3.23. The existence of such comments is surprising, given that the programming languages used all have the ability to define named constants. For the comment in Figure 3.23, a likely reason is that the GCC inline assembly language syntax is arcane enough that the programmer was unsure how to use a named constant as a literal operand. Indeed, I only know the syntax from reading one of the programs in this data set. I cannot imagine a similarly convincing reason for the comment in Figure 3.24, but this is insufficient justification to assume that no such reason exists.

Clarification comments are more about the form of code than about its function. If a named constant were used in line 114 of Figure 3.23, the program behaviour would be unaffected, but the comment would lose its referent, since "`#0x04`" would no longer exist in the code. This example demonstrates how clarification comments are sensitive to the surface characteristics of code, in contrast to execution narrative comments which describe code on a functional level.

Name Clarification Comments

Name clarification comments explain the meaning of a named entity in the code. Often this name is imposed by another programmer, and thus cannot simply be changed, such as the GCC compiler option names clarified by the comment in Figure 3.26.

```
CFLAGS  = -g -fPIC -Wall -I. -I../include -mcpu=arm920t
          -msoft-float -MMD -Wno-return-type
OPTIMIZE = -O2
# -g: include hooks for gdb
# -c: only compile
# -mcpu=arm920t: generate code for the 920t architecture
# -fpic: emit position-independent code
# -Wall: report all warnings
# -MMD: make dependencies
# -Wno-return-type: ignore warnings about not having
                    returns for system calls
```

Figure 3.26: Excerpt from `a02/k4/src/Makefile`

End Block Comments

Many clarification comments have fairly explicit meaning that can be understood mechanically. *End block comments* link the end of a statement block to its beginning, as in Figure 3.27.

```
104                            // Set the next action
105   #ifdef RANDOM_ACTION
106                            random_number(&action_seed);
107                            action_number = action_seed % 6;
108   #else
109                            action_number = (action_number + 1) % 6;
110   #endif /* RANDOM_ACTION */
```

Figure 3.27: Excerpt from `a03/p3/src/dumb_user.c`

This comment reminds the reader that the block ending on line 110 began with the line “`#ifdef RANDOM_ACTION`”. Some programming languages have features like this in their syntax, such as BASIC where each type of block statement has a different end marker. When this is part of the programming language, the compiler detects mismatches.

Positional Syntax Clarification Comments

Positional syntax clarification comments help the reader track the role of syntax elements distinguished only by position. An example is given in Figure 3.28, which names the first four arguments of “`UpdatePhysics`”; Figure 3.29 shows the function prototype.

```
UpdatePhysics(
    (timestamp - state->time),  // time_delta
    ti->v_before_a,  // start_velocity
    ti->end_velocity,  // end_velocity
    &delta_distance,  // displacement
    &ti->velocity,
    &ti->acceleration,
    &ti->jerk);
```

Figure 3.28: Excerpt from `a02/p3/src/model.c`

That the names match the prototype could easily be checked mechanically. Indeed, several programming languages, such as Python, allow function call arguments to be optionally named.

```
void UpdatePhysics(
    int time_delta,
    double start_velocity,
    double end_velocity,
    double* displacement,
    double* velocity,
    double* acceleration,
    double* jerk);
```

Figure 3.29: Excerpt from `a02/p3/src/model.c`

Other comments clarify the name of a structure member or array index in long initializers, as in Figure 3.30.

```
struct A3CLIENTS {
  int priority;
  struct A3CLIENTARGS args;
} clients[] =
{
  // Priority,{Times,Delays}
  { -3,   { 10,   20 } },
  { -4,   { 23,   9  } },
  { -5,   { 33,   6  } },
  { -6,   { 71,   3  } }
};
```

Figure 3.30: Excerpt from `a01/k4/src/testprogs.c`

This comment clarifies that -3 is the *priority* of the first array member. Recent versions of C support using syntax like "`.priority=-3`" in structure initializers, allowing this comment to be replaced by code.

An interesting feature of the comment in Figure 3.30 is the mimicry of the programming language syntax. This is a major source of difficulty for automatically distinguishing between commented-out code and comments written by programmers as documentation.

Arithmetic Clarification Comments

Arithmetic clarification comments express a calculation which produces a constant in the code, as in Figure 3.31.

```
#define NUM_TASKS       128
#define GEN_BITS        25      // 32 - log(NUM_TASKS)
```

Figure 3.31: Excerpt from `a05/p3/src/task.h`

In principle, "`25`" here could be replaced by code resembling the comment text. However, this may be prevented by limitations in the programming language, for example if "`GEN_BITS`" must be a known constant at compile time. Also, perhaps the programmer wants to see both the calculation and the final result in the code.

On Automation

There is a temptation to call many of these comments "bad" because what they express is not being checked by the compiler. However, this is true of all comments! These comments are simply the best candidates for automation. Also, in the execution narrative comments, some state comments could be replaced with assertion statements, which are verified when the program is run.

Summary

Clarification comments explain an aspect of the code that is particular to its form. Nearly all of these comments could have their meaning automatically verified, at least in principle. However, in some cases the programmers appear to be deliberately avoiding the existing tools for automation. The different kinds of clarification comments are particular to the habits of each group.

3.2.3 Data Definition Comments

Data definition comments are simply comments which refer to a data definition. The typical example is a comment which elaborates a variable name; two such comments are shown in Figure 3.32.

```
task_descriptor* active_task; // Active task fetched by scheduler
syscall_request active_request; // Requests sent from active task
```

Figure 3.32: Excerpt from `a03/k4/src/redactedkernel.c`

The first comment clarifies that the identifier name "`active_task`" means the active task *fetched by the scheduler*. The second comment says that "`active_request`" is the request *from the task which is* active.

This category applies to all data definitions, including:

- variable definitions (global and local),
- function parameter definitions,
- constant definitions (including definitions made with `#define`),
- structure member definitions,
- type definitions (`struct`, `typedef`), and
- enumeration definitions.

These are statements which bind a piece of (non-executable) code to an identifier name. They are treated together because similar patterns were observed for all data definitions.

Figure 3.33 shows the breakdown into subcategories. The subcategories of data definition comments are murky; many comments show multiple features. Because they do this in very few words, it is unclear how to split them into multiple comments, if this is even appropriate. However, there are general trends in the additional information these comments provide. Owing to the absence of cleanly divided subcategories, a more detailed breakdown is not given.

Code Base	Count	name expansion	name expansion with other content	other
Overall k4	379	66%	6%	28%
Overall p3	769	64%	6%	30%
A01/k4	24	38%	25%	37%
A01/p3	65	31%	14%	55%
A02/k4	53	58%	8%	34%
A02/p3	72	46%	10%	44%
A03/k4	177	69%	6%	25%
A03/p3	331	68%	6%	26%
A04/k4	45	80%	–	20%
A04/p3	121	75%	2%	23%
A05/k4	12	17%	17%	66%
A05/p3	58	72%	9%	19%
A06/k4	68	72%	1%	27%
A06/p3	122	67%	5%	28%

Figure 3.33: Proportions of data definition comment subcategories by code base in the A-series

Name Expansion Comments

Name expansion comments give a longer version of the identifier name, as in Figure 3.32. I consider the longer version to be the true name, and the identifier to be an abbreviation. This is the only subcategory of data definition comments that is clear enough to be isolated.

To reference a named entity in a programming language, the identifier must be repeated without variation. Therefore, programmers are motivated to keep identifiers short. However, longer names are often more meaningful. The practice of using a short name as the identifier and providing the longer name in a comment is a popular compromise as seen in the data.

The wording of these expanded names is unusual even for coding styles which encourage long identifiers. The expanded names in Figure 3.32 are unlikely to be identifier names. and Figure 3.34 shows a name which is even less likely.

```
//how far we should have travelled in this time
int dist = 0;
```

Figure 3.34: Excerpt from `a04/p3/src/lib/train_state.c`

The expanded name begins with “how”, a word which is rarely found in identifiers. Indeed, “how” does not occur in any identifier name in the data set except as part of the word “show”.

Identifier Names as Comments

Programmers freely choose the identifiers of language entities they define; the content of the names is ignored by the compiler. Therefore, it is appropriate to also consider identifier names in studying how programmers use natural language in their code. However, for this project I study only the literal comments.

Other Data Definition Comments

Many comments describe the units of a quantity, as in Figure 3.35.

```
int stopdist[NUM_LOGICAL_SPEEDS]; //mm
int stopdistsafe[NUM_LOGICAL_SPEEDS]; //mm
```

Figure 3.35: Excerpt from `a04/p3/src/tasks/train_data.h`

This comment clarifies that the stopping distances are measured in millimetres. For the train project, values measured in physical units are common because the programs interact with the physical world.

Other comments describe how to interpret coded values. Figure 3.36 provides an example of such a comment.

```
//0 = not reserved for this train,
    INT_MAX = fully reserved for this train,
//any other value = that many mm reserved for this train
int length_reserved;
```

Figure 3.36: Excerpt from `a04/p3/src/lib/route.h`

This comment explains the meaning of special values, such as how the large integer "`INT_MAX`" is used for a track segment which is fully reserved, even though the value of "`INT_MAX`" (approximately 2.1 billion) is much larger than the actual length of that track segment. Both of these comments resemble the clarification comments seen earlier, and are perhaps just another manifestation of them. Finally, some comments describe how the defined entity is used, as in Figure 3.37.

```
//used if an event was triggered and no task was waiting
int too_slow;
```

Figure 3.37: Excerpt from `a04/k4/src/kern/kern_globals.h`

These comments may also explain why the defined entity exists in the first place, and where in the code it is used. This is especially important if its use is far away from its definition. Another example is provided in Figure 3.38.

```
int final_game_state;
    // Final state the game ends up in...Only used by the UI
```

Figure 3.38: Excerpt from `a03/p3/src/redacted.h`

This comment also provides the expanded name, and is thus an example of a name expansion comment with other content.

Summary

Comments referring to data definitions primarily give a longer version of the identifier name. Many data definition comments also provide other information about the defined entity, but they fail to divide cleanly into subcategories because many comments provide several kinds of information.

3.2.4 Sectioning Comments

Sectioning comments divide the code into logical units. A typical example is a heading which groups together several statements, as in Figure 3.39.

```
/* Time */
#define TIME_USE_CLK                     CLK_3
#define TIME_CLK_MODE                    CLK_MODE_FREE_RUN
#define TIME_CLK_SRC                     CLK_SRC_508KHZ
#define TIME_CLK_INITIAL                 0xffffffff
```

Figure 3.39: Excerpt from `a06/k4/src/config.h`

The breakdown into subcategories is given in Figure 3.40.

Code Base	Count	heading	end marker	divider
Overall k4	234	87%	7%	6%
Overall p3	446	88%	7%	5%
A01/k4	8	88%	–	12%
A01/p3	9	89%	–	11%
A02/k4	6	67%	33%	–
A02/p3	18	72%	11%	17%
A03/k4	28	100%	–	–
A03/p3	51	100%	–	–
A04/k4	76	82%	18%	–
A04/p3	193	85%	14%	1%
A05/k4	2	100%	–	–
A05/p3	6	100%	–	–
A06/k4	114	88%	–	12%
A06/p3	169	89%	1%	10%

Figure 3.40: Proportions of sectioning comment subcategories by code base in the A-series

Every comment in the data fits well into one of these three subcategories. Heading comments dominate, and are the only subcategory occurring in every code base.

Heading Comments

The comment shown in Figure 3.39 is an example of a *heading comment*. The text of heading comments is usually a noun phrase. Compared to other comments, heading comments provide very little information. For example, execution narrative comments often contain enough information to reconstruct their referent, but heading comments focus on a single aspect that ties the referent together.

Heading comments often occur as a sequence of related headings, as in Figure 3.41. They may even form a hierarchy, as in Figure 3.42, where the "`VIC`" sections have subsections such as "`COM1`". Programmers often indent hierarchical structures in code to show the hierarchy. In contrast, heading comments are rarely indented in this way; indeed they are not in Figure 3.42. So, even for these straightforward comments, determining the referent requires understanding the meaning of the comment text.

```c
// The parameters of a level
typedef struct
{
        // User Parameters
        int user_train_id; // ID of the train that the user uses
        int user_speed; // Speed the train can go at
        // Guard Parameters
        int num_guards; // Number of guards for the level
        int guard_intel[MAX_GUARDS]; // Intelligence level of each guard
        int guard_train_id[MAX_GUARDS]; // Train used by each guard
        int guard_speed[MAX_GUARDS]; // Speeds of the guards
        // Treasure Parameters
        int num_treasures; // Number of treasures
        int treasure[MAX_TREASURES]; // Locations of the treasures
        // Time Parameters;
        int minutes;
        int seconds;
        int tenth_seconds;
} level_definition;
```

Figure 3.41: Excerpt from a03/p3/src/redacted.h

```c
/* VIC 1 */
/* Clock 1 */
interrupt_init_one( VIC1_BASE, 0, INTERRUPT_SRC_TC1UI );

/* COM1 */
interrupt_init_one( VIC1_BASE, 1, INTERRUPT_SRC_UART1RXINTR1 );
interrupt_init_one( VIC1_BASE, 2, INTERRUPT_SRC_UART1TXINTR1 );

/* COM2 */
interrupt_init_one( VIC1_BASE, 3, INTERRUPT_SRC_UART2RXINTR2 );
interrupt_init_one( VIC1_BASE, 4, INTERRUPT_SRC_UART2TXINTR2 );

/* VIC 2 */
/* COM */
/* General interrupt must be lower priority than RX/TX */
interrupt_init_one( VIC2_BASE, 0, INTERRUPT_SRC_INT_UART1 );
interrupt_init_one( VIC2_BASE, 1, INTERRUPT_SRC_INT_UART2 );
```

Figure 3.42: Excerpt from a06/k4/src/interrupt_handler.c

End Marker Comments

End marker comments explicitly mark the end of another comment's referent, as in Figure 3.43.

```
//helper functions
void draw_tab_outline( );
void draw_entry( struct train_display_data* DATA, int i );
void update_entry( struct train_display_data* DATA, int i,
    struct train_position_output_message* msg );
int get_entry_num( struct train_display_data* DATA, int train_num );
void initialize_display_data( struct train_display_data* DATA );
//end of helpers
```

Figure 3.43: Excerpt from `a04/p3/src/tasks/user_train_display_tab.c`

Based on the hypothesis that programmers only write necessary comments, I conclude that programmers themselves believe that some comments have ambiguous referents.

Divider Comments

Some sectioning comments, such as the comment in Figure 3.44, have no English text at all!

```
#define SYS_VIC_MASK(intno)     (1 << (intno % 32))
#define SYS_VIC_BASE(intno)     ((intno / 32) ? SYS_VIC2 : SYS_VIC1)

////////////////////////////////////////////////////////////////////////////

#define TIMER1_BASE     0x80810000  // 16-bit timer
#define TIMER2_BASE     0x80810020  // 16-bit timer
#define TIMER3_BASE     0x80810080  // 32-bit timer
```

Figure 3.44: Excerpt from `a01/k4/src/ts7200.h`

These are *divider comments*, devoid of linguistic content. Thus these comments could be viewed as an alternative to whitespace. Some resolve this by considering all (unnecessary) whitespace as a kind of comment. For example, Scowen and Wichmann [26] define a comment as "characters which have no significance [to the compiler]", which certainly includes all whitespace (except for whitespace required to separate adjacent tokens). As before, I persevere with the programming language's definition of what constitutes a comment.

Summary

Sectioning comments separate into subcategories more cleanly than any other category. There are simply three types: heading comments which group together a section of code, end marker comments which mark the end of a comment's referent, and divider comments which contain no actual text. All sectioning comments have minimal linguistic content.

3.2.5 Development Narrative Comments

Development narrative comments describe the development of the program's source code. A typical example is a reminder that work is unfinished, as in Figure 3.45.

```c
// TODO Make this faster
for ( int i = NUM_PRIORITIES - 1; i >= 0; i-- ){
    if ( !queue_empty( &queues[i] ) ){
        priority = i;
        break;
    }
}
```

Figure 3.45: Excerpt from `a05/k4/src/kern/main.c`

This comment states that there is something "to do"; in this case the following code must be made faster.

Development narrative comments often explicitly mention the programmers or talk directly to the reader, as seen in Figure 3.46.

```c
//Pointer for ungodly pointer copy  NEVER DO THIS AGAIN
//(If you do decide to do this again,
   at least copy this warning message)
rrmsg.route = &DATA->route_buffer;
```

Figure 3.46: Excerpt from `a04/p3/src/lib/train_state.c`

Development narrative comments tell how the program has changed, or will change, or should change. Like execution narrative comments, these comments are part of a story, but the setting is different. Execution narrative comments talk about events that happen during a program run. Development narrative comments talk about events in the real world, the world in which the program is being written.

Figure 3.47 shows the major subcategories of development narrative comments. Most consistently popular and universal are "to do" comments; the other subcategories vary strongly from group to group.

Code Base	Count	to do	warning	instruction	attribution	other
Overall k4	90	38%	14%	33%	6%	9%
Overall p3	190	46%	19%	22%	3%	10%
A01/k4	24	46%	33%	–	4%	17%
A01/p3	52	33%	40%	4%	4%	19%
A02/k4	4	50%	–	25%	25%	–
A02/p3	14	86%	–	7%	7%	–
A03/k4	1	–	–	100%	–	–
A03/p3	10	30%	40%	20%	–	10%
A04/k4	25	36%	4%	52%	8%	–
A04/p3	47	53%	2%	38%	4%	3%
A05/k4	21	14%	–	71%	5%	10%
A05/p3	43	40%	5%	40%	2%	13%
A06/k4	15	60%	27%	–	–	13%
A06/p3	24	54%	33%	4%	–	9%

Figure 3.47: Proportions of development narrative comment subcategories by code base in the A-series

"To Do" Comments

Most common are annotations of unfinished work, as in Figure 3.45. Such comments almost always contain the string "`TODO`", and so I call them *"to do" comments*, but Figure 3.48 shows another phrasing. Many programmers use a consistent tag such as "`TODO`" so that it is easy to search for all such comments when checking for unfinished work [21].

```
// WARNING: Can add a check here regarding
   the center of the track later
```

Figure 3.48: Excerpt from `a03/p3/src/redacted.c`

Warning Comments

Warning comments mark code which is less than ideal, but give no indication that it should be changed. Perhaps it cannot be changed, as the comment in Figure 3.49 implies.

```
// Bad stuff seems to happen to RedBoot when I don't do this.
*SYS_VIC_INTENCLR(SYS_VIC1) = 0xFFFFFFFF;
for(volatile int i = 0; i < 10000; i++);
*SYS_VIC_INTENCLR(SYS_VIC2) = 0xFFFFFFFF;
```

Figure 3.49: Excerpt from `a01/k4/src/system.c`

Here the programmer expresses their feelings about the code: they are skeptical that the referent code should even exist. These value judgements are almost always negative in the data set; programmers take for granted that most code is good.

Instruction Comments

Instruction comments provide helpful information to future programmers or maintainers, as in Figure 3.50.

```
# Add any subdirectories that need to be made here
SUBDIRS = kern lib usr track
```

Figure 3.50: Excerpt from `a05/p3/src/Makefile`

This comment tells future programmers that any new subdirectories with files to be built must be added to the "`SUBDIRS`" list. These comments retain knowledge from the design process, which may help new readers understand the intentions of the original programmers.

Attribution Comments

Attribution comments indicate that code was taken from elsewhere, as in Figure 3.51.

```
// Copied from
// http://graphics.stanford.edu/~seander/bithacks.html#IntegerLogLookup
int FirstBitSet(unsigned int flag, int* LogTable256) {
  ...
```

Figure 3.51: Excerpt from `a02/k4/src/kernel_asm.c`

This kind of comment may be overrepresented in the data set since the code is from an academic environment, and academic integrity requires the students to clearly mark code written by others.

Commenting as a Social Activity

Some comments serve primarily a social function; these comments have little to do with the code. This is only natural because the programmers working on a given software project form a community, and their primary interaction as a part of this community occurs via the code base itself [20].

Some of these utterances have little content because their primary purpose is social grooming [12]. These comments serve to reinforce the bonds between the group members. Figure 3.52 is a good example of this social grooming.

```
            ...
            REDACTED_CONDUCTOR_RT_END = 21, // Conductor done rt'ing
            REDACTED_BOMB = 22 // Someone set us up the bomb
    } redacted_code;
```

Figure 3.52: Excerpt from `a03/p3/src/redacted.h`

This comment makes a certain cultural reference associated with the word "bomb", but is otherwise unrelated to the code. Mawler calls such comments *identity-oriented*, as opposed to comments which focus on the programming task. Each individual comment falls somewhere on the continuum between these two extremes. For example, some comments in the data show a playful nature, but also have a clear task-oriented meaning, as in Figure 3.53.

```
//NEVER USE THIS TASK!!! MWAHAHAHAHAH!!!!
```

Figure 3.53: Excerpt from `a04/p3/src/tasks/system_extra_stack.c`

The amount of identity-orientation, and the form it takes, naturally depends heavily on the individuals. Some group norms require never writing comments in a playful tone. The extent of identity-orientation in the data set may be the result of assignments being done in pairs.

For a more thorough treatment of this topic, please see Mawler's thesis [20]. However, since it is important to recognize the human element in discussions of programming behaviour, selected examples are presented in Figures 3.54–3.64 without further commentary.

Summary

Development narrative comments discuss the development of the code itself. This is where the programmers criticize the code, give advice to those who will follow, and express their wishes for the future.

```
// On your mark! Get set! GO!
```

Figure 3.54: Excerpt from `a03/k4/src/user.c`

```
// TODO: I think there may be a bug somewhere here. (helpful, eh?)
```

Figure 3.55: Excerpt from `a01/p3/src/userlib.h`

```
//TODO:WOWOWOWOW
```

Figure 3.56: Excerpt from `a05/p3/src/usr/shell.c`

```
for(volatile int i = 0; i < 54; i++)
{
  __asm__("NOP"); // AHHHHHHHH!!!!!!!!!
}
```

Figure 3.57: Excerpt from `a01/k4/src/userevents.c`

```
// Estimate a stop distance for the train based on some beautiful,
   complex mathematics
```

Figure 3.58: Excerpt from `a03/p3/src/train.c`

```
#include <devices/clock.h>
         /* Pretty hacky here, don't do this in a normal app */
#include <context.h>
                      /* So damn hacky, I can't go to heaven */
```

Figure 3.59: Excerpt from `a06/k4/src/userland/apps/srr_timing/main.c`

```
//Do we have enough route? If not, yell at the route
  server! Yelling is therapeutic, and likely to make
//people like you!
```

Figure 3.60: Excerpt from `a04/p3/src/tasks/user_train.c`

```
else  // Nada persona esta escuchando para este,
         most likely este es informacion redundante
{
        // Extract the input character from the UART
        irq_status2 = *(int*)(UART1_BASE + UART_DATA_OFFSET);
}
```

Figure 3.61: Excerpt from `a03/k4/src/redactedkernel.c`

```
//shut'er down
```

Figure 3.62: Excerpt from `a04/p3/src/tasks/user_train_command_dispatcher.c`

```
if(trainStateNo == ATTRIBUTION_IGNORE)
{
  // I can do that!
}
else if(trainStateNo == ATTRIBUTION_UNKNOWN)
{
  ...
```

Figure 3.63: Excerpt from `a01/p3/src/trackingserver.c`

```
default: // Mystery uh-oh case
    ...
```

Figure 3.64: Excerpt from `a03/p3/src/redactedkernel.c`

3.2.6 Prologue Comments

Prologue comments give introductory remarks before a major section of code. The typical example is a function prologue comment, which precedes a function body and can include the function's purpose, return value, constraints on input, or even implementation details. Figure 3.65 gives an example.

```
/**
 * Returns a PATHEVENT corresponding to the next thing that should be done to
 * go to the destination, assuming we start at the given location.
 *
 * PATHEVENT_SENSOR:
   the next thing to happen is a sensor. ie, you don't need to
 * do anything before you hit a particular sensor, which is returned in event.
 *
 * PATHEVENT_REVERSE: you need to initiate a reverse after a certain amount of
 * distance, returned in event.
 *
 * PATHEVENT_STARTSTOP: you need to initiate a reverse after a certain amount
 * of distance, returned in event.
 *
 * If the distance pointer is provided, the estimate distance remaining is
 * placed there.
**/
PATHEVENT Destination::GetNextEvent(Location &location, int stoppingDistance,
        bool forward, int *event, int *distance, int *pBuf)
{
  ...
```

Figure 3.65: Excerpt from `a01/p3/src/track.c`

Consider the comment in Figure 3.66 which establishes a naming convention.

```
/*
 *
 * SENDER ALWAYS REFERS TO THE TASK WHICH CALLED SEND
 *
 */
```

Figure 3.66: Excerpt from `a04/k4/src/kern/syscall_handlers.c`

The operating system supports a communication mechanism which allows one task to *send* a message to another task, which *receives* it. The receiver must then *reply* to the sender. This comment removes the potential for confusion, since otherwise they may speak of the receiver *sending* a reply. The referents of such comments are somewhat vague and all-encompassing: for this example the referent includes any occurrence of the word “sender” in the file.

The subcategory breakdown is given in Figure 3.67. As with the data definition comments, many categories show multiple features, so the subcategories presented here are taken from the clearest trends.

Code Base	Count	function summary	file	complex function prologue	other
Overall k4	475	57%	23%	15%	5%
Overall p3	864	64%	16%	16%	4%
A01/k4	41	24%	7%	41%	28%
A01/p3	77	30%	4%	43%	23%
A02/k4	29	41%	–	52%	7%
A02/p3	35	46%	–	51%	3%
A03/k4	237	78%	16%	5%	1%
A03/p3	422	81%	14%	5%	$<$ 1%
A04/k4	80	29%	49%	20%	2%
A04/p3	196	52%	21%	25%	2%
A05/k4	26	8%	77%	4%	11%
A05/p3	32	6%	72%	9%	13%
A06/k4	62	60%	16%	19%	5%
A06/p3	102	69%	13%	12%	6%

Figure 3.67: Proportions of prologue comment subcategories by code base in the A-series

Function Summary Comments

Function summary comments refer to an entire function body and give a single sentence summary of the function's effect. An example is shown in Figure 3.68.

```
// Print cpsr, then the current state, and interrupt status.
void PrintCpsr() {
  ...
```

Figure 3.68: Excerpt from `a02/k4/src/common.c`

These simpler prologue comments resemble execution narrative comments.

In C, a function definition comes with a *function prototype* which provides just the types of the function's arguments and return value so that other source files can use the function. Comments referring to a function prototype, such as the comment in Figure 3.69, are also counted as prologue comments even though a function prototype is not a major section of code.

```
/* Read the difference of clock cycles between clock reads */
int clk_diff_cycles( Clock* clk, uint* val );
```

Figure 3.69: Excerpt from `a06/p3/src/devices/clock.h`

This is justified by how their code was written. For each function, the implementer and user are one and the same, since each group is the only user of the operating system they write. Therefore, there is little difference between a comment placed by the function prototype and a comment placed by the function body.

File Prologue Comments

File prologue comments refer to an entire file, often following a fixed template which is filled in. These templates are particular to the group; an example is shown in Figure 3.70.

```
// Name(s) Redacted
// Created by Name(s) Redacted
// Created 2011/05/12
// Last updated 2011/06/07
// This file is a simple task for testing context switches
```

Figure 3.70: Excerpt from `a03/k4/src/user.c`

Not all file prologue comments follow an elaborate template: some summarize the file's contents in a single sentence, such as in Figure 3.71.

```
/* Simple doubly linked circular list implementation */
#ifndef _LIST_H_
#define _LIST_H_
...
```

Figure 3.71: Excerpt from `a06/p3/src/lib/list.h`

Complex Function Prologue Comments

The function prologue comment shown earlier in Figure 3.65 is fairly elaborate, but only describes the effect and use of the function it refers to. However, prologue comments often show features from other categories. For example, the boxed sentence in Figure 3.72 would be classified as a development narrative comment if it occurred alone.

```
// Turn on the instruction cache
// NB: this function is here for convience.
   It should NOT be called by user tasks!
void enable_icache ()
{
            ...
```

Figure 3.72: Excerpt from `a03/k4/src/syscall.c`

This subcategory of *complex function prologue comments* is a "catch-all" for these multi-featured comments. Further analysis would require classifying the individual sentences of each comment separately.

Summary

Prologue comments are comments which precede a major section of code. Most of these comments summarize the effect of a function in one sentence, and are thus similar to execution narrative comments. Other prologue comments summarize an entire file. Analyzing the remaining comments in more detail requires looking below the level of individual comments.

3.3 Reflections on the Taxonomy

One problem with rigid classification of comments is that many comments show features from multiple categories. Even so, my classification succeeded for most of the comments in the data set: 96% of the comments were classified into one of the six basic categories, and 87% of the comments were further classified into a definitive subcategory. That is, most comments have a single purpose. The most robust trends I observed are:

- Execution narrative comments, which describe the effect of executing the code, the current state of program execution, or both.
- Clarification comments, which clarify the specific form of the code, and whose meaning can often be checked mechanically.
- Sectioning comments, which group statements under a heading, clarify the division of code, or mark the referent of a comment.
- Comments which provide the expanded form of an identifier name.
- Comments which indicate that work is unfinished, commonly containing the string "`TODO`".

Data definition and prologue comments often exhibit multiple features, which makes them difficult to classify further. This may be a characteristic of comments which refer to *definitions*.

What is a Comment?

There may be benefit in broadening the definition of what counts as a comment. The purpose of many comments overlaps with other language features where variation in expression is permitted, such as whitespace and identifier names.

Also, some string constants serve similar purposes to comments seen in the taxonomy. For example, Figure 3.73 shows a string containing an error message.

```
case REQ_UPDATE:
    // Reply right away, minimize notifier delay
    reply( tid, &request, 0 );
    assert( tid == notifiertid,
        "clockserver: Notified by wrong task\n" );
    time += request.value;
    break;
```

Figure 3.73: Excerpt from `a05/k4/src/lib/usr/clockserver.c`

This error message serves the same purpose as a state comment: a state comment saying the same thing would be redundant here. Moreover, this error message is part of an assertion, and assertions overlap with state comments.

Any part of the code not constrained by the programming language contains free personal expression. Depending on what aspects of behaviour are being studied, there may be benefit in being more inclusive in the definition of a comment.

Conclusion

This chapter examined commenting behaviour by walking through a detailed taxonomy of comments found in the pilot data set. This taxonomy is a starting point for developing a real understanding of what comments programmers write, and what they hope to accomplish by writing those comments.

Chapter 4

Quantitative Observations

Quantitative analysis gives a more objective perspective on programming behaviour. Testable hypotheses can be formed using tools from statistics. However, these tools apply only to aspects of the data that can be quantified. Therefore, this chapter takes a higher level view. The analysis reveals patterns which suggest unexpected regularities in programming.

4.1 Commenting by File

First the data is examined at the granularity of the individual file, looking at variation over two factors: group and milestone. Each file was written by an individual group; inter-group variation reveals differences between programming teams. Each file was also written for a specific assignment milestone, which reveals differences as the programming task varies.

Four *basic file types* are distinguished, as shown in Figure 4.1. They account for differences in file content which I noticed were affecting the analysis. I chose these file types to make the most important distinctions without excessively subdividing the data.

Type	Description	Programming Languages
implementation	C source file, with a “ `.c` ” extension, containing the definitions of functions, that is, the actual code executed by the program.	C (occasionally C++), may contain some assembly language code
header	C source file, with a “ `.h` ” extension, containing type definitions and function prototypes for the corresponding “ `.c` ” file.	C (occasionally C++)
assembly	Source file written in assembly language; directly compiled to object code.	ARM™ assembly language
script	Support code written in other languages, mostly to support the compilation process, or for offline processing of data pertaining to the model train set.	Make, GNU linker language, Shell scripts (sh, bash), Matlab, Perl, Python, Racket, Ruby

Figure 4.1: The four basic types of source code files

Each file is divided into *lines of code* based on line break characters entered manually by the programmers. This representation can differ from the displayed code when line wrapping is used. For this part of the analysis, measurements of the data are all of the form: “number of lines in the file satisfying such-and-such a property”, for example:

- the total number of lines in the file,
- the number of nonblank lines in the file, and
- the number of commented lines, that is, the number of lines containing part of a comment.

These quantities all have the same units (lines of source code), and can thus be compared to one another.

4.1.1 Measuring the Size of Code

The histogram in Figure 4.2 shows the distribution of total line count for each file in the data.

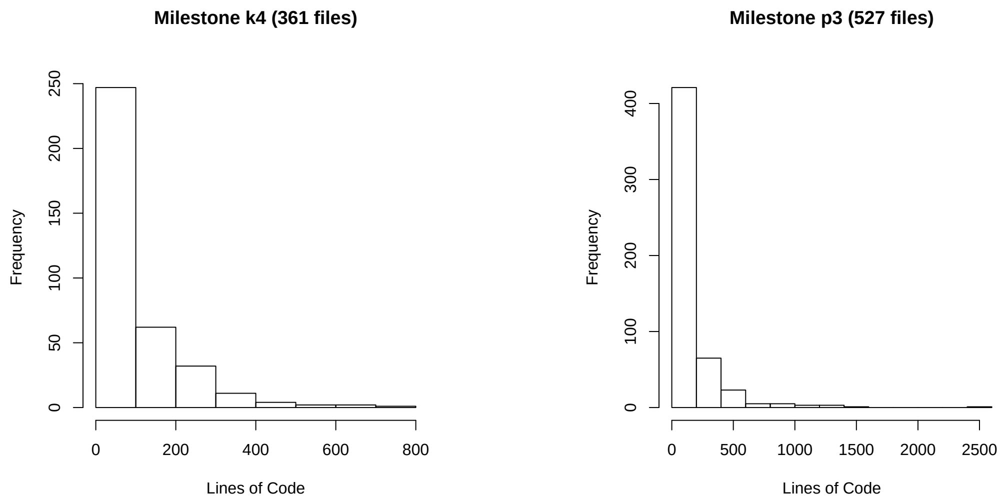


Figure 4.2: Distribution of source file line count in the A-series

These graphs are hard to interpret because the data bunches up at small values. Thus, data should be transformed before it is graphed, primarily so that it fills the available area [7]. In Figure 4.2 almost all of the space is wasted because it communicates nothing.

Transforming the Data

The numeric measurements most common for this data are what Tukey calls *counts*: nonnegative integer values obtained by counting occurrences of something in the data [28]. Examples include the number of lines in a file, the number of letters in a word, and the number of words in a sentence. Re-expressing such data by taking its logarithm or square root is generally appropriate [28,30]. These are rules of thumb developed by experienced practitioners of exploratory data analysis.

There is a continuous family of power transformations, called Box-Cox transformations [30]. The Box-Cox transformation has a continuous parameter p, the exponent of the transformation. Both the logarithm and square root are part of this family, because if $p = 0$, then it takes the logarithm, and if $p = \frac{1}{2}$, it takes the square root (up to a scaling factor). Other values of p produce other power transformations.

This parameter p can be manually optimized by watching the transformed data change as p is varied. By doing so, I found the logarithm to be the most effective transformation for this data. The histogram in Figure 4.3 shows the line count data transformed by a logarithm, giving a nice bell curve shape.

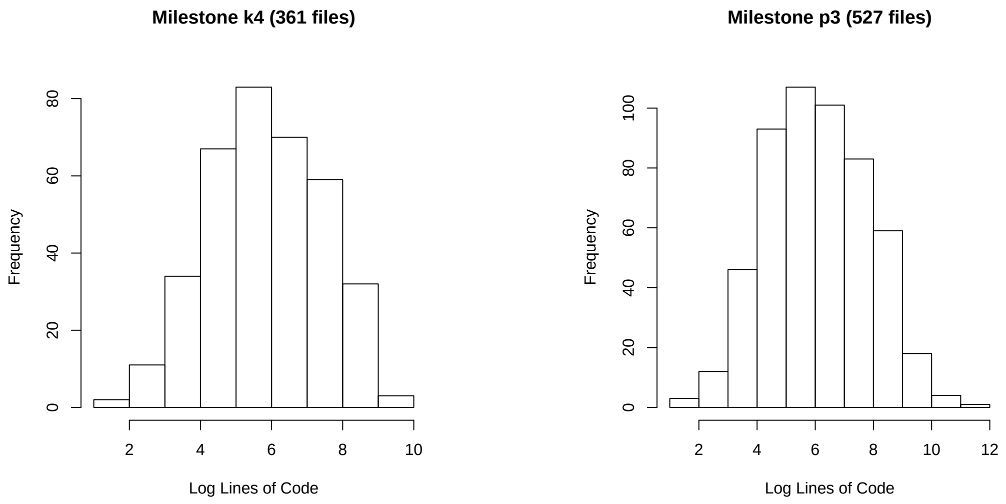


Figure 4.3: Distribution of source file line count in the A-series, transformed by Log

Some technical details about my logarithm transform are worth mentioning. The transformation actually used is a function I call Log, defined by:

$$\mathrm{Log}(x) \;=\; \log_2(x+1).$$

Log is an increasing bijective map from $[0, \infty)$ to itself. That is, if x is a nonnegative real number then so is $\mathrm{Log}(x)$. Log also preserves the ordering of the data points. The input is shifted by one before taking the logarithm because the data has many zero values which would otherwise be discarded (since the logarithm of zero is undefined). This adjustment is a common practice known as “starting” the count [28].

I chose the base two logarithm because it makes graphs easier to interpret than either base e (whose powers are difficult to compute), or base 10 (whose powers are spread too far apart for the scale of the data). The choice of base does not affect the analysis as the data itself is unchanged, except for a scaling factor.

4.1.2 Distribution of File Sizes

Studies comparing different size metrics for a code base have failed to find a metric which outperforms simple metrics such as statement or line count [2,19]. This justifies counting the number of lines of code to measure the size of each file.

Figure 4.4 shows a normal QQ plot of the Log-transformed data, separated by milestone. This plot displays the data compared to what is expected of a standard normal distribution. The points in this QQ plot lie on a straight line when the untransformed data has a lognormal distribution. It is much easier to recognize subtle deviations from normality on a QQ plot than on a histogram.

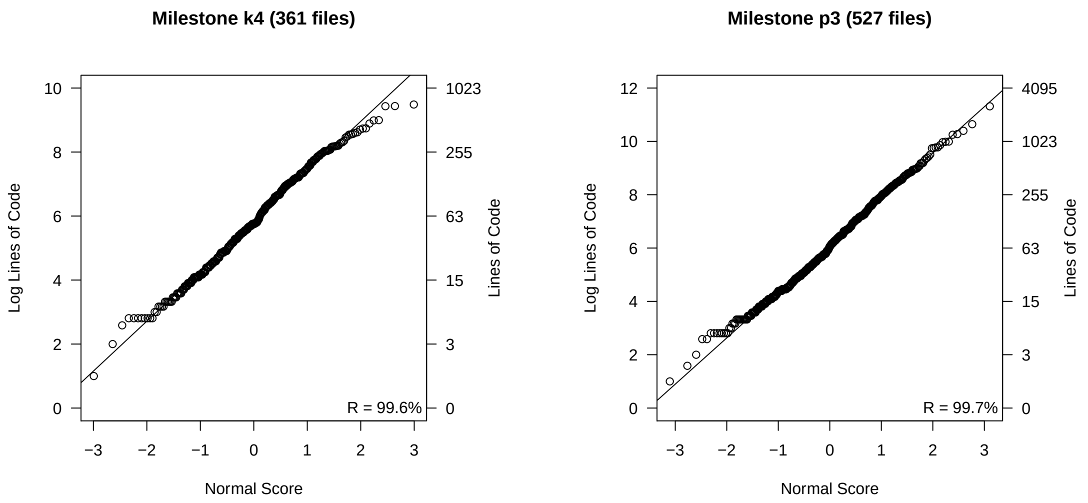


Figure 4.4: Normal QQ plots for Log line count in the A-series

Graphically, the Log-transformed data is well approximated by a normal distribution, especially centrally. What might be anomalies at low values are of little concern, since they are likely artifacts of the discrete nature of the data. There are not enough points at high values to come to strong conclusions merely by looking. Ignoring low values, the fit looks very good for the p3 milestone, and only a single point for the k4 milestone deviates from the line, which is easily attributable to sampling error. The R values shown in Figure 4.4 are a "goodness of fit" measure. I eschew hypothesis tests for normality because they are too sensitive to small amounts of non-normality in the data.

These observations justify a simple model of software development, where lines of code are added at (uniformly) random locations in the code. In this model, each file receives additions proportional to its size. According to Gibrat's rule of proportionate growth [27], dynamic processes with this property naturally give rise to lognormal distributions.

Factor Analysis

An analysis of variance (ANOVA) performed on the milestone, group, and file type shows that all three factors have a significant effect on Log line count. The results of the ANOVA are reported in Figure 4.5. Asterisks indicate p values significant at the 0.05 level.

	df	sum of squares	mean square	F	p
milestone	1	11.695	11.695	7.271	0.007 *
group	5	114.775	22.955	14.273	< 0.001 *
file type	3	947.955	315.985	196.468	< 0.001 *
error	878	1412.112	1.608		
Total	887	2486.538			

Figure 4.5: ANOVA summary for Log line count in the A-series

Tukey post-hoc comparisons are shown in Figure 4.6. The levels of the features are in decreasing order by sample mean so that each cell has the alternative hypothesis that the column label has greater mean than the row label.

	implementation	assembly	script
assembly	< 0.001 *		
script	< 0.001 *	0.991	
header	< 0.001 *	< 0.001 *	0.001 *

	A01	A03	A05	A02	A04
A03	0.799				
A05	< 0.001 *	0.016 *			
A02	0.001 *	0.046 *	1.000		
A04	< 0.001 *	< 0.001 *	0.893	0.954	
A06	< 0.001 *	< 0.001 *	0.251	0.478	0.848

Figure 4.6: Tukey post-hoc comparisons for Log line count in the A-series

Implementation files are the longest ($p < 0.001$), and header files are the shortest ($p < 0.002$). The difference between assembly and script files is not significant. Groups A01 and A03 write longer files than the other four groups ($p < 0.05$). Other differences are not statistically significant.

Figure 4.7 shows normal QQ plots for each code base individually. No group shows significant deviations from the behaviour for the data set as a whole, which suggests that the distribution of file sizes is independent of individual differences. The choppy appearance especially visible in the graphs for A02/k4 and A03/k4 is an artifact of too few sample points, which can be verified by looking at normal QQ plots for small random samples from a normal distribution, which often have a similar appearance.

Figure 4.8 shows normal QQ plots for each file type. Only the script files appear to deviate from lognormality, but again there are too few files to be conclusive.

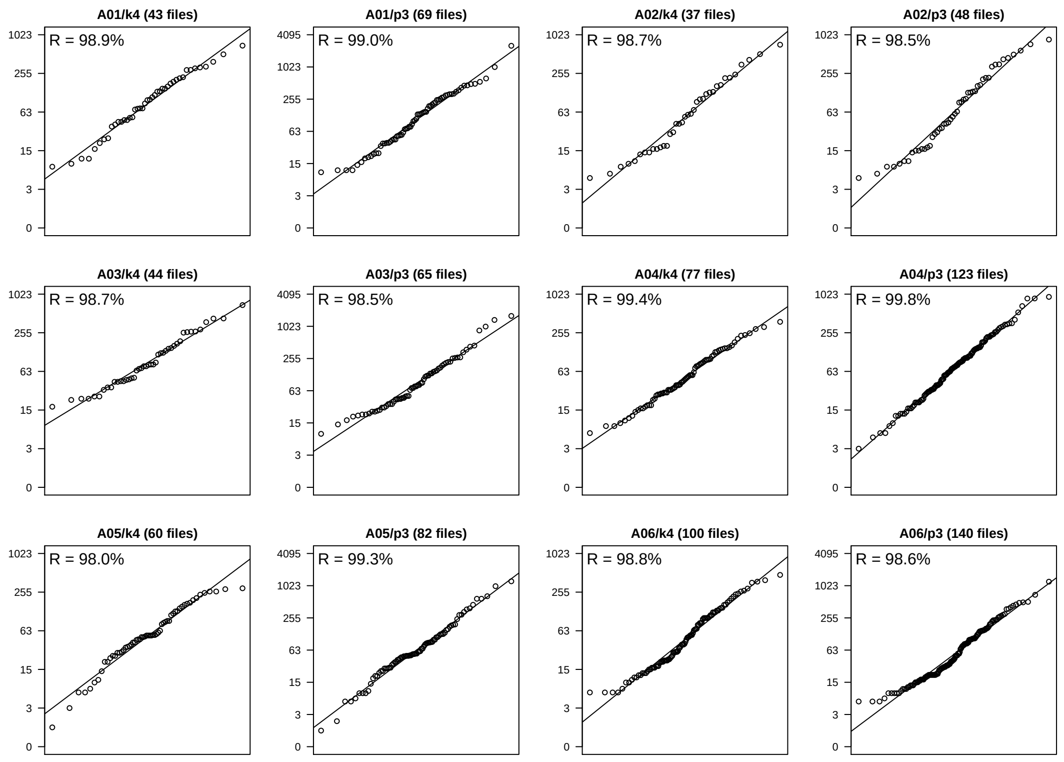


Figure 4.7: Normal QQ plots for Log line count in the A-series, by code base

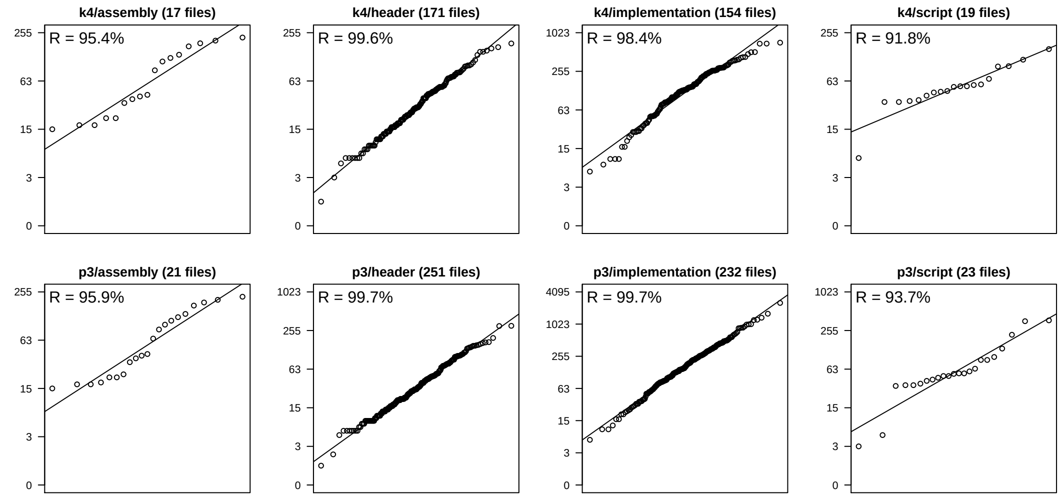
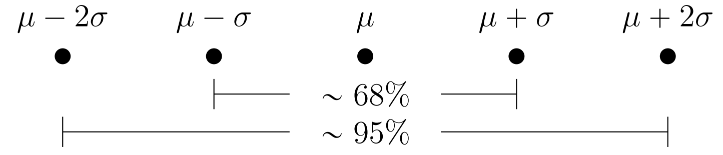


Figure 4.8: Normal QQ plots for Log line count in the A-series, by file type

Location and Scale Parameters

The QQ plots show that line count has approximately the same distribution in each code base, varying only in location and scale. The natural location and scale parameters for a lognormal distribution are its geometric mean and multiplicative standard deviation [18]. The geometric mean μ^* is the untransformed mean of the Log-transformed data, and is easier to comprehend since it is expressed in the same units as the original data. This value μ^* can be thought of as the size of an "average" file.

The multiplicative standard deviation σ^* is equal to 2^σ, where σ is the standard deviation of the transformed data. It is a unitless multiplicative factor, and is analogous to the standard deviation of a normal distribution. For a normal distribution with mean μ and standard deviation σ, approximately 68% of the values lie within one standard deviation of the mean, and approximately 95% of the values lie within two standard deviations of the mean:



For a lognormal distribution, approximately 68% of the values fall between μ^*/σ^* and $\mu^* \cdot \sigma^*$, and approximately 95% of the values fall between $\mu^*/(\sigma^*)^2$ and $\mu^* \cdot (\sigma^*)^2$. This rule gives a sense of values typical of the distribution. In the diagram below, $\sigma^* = 1.5$, to show a typical example of the skewed appearance of a lognormal distribution.

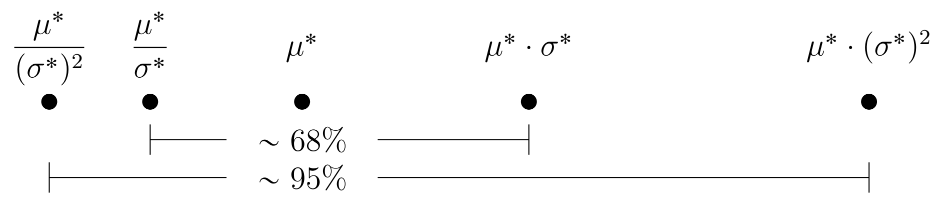


Figure 4.9 tabulates the estimated parameters for the lognormal distributions fit to the data.

	k4 μ^*	k4 σ^*	p3 μ^*	p3 σ^*
Overall	56 lines	3.0x	67 lines	3.3x
Group A01	85 lines	3.0x	103 lines	3.3x
Group A02	52 lines	3.6x	63 lines	3.9x
Group A03	86 lines	2.5x	86 lines	3.1x
Group A04	50 lines	2.6x	68 lines	3.1x
Group A05	51 lines	2.9x	62 lines	3.4x
Group A06	46 lines	2.9x	52 lines	3.1x

Figure 4.9: Parameters for the lognormal distributions fit to line count in the A-series

First, σ^* is quite large; for milestone k4, the above rule says that 95% of the files span the range from 6 to 500 lines, which is a wide range of file sizes. The values of σ^* for the data capture this large size disparity, and indeed they are larger than normally found in nature: a study of the lognormal distribution across various branches of science found that typical values of σ^* are between one and two [18].

Both parameters appear to vary from one code base to another. This is somewhat intriguing; σ^* is thought of as a qualitative parameter, and is sometimes found to *not* vary with the individual [18]. However, each code base has few sample points, so the ability to estimate σ^* for each code base is too weak to rule out the possibility that the underlying value is the same for all groups.

For example, there are only 43 files in code base A01/k4. The measured multiplicative standard deviation for a sample of this size from a lognormal distribution with $\sigma^* = 3.0$ falls between 2.4 and 3.6, approximately 95% of the time. So, the measured value has less than one digit of precision.

This idea is given more rigour by the *bootstrap confidence interval* [31]. Bootstrap methods have the advantage of being unaffected by deviations from lognormality in the data. Figure 4.10 gives the 95% bootstrap confidence intervals for the σ^* estimates.

	k4 σ^*	p3 σ^*
Overall	2.77x–3.18x	3.12x–3.57x
Group A01	2.41x–3.62x	2.81x–3.95x
Group A02	2.87x–4.47x	3.23x–4.79x
Group A03	2.12x–2.93x	2.56x–3.69x
Group A04	2.30x–2.93x	2.70x–3.51x
Group A05	2.39x–3.66x	2.79x–4.18x
Group A06	2.60x–3.29x	2.83x–3.52x

Figure 4.10: 95% bootstrap confidence intervals for the multiplicative standard deviation estimates for line count in the A-series

Within a milestone, each pair of confidence intervals overlap, which means that the observed variation in estimates of σ^* is not statistically significant.

Excluding Blank Lines

The lognormal distribution fit worsens if blank lines are excluded from the line count. This suggests that complicating the definition adds more noise to the data, which justifies my preference for simpler measurements at this stage of the analysis.

Furthermore, the plots in Figure 4.11 compare the line count excluding blank lines to the total line count. The left plot is shown on the scale of the data with linear regression, and the right plot is shown on Log scale, with Log regression.

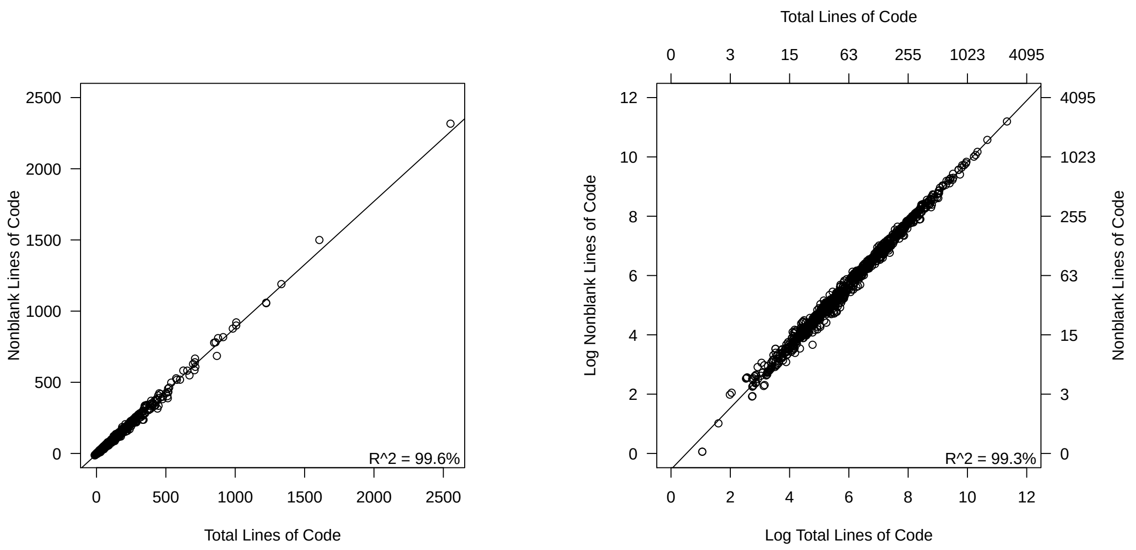


Figure 4.11: Nonblank versus total lines of code in the A-series

Owing to the discrete nature of the data, many values are exactly equal, and would overlap completely if plotted directly. This would conceal the true density of the data. Therefore, the points have been *jittered*, that is, moved by a small random amount before plotting [7].

The plots in Figure 4.11 show that the nonblank line count is so highly correlated to the total line count that there is little difference between the two. This redundancy justifies ignoring the nonblank line count since it provides little additional information.

4.1.3 Distribution of Comments by File

Figure 4.12 gives the normal QQ plots for the number of commented lines of code in each file. The main difference is that 17% of the files have no comments, which truncates the distribution to the left. Otherwise, the overall distribution is also lognormal. This suggests that code and comments grow in qualitatively similar ways.

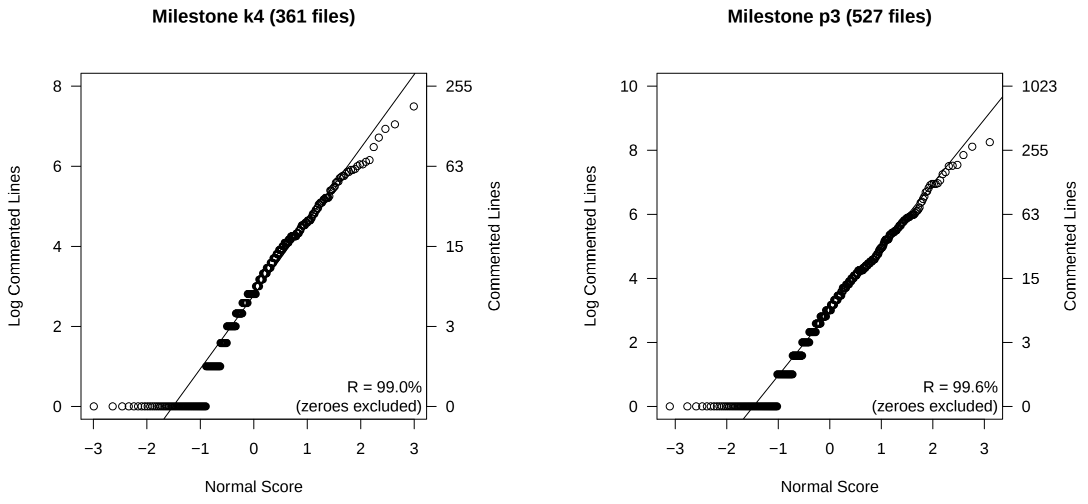


Figure 4.12: Normal QQ plots for Log commented line count in the A-series

Owing to the truncation, the lines in Figure 4.12 are fit by ignoring all data points representing zero values. These points are also ignored when computing the R value which measures the quality of fit.

Factor Analysis

The ANOVA for Log commented line count is given in Figure 4.13. Zero values are excluded so that the input to the ANOVA is closer to a normal distribution.

	df	sum of squares	mean square	F	p
milestone	1	2.993	2.993	1.574	0.210
group	5	238.341	47.668	25.075	< 0.001 *
file type	3	119.435	39.812	20.942	< 0.001 *
error	730	1387.765	1.901		
Total	739	1748.534			

Figure 4.13: ANOVA summary for Log commented line count in the A-series

Here the milestone effect is not significant, but group and file type are still significant factors. Figure 4.14 shows the post-hoc comparisons.

	implementation	assembly	script
assembly	0.970		
script	0.123	0.579	
header	< 0.001 *	0.013 *	0.490

	A03	A02	A04	A01	A05
A02	0.013 *				
A04	< 0.001 *	1.000			
A01	< 0.001 *	0.998	1.000		
A05	< 0.001 *	0.997	1.000	1.000	
A06	< 0.001 *	0.001 *	< 0.001 *	< 0.001 *	< 0.001 *

Figure 4.14: Tukey post-hoc comparisons for Log commented line count in the A-series

Header files have fewer comments than both implementation and assembly files ($p < 0.02$). Group A03 writes the most comments ($p < 0.02$), and group A06 writes the fewest comments ($p < 0.002$).

Figure 4.16 shows the breakdown by file type; no large deviations from lognormality are visible. Figure 4.15 shows the breakdown by code base. In the case of code base A02/k4, the median data value is actually zero! Here most groups are similar to the overall picture, although group A02 has a higher proportion of files without comments. Also, every single source file written by group A03 has at least one comment.

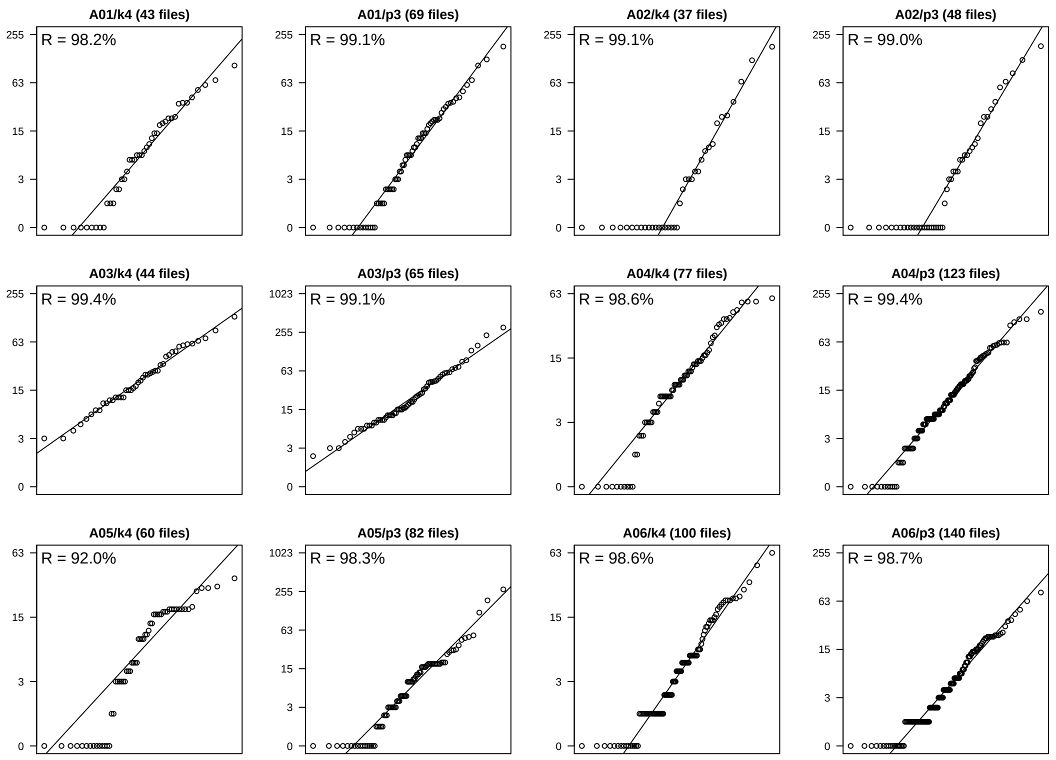


Figure 4.15: Normal QQ plots for Log commented line count in the A-series, by code base

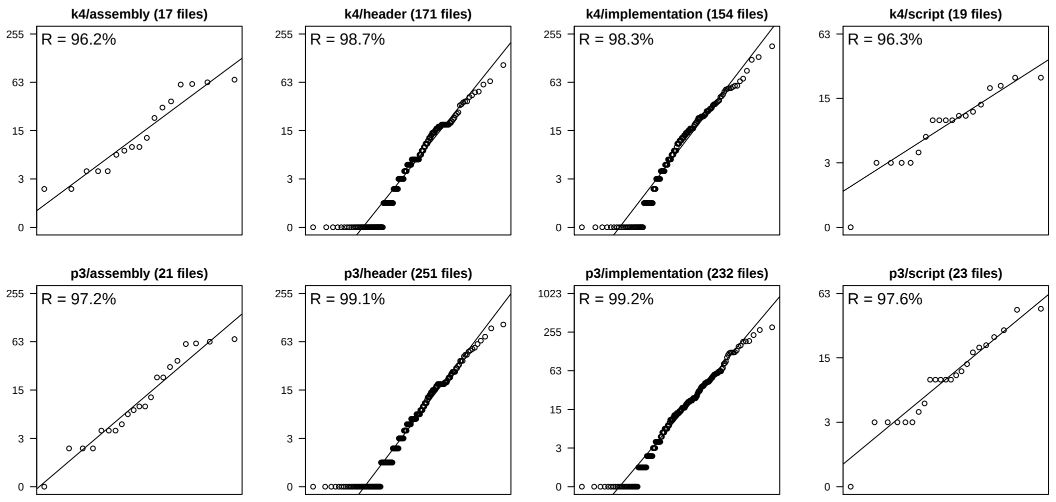


Figure 4.16: Normal QQ plots for Log commented line count in the A-series, by file type

Something different is occurring with group A05, however. There is a bunching up around 17 commented lines because many files have almost identical commented line counts. This is caused by a file prologue comment in several of their header files, which in many cases is the only comment. An example is shown in Figure 4.17.

```
/*
 * ==================================================================================
 *
 *       Filename:  random.h
 *
 *    Description:
 *
 *        Version:  1.0
 *        Created:  05/28/2011 10:38:52 PM
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Name(s) Redacted (20redacted), redacted@uwaterloo.ca
 *        Company:
 *
 * ==================================================================================
 */
```

Figure 4.17: Excerpt from `a05/k4/src/random.h`

This adversely affects the fit for A05/k4, but the quality of fit improves with the larger number of data points present in A05/p3.

Location and Scale Parameters

The location and scale parameters for commented line count are given in Figure 4.18.

	k4 μ^*	k4 σ^*	p3 μ^*	p3 σ^*
Overall	6 lines	3.6x	7 lines	4.0x
Group A01	6 lines	4.0x	6 lines	4.5x
Group A02	1 line	8.8x	2 lines	7.2x
Group A03	20 lines	2.3x	21 lines	2.7x
Group A04	8 lines	2.8x	10 lines	3.3x
Group A05	7 lines	2.5x	6 lines	4.0x
Group A06	3 lines	3.1x	4 lines	3.2x

Figure 4.18: Parameters for the lognormal distributions fit to commented line count in the A-series

Here the variation looks more extreme, especially for the σ^* estimates. To be sure, the bootstrap confidence intervals in Figure 4.19 should be examined.

	k4 σ^*	p3 σ^*
Overall	3.35x–3.85x	3.58x–4.11x
Group A01	3.24x–4.90x	3.50x–5.00x
Group A02	3.14x–6.47x	3.56x–6.30x
Group A03	1.98x–2.71x	2.22x–3.11x
Group A04	2.59x–3.56x	2.95x–3.89x
Group A05	2.87x–3.77x	3.29x–4.72x
Group A06	2.57x–3.20x	2.69x–3.26x

Figure 4.19: 95% bootstrap confidence intervals for the multiplicative standard deviation estimates for commented line count in the A-series

Estimation of σ^* is very poor when many files have no comments, especially for group A02. Even so, some pairs of intervals do not overlap, and so individual variation has significant effect on the value of σ^*. However, these location and scale parameters may not be ideal for describing a lognormal distribution with this much truncation.

Taming the Data

This analysis was done *without* excluding the undesirable comments mentioned in Section 2.2.2. This is because the lognormal distribution fits slightly worse if these comments are excluded. So even this straightforward attempt to improve data quality fails to improve the results. This suggests that even though the data is “wild”, my attempts to tame its wildness did not isolate the correct factors.

4.1.4 Summary of Commenting by File

The distributions of line count and commented line count for the files in the data are well approximated by a lognormal distribution. Each code base follows the same distribution, but the location and scale parameters vary. However, the scale parameter σ^* for total line count may have little individual variation; the present data is inconclusive. Ignoring blank lines, and discarding the presumably undesirable comments both fail to improve the quality of fit, suggesting that I lack the understanding necessary for complex preprocessing of the data.

4.2 Comment Density

Figure 4.20 compares commented line count to total line count on Log scale. The dashed line shown on these plots is the line $y = x$. A point on this line would correspond to a file in which every line is commented.

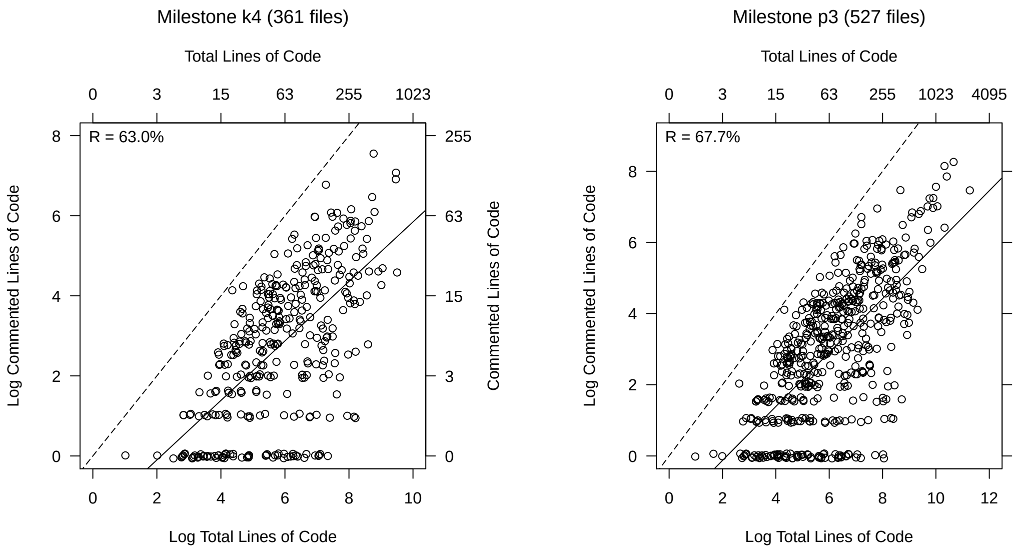


Figure 4.20: Log commented line count versus Log total line count in the A-series

Every data point must be on or below this line, and the area below this line is almost completely filled. Therefore, these two properties vary quite a lot relative to each other. This is different from the comparison of nonblank line count to total line count (Figure 4.11, page 78).

There is moderate positive correlation between these two variables: the correlation coefficients for the Log-transformed data are $R = 63.0\%$ for milestone k4 and $R = 67.7\%$ for milestone p3. However, this means little since every point lies below the line $y = x$. For example, the data set

$$\big\{(x,\, y) : 0 \leqslant y \leqslant x \leqslant 100,\, x, y \in \mathbb{Z}\big\}$$

consisting of uniformly spaced points, unrelated except that $y \leqslant x$, has correlation coefficient $R = 50\%$.

Figure 4.20 shows that the proportion of lines with comments varies greatly by file. Therefore it is worth studying this proportion D, the *comment density*, as a property of each file.

Scaling Properties of Density

A true measure of density would be unaffected by changes in the scale of the data. Defining the comment density as

$$D \quad = \quad \frac{(\#\text{ of commented lines})}{(\text{total }\#\text{ of lines})}$$

implicitly assumes that the number of commented lines grows proportionally to the total number of lines. That is, it assumes that if a given file were twice as long, then it would have roughly twice as many comments. However, the relationship may be different. A more general model is:

$$(\#\text{ of commented lines}) \;\approx\; c \cdot (\text{total }\#\text{ of lines})^{e}$$

where c is a scaling constant, and e is a positive exponent. This model allows the proportion of comments in a file to depend on its size, which cannot be ruled out without more investigation. For example, if a long file is more complex than the sum of its parts, then long files may naturally have more comments per line of code.

4.2.1 Comment Density within a File

This section investigates the uniformity of comment density within each file. For example, the densities of each half of the file could be compared. This could help validate my measure of density, because if the density is uniform, then there is reason to believe that doubling the size of a file would also double the number of commented lines. Figure 4.21 shows this comparison.

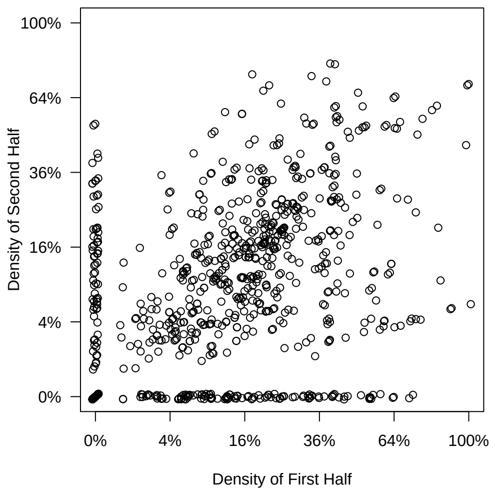


Figure 4.21: Comparison of comment density within the files of the A-series, plotted on square root scale

The points are transformed by a square root for easier viewing, and they are jittered to prevent overlap. No relationship between the two is visible. There is weak positive correlation: $R = 47.4\%$ for the transformed data, and other power transformations give little improvement.

Several factors might mask a possible result. Large prologue comments cause some files to have significantly more commented lines at the beginning. There may be factor effects from the file type, group, or milestone. Also, this density measure may be unreliable for very short files, where adding or removing a single comment has a disproportionately large effect.

To deal with these problems, the following is done. Files shorter than 50 lines of code are excluded. Each file is split into halves, thirds, quarters, and fifths, so even if the ends of the files behave differently, comparing two portions from the middle avoids those problems. Also, each file type and code base is examined individually.

Figure 4.22 compares fractional density measures for each file divided into halves, thirds, quarters, and fifths. Figure 4.23 shows the comparison of the third fifth versus the fourth fifth broken down by file type and by code base. This pair was chosen for presentation since it has the highest correlation coefficient, so it is the most likely to show a relationship. The other pairs were also examined and are no better.

No file type, milestone, or group shows a relationship between any pair of these fractional densities. Any relationship, linear or non-linear, between any two fractions would show as a one-dimensional curve on the plot, but in all cases the points spread out to fill the plotting area.

I conclude that comment density varies too much within a file for this approach to validate my density measure. Perhaps the very concept of “the same source code file, but twice as long” is ill-defined.

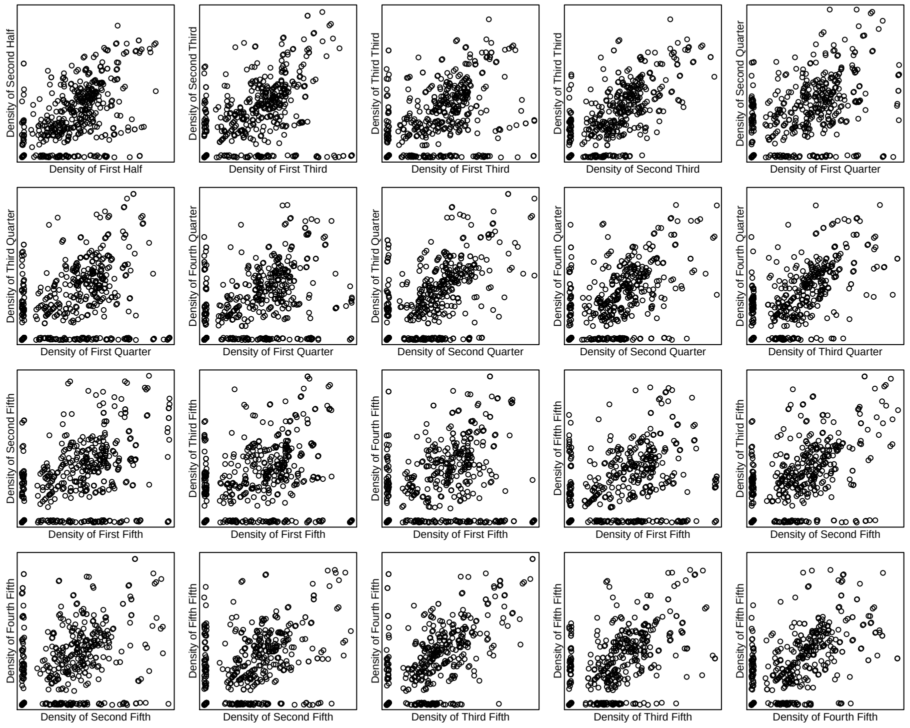


Figure 4.22: Comparison of comment densities from fractions of the files in the A-series, plotted on square root scale

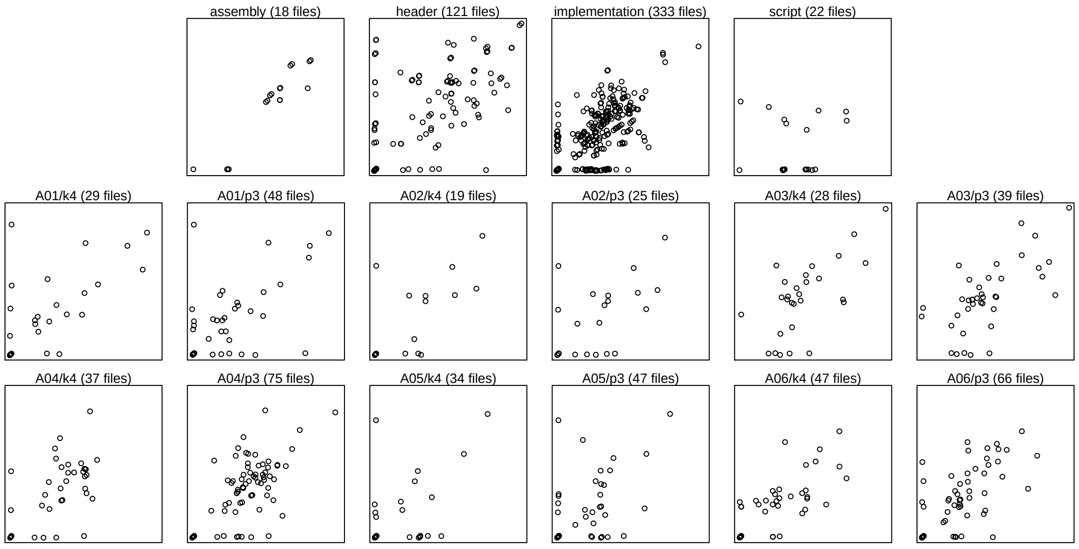


Figure 4.23: Comment density of third fifth versus comment density of fourth fifth in the A-series, plotted on square root scale, by file type and by code base

4.2.2 Comment Density over Time

Another way to think about "the same file but larger" is to examine the files as they change over time. Unfortunately, the data was not processed to track code changes in a way which is thorough enough to make any definite conclusions. For a preliminary test, I assume that a file in milestone p3 corresponds to the later version of a file from milestone k4 if they have the exact same file name (including the full path).

There are 341 such matching pairs of files, 158 with changes between the two milestones. Figure 4.24 shows a comparison of the changed files.

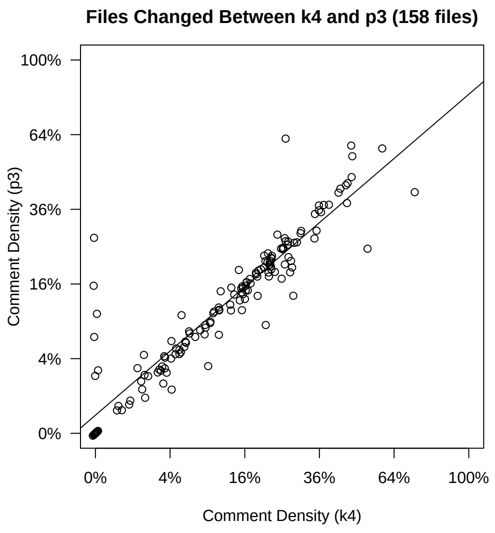


Figure 4.24: Comparison of comment density in the A-series for source files changed between milestones, plotted on square root scale

Linear regression on the transformed data gives an R^2 value of 85.1% (the untransformed data gives a slightly lower R^2 value). This is much better than the previous attempt! Further investigation is required, but this data suggests that the comment density of a file changes little over its development lifetime.

4.2.3 Factor Analysis for Comment Density

This section looks at how group, milestone, and file type affect comment density. Transforming the data by a square root gives the distribution shown in Figure 4.25.

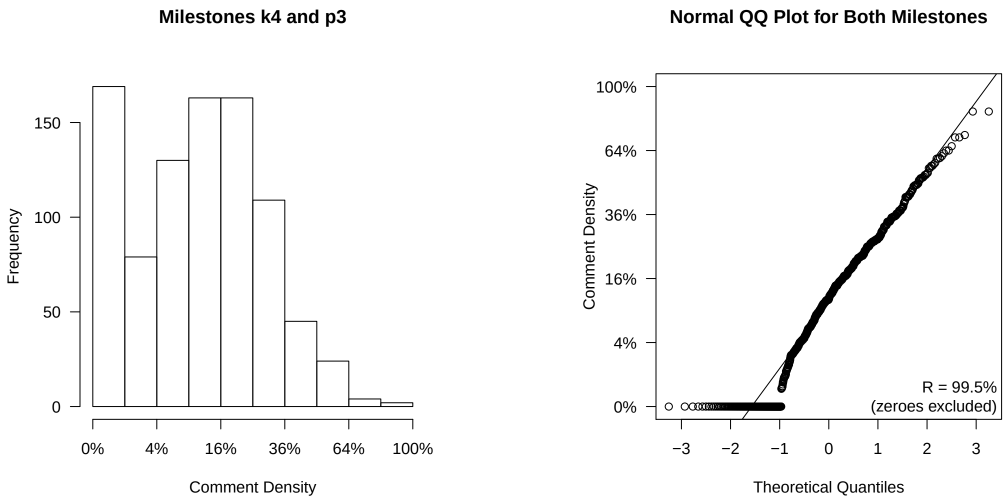


Figure 4.25: Histogram and normal QQ plot for square root transformed comment density in the A-series

The square root of density is close to a normal distribution, but is truncated to the left at zero. Therefore it is reasonable to perform an ANOVA, the results of which are shown in Figure 4.26. Files with no comments are excluded from the ANOVA.

	df	sum of squares	mean square	F	p
milestone	1	0.009	0.009	0.485	0.486
group	5	2.983	0.597	33.703	< 0.001 *
file type	3	4.193	1.398	78.974	< 0.001 *
error	730	12.921	0.018		
Total	739	20.105			

Figure 4.26: ANOVA results for square root comment density in the A-series

The difference between milestones is not significant, which is consistent with comment density being stable over time. Both file type and group significantly affect the comment density. Figure 4.27 gives the post-hoc comparisons for these factors.

	assembly	header	script
header	0.395		
script	0.008 *	0.034 *	
implementation	< 0.001 *	< 0.001 *	< 0.001 *

	A03	A04	A05	A02	A06
A04	< 0.001 *				
A05	< 0.001 *	0.207			
A02	< 0.001 *	0.004 *	0.384		
A06	< 0.001 *	< 0.001 *	0.002 *	0.992	
A01	< 0.001 *	< 0.001 *	0.004 *	0.948	0.994

Figure 4.27: Tukey post-hoc comparisons for square root comment density in the A-series

Implementation files are the least densely commented ($p < 0.001$), followed by script files ($p < 0.04$). Group A03 comments the most densely ($p < 0.001$). The other groups are ordered as:

$$\text{A04} > \text{A05} > \text{A02} > \text{A06, A01}.$$

where the difference is significant for groups more than two spaces apart in this chain ($p < 0.005$).

Figure 4.28 gives the average density for each group, milestone, and file type as seen by the ANOVA. Zero values are excluded from the calculation of the mean, and the presented value is actually the untransformed mean of the transformed data. This is analogous to using the geometric mean for lognormally distributed data to present the values in the original units.

Figure 4.29 shows normal QQ plots for square root density by file type and by code base. Each file type and code base appears to be roughly normally distributed, but looking more closely, there are some pronounced deviations. Group A04 and code base A01/k4 have negative curvature throughout the centre of the distribution, code base A03/p3 has a noticeable sigmoid shape, and group A02 has a consistent bend in the centre. All of these suggest

Milestone	Cell Mean
k4	15.6%
p3	15.0%

Group	Cell Mean
A01	10.5%
A02	11.9%
A03	25.9%
A04	18.3%
A05	15.3%
A06	11.1%

File type	Cell Mean
assembly	24.5%
header	21.1%
implementation	9.5%
script	15.9%

Figure 4.28: Untransformed cell means for square root comment density in the A-series

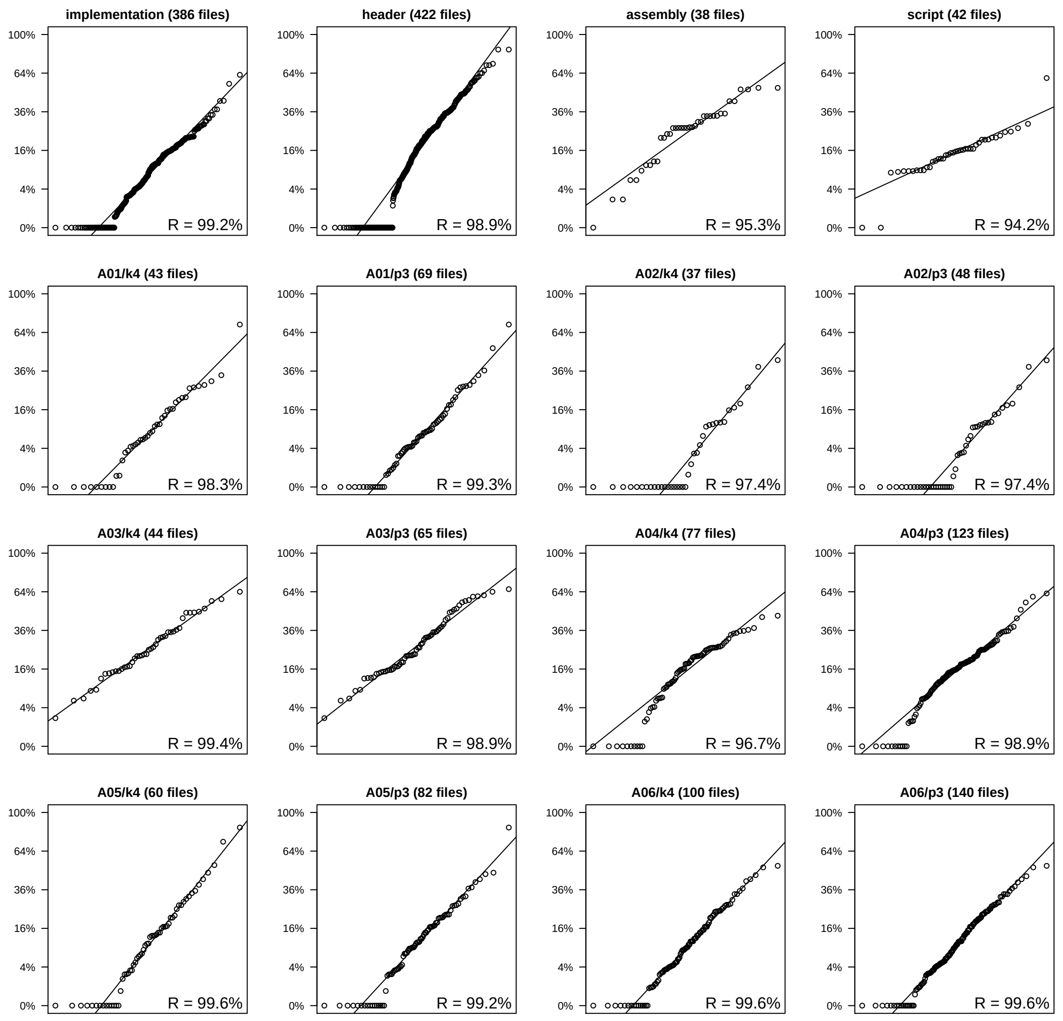


Figure 4.29: Normal QQ plots for square root comment density in the A-series by file type and by code base

systematic deviations from normality which are not common to all groups. Still, the square root transformation improves normality, which is useful for statistical techniques intended for normally distributed data.

4.2.4 Comment Density versus Task Difficulty

While extracting comments from the data, I saw many files with very few comments. Often, these files solve straightforward problems, such as implementing utility or helper functions. It has been suggested in the literature that difficult code is commented more densely [19].

The implementation and assembly files in the data set can be classified based on the nature of the task they are solving. I manually classified each implementation and assembly file from milestone p3 into one of five *task categories*, described below.

The students are not given a full standard library compatible with their compiler, so they have to implement common library functions themselves. I expect these *library files* to contain the fewest comments because the code in these files solves basic problems familiar to the students.

The most difficult part of the operating system assignments is writing a correct context switch. It must be written in assembly language, and is notoriously difficult to debug. Thus, the *context switch files* containing its implementation are likely to have the most comments.

Many context switch files are C implementation files, since many students embed the assembly language code in C source files. Also, some library code is written in assembly language files. So, this is not just reiterating the file type distinction.

Some files contain code which implements their operating system. I expect these *operating system files* be moderately commented.

Other files implement the train project as a set of user programs on top of their operating system. The train project is more difficult than the operating system project, so I expect the *train project files* to be commented more densely than the operating system files.

Finally, some files contain code written *for* their operating system, but which is not part of the train project. These are mostly test programs, or other useful programs, such as those providing the user interface. These are the *user program files*, expected to have relatively few comments because they are more straightforward to implement and less essential.

Figure 4.30 gives the results of a Wilcox rank sum test applied to comment density by task category. This test only uses the relative ordering of the data values, so there is no need to transform the data.

	context switch	operating system	train project	user program
operating system	< 0.001 *			
train project	< 0.001 *	0.321		
user program	< 0.001 *	0.050 *	0.321	
library	< 0.001 *	0.010 *	0.103	0.627

Figure 4.30: Pairwise Wilcox rank sum test results for the effect of task category on comment density in the A-series

Context switch files are by far the most densely commented ($p < 0.001$). The user program and library files are less densely commented than the operating system files ($p < 0.05$). Interestingly, the operating system and train project files are not significantly different, and this data suggests that the operating system files may be more densely commented.

4.2.5 Summary of Comment Density

The proportion of commented lines varies tremendously among their source files. It even varies within each file, but it may be worthy of the name “density”, since it appears to be stable as the files change during the development of the software. The distribution of comment density is affected by group differences, but transforming by a square root gives some approximation of normality. The data provides further evidence that difficult code is commented more densely.

4.3 Comment Length

Now the data is investigated at the level of individual comments. Dividing the raw data into individual comments required manually merging some comment fragments split over several lines. As in previous sections of this chapter, I retain the undesirable comments identified in Section 2.2.2, because excluding them fails to improve the results of this section.

4.3.1 Measuring Comment Length

There are many ways to measure the amount of text in a comment. An obvious measure is word count, but I found that character count has a better distribution. Again, simpler measurements fare better. Word count is also a coarser measure than character count since it ignores the lengths of individual words, which discards a lot of information.

Defining the Text of a Comment

Which characters in the comment text should be counted? Consider the comment in Figure 4.31.

```
default:
        colour = 31;    //RED
        break;
```

Figure 4.31: Excerpt from `a04/p3/src/sys_calls/display.c`

The literal text of this comment is six characters long: "`/`", "`/`", "`R`", "`E`", "`D`", and a newline character which terminates the comment. Perhaps the comment indicator "`//`", as well as the terminating newline, should be excluded. Even more extreme is the comment shown in Figure 4.32.

```
116   /**
117    * Returns the CPSR.
118   **/
119   NOMANGLE_DECL int asm_get_cpsr(void);
```

Figure 4.32: Excerpt from `a01/k4/src/system.h`

This comment has 28 characters in total, including two newlines, four spaces, five asterisks, and two forward slashes. The space at the beginning of line 117 is indeed part of the comment's text.

Therefore, I define the *corrected text* of a comment, as distinguished from its *raw text*, as follows. First, any comment indicators and terminators are removed. For comments with the indicator "`/*`", asterisks are removed from both ends of each line. For comments with other indicators, repetitions of the indicator symbol are removed from the beginning of each line. Then, redundant whitespace is deleted: all whitespace is removed from both ends, and each run of consecutive whitespace characters is replaced by a single space. Finally, runs of five or more of any punctuation symbol are also removed since they are likely parts of banners (in the data, only "`*`", "`#`", "`=`", and "`-`" occurred in this way).

The corrected text of the comment shown in Figure 4.31 is the three-character string "`RED`", and the comment in Figure 4.32 has corrected text "`Returns the CPSR.`", which has only two spaces separating the three words. Some comments are dividers whose raw text is just a banner, such as the comment in Figure 4.33. This comment has a corrected text length of zero since it has no English text.

```
@ Save out IRQ handler data
stmdb   a2!, {ip, sp, lr}
@ ==============================================================
@ Enter supervisor mode
mrs     ip, cpsr
...
```

Figure 4.33: Excerpt from `a06/p3/src/trap.s`

Redacting the data changes some of the text in the files. In previous sections of this chapter, the analysis is unaffected because the files are examined line by line, and redaction does not add or remove lines. In this section, comments with any redacted text are excluded, since the true lengths of these comments are unavailable. This affects relatively few comments, as shown in Figure 4.34, and group A02 was completely unaffected, since none of their comment text was redacted.

Group	Redacted in k4	Redacted in p3
Overall	1.7%	2.5%
A01	0.2%	0.1%
A02	0.0%	0.0%
A03	4.0%	7.1%
A04	0.1%	0.2%
A05	7.9%	3.6%
A06	0.2%	0.1%

Figure 4.34: Proportions of comments with redacted text in the A-series

4.3.2 Distribution of Comment Length

Figure 4.35 gives normal QQ plots of the Log-transformed corrected comment text length.

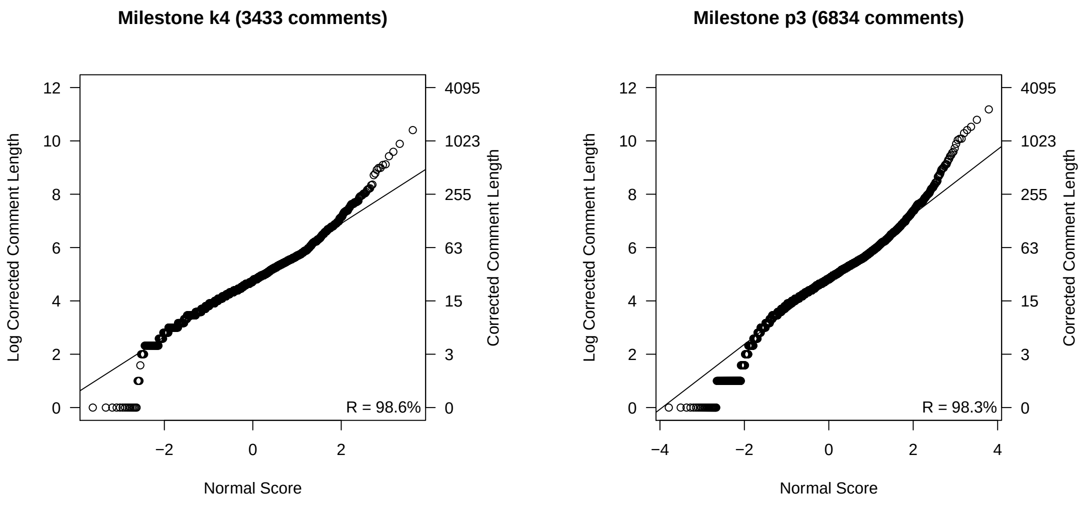


Figure 4.35: Normal QQ plots for Log corrected comment length in the A-series

Centrally the data is well approximated by a lognormal distribution. Deviations from normality for short comments are of little concern because they are inevitable in discrete data. However, there is significant deviation at high values. In the code base breakdown shown in Figure 4.36, every code base except A05/k4 deviates in the same direction. In the earlier results for file length, individual code bases deviated to both sides of the line (Figure 4.7, page 74).

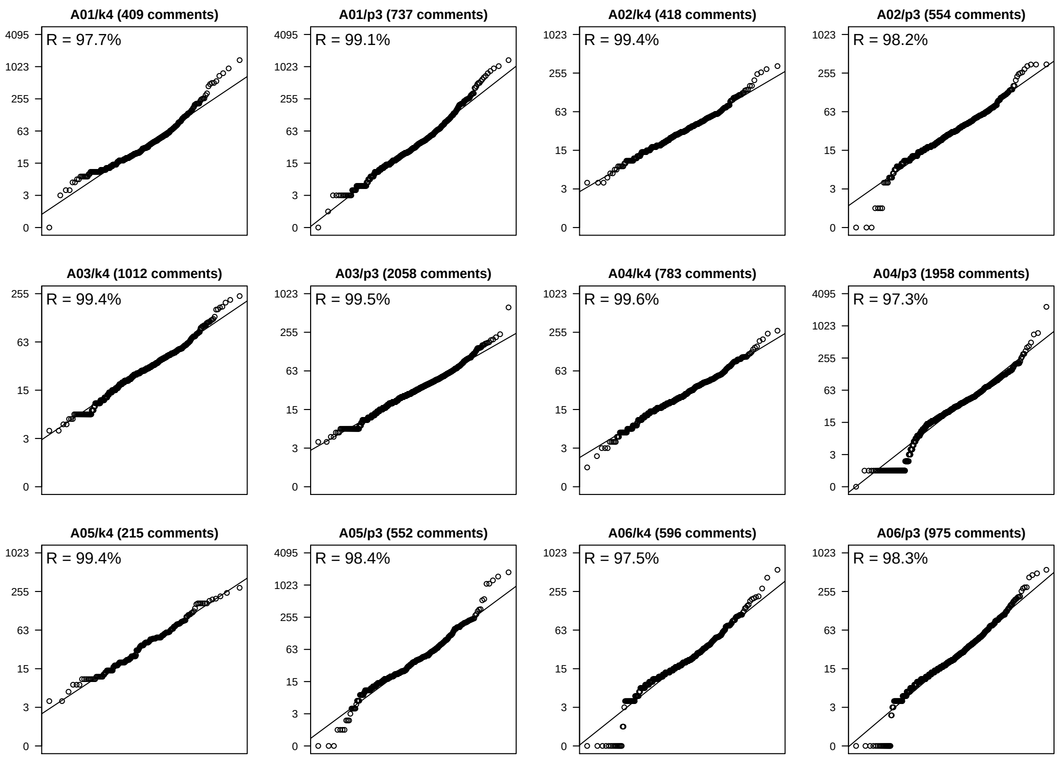


Figure 4.36: Normal QQ plots for Log corrected comment length in the A-series, by code base

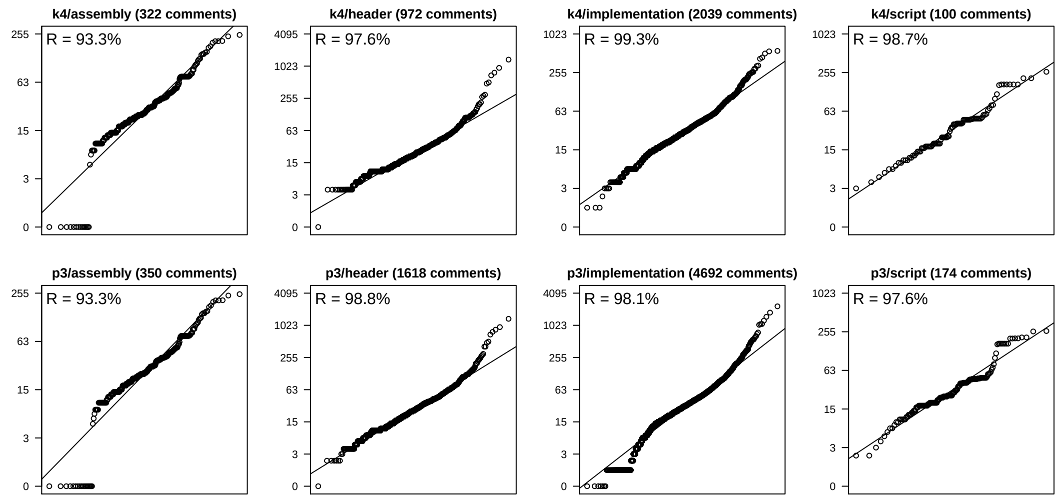


Figure 4.37: Normal QQ plots for Log corrected comment length in the A-series, by file type

4.3.3 Long Tail Behaviour

Thus, it may be significant that the high values for comment length increase more rapidly than would be expected of a normal distribution. In other words, the distribution may have a *long tail*, so named because on a histogram of such data, the right tail has a stretched out appearance.

A random variable X has a long tail if its complementary cumulative distribution function (CCDF) approximates a power law for sufficiently large values, that is, if there are constants c and α such that

$$\Pr[X > x] \approx cx^{-\alpha}$$

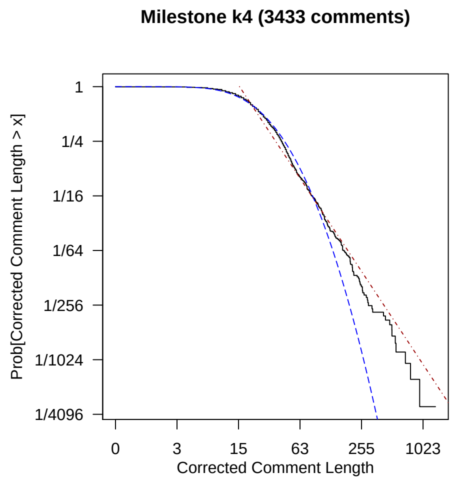
for sufficiently large x. The parameter c determines the scale of the data. The parameter $\alpha > 0$ is the *tail index* of the distribution. This tail index measures the degree of falloff in the right tail, with high values corresponding to a faster falloff. The lognormal distribution falls off faster than any long-tailed distribution, no matter how large its tail index.

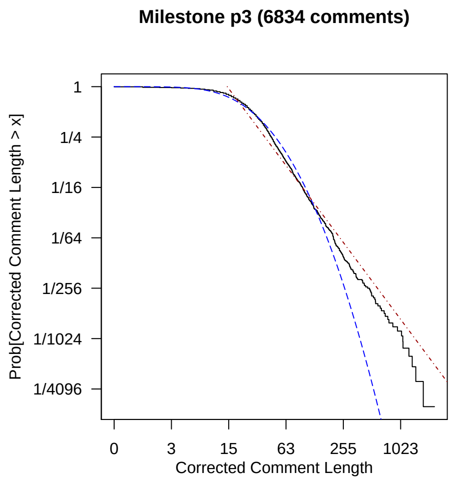
The parameter α has special significance. If $\alpha \leqslant k$, then the kth moment of X is infinite. In particular, if $\alpha \leqslant 2$, then X has infinite variance, and the Central Limit Theorem does not hold for X. For such random variables, the few largest values dominate in any sample. Some authors consider this property as the main characteristic of long tail behaviour, and require $\alpha \leqslant 2$ for a distribution to have a long tail.

The logarithm of the above equation is

$$\log_2 \Pr[X > x] \approx -\alpha \log_2 x + \log_2 c.$$

So, on a log-log plot the CCDF of a long-tailed distribution converges to a straight line with slope $-\alpha$. When a lognormal distribution is graphed in this way, its CCDF curves downward with continually decreasing slope. The empirical CCDF plots for the data are shown in Figure 4.38.





In each plot, the blue dashed curve is the CCDF for the lognormal distribution fit to the data, and the red dot-and-dash line is the CCDF for the power law fit to the upper sixteenth of the data.

Figure 4.38: Empirical CCDFs for corrected comment length in the A-series

The estimated parameters μ^* and σ^* can be used to fit a lognormal distribution to the data. The CCDF of this fitted lognormal distribution is shown as blue dashed lines in Figure 4.38.

I use the method of Crovella and Taqqu to estimate the tail index α; this method is more reliable than linear regression [8]. The estimated tail index is $\alpha = 1.57$ for the data set as a whole. It is best to use the entire data set since this estimate requires a large number of data points.

The estimate for c is chosen so that the fitted distribution agrees with the empirical CCDF at the upper sixteenth quantile of the data. The CCDF of the fitted power law is shown as red dot-and-dash lines in Figure 4.38. It appears to fit the right tail of the data better than a lognormal distribution.

This graphical test can be made more objective, using a method of Downey [11]. Downey defines a measure of *tail curvature*, which approximates the second derivative of the log-log plotted empirical CCDFs shown in Figure 4.38. For the null hypothesis of a lognormal or power law distribution, by drawing many samples from a fit of this distribution to the data, an empirical distribution for the tail curvature of the sample under the null hypothesis is obtained. Then, the tail curvature of the actual data is compared with this empirical distribution of tail curvatures to obtain a simulation-based p value. This test is completely unaffected by the value of c, so the arbitrary choice of c poses no problem.

I follow Downey's convention, and consider the tail region to be the upper sixteenth of the data for the purposes of this test. The test rejects that the tail of the comment length distribution is lognormal with $p < 0.001$. This validates the presence of long tail behaviour in the data. It fails to reject that the tail follows a power law, with $p = 0.629$.

There are not enough data points to compute estimates of the tail index or to apply Downey's test to any individual code base. The same is true for applying the test to the distributions of line count and commented line count from Section 4.1. In all cases there is failure to reject that the distribution is lognormal, as well as failure to reject that it follows a power law. A large number of data points are required because most are ignored: Downey's test uses less than 7% of the data!

In light of the upper tail being decidedly non-normal, it makes sense to exclude the upper tail from linear regression on the normal QQ plots. Figure 4.39 shows the R values fit to just the central fourteen sixteenths of the data, which is delimited by dashed lines. I also exclude the lower tail since I am not concerned by deviations there. Figures 4.40 and Figures 4.41 show the breakdown by code base and by file type, and there are greatly improved R values.

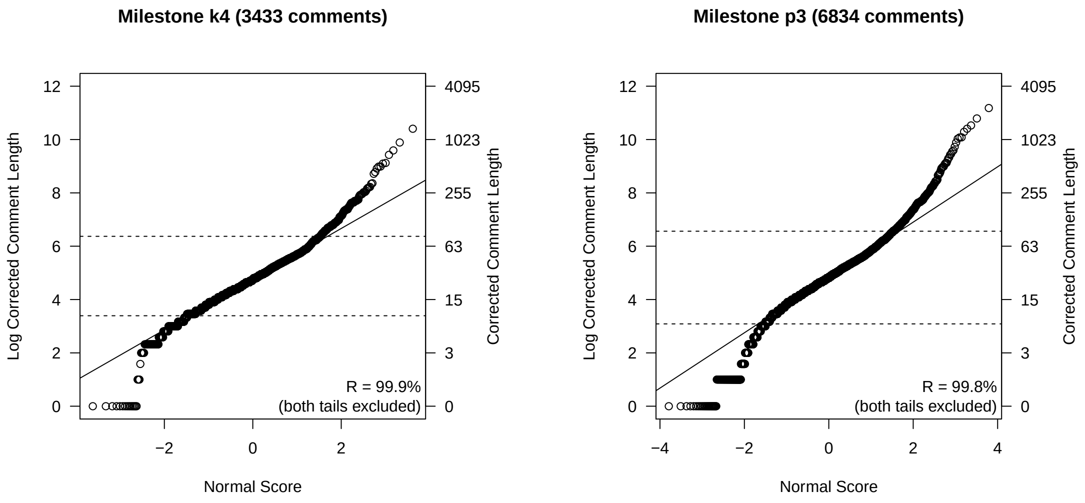


Figure 4.39: Normal QQ plots for Log corrected comment length in the A-series, with both tails excluded from the fit

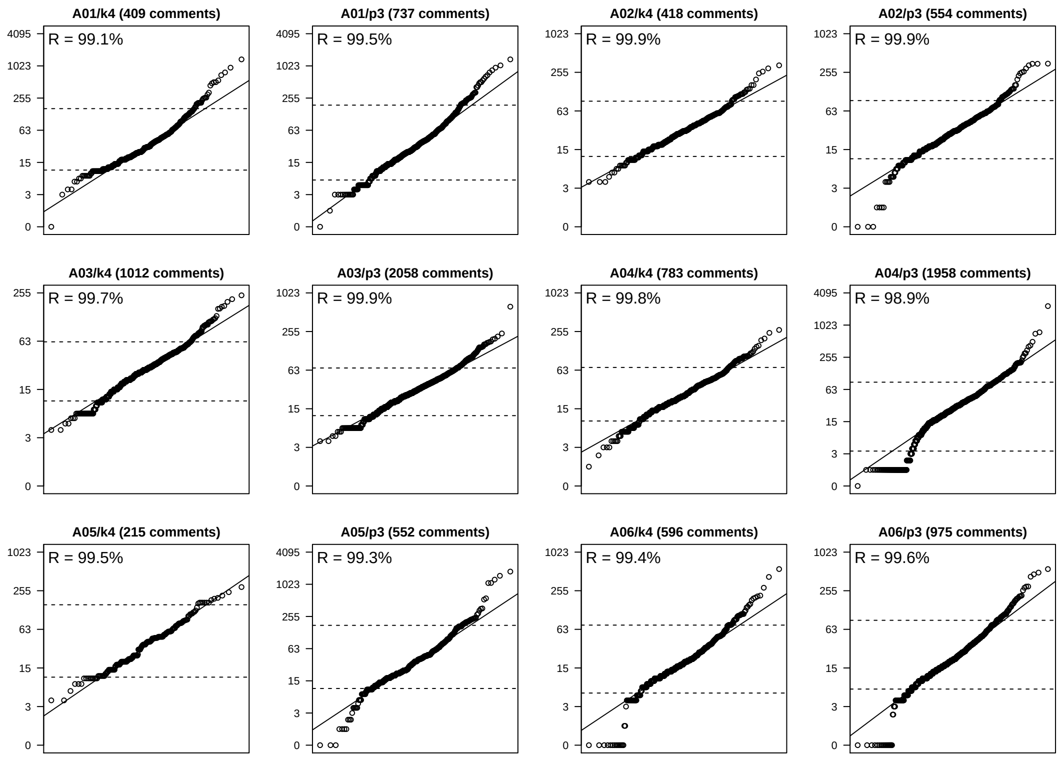


Figure 4.40: Normal QQ plots for Log corrected comment length in the A-series, with both tails excluded from the fit, by code base

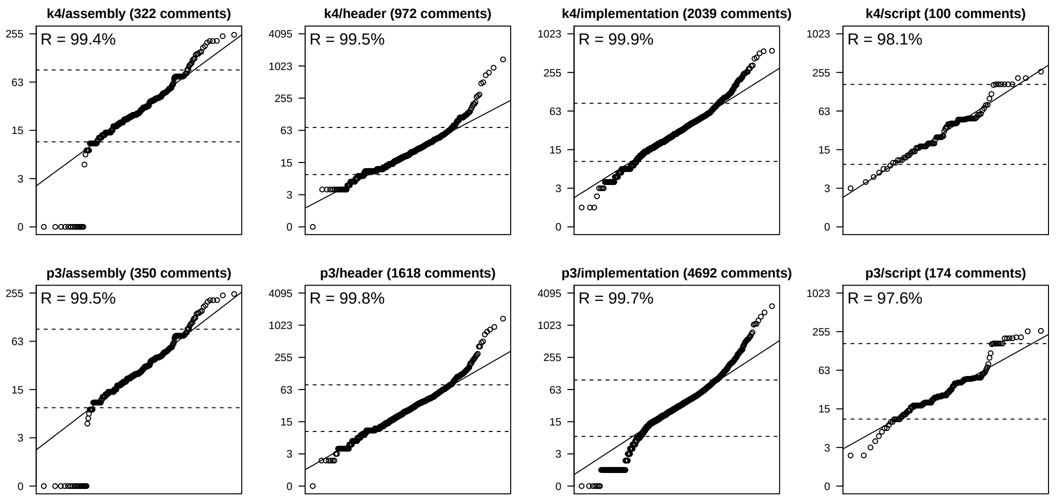


Figure 4.41: Normal QQ plots for Log corrected comment length in the A-series, with both tails excluded from the fit, by file type

4.3.4 Factor Analysis for Comment Length

Since the bulk of the data is lognormally distributed, an ANOVA can be performed on the Log-transformed data. Both the upper and lower sixteenths of the data are excluded to improve normality. Figure 4.42 provides the results of the ANOVA.

	df	sum of squares	mean square	F	p
milestone	1	6.875	6.875	12.963	< 0.001 *
group	5	95.838	19.168	36.140	< 0.001 *
file type	3	94.442	31.481	59.355	< 0.001 *
error	8652	4588.827	0.530		
Total	8661	4785.982			

Figure 4.42: ANOVA summary for Log corrected comment length in the A-series

Every tested factor is significant, so Figure 4.43 gives the post-hoc comparisons.

	implementation	assembly	script
assembly	0.031 *		
script	0.203	0.997	
header	< 0.001 *	< 0.001 *	0.025 *

	A05	A03	A02	A01	A04
A03	0.095				
A02	0.016 *	0.745			
A01	0.005 *	0.446	0.999		
A04	< 0.001 *	0.091	0.995	1.000	
A06	< 0.001 *	< 0.001 *	< 0.001 *	< 0.001 *	< 0.001 *

Figure 4.43: Tukey post-hoc comparisons for Log corrected comment length in the A-series

Header files have the shortest comments ($p < 0.03$). Implementation files have longer comments than assembly and header flies ($p < 0.04$). Group A05 writes longer comments than every other group except A03 ($p < 0.02$). Group A06 writes the shortest comments ($p < 0.001$).

Given that the group differences are significant, it is worth looking at the estimated location and scale parameters shown in Figure 4.44.

	k4 μ^*	k4 σ^*	p3 μ^*	p3 σ^*
Overall	26 characters	2.1x	27 characters	2.3x
Group A01	33 characters	2.5x	32 characters	2.7x
Group A02	30 characters	1.9x	28 characters	2.2x
Group A03	27 characters	1.8x	29 characters	1.8x
Group A04	25 characters	1.9x	24 characters	2.5x
Group A05	35 characters	2.2x	36 characters	2.7x
Group A06	19 characters	2.4x	21 characters	2.4x

Figure 4.44: Parameters for the lognormal distributions fit to corrected comment length in the A-series

Here the variation is less extreme than seen for file length and commented line count. However, the differences in μ^* between some pairs of groups are significant by the ANOVA (Figure 4.43). Figure 4.45 shows the bootstrap confidence intervals for the estimates of σ^*.

	k4 σ^*	p3 σ^*
Overall	2.05x–2.17x	2.31x–2.42x
Group A01	2.31x–2.74x	2.58x–2.91x
Group A02	1.84x–2.05x	2.04x–2.34x
Group A03	1.71x–1.82x	1.72x–1.79x
Group A04	1.83x–1.97x	2.49x–2.68x
Group A05	2.09x–2.41x	2.48x–2.93x
Group A06	2.25x–2.60x	2.31x–2.59x

Figure 4.45: 95% bootstrap confidence intervals for the multiplicative standard deviation estimates for corrected comment length in the A-series

There are many more comments than files: this time there are over ten times as many data points. The estimates of σ^* are much more precise, and many pairs of intervals within each milestone are disjoint. Therefore the value of σ^* for the comment length distribution cannot be universal across groups.

4.3.5 Summary of Comment Length

The character count of individual comments follows a lognormal distribution, except for the right tail, which shows long tail behaviour with tail index approximately $\alpha = 1.57$. The location and scale parameters vary by group and by milestone. There are not enough data points to validate the long tail behaviour of individual groups, nor to estimate the tail index of each group.

4.4 Evaluation on the B-series

As described in Chapter 2, the data set was collected from three different offerings of the course, and these three portions of the data are labelled the A-, B-, and C-series. All investigation so far was exploratory analysis of the A-series, the pilot data set. Now the B- and C-series which make up the test data set are investigated. I processed the B-series first, so that the hypotheses could be adjusted for the C-series if necessary.

4.4.1 Evaluation Philosophy

The most robust observations from analyzing the results of the A-series data in Sections 4.1–4.3 were collected into a set of explicit hypotheses. The hypotheses were then tested against the B-series data set, which was hidden until the hypotheses were finalized. Indeed, I wrote the complete draft of this work up to the end of Section 4.4.2 before contaminating myself by analyzing any of the test data set.

I tested only the strongest observations to ensure that this evaluation is worthwhile. Also, my understanding of many of the observations is not precise enough to permit them to be tested objectively. However, the test data set does not have to be completely used up now; if the right tool is developed later, then additional hypotheses can be tested, either by me or by other researchers.

It is important to make the hypotheses objective to remove the bias in human judgement. Fortunately, excluding the undesirable comments, which were identified subjectively, failed to improve the results in Sections 4.1–4.3, so this step is omitted for the test data set. Only the automatic merging of line comments was applied to the test data set. The quantitative features of the data should be little affected, because relatively few comments – only 3% – received manual merging or splitting in the pilot data set.

Owing to my methodology, arbitrary choices in the hypotheses need not be justified because a bad hypothesis simply fails on the test data set. In some sense, the ends justify the means: if the hypotheses are validated then they are correct. Adding unnecessary detail increases the risk, and is punished by testing on hidden data.

4.4.2 Forming Explicit Hypotheses

I consider data to be lognormally distributed if its Log-transform is normally distributed, where Log is the *shifted* logarithm transformation

$$\mathrm{Log}(x) \;=\; \log_2(x+1).$$

Quality of fit to the lognormal distribution is measured graphically using a normal QQ plot, and quantitatively using the R value obtained from linear regression on the normal QQ plot.

Total Line Count by File

These hypotheses are for total number of lines of code by file, and are based on the results in Section 4.1.2.

- (H1a) For each milestone, total line count by file fits a lognormal distribution.
- (H1b) For each code base, total line count by file fits a lognormal distribution.

Commented Line Count by File

These hypotheses are for commented lines of code by file, and are based on the results in Section 4.1.3. A *commented line* is a line of code containing part of a comment, as defined by the programming language syntax. For linear regression on the QQ plot, zero values are excluded from the fit but are still assigned the appropriate normal scores for their rank.

- (H2a) For each milestone, commented line count by file fits a lognormal distribution.

- (H2b) For each code base, commented line count by file fits a lognormal distribution.

Corrected Comment Text Length

The *corrected text length* of a comment is the number of characters in the comment text after removing superfluous punctuation and whitespace (see Section 4.3.1 for the precise definition). These hypotheses are for the distribution of corrected comment text length for individual comments, and are based on the results in Section 4.3.2. For linear regression on the QQ plot, values from the upper and lower sixteenths of the data are excluded, but are still assigned the appropriate normal scores for their rank.

- (H3a) For each milestone, the central part of the distribution of corrected comment length fits a lognormal distribution.
- (H3b) For each code base, the central part of the distribution of corrected comment length fits a lognormal distribution.

Long Tail Behaviour of Comment Length

These hypotheses test the long tail behaviour of corrected comment text length, and come from Section 4.3.3. Downey's test distinguishes between lognormal and power law distributions in the upper tail of the data [11]. The method of Crovella and Taqqu estimates the tail index [8]. The tail index is hypothesized to be between 1 and 2 because the accuracy of the measurement is unknown, and this range captures the most important qualitative properties (finite mean and infinite variance). Each of these hypotheses applies to the B-series as a whole, because a large number of data points is needed when studying tail behaviour.

- (H4a) Downey's test rejects that the upper tail of the corrected comment length distribution is lognormal.

- (H4b) Downey's test fails to reject that the upper tail of the corrected comment length distribution is a power law. This is included as a sanity check, since failure to reject is not significant.
- (H4c) The estimated tail index α for the corrected comment length distribution is between 1 and 2.

Estimates of σ^* for Total Line Count by File

Based on the results of Section 4.3.4, the estimated multiplicative standard deviation σ^* for comment length varies by group. In Section 4.1.2 the differences in σ^* for file length are just short of significance. So, the test data set should provide enough additional data for these differences to be significant. Therefore, I also attempt to falsify the following hypotheses, which are consistent with the A-series data.

- (H5a) The 95% bootstrap confidence interval for σ^* of the lognormal distribution fit to line count for each code base of milestone k4 contains 2.9.
- (H5b) The 95% bootstrap confidence interval for σ^* of the lognormal distribution fit to line count for each code base of milestone p3 contains 3.3.

Validation on the Pilot Data Set

As a sanity check, these hypotheses were first tested on the pilot data set, excluding H5a and H5b which cannot be falsified by the pilot data. The only failure is that the commented line count distribution in code base A05/k4 has a relatively poor fit with $R = 92.0\%$ (Figure 4.15, page 81). The likely reason for this was addressed in Section 4.1.3. The stated hypothesis H2b makes no attempt to compensate for this, because I do not want to introduce additional complications for the sake of a single exception in the pilot data set.

4.4.3 B-series Results

Total Line Count by File

Figures 4.46–4.48 show the results for lognormality of total line count in the B-series. The R values shown in Figure 4.46 are comparable with those for the A-series, excluding group B07. Looking at Figure 4.48, however, group B07 has many fewer files than the other groups. Indeed, the file `kernel.c` contains essentially their entire program. As a post-hoc test, a Wilcox rank sum test can be applied to compare the R values in the A-series with those obtained here for the B-series. However, this test falls short of significance ($p = 0.195$). Graphically, deviations from normality are no worse than in the A-series (compare to Figure 4.7, page 74), with the possible exception of code base B04/p3.

	k4 R value	p3 R value
Overall	99.8%	99.8%
Group B01	99.0%	99.1%
Group B02	98.2%	98.1%
Group B03	98.3%	98.3%
Group B04	97.6%	96.0%
Group B05	96.8%	98.9%
Group B06	97.8%	99.0%
Group B07	88.7%	93.0%
Group B08	99.2%	99.0%
Group B09	99.2%	99.6%
Group B10	99.3%	98.4%
Lowest Group	88.7%	93.0%
Second Lowest Group	96.8%	96.0%
Third Lowest Group	97.6%	98.1%
A-series Lowest Group	98.0%	98.5%
A-series Second Lowest Group	98.7%	98.5%
A-series Third Lowest Group	98.7%	98.6%

Figure 4.46: Goodness of fit of the lognormal distribution for total line count by file in the B-series

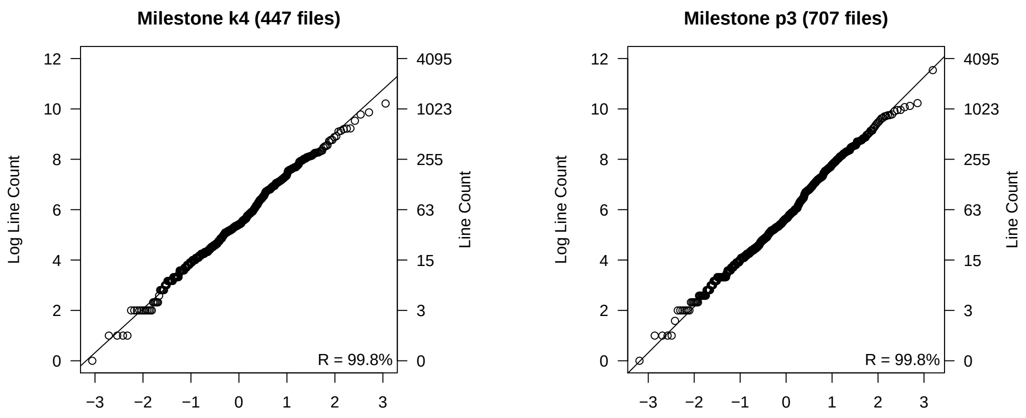


Figure 4.47: Normal QQ plots for Log line count in the B-series

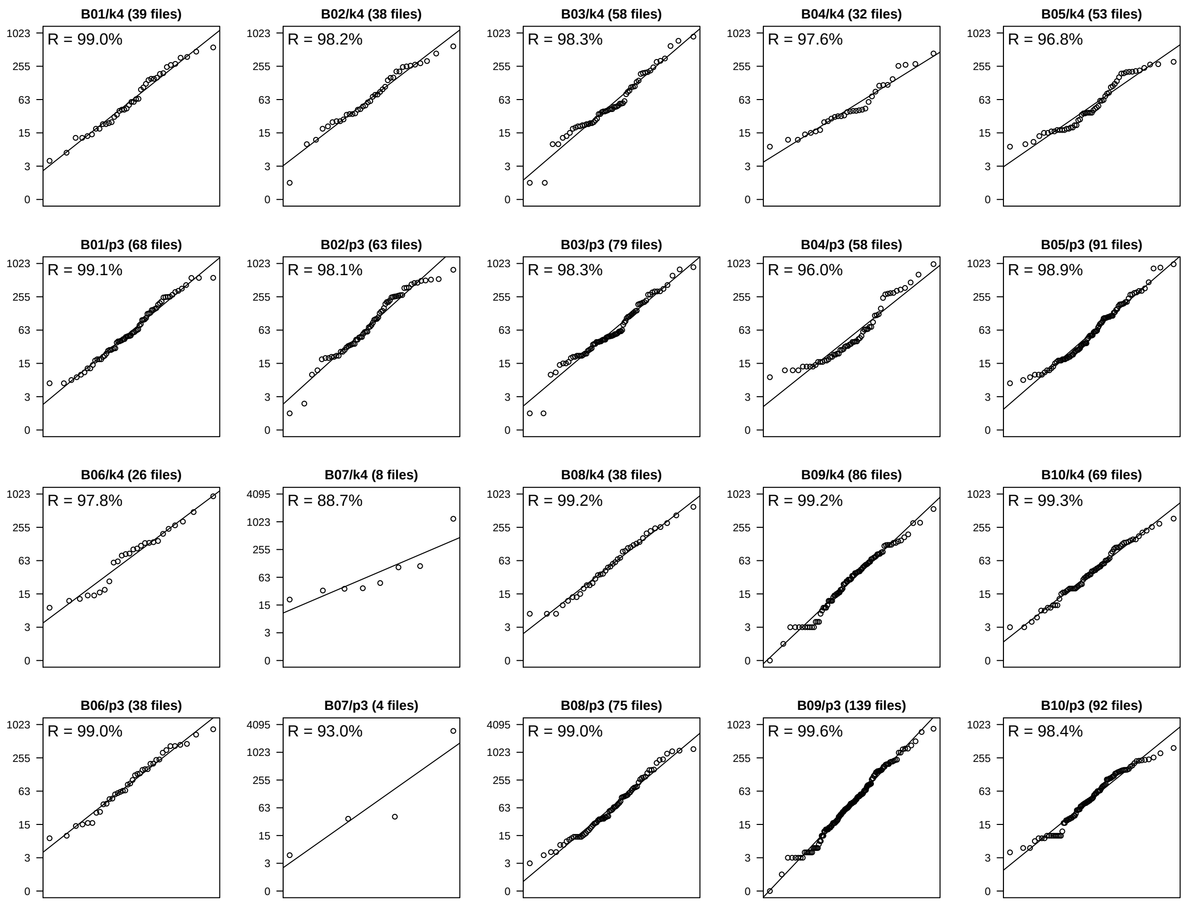


Figure 4.48: Normal QQ plots for Log line count in the B-series, by code base

Commented Line Count by File

Figures 4.49–4.51 show the results for the test of lognormality for commented line count in the B-series. In the A-series, excluding the outlier A05/k4, the lowest R value is $R = 98.2\%$. Here the R values are lower, even if the few lowest values are excluded as potential outliers. A Wilcox rank sum test finds that the differences in R values between the A- and B-series are significant at the 0.05 level ($p = 0.029$), but the differences themselves are small. In the B-series, the fit for milestone p3 is much better than for milestone k4; the R values for milestone p3 are closer to those seen for the A-series. Deviations from normality are no worse than for the A-series (compare Figure 4.15, page 81), even for groups B04 and B07 which have few files with comments.

	k4 R value	p3 R value
Overall	99.2%	99.5%
Group B01	94.9%	99.0%
Group B02	98.6%	98.5%
Group B03	97.9%	98.2%
Group B04	93.9%	97.3%
Group B05	97.6%	99.2%
Group B06	98.5%	98.8%
Group B07	92.1%	98.2%
Group B08	94.8%	96.7%
Group B09	98.4%	99.2%
Group B10	98.9%	98.3%
Lowest Group	92.1%	96.7%
Second Lowest Group	93.9%	97.3%
Third Lowest Group	94.8%	98.2%
A-series Lowest Group	92.0%	98.3%
A-series Second Lowest Group	98.2%	98.7%
A-series Third Lowest Group	98.6%	99.0%

Figure 4.49: Goodness of fit of the lognormal distribution for commented line count by file in the B-series

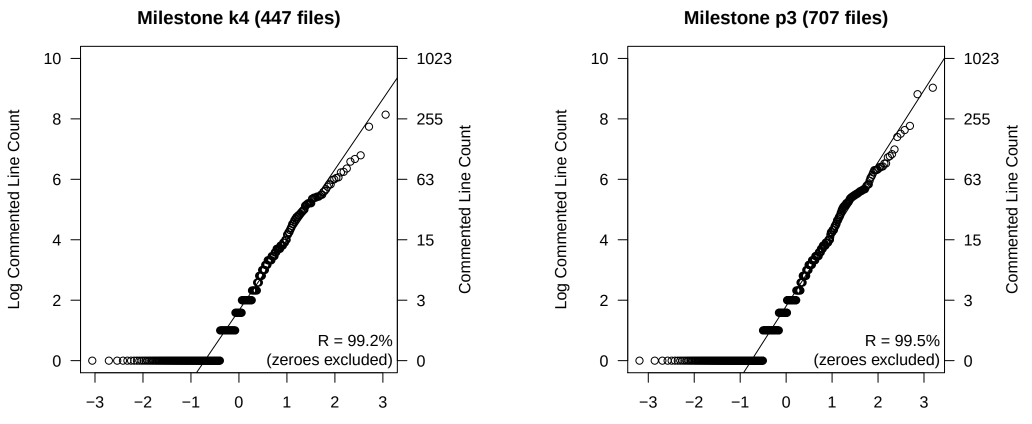


Figure 4.50: Normal QQ plots for Log commented line count in the B-series

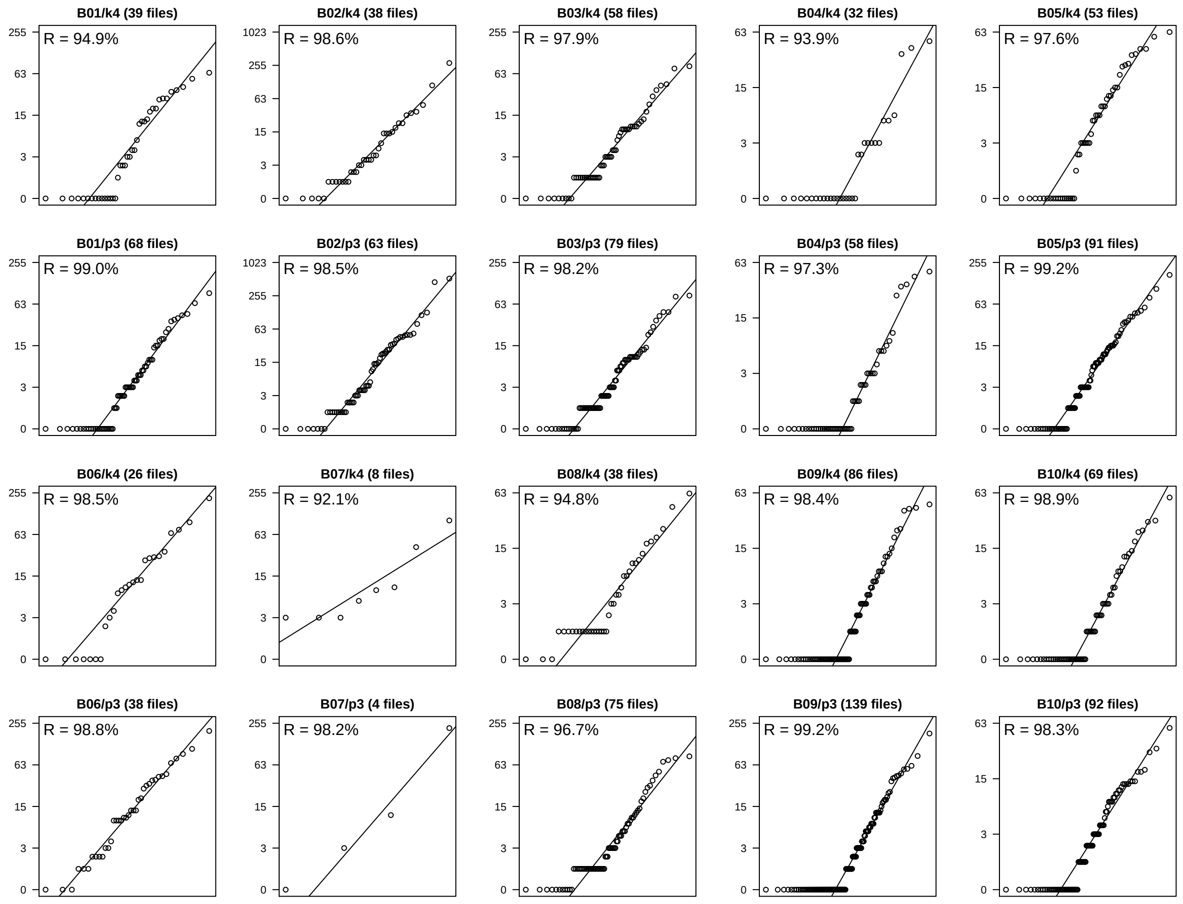


Figure 4.51: Normal QQ plots for Log commented line count in the B-series, by code base

Corrected Comment Text Length

Figures 4.52–4.54 show the results for lognormality of the central part of the distribution of corrected comment length in the B-series. The fit is good in the centre of the distribution, and the R values are relatively high: the lowest is $R = 97.8\%$. However, again the R values are lower than in the A-series where the minimum R value was $R = 98.9\%$ (Figure 4.40, page 104). This difference is again significant at the 0.05 level by a Wilcox rank sum test, with $p = 0.036$. The R values seen here may be more realistic, due to overfitting in the analysis of the A-series.

	k4 R value	p3 R value
Overall	99.8%	99.7%
Group B01	99.2%	99.1%
Group B02	99.7%	98.8%
Group B03	97.9%	99.4%
Group B04	98.5%	99.4%
Group B05	99.8%	99.9%
Group B06	97.8%	98.1%
Group B07	99.4%	99.7%
Group B08	98.8%	99.8%
Group B09	99.3%	99.5%
Group B10	98.9%	99.4%
Lowest Group	97.8%	98.1%
Second Lowest Group	97.9%	98.8%
Third Lowest Group	98.5%	99.1%
A-series Lowest Group	99.1%	98.9%
A-series Second Lowest Group	99.4%	99.3%
A-series Third Lowest Group	99.5%	99.5%

Figure 4.52: Goodness of fit of the lognormal distribution for corrected comment text length in the B-series

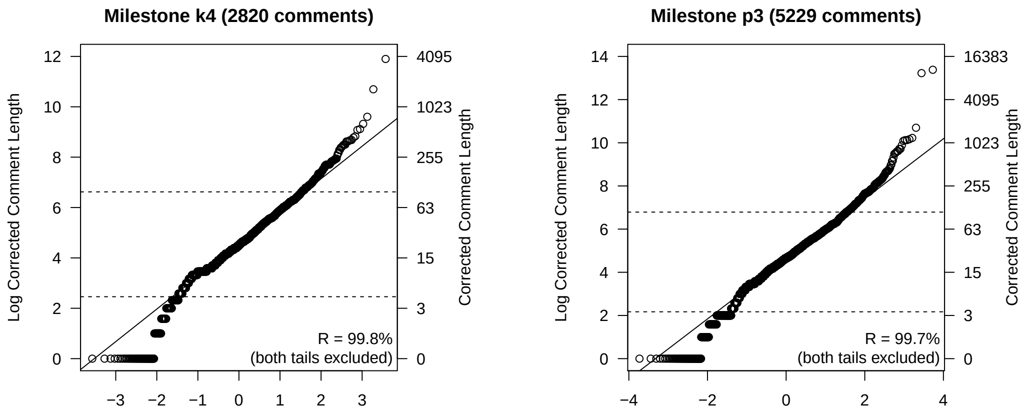


Figure 4.53: Normal QQ plots for Log corrected comment length in the B-series

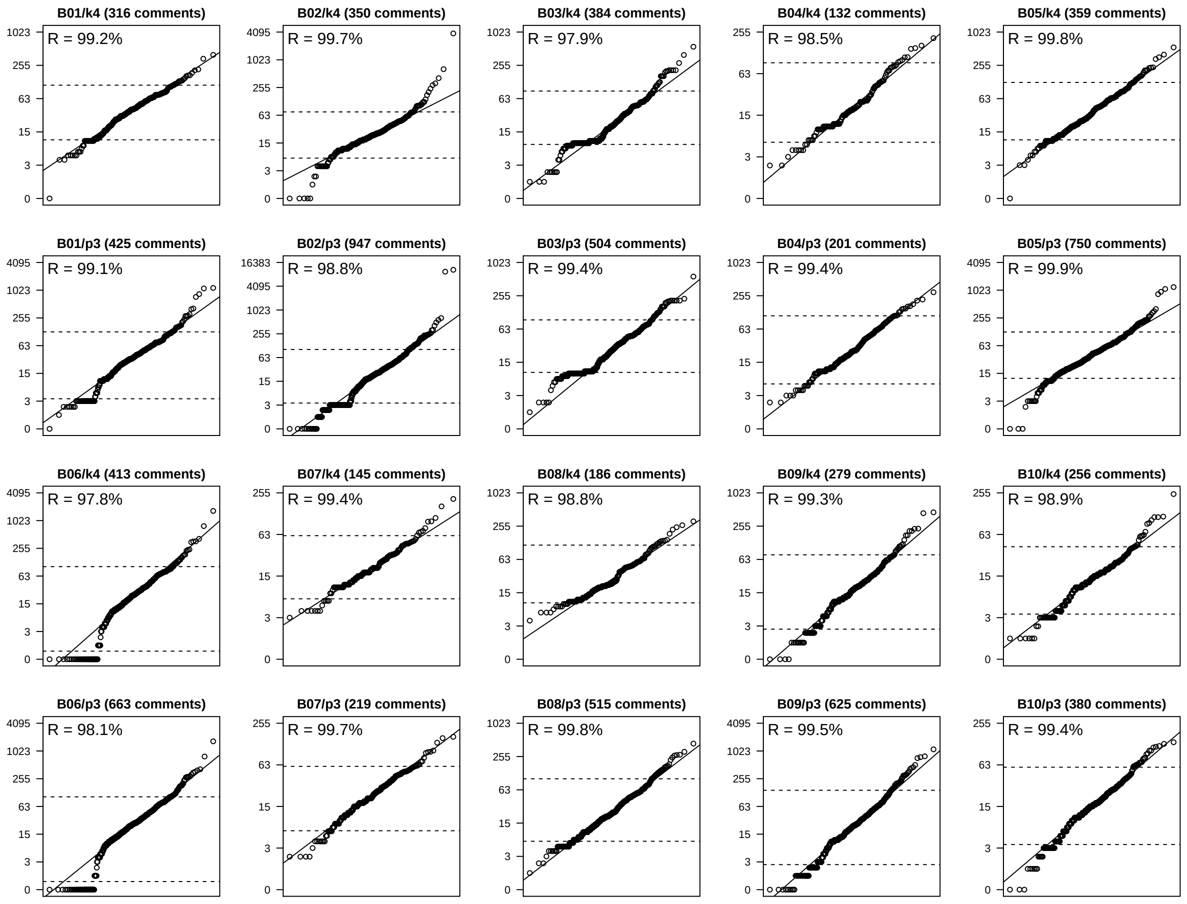
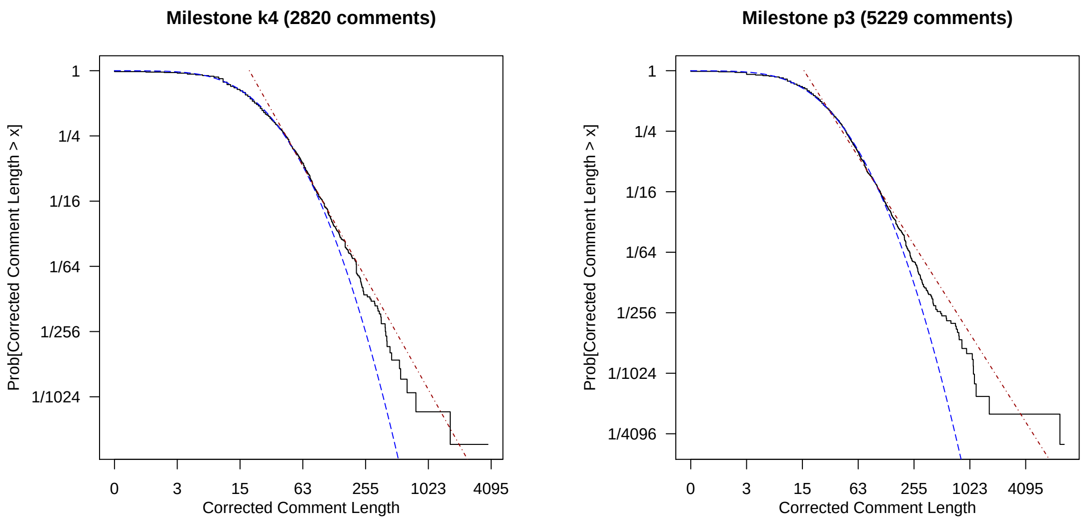


Figure 4.54: Normal QQ plots for Log corrected comment length in the B-series, by code base

Long Tail Behaviour of Comment Length

The hypotheses for long tail behaviour of corrected comment length are consistent with the B-series data, as shown in Figure 4.56. The tail index estimate for the B-series is also quite close to the A-series tail index estimate of 1.57. Unfortunately, it is difficult to obtain a confidence interval for this measurement to determine whether the difference in estimated tail index is significant.



In each plot, the blue dashed curve is the CCDF for the lognormal distribution fit to the data, and the red dot-and-dash line is the CCDF for the power law fit to the upper sixteenth of the data.

Figure 4.55: Empirical CCDFs for corrected comment length in the B-series

p value for lognormal distribution	0.001 *
p value for power law	0.199
estimate of tail index α	1.54

Figure 4.56: Results for Downey's test applied to corrected comment length in the B-series

Estimates of σ^* for Total Line Count by File

I realized after testing on the B-series that hypotheses H5a and H5b were poorly designed. For milestone p3, the value 3.3 was arbitrarily chosen, but any value of σ^* between 3.233 and 3.507 is consistent with the A-series data. Also, nothing was done to adjust the 95% confidence levels to compensate for multiple testing. The results for the B-series data technically falsify hypotheses H5a and H5b, but this means little in light of these problems.

Since line count is lognormally distributed with no truncation, traditional tests for homogeneity of variance can be used. However, both Bartlett's test and the Fligner-Killeen test fail to reject the null hypothesis that the variance of the Log-transformed data is the same for each code base. They even fail to reject if the A- and B-series data are combined. Therefore I abandoned these hypotheses, since there is insufficient statistical power to obtain a precise enough estimate of σ^*.

4.4.4 Summary of B-series Results

Overall, the hypotheses in Section 4.4.2 are consistent with the B-series data.

The hypotheses H1a, H2a, and H3a concerning lognormal distributions fit to various quantities by milestone were validated, and give comparable R values to the A-series. The hypotheses H1b, H2b, and H3b which examine the distribution of these same quantities for each individual code base were also validated, but here the quality of fit is slightly lower than for the A-series, and this difference is statistically significant for H2b and H3b.

Hypotheses H4a, H4b, and H4c concerning long tail behaviour of the comment length distribution were validated perfectly.

Hypotheses H5a and H5b were validated, but were too poorly stated to be interesting. More careful post-hoc testing failed to give any significant result, so I abandoned further attempts to validate these hypotheses using the test data set.

4.5 Evaluation on the C-series

Minor changes were made to the hypotheses before I looked at the C-series. Hypotheses H5a and H5b are dropped. Hypotheses H1b, H2b, and H3b are extended by comparing the R values for the C-series to those of the A- and B-series using a Wilcox rank sum test with significance at the 0.05 level. Comparing to the B-series may be more realistic, since both the B- and C-series data are complete, unlike the A-series.

The hypotheses are summarized below; see Section 4.4.2 for the full descriptions. For hypotheses H4a, H4b, and H4c, the data for milestones k4 and p3 is combined.

- (H1a) For each milestone, total line count by file fits a lognormal distribution.
- (H1b) For each code base, total line count by file fits a lognormal distribution.
- (H2a) For each milestone, commented line count by file fits a lognormal distribution.
- (H2b) For each code base, commented line count by file fits a lognormal distribution.
- (H3a) For each milestone, the central part of the distribution of corrected comment length fits a lognormal distribution.
- (H3b) For each code base, the central part of the distribution of corrected comment length fits a lognormal distribution.
- (H4a) Downey's test rejects that the upper tail of the corrected comment length distribution is lognormal.
- (H4b) Sanity check: Downey's test fails to reject that the upper tail of the corrected comment length distribution is a power law.
- (H4c) The estimated tail index α for the corrected comment length distribution is between 1 and 2.

4.5.1 C-series Results

Total Line Count by File

Figures 4.57–4.59 show the results for lognormality of total line count in the C-series. The fit to a lognormal distribution looks good for each code base in Figure 4.59, except for code base C02/p3 (compare to Figure 4.7 on page 74 and Figure 4.48 on page 114). Code base C09/k4 has a markedly lower R value than the others, but it has relatively few data points. The C-series R values are not significantly lower than those for the A-series ($p = 0.501$), nor are they significantly lower than those for the B-series ($p = 0.495$).

	k4 R value	p3 R value
Overall	99.6%	99.6%
Group C01	99.2%	98.9%
Group C02	97.5%	97.3%
Group C03	98.7%	98.7%
Group C04	98.3%	98.8%
Group C05	98.6%	99.2%
Group C06	99.3%	99.1%
Group C07	98.1%	98.8%
Group C08	98.4%	99.7%
Group C09	95.9%	99.1%
Group C10	98.3%	98.8%
Lowest Group	95.9%	97.3%
Second Lowest Group	97.5%	98.7%
Third Lowest Group	98.1%	98.8%
A-series Lowest Group	98.0%	98.5%
A-series Second Lowest Group	98.7%	98.5%
A-series Third Lowest Group	98.7%	98.6%
B-series Lowest Group	88.7%	93.0%
B-series Second Lowest Group	96.8%	96.0%
B-series Third Lowest Group	97.6%	98.1%

Figure 4.57: Goodness of fit of the lognormal distribution for total line count by file in the C-series

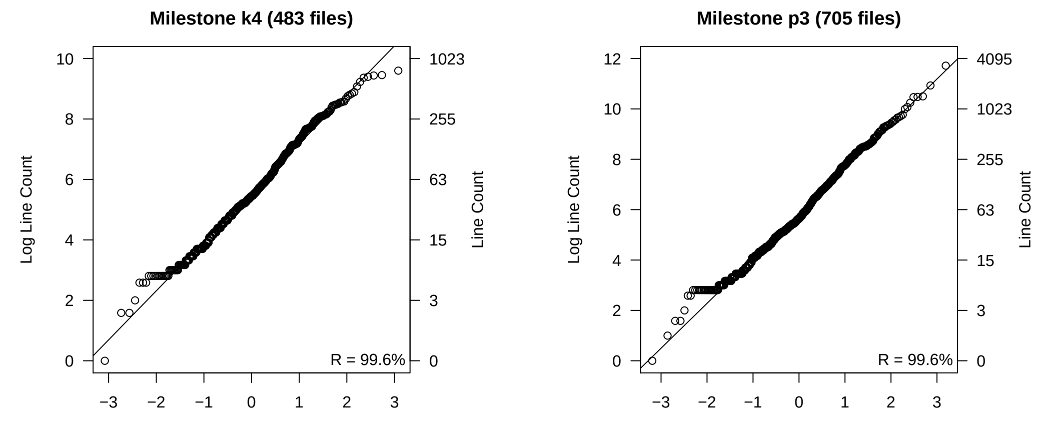


Figure 4.58: Normal QQ plots for Log line count in the C-series

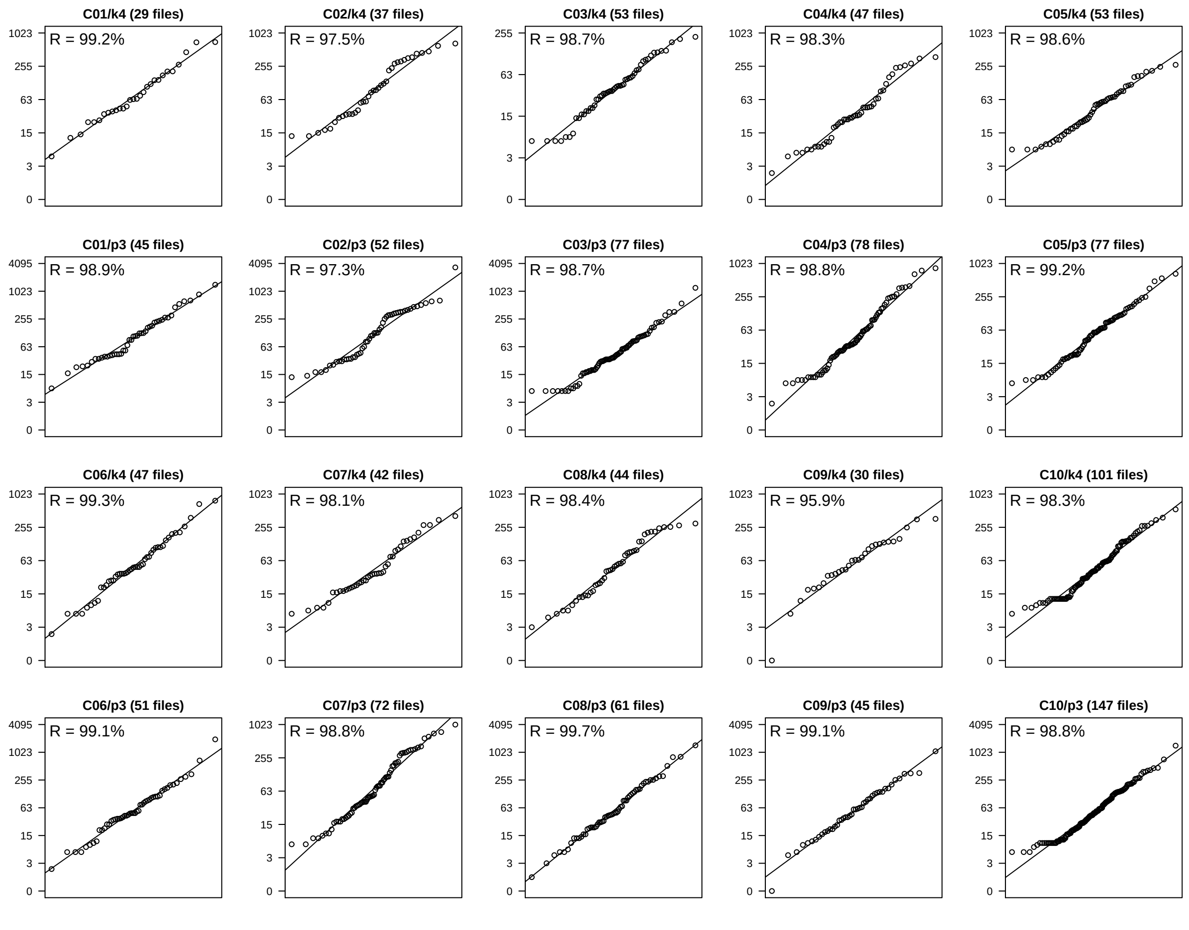


Figure 4.59: Normal QQ plots for Log line count in the C-series, by code base

Commented Line Count by File

Figures 4.61–4.63 show the results for lognormality of commented line count in the C-series. Graphically, the deviations from normality for most groups are comparable to the A- and B-series (Figure 4.15 on page 81 and Figure 4.51 on page 116). Groups C03, C06, and C07 deviate by having too many files with exactly the same commented line count. For group C06 this is caused by six similar Makefiles in each code base with exactly ten commented lines each. For groups C03 and C07, nearly all files with one commented line are header files whose only comment is an end block comment like the one shown in Figure 4.60.

```
#ifndef CLOCK_NOTIFIER_H
#define CLOCK_NOTIFIER_H

void ClockNotifierTask(void);

#endif // CLOCK_NOTIFIER_H
```

Figure 4.60: The entirety of `c03/p3/src/clock/clocknotifier.h`

The C-series R values are significantly lower than those for the A-series ($p = 0.005$). However, they are *not* significantly lower than those for the B-series ($p = 0.355$). The B-series is a fairer comparison since it has the same number of groups, and since the A-series has bias because four groups did not respond with consent for their code to be studied.

	k4 R value	p3 R value
Overall	99.4%	99.4%
Group C01	98.0%	98.3%
Group C02	99.1%	99.3%
Group C03	85.3%	86.7%
Group C04	97.7%	97.0%
Group C05	95.5%	98.4%
Group C06	96.0%	96.8%
Group C07	97.0%	98.0%
Group C08	98.9%	98.9%
Group C09	96.0%	98.3%
Group C10	97.1%	97.1%
Lowest Group	85.3%	86.7%
Second Lowest Group	95.5%	96.8%
Third Lowest Group	96.0%	97.0%
A-series Lowest Group	92.0%	98.3%
A-series Second Lowest Group	98.2%	98.7%
A-series Third Lowest Group	98.6%	99.0%
B-series Lowest Group	92.1%	96.7%
B-series Second Lowest Group	93.9%	97.3%
B-series Third Lowest Group	94.8%	98.2%

Figure 4.61: Goodness of fit of the lognormal distribution for commented line count by file in the C-series

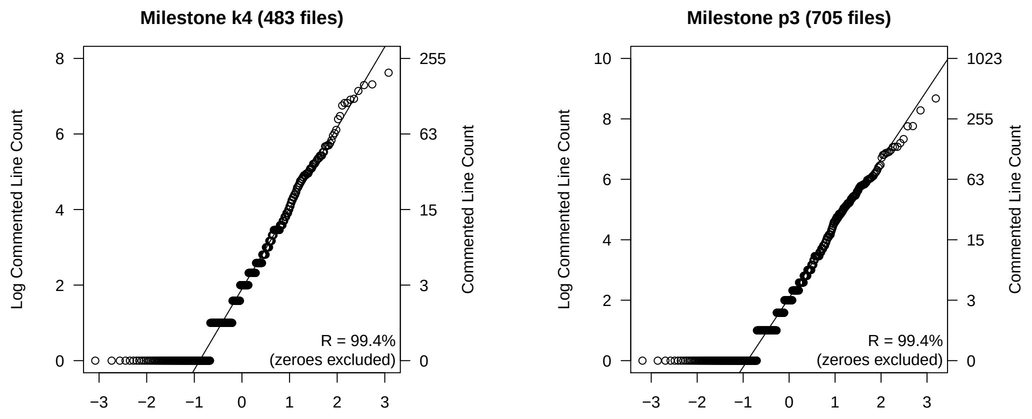


Figure 4.62: Normal QQ plots for Log commented line count in the C-series

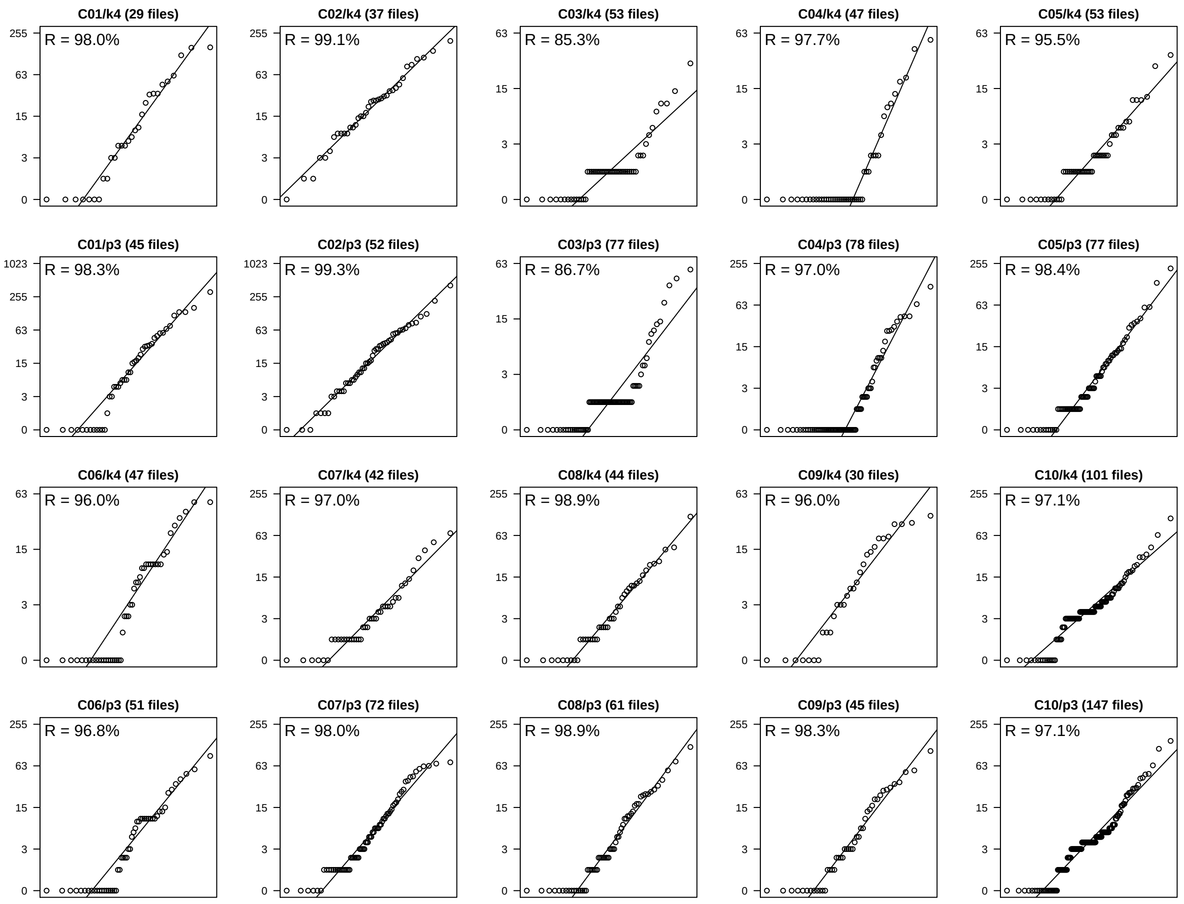


Figure 4.63: Normal QQ plots for Log commented line count in the C-series, by code base

Corrected Comment Text Length

Figures 4.64–4.66 show the results for lognormality of the central part of the distribution of corrected comment length in the C-series. Each code base is well-fit by a lognormal distribution: there are no significant deviations in the central part of the distribution (compare to Figure 4.40 on page 104 and Figure 4.54 on page 118). The differences in R values between the C- and A-series are nearly significant ($p = 0.064$). The C-series R values are not significantly lower than those for the B-series ($p = 0.904$).

	k4 R value	p3 R value
Overall	99.8%	99.8%
Group C01	99.9%	99.6%
Group C02	99.9%	99.9%
Group C03	99.4%	99.3%
Group C04	98.6%	99.8%
Group C05	98.1%	98.4%
Group C06	98.9%	98.9%
Group C07	99.6%	99.7%
Group C08	99.3%	99.2%
Group C09	98.7%	98.7%
Group C10	97.6%	98.5%
Lowest Group	97.6%	98.4%
Second Lowest Group	98.1%	98.5%
Third Lowest Group	98.6%	98.7%
A-series Lowest Group	99.1%	98.9%
A-series Second Lowest Group	99.4%	99.3%
A-series Third Lowest Group	99.5%	99.5%
B-series Lowest Group	97.8%	98.1%
B-series Second Lowest Group	97.9%	98.8%
B-series Third Lowest Group	98.5%	99.1%

Figure 4.64: Goodness of fit of the lognormal distribution for corrected comment text length in the C-series

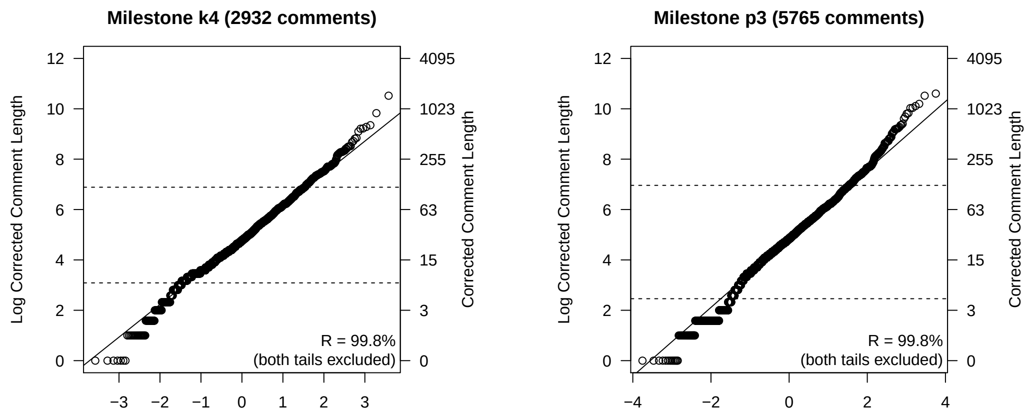


Figure 4.65: Normal QQ plots for Log corrected comment length in the C-series

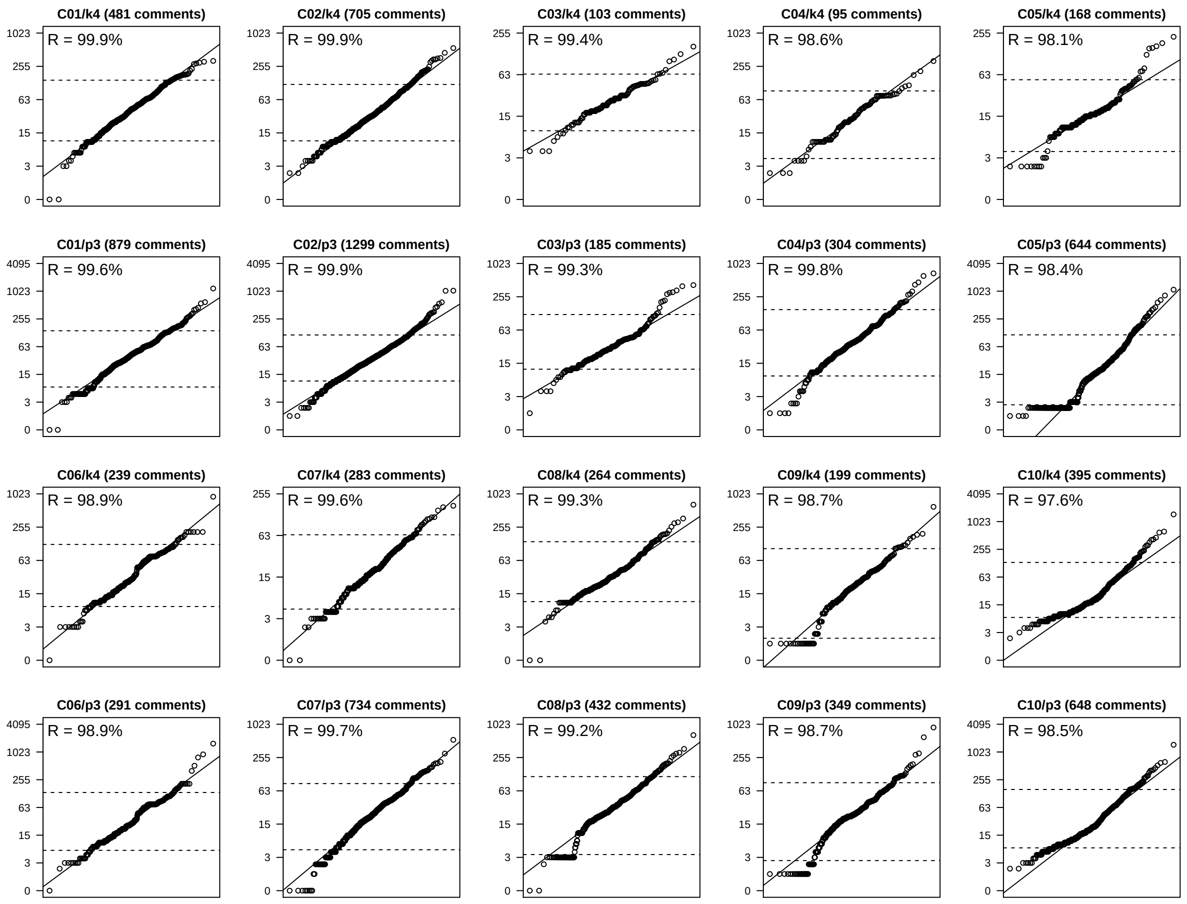
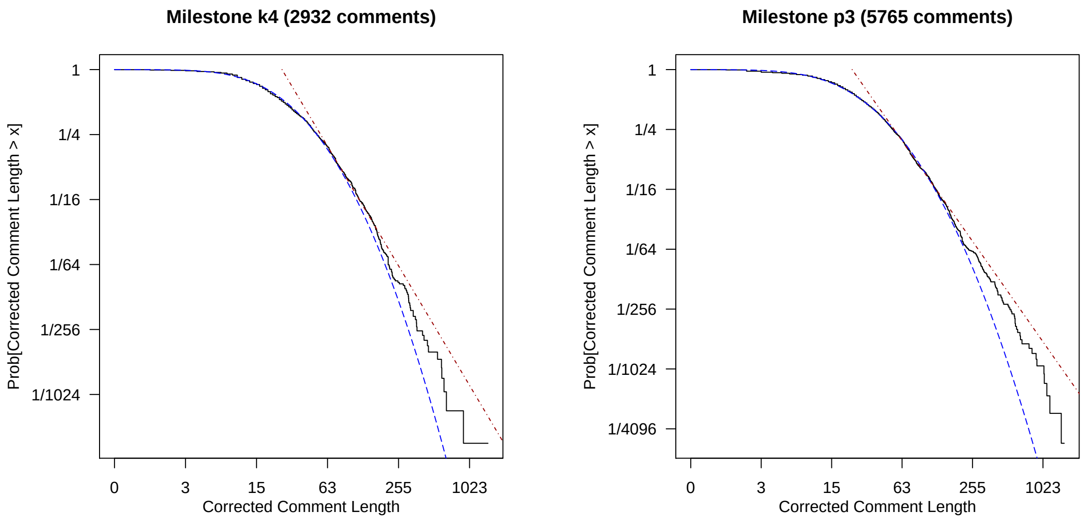


Figure 4.66: Normal QQ plots for Log corrected comment length in the C-series, by code base

Long Tail Behaviour of Comment Length

The C-series data conforms with the long tail hypotheses for corrected comment length, as shown in Figure 4.68. The tail index is larger than for the A- or B-series, but the significance of this result cannot be determined.



In each plot, the blue dashed curve is the CCDF for the lognormal distribution fit to the data, and the red dot-and-dash line is the CCDF for the power law fit to the upper sixteenth of the data.

Figure 4.67: Empirical CCDFs for corrected comment length in the C-series

p value for lognormal distribution	0.003 *
p value for power law	0.899
estimate of tail index α	1.76

Figure 4.68: Results for Downey's test applied to corrected comment length in the C-series

4.5.2 Summary of C-series Results

The results for the C-series are very similar to those for the B-series. All of the hypotheses are consistent with the C-series data.

The R values for hypotheses H1b, H2b, and H3b are comparable to those for the B-series. This is fairer than comparing to the A-series, since both the B- and C-series were collected in the same way, while the A-series was collected after the fact and had an imperfect response rate.

There are more examples of deviation from normality because of many files with the same commented line count. With more cases like this available, a method might be developed to compensate for such files, especially those that arise as multiple derived copies of a single original file.

Chapter 5

Discussion

This work exhibits commonalities in natural commenting behaviour, which demonstrate that science can provide insight into programming. Of course, the human element is ubiquitous in commenting, and there are interesting outliers that fall outside any description.

Chapter 3 organizes the comments of the pilot data set into a taxonomy of commenting. In analyzing the role of each comment, I assumed that each comment is written for a reason. The main result is that most comments in the data have a single purpose, so there is merit in classifying them.

The categories in Chapter 3 can be used deliberately when writing comments. Indeed, if these categories are popular because they are useful, then this is what novice programmers should be taught about commenting, because these are the comments that experts write! This could also benefit the design of commenting standards: a standard based on natural behaviour is more likely to be followed.

Section 4.2 studies *comment density*: the proportion of commented lines by source code file. The data rules out uniformity of comment density within a file. However, it looks as though the overall comment density of a file changes little over its lifetime, and that more difficult code is more densely commented.

Section 4.1 studies file size in terms of number of lines of code, and amount of commenting per file in terms of number of commented lines. Section 4.3

studies the length of comment text measured by character count. Each of these is well approximated by a lognormal distribution, although the distribution of commented line count is truncated to the left at zero, and the distribution of comment length has a long tail.

Even more important than the particular distribution is that every code base has the same distribution, up to location and scale parameters. So, even though programmers show a lot of individual variation, this variation has constraints. The distribution is also common to both the operating system and train project milestones, which are very different programming tasks.

These results from Sections 4.1 and 4.3 are validated in Sections 4.4 and 4.5. This was done using the test data set which was kept hidden during the investigation of the pilot data set presented in the preceding sections.

5.1 Why do we Comment?

What programmers cannot express with code, they express with comments. This is indeed the purpose of commenting. Therefore, programming language designers should read comments to see which aspects of current programming languages programmers find inadequate.

For example, programming languages provide the ability to define named constants. However, Section 3.2.2 shows comments in the data whose purpose is to associate a name with a constant in the code. The presence of these comments in the data set implies that programmers find the current facilities for defining named constants lacking! These comments occur in the data most often for constants that are used only once; perhaps there is a mental barrier against creating a definition unless it is referenced multiple times.

I suspect that every major programming language feature could have been predicted by looking at comments in earlier programming languages, although the data to confirm this would be difficult to collect. The very concept of a high level language must have been predated by comments in assembly language code, written in notations that eventually became the syntax encoded in today's programming languages. Indeed, even assemblers must have been invented after the use of hand written instruction mnemonics on machine code listings.

This work studied only the literal comments, which may be too restrictive for some research goals. The role of commenting overlaps with whitespace, identifier names, assertions, pseudocode, and even string constants. For example, identifier names, which are more constrained than comments and thus easier to study, may provide valuable insight into how programmers use language to express themselves in code.

5.2 Considerations for Experimental Design

The primary problem when collecting code from the wild is that the raw data lacks the structure needed for automated extraction of the objects under study. In Chapter 2, every step in recovering this structure needed manual intervention, because the required information was not explicit in the data. It was even difficult to identify which files contained source code, although this could be solved in the experimental design by specifying a more rigid format for the students' code submissions. Detecting commented-out code is particularly problematic, since programmers often mimic programming language syntax in legitimate comments.

Requiring so much subjective interpretation of the data inevitably biases the analysis. However, plenty of information is explicit in the raw data. For example, the semantics of the code in the data set is well-defined in the sense that it always compiles. This is true because the students had to produce code that the compiler would accept in order to achieve their own goals.

In other words, the programmers' goals often align themselves with the experimenters' goals. For an experiment studying a particular aspect of programming, it may be worth setting up the programming environment so that the programmers themselves ensure that the desired information is present. For example, if the students' programming languages had only bracketed comments, then there would be no need to manually merge adjacent line comments.

In Sections 4.1 and 4.2, the analysis suffered when a code base had too many files without comments, as this caused distributions to bunch up at zero. Occurring to varying degrees in almost every code base, this adversely affected estimates of location and scale parameters. Group A03, in contrast, commented every file: for this group, even the lowest values are well-fit by a lognormal distribution (Figure 4.15, page 81). Thus, there is reason to believe that the true distribution is lognormal over its entire range, but this is apparent in the data only when every file is commented.

There are further statistical problems when measuring "goodness of fit" as R values obtained from linear regression on the QQ plot. These R values are overly sensitive to sample size: larger samples have higher R values just by being larger.

For most of the analysis in Chapters 3 and 4, the data was separated by assignment milestone. I did this because I did not want code or comments in milestone k4 which remained unchanged in milestone p3 to receive double weight. However, this problem is more general; there are other ways that code can be duplicated. Indeed, the provided comments and the compiler-generated comments excluded in Section 2.2.2 deserve special treatment simply because they are duplicates of a small number of original comments. Furthermore, some observed anomalies in the commented line count distributions are the result of many copies of a comment (Figure 4.15, page 81: group A05, and Figure 4.63, page 126: groups C03, C06, and C07).

Many of these problems are inherent to studying code in the wild, and many of my suggestions undermine the idea of collecting code written under natural conditions. However, there are good reasons to know which aspects of wild data are the most problematic. In an experiment, which is already artificial, this knowledge helps the experimenter decide which variables should be explicitly controlled. Furthermore, naturally written code may lack one or more of these problematic features. Such a data set may be especially valuable in giving an unusually clear picture of programming behaviour.

5.3 Future Work

Ideally, this work would be validated for different programming tasks, computing environments, programming languages, and cultural environments, to determine which results are universal and which are particular to the conditions of this study.

The taxonomy developed in Chapter 3 must be improved to handle comments that refer to definitions, because such comments fail to divide cleanly into distinct categories. The data shows that there is a variety of information relevant to a definition, but only one natural place in which to put it. Perhaps disjoint categories could be generalized into mutually interacting features. Alternatively, the text of these comments could be analyzed more finely: at the sentence or word level.

The results of Chapter 3 were not validated on the test data set because I failed to make the categories objective enough to eliminate human bias. It might suffice to create *operational definitions* for the comment categories. For example, clarification comments describe the surface structure of code, suggesting a test for identifying clarification comments: "Would this comment make sense if the code to which it refers were replaced with an equivalent implementation?". Subjectivity would remain, but it could be measured by comparing the judgements of programmers using the operational definitions. Along these lines, an experiment must be performed to test the robustness of *referent*; if the referent of a comment has an objective definition, then different programmers should agree.

The empirical laws validated in Chapter 4 show that aggregate properties of source code originate in certain statistical distributions, which suggest tools for comparing code written under different conditions. Owing to limitations of space and time, only a few factors were analyzed in this work, and so a more comprehensive study could be done. The relationship between aspects of commenting and features in the code could also be investigated; this was excluded from this study since the content of the code was largely ignored.

For example, specific features of a programming language may receive a disproportionate number of clarification comments.

Section 4.2.2 studied changes in comment density between the fourth and seventh assignment milestones. It is possible to study how the code and comments changed over the seven milestones with the existing data. However, it is difficult to accurately track changes, especially when code is moved or duplicated. It may be beneficial to collect data with more fine-grained revision history, perhaps even monitoring every single change to the source files as they are edited. This extra information would make it easier to recover accurate change information in an automated way.

5.4 Closing Remarks

Simply reading so much code has greatly affected my commenting practice. I read every single comment in the pilot data set multiple times, and spent much time struggling to understand the intent behind each comment and the context surrounding it. I now have a greater appreciation of just how little information is possessed by a reader who is unfamiliar with the code being read.

I undertook this research project because I believe that programming is a fascinating human activity. While reading the comments, I observed much more than was discussed in this thesis. After all, Chapter 3 only discussed observations directly relevant to the taxonomy. There is insufficient space here for a full description of everything I saw. Indeed, such a description would likely exceed the size of the data set itself! In other words, to gain a better understanding of programming, there is no substitute for reading code.

Bibliography

[1] Ronald M. Baecker and Aaron Marcus. *Human factors and typography for more readable programs.* ACM, New York, NY, USA, 1989.

[2] V. R. Basili and D. H. Hutchens. An empirical study of a syntactic complexity family. *IEEE Transactions on Software Engineering*, 9:664–672, 1983.

[3] Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors. *History of programming languages—II.* ACM, New York, NY, USA, 1996.

[4] Lucy M. Berlin. Beyond Program Understanding: A Look at Programming Expertise in Industry. Technical report, November 1992.

[5] Frank W. Calliss. Problems with automatic restructurers. *SIGPLAN Notices*, 23(3):13–21, March 1988.

[6] D. R. Cheriton. *The Thoth system: multi-process structuring and portability.* Operating and programming systems series. North Holland, 1982.

[7] William S. Cleveland. *The elements of graphing data.* Wadsworth Publishing Company, Belmont, CA, USA, 1985.

[8] Mark Crovella and Murad S. Taqqu. Estimating the heavy tail index from scaling properties. *Methodology and Computing in Applied Probability*, pages 55–79, 1999.

[9] Bill Curtis. Substantiating programmer variability. *Proceedings of the IEEE*, 69(7):846, 1981.

[10] Bill Curtis. Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *Proceedings of the 7th international conference on Software engineering*, ICSE '84, pages 97–106, Piscataway, NJ, USA, 1984. IEEE Press.

[11] Allen B. Downey. Lognormal and pareto distributions in the internet. *Computer Communications*, 28:790–801, 2005.

[12] Robin Dunbar. *Grooming, Gossip, and the Evolution of Language.* Harvard University Press, October 1998.

[13] Beat Fluri, Michael Wursch, Martin Pinzger, and Harald C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[14] W. Morven Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software: Practice and Experience*, 11(5):435–466, 1981.

[15] Peter Grogono. Comments, assertions and pragmas. *SIGPLAN Notices*, 24(3):79–84, March 1989.

[16] Maurice H. Halstead. *Elements of Software Science.* Operating and programming systems series. Elsevier Science Inc., New York, NY, USA, 1977.

[17] M. J. Kaelbling. Programming languages should not have comment statements. *SIGPLAN Notices*, 23(10):59–60, October 1988.

[18] E. Limpert, W. A. Stahel, and M. Abbt. Log-normal Distributions across the Sciences: Keys and Clues. *BioScience*, 51(5):341–352, May 2001.

[19] Randy K. Lind and K. Vairavan. An experimental investigation of software metrics and their relationship to software development effort. *IEEE Transactions on Software Engineering*, 15(5):649–653, May 1989.

[20] Stuart Mawler. *Executable Texts: Programs as Communications Devices and Their Use in Shaping High-Tech Culture.* Master's thesis, Virginia Polytechnic Institute and State University, 2007.

[21] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[22] L. A. Miller. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 20(2):184–215, 1981.

[23] Paul W. Oman and Curtis R. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, May 1990.

[24] Esmond Pitt. On the typesetting of computer programs. `http://www.rmiproxy.com/javarmi/TypesettingComputerPrograms.pdf`, 2003. Accessed 2013 June 24.

[25] R. D. Riecken, J. Koenemann Belliveau, and S. P. Robertson. What do expert programmers communicate by means of descriptive commenting? In *Empirical studies of programmers: Fourth workshop*, pages 177–195.

[26] R. S. Scowen and B. A. Wichmann. The definition of comments in programming languages. *Software: Practice and Experience*, 4(2):181–188, 1974.

[27] John Sutton. Gibrat's legacy. *Journal of Economic Literature*, 35(1):40–59, 1997.

[28] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

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

[30] Paul F. Velleman. *The Datadesk Handbook*. Ithaca, New York, Odesta, 1988.

[31] Rand R. Wilcox. *Fundamentals of modern statistical methods: substantially improving power and accuracy*. Springer, New York, 2001.

[32] University of Waterloo. CS 452 real-time programming. `https://cs.uwaterloo.ca/current/courses/course_descriptions/cDescr/CS452`, 2013. Accessed 2013 August 16.