MODULAR PROGRAM SIZE COUNTING

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAIʻI IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

INFORMATION AND COMPUTER SCIENCES

DECEMBER 1999

By
Joseph A. Dane

Thesis Committee:

Philip M. Johnson, Chairperson
Wesley Peterson
Edoardo S. Biagioni

We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Information and Computer Sciences.

THESIS COMMITTEE

Chairperson

Table of Contents

List of Tables

List of Figures

Abstract

Effective program size measurement is difficult to accomplish. Factors such as program implementation language, programmer experience and application domain influence the effectiveness of particular size metrics to such a degree that it is unlikely that any single size metric will be appropriate for all applications. This thesis introduces a tool, LOCC, which provides a generic architecture and interface to the production and use of different size metrics. Developers can use the size metrics distributed with LOCC or can design their own metrics, which can be easily incorporated into LOCC. LOCC pays particular attention to the problem of supporting incremental development, where a work product is not created all at once but rather through a sequence of small changes applied to previously developed programs. LOCC requires that developers of new size metrics support this approach by providing a means of comparing two versions of a program. LOCC's effectiveness was evaluated by using it to count over 50,000 lines of Java code, by soliciting responses to a questionnaire sent to users, and by personal reflection on the process of using and extending it. The evaluation revealed that users of LOCC found that it assisted them in their development process, although there were some improvements which could be made.

Chapter 1

Introduction

At first glance, it seems trivial to determine the size of a computer program. There are any number of reasonably simple means for calculating program size. Since computer programs are generally stored on computer systems, it requires little effort to ask the operating system for a count of the number of bytes in the program. If the program file is actually object code, however, we may be including in the count the chaff of system library code, linker directives, and symbol tables along with the wheat of the program instructions themselves.

We may then decide to count the program source instead of the compiled object code. We could count the number of characters, or perhaps the number of a certain type of character, such as the newline character. But what if we are dealing with the work of a perverted programmer, who has placed all his code on a single line, or spread what should reasonably be considered a single line over many lines? Even non-perverse programmers often have different opinions on how to best format code.

We see that we can avoid problems here if we count something intrinsic to the program itself, instead of something which is an artifact of the representation of the program source text. So we may try to quantify some aspect of the program's complexity, structure, or function. We could, for instance, count the number of language expressions denoted by the program text. We have now clearly left the domain of "trivial" in which we expected to find an answer to the question: "How large is this program?"

Even assuming we have found a satisfactory solution to the problem of measuring the size of a program, our problems are not over. The source of the new problem is that software development in fact rarely begins with a clean slate. The developer's effort is usually spent modifying or extending an existing application. Say a programmer takes a file with 1000 lines of code (or 1000 expressions — the metric used is not relevant), deletes 400 lines and adds 500. A total count would give a size of 1100 lines, only 100 more than the initial count. The total count has not given us the information we are interested in, namely the amount of new work added to the file. This is because the total count is incompatible with the incremental software development regime in which most development occurs.

Accurate size data is vital to project planning, estimation, and monitoring. A measure of program size is needed to determine programmer productivity, which can then be used in estimating costs for future projects. Despite the clear need for accurate size measures, there is often a disconnection between those developing the metrics and those using them. A recent issue of IEEE Software contained a "Status Report on Software Measurement"[25] which noted:

> *Practitioners continue to use what is readily available and easy to use, regardless of its appropriateness. This is in part the fault of researchers, who have not described the limitations of and constraints on techniques put forth for practical use. . . . We need to encourage researchers to fashion results into tools and techniques that practitioners can easily understand and apply.*

This thesis responds directly to this desire with a new approach to program size measurement. Instead of attempting to determine, once and for all, what the best size measure is, it approaches the problem from a more practical perspective — that of the practicing software engineer. What a programmer needs is a tool which can be used to give one or more ideas as to the size of his programs. Since the best measure for this size might vary with changes in programming language, application domain, experience, and formal or structural requirements, a tool which can support a number of size measures will be an important addition to the programmer's tool chest. The tool should present the same interface to the user regardless of the size measure being used, so that a new set of rules need not be memorized for each circumstance. It should address the issue of incremental development. The tool must additionally be flexible enough to incorporate new size measures as the need arises.

1.1 Motivation — Why measure size?

LOCC was designed as a tool to simplify size data collection in a software development process. By "process", I mean not simply the act of writing software, but a framework for observing and analyzing that act, with the goal of making it more efficient and precise. A person who comes to work, turns on his or her monitor, and begins coding is using a process of a sort, but when I use the term "process" I am referring to something more defined. In particular, a software development process might proceed in a number of well-defined stages, such as planning, design, coding, compiling, testing, etc.. It might also define a number of activities to be performed at certain stages, such as measurement of the amount of time taken in each stage, size of the product developed at each stage, number of defects injected or removed at each stage, and so on. A number of distinct processes have been outlined in the literature[3, 10], and I refer not to any one in particular, but to the general idea that one should try to keep track of one's activities to better understand and improve them.

There are a number of reasons why a developer would use a well-defined software development process. Developers are interested in improving the quality of their designs, in reducing the number of defects introduced into their products, and in making accurate predictions about the size and cost of the products they develop. This last goal, accurate size and time estimation, is of particular interest to this thesis. Different processes will approach the estimation problem in different ways, but they all have in common a need for accurate size measurement. Program size measurement is used to establish the framework within which estimation is carried out. Whichever process the developer is using, accurate size data will be required as an input to the estimation system.

Programmers, typically, are interested in programming, and not in such bookkeeping as is inevitably required by a process. The more administrative overhead is involved in a particular process [1], the more likely that programmers will not follow its guidelines, or will follow them incorrectly[4]. It is vital that the focus of the programmer be the task at hand, and not on the guidelines and measurements required by the process. As such, automated tools which assist the programmer in managing the details of the process are essential[8, 10].

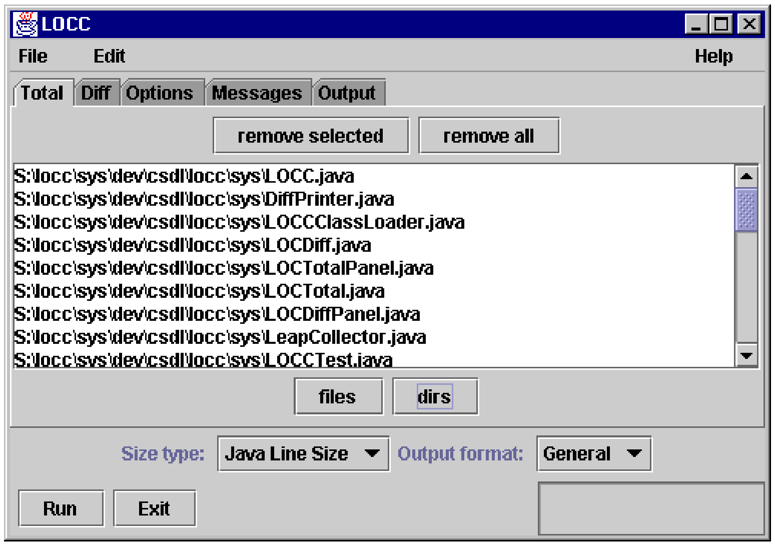
The process detail LOCC assists in automating is, of course, program size measurement. LOCC presents a programmer with a uniform interface for measuring program size, and allows the developer to quickly and efficiently obtain a measurement of program size. LOCC makes it simple to apply a size metric to even large software systems. It also supports the concept of iterative development, which will be discussed further in Section 2.4 on difference measuring.

The actual metric being used to obtain the size measure is independent of the interface. LOCC is shipped with a set of predefined size measures for the Java Programming Language, C++ and plain ASCII text files. If the particular size metrics which exist by default in LOCC are not sufficient for the programmer's task at hand, there is a simple mechanism for adding new size metrics to the system. LOCC ships with both line-oriented metric and metrics which focus on syntactic units such as statements or expressions. LOCC places no restrictions on size metrics, leaving developers free to utilize whichever size metric a particular situation demands.

1.2 Objectives

This goal of this thesis is to develop, deploy, and evaluate a high quality tool to assist software engineers in the collection of useful program size data. It should additionally encourage experimentation with new or modified size metrics. The claim of this thesis is therefore twofold:

[1]Hall and Fenton[9] note that 7% overhead has been suggested as maximal



```
S:\locc\sys\dev\csdl\locc\sys\LOCC.java
S:\locc\sys\dev\csdl\locc\sys\DiffPrinter.java
S:\locc\sys\dev\csdl\locc\sys\LOCCClassLoader.java
S:\locc\sys\dev\csdl\locc\sys\LOCDiff.java
S:\locc\sys\dev\csdl\locc\sys\LOCTotalPanel.java
S:\locc\sys\dev\csdl\locc\sys\LOCTotal.java
S:\locc\sys\dev\csdl\locc\sys\LOCDiffPanel.java
S:\locc\sys\dev\csdl\locc\sys\LeapCollector.java
S:\locc\sys\dev\csdl\locc\sys\LOCCTest.java
```

Size type: Java Line Size
Output format: General

Figure 1.1: LOCC's graphical interface

- *LOCC is a generic, effective tool for the collection of size data to support software engineering tasks such as project planning, estimation, and process improvement.*

- *LOCC provides an environment in which size measures can be quickly developed and evaluated, leading to more accurate and efficient size measurement.*

1.3 Use and evaluation

LOCC is currently being used both in the Collaborative Software Development Laboratory at the University of Hawai‘i, and in a graduate software engineering class. The initial version of LOCC was deployed in July, 1999 in the CSDL. It has since gone through two major revisions and numerous minor revisions. LOCC has been used to count over 50,000 lines of Java source code. Figure 1.1 shows a screen shot of an LOCC session.

LOCC has been evaluated in three ways: through its use in measuring the size of Java programs; through an online survey of current LOCC users; and through a retrospective look at the process of extending LOCC to include additional size metrics. The results of the evaluation indicate that LOCC provides a useful measurement tool, and that current users of LOCC feel that it has assisted them in the collection of their software process data.

The name “LOCC” was originally an acronym expanding to “Lines of Code Counter”. This has become a misnomer, since LOCC is in no way limited to counting lines of code. But, inertia being what it is, the name has stuck.

1.4 Contributions

LOCC makes a number of important contributions to the software engineering community:

- A Java program, portable to any platform on which Java is supported, which assists in the collection of program size data
- A graphical interface for ease of use and a flexible command line interface for use within scripts
- A method of collecting numerous size metrics together for cross checking or counting different types of work products
- A specific proposal for counting Java source code which considers program structure and iterative development

The fact that LOCC is implemented as a Java program means that it is instantly portable to the numerous platforms which support Java. The idea that one tool should support more than one size metric is, as far as I know, novel. LOCC provides an environment conducive to experimentation with size metrics. Changes can be made to existing metrics, or new metrics installed, with a minimal expenditure of effort. Of course, the problem of “which size metric should I use” is still difficult. LOCC can assist by allowing a developer to use multiple metrics, deferring the decision on which is most appropriate until enough data has been collected to make a rational judgment.

1.5 Outline of the thesis

The thesis will cover the following topics. Chapter 2 will discuss size measurement in general. Some reasons why program size measurement is interesting will be discussed, and several size measures will be introduced and described. Chapter 3 will describe the operation of LOCC. Chapter 4 will describe the architecture and implementation of LOCC. Chapter 5 will examine the steps needed to extend LOCC with new size metrics, and will go through a simple example. Chapter 6 will give an evaluation of LOCC based on its use in a graduate software engineering class. Finally, Chapter 7 will briefly describe some possible future directions for LOCC.

Chapter 2

Size Measurement

We begin this chapter with a discussion about why size measurement is an important part of the software engineering process. A brief survey of size measurement techniques will be followed by a more detailed description of the use of lines of code (LOC). Particular attention will be paid to LOC because, for better or worse, LOC continues to be the most frequently used size metric. It should be emphasized, however, that LOCC in no way dictates the use of a particular size metric. We then define structure-sensitive counting and why it is desirable to have a size metric which considers program structure when producing a measurement. The problems involved in using a size measure which counts total program size in an environment in which most development proceeds iteratively is then discussed. Finally, we close with some comments about the desirability of automated tool support in a size measurement system.

2.1 Why is size measurement interesting?

As a start we should be clear about why program size is an interesting feature and why we should want to measure it. There are a number of reasons, some of which are:

- Cost estimation
- Validation of estimates

- Productivity measurement
- Documentation

2.1.1 Cost Estimation

If the object we are measuring is something produced in the early stages of the development process, we may want to measure its size so we can estimate the cost of future stages. Cost is often expressed in units of time, as in “we expect the product will require N man-months for completion”. The object being measured in this situation may be a formalized requirements description or some abstract representation of the design of the final product.

For example, say we have defined a formalized requirements language. Using this language, we describe the product we wish to develop. The language may contain extensive detail, or may be a high-level and more abstract description. A size metric is applied to the “program” in this language, and the resulting measurement can be used to predict the cost of complete development. To do this we would need to have some idea of how a certain size measurement in the requirement language translates into cost. This could come from historical data relating these quantities, or from historical data relating requirement size to final program size, and other data relating final program size to development cost. Both of these translations would have to consider factors such as development language, problem domain, problem complexity, staff experience, etc.

2.1.2 Validation of Estimates

Software cost estimation is notoriously inaccurate: programmers seem often to be overly optimistic in estimates of their productivity[26, 3]. A measurement of the actual size of the finished product can be compared to the estimated size, and errors can be used to correct the estimation process. Another way of looking at this is to consider a “database” of estimated size vs. actual size measurements. Linear regression can then be used to predict cost from estimated size for future products.

Linear regression can also be used to gauge the confidence we can have in its estimates. As more data is added to our historical database we may find that we can predict with increasing confidence the effort required to produce products in the future. If, however, we find that we cannot predict future effort with sufficient confidence — that our size vs. effort measurements are not strongly correlated — then we need to more closely examine both the process and the tools we use to analyze it. We may be able to improve estimation by identifying factors other than program size which may affect cost: target programming language, experience level of the programmers, or special factors such as programs requiring multithreading, specialized numerical calculations or other complications. Or we might consider using a different size measure, something LOCC is perfectly suited for. After taking these additional factors into consideration we may improve our predictive abilities.

2.1.3 Productivity measurement

If a developer has at his disposal a means for evaluating the size of a software product, and additionally has a measure of the time spent developing the product, then a productivity measure is a simple calculation away. Productivity is clearly an interesting quantity, and one which both programmers and their managers would be interested in measuring. However, see Section 2.2.3 for some potential complications with using size measurement to deduce productivity.

2.1.4 Documentation

We may be interested in recording the size of a product as part of the process of describing and archiving it. This may be because of institutional guidelines requiring such documentation, or may be done simply in the course of the programmer's development process. We may also wish to document progress toward some goal which has been identified, e.g. in a GQM [10] model. For example, a goal may have been set to keep program module size below certain levels.

2.2 Some types of size metrics

A large number of size metrics have been devised in both academic and industrials settings, each with its own strengths, weaknesses and peculiarities. Some have passed into obsolescence while others continue to be used regularly. We discuss some of these below.

2.2.1 Complexity Metrics

Beginning with Halstead's Software Science, a number of metrics have been developed which try to measure the complexity of a program, and to associate that complexity with a number.

Halstead proposed measuring program size in the following way. A language has some number of distinct primitive operators. Call the number of these operators η_1. Additionally, in any piece of code there will be some number of distinct operands. Call this number η_2. Additionally, let N_1 and N_2 be the total number of operator and operand occurrences. Halstead defined three metrics based on these primitive quantities.

The *vocabulary* of the program is given by

$$\eta = \eta_1 + \eta_2$$

The *length* of the program is given by

$$N = N_1 + N_2$$

And the *volume* of the program is given by

$$V = N \log_2 \eta$$

Halstead meant for these metrics to be applied to small units, such as individual functions or procedures, and the total program size was to be found by summing the individual sizes.

Halstead's approach is significant today mainly because it was one of the earliest attempts at applying rigorous metrics to software. Software Science was intended to be just that: a scientific (and therefore precise) quantification of software and the software development process. However, the specific proposals Halstead advanced have been generally criticized and found wanting. One problem [16] is Halstead's neat division of program text into the mutually exclusive categories of "operator" and "operand". This division seems reasonable when considering some languages, e.g. FORTRAN and Pascal, but in other languages such as LISP and other functional languages the status of an expression may change from being a value (operand) to being a function (operator).

A more refined measure of complexity was provided by McCabe. The basic idea is to transform the written description of the program into a control graph, and to then assign a size to the program as follows:

$$N = e - n + 2$$

where the graph has e arcs and n nodes. This number is referred to as the "cyclomatic" complexity of the program. The complexity of a program is then related to its size.

Norman Fenton has written about the utility of cyclomatic complexity as a measure of program size[5]. Fenton claims that, while cyclomatic complexity may be useful in some situations (for example, in predicting how difficult a program will be to maintain) it does not correspond to an intuitive understanding of complexity, and therefore is not useful in prediction and estimation. Fenton gives an example of two programs, one simple and the other seemingly complex, which have the same cyclomatic complexity.

2.2.2 Function Points

Function point analysis has probably been the most widely adopted size measure, aside from lines of code. Function points were first defined by Allan Albrecht in 1979.

Function point analysis works by examining the function of a software product. As Albrecht says, the approach is “to list and count the number of external user inputs, inquiries, outputs, and master files to be delivered by the development project”, and to weight these counts by the relative “value” of the component. Function point analysis has a number of attractive features, including:

- Function point analysis can arrive at a size measure relatively early in the development process. No code is required. If a customer can supply a detailed description of the product to be developed, function point analysis can arrive at a size for the product. Since the inputs to function point analysis are the same requirements specifications needed to develop the product itself, a size estimate can be produced almost at the same time as the requirements.
- The customer is also more likely to be able to relate to a measure given in terms of functional units than in lines of code, which may be meaningless to a person not versed in software development.
- Function points are largely language independent. It makes little difference whether the final product is implemented in COBOL, C++ or Basic, since the analysis is concerned only with the abstract functionality of the product. Function points are also independent of the coding style used in writing the code.
- Function points are consistent. There is now a International Function Point Users Group, which defines standards and concepts to promote consistency between different applications of function points.

Function point analysis has proved its usefulness simply by the fact that it continues to be used today, while other metrics have vanished into academic oblivion. Despite its obvious utility, however, there are serious problems with the function point approach:

- Function points, as originally conceived and as usually practiced, have a limited domain of application. One can tell by the phrases used by Albrecht — “master files”, “user input screens" — that function points began in an environment of data processing. How should one go about applying function points to, say, a large physics simulation problem, or a game?

- There are problems in classifying function point units. [14]. Functional items are identified and classified along an ordinal scale from "simple" to "complex". Kitchenham identifies problems with adding items across classifications.
- Kitchenham also identifies problems with the predictive ability of function points. Function points in themselves are mostly useless. They are interesting in that they allow estimation of other, more important quantities, like development effort. As it turns out, it may be difficult to have confidence in predictions made based on function point analysis. Graham Low and Ross Jeffrey discovered a 30% variation *within a single organization* in estimation based on function point counting.
- Function points are not immediately amenable to automatic counting. See [2] for a counter-example, though.

Function points will continue to be used in industry, and will be continued to be studied and refined in academia. But despite their obvious utility, they have not been able to dislodge the most popular, and equally troubling measure, lines of code.

2.2.3 Lines of Code

The simplest measure of them all, lines of code (also referred to as LOC, SLOC, DSI (Delivered Source Instructions)) is like an unpleasant relative : its faults are obvious to everyone, but no one can seem to make it go away. LOC has problems equally as troubling as those associated with function points, but it remains the unit most often used when discussing program size. I will enumerate some of the problems with using LOC in measuring size, then discuss some reasons why, despite the problems, its use continues.

Measurement Dysfunction

Werner Heisenberg, the great German physicist, made many contributions to physics. The one he is most often remembered for is, of course, what has become known as the Heisenberg Uncertainty Principle, which states that there is a limit to the knowledge one can have simultaneously about the momentum and position of a particle. The principle is often characterized (somewhat misleadingly, I think) thus: “The act of taking a measurement inevitably disturbs that which was being measured.”

The application of the Uncertainty Principle to software engineering can be stated, “The act of measuring program size changes the size of the program being measured.” No laws of causality need be violated for this to be true. All that need happen is that the programmer, the one responsible for producing the code to be measured, feel that his performance will be evaluated based on the results of the measurement. If these conditions obtain, then the programmer may be pressed into manipulating the very code he is producing, with the measurement more in mind than the function of the code.

The most obvious and simple example of such a situation is when measuring productivity. Rewards may be given, or penalties assessed, depending on the programmer meeting some productivity goal. When the measure of programmer activity is something tied to the behavior of the program, like function points, then such productivity measures may be meaningful. It is, however, quite another story with LOC, which can be inflated or deflated with ease by experienced programmers. Even when code must pass review, and even when strict code formatting standards are enforced, a programmer who is primarily concerned about meeting goals expressed in LOC will be able to manipulate his output to that end. The result may or may not be poor quality code. At any rate, it is generally agreed that the programmer’s focus should be on solving the problem at hand, and not artificially meeting productivity goals.

The conclusion one draws from this is simple and drastic: LOC is incompatible with productivity measurement when there is some possibility that the productivity measurement might affect the programmer. Productivity measurement must either take place using some measure other than lines of code, or must not be allowed to affect the programmer. The only way a programmer can feel certain that his productivity measurement will not be used against him is if he keeps it secret. It is this option we advocate with LOCC.

Size counting in a personal software process

Watts Humphrey, of the Software Engineering Institute at Carnegie Melon University, has described a new direction in software process development [10]. Instead of formulating a software process as an organization wide process, the Personal Software Process focuses on the individual programmer. Use of the PSP, or any other process centered on the individual, does not preclude participation in an organizational process. Nor does the PSP exist primarily as a response to the measurement dysfunction problems outlined above. One might even say that the PSP solves the measurement dysfunction problem as a side effect of its focus on the developer.

In any case, LOCC was designed with a personal software process in mind. LOCC as a tool is intended to be used by individual programmers to help them gain insight into their personal software process. Data produced by LOCC is generally provided as text files of some format, which can be analyzed by the programmer and delivered to management as the programmer sees fit. While there is nothing in the design of LOCC which limits its use to such circumstances, it is this personal, individual framework in which LOCC has thus far been applied.

Problems with LOC

Arguably the most vivid problem with the use of LOC as a program size metric is its tendency to cause code bloat when used to derive productivity measures. If one envisions a process in which process data can be kept private to the developers producing the code, then this problem is ameliorated. Other problems with LOC still exist, however.

The second major fault often found with the use of LOC is that the precise definition of a "line of code" is actually rather difficult to pin down. A strict interpretation in which each newline is counted does not seem to be best, especially when considering the free-form languages which are popular today. A decision to ignore blank lines follows immediately from this.

What about lines with more than one executable statement, or lines with a partial statement which is continued on the next line? Is a line which contains a series of complex logical tests equivalent to a line which simply increments a counter? Clearly, some lines are more "expensive" than others, and a measure which counts all lines equally will miss this complexity.

A decision must be made on how to count comments, and the usual decision [10, 3, 8] seems to be to ignore them. The decision seems somewhat arbitrary, and could even lead to poor documentation, which is another perpetual bane of the programmer's existence. LOCC's default size metrics, however, follow tradition and ignore all comments, although alternative decisions could easily be implemented with minimal changes.

Why LOC is still an interesting measure

The two fundamental reasons why LOC persists as a measure of program size are convenience and familiarity. A naïve line counting tool can be written in a matter of minutes. Alternatively, programmers can often use general line counting utilities provided by their programming environment, like the UNIX "`wc`" utility[23]. Unlike many other size metrics, LOC does not require any deep analysis of the design, requirements, or structure of the program. The flip side of this is, naturally, that LOC is not directly applicable to the measurement of these features. LOC is simply "lines of code", not units in an abstract design or specification language.

Also, and most importantly, programmers tend to think naturally in terms of LOC. Arguably the most important use of the results of a program size measurement is in developing an accurate size estimation regime, leading from there to more accurate cost estimation. Estimation always asks a question, "How large do I think this program element will be when coded?", and effective estimation depends on accurate answers to this question. Since programmers are used to thinking in terms of lines and not in terms of cyclomatic complexity or expression count, they should feel more comfortable answering this all-important question in terms of lines of code. While it could be argued that programmers simply need time to adjust to thinking in terms of other size metrics, as long as development is done using text-based languages in text-based environments, lines of code will probably be the most comfortable metric for developers.

2.2.4 Other program size metrics

In addition to Halstead's metrics, complexity metrics, function points, and lines of code, other metrics have been proposed over the years. Briefly, some of the size metrics proposed are:

- Storage size (in bytes) of the program text or compiled object code
- Number of executable program statements
- Number of input tokens [17]
- Abstract information flow [29]

2.3 Structure sensitive counting

Although LOC as a size metric does not strictly require analysis into the structure of a program, it has been shown to be beneficial to add this as a feature of a line counting regime[10, 15, 19]. Knowing the number of lines in a program unit is important in its own right, but one also may be interested in metrics such as LOC per function, module, class, or other syntactic unit. One important reason for gathering this sort of data is that it may help the programmer better understand his development process. It is important for the developer to become aware of his habits and routines in order to make changes which improve the development process. Productivity, in terms of LOC per unit time, is one such interesting statistic. Other interesting statistics (assuming the developer is using an object-oriented programming environment) are the number of lines per method, the number of lines per class, and the number of methods per class. Once the developer has analyzed a number of his work products, he can begin to see the trends in his personal development process. He may be able to compare these numbers to those of other developers in an organization, or against organizational or personal goals. There may be, for example, a desire to limit the number of lines per method, to keep the logic in each method as simple as possible. A LOC size metric can work in such a situation, but the metric must be enhanced to have knowledge about the structure of the development language.

Another important reason to be interested in structure sensitive sizing is that it may help cost estimation to be more effective and accurate. A typical cost estimation regime [10, 31, 15] works something like this, again assuming a development environment in which the terms “class”, and “method” make sense:

1. Produce a design document.
2. From the design, enumerate the classes needed.
3. For each class, enumerate the methods needed.
4. For each method, categorize the method along a scale from simple to complex.
5. Assign a cost to each method, and sum to find the total cost.

As a concrete example, consider a developer who has completed a design and is about to begin coding. The design has been formalized as a UML diagram which displays the classes, attributes, and relationships between entities. Each possible method which can be called must be considered and classified according to its expected complexity. The developer, who has previously collected data on the actual sizes of classes and methods he has produced, can use his historical database to assign an expected size to each method. By summing the expected sizes of each method, and adding the expected sizes of other class members, the developer arrives at an estimated size for the project. The developer can then use historical productivity measurements to estimate the time which must be alloted to the current project.

The routine described here is a simplification, of course, but generally describes many of the cost estimation systems in existence. In order to use such a system, the developer needs to know such numbers as his historical cost in developing a complex method.

Finally, historical data on such structure sensitive information as average number of methods per class may be used to validate new designs. If a new design is proposed which is out of line with historical data, it may be worthwhile to spend extra time on the design to see if it may be improved.

The important insight, and one of the most interesting and valuable contributions of LOCC, is that the inclusion of a parser in the size measurement tool can offer extraordinary flexibility when producing size measurements. Should comments be counted or not? A parser can recognize comments and ignore them, count them, or count them with a weight assigned by the type of the comment. For instance, documentation comments can often be syntactically recognized, either by their location in the parse tree (e.g. LISP) or by convention (e.g. Java, whose documentation comments begin with "/**").

This flexibility is useful because it provides an opportunity to empirically determine which size metric is best correlated with effort. Decisions on what to include in a size count are often motivated by convenience instead of by a real consideration of the effect of the decision. Frequently, lines of code are counted simply because they are easy to count. Once a parser is included in the size metric it is a relatively simple matter to make and change decisions on what to count and on what weight to give to those things that are counted. These decisions can be examined by empirically testing the predictions based on the measurements derived from the decisions. In this way, the developer can converge on a more accurate size measure.

A size measurement tool which incorporates a parser for the language being counted can count lines, statements, expressions, or weighted combinations of those or any other syntactically recognizable features. LOCC encourages the use of parsers in its size metrics, and includes parsers for the two programming languages (Java and C++) it supports, but does not require it. Again, LOCC values flexibility above all other virtues.

2.4 Size measurement for iterative development

There is an important aspect of program development which is often overlooked in discussions about size measurement. This is that a large percentage of software development begins with a base of old, previously developed code which is adapted to suit the needs of the current project. Walker Royce, in a talk at the 1997 International Conference on Software Engineering, put it this way:

> *In a modern development process the most important software metrics are simple, objective measures of how various perspectives of the product/project are changing. The absolute measures are usually much less important than the relative changes with respect to time.*

It should be emphasized that there is very little in the size measurement literature about iterative development. Most software size metrics focus on measurement of an entire program unit in isolation. While this research is certainly important, it ignores the reality that most software development is concerned not with producing code from scratch but with making modifications to existing code. There are a number of ways in which this modification takes place:

- Code may be reused without modification, as when a previously developed library is used in new development.
- Code may be modified during the course of bug fixing. No major modifications are enacted.
- Code may be improved, or features added, during the course of preparing a new release or version. In this case, major modifications may or may not occur.

The first item above is not really of concern to us. No effort is expended on using the old code, and we can simply take the old code to be part of the landscape, in much the same way as we do not count the C libraries when counting programs developed in that language. The second and third items, however, present a problem. Assuming that the program text itself is modified, how are we to distinguish new code from old? A simple line count will clearly not suffice in this circumstance. Possibly we can use something like the UNIX "`diff`"[22] utility to simply compare versions of the source files, and take as the number of new lines the number reported by `diff`.

However, we need to assess what we mean by the "difference" between two versions of a program. We are generally interested in program size because of its relation to cost, in terms of the number of man-hours required to produce a product of a given length. Therefore, we would like a difference measure which would correlate well with effort. Changes which are "easy" to make should be afforded less weight that changes which are "hard".

Making this determination is, however, not always straightforward. Programs like `diff` have no information to assist them in this determination. Since we have already identified sensitivity to program structure as a desirable characteristic of a size measure we can appeal to the same intelligence which endows a measure with this ability to solve the diff problem. Simply put: size metrics which contain parsers can make more accurate measurements of size difference than simple `diff`-like tools can.

While relatively little has been said in the literature on the diff problem, there has been quite active research in methods to quantify software reuse, especially since the advent of object oriented languages. This research may not be applicable to the problem at hand, however. Often, object component libraries can be classified as in the first category above, as code which is used without modification. In that case, there is no measurement problem to worry about.

Other researchers have proposed systems for categorizing reused code, depending upon the amount of intervention required. This research is often quite interesting. For example, NASA's Goddard Software Engineering Laboratory [5] arrived at a 4 level categorization of code, from

New code	1.0
Reused — lines of code in modules which will be reused	0.27
Added — lines added to reused modules	0.53
Changed — lines from reused modules which are changed	0.24
Deleted — lines deleted from reused modules	0.15
Removed — lines deleted from modules as a whole entity	0.11
Tested — lines reused without modification which still required testing	0.12

Table 2.1: Classification and weights of modified code

reused verbatim to completely new code. Other systems categorize each line in a source file depending on whether it was unmodified, modified, or new, assigning a weight to each line. Tausworthe proposed a system in which new code lines are given a weight of unity and modified lines some value less than one. His categorization is shown in Table 2.1.

The problem with many of these systems is that, despite their potential utility, they are not supported by any automated tools. Developers are left to themselves to determine how much code is new and how much is reused. The result can be incorrect counting, decreased productivity due to the time it takes to do the analysis, and increased developer dissatisfaction with the software development process. A programmer can't help but think, while sorting through lines of code, that this is exactly the sort of thing computers were meant to do.

2.5 Automated vs. manual size counting

Despite the general lack of agreement on which size measure should actually be used, everyone seems to agree that tools are needed to assist in the actual data collection. A software development process needs to be as unobtrusive as possible to encourage compliance and avoid burn-out. Requiring programmers to manually count lines, tokens, or even function points places an unacceptable burden on people whose desire, usually, is to be coding, not counting.

There are very few descriptions of such tools in the literature. Most academic literature on size counting focuses on the metric, or on their validation or interpretation. Rajiv Banker, et al[2] describe their experiences in developing an automated function point counter. They were using an application specific development system in which objects were stored in a central repository. A mapping from language objects to function points was defined, and this mapping was used to perform an automatic function point analysis on objects in the repository.

At least within the UNIX community, the utilities `wc` and `diff` are usually considered the standard tools for program size measurement. `wc` computes word and line counts from ASCII input files, and `diff` can be used to compute the difference between two ASCII files. The primary advantage displayed by these tools are their ease of use. A simple script can be written to count large numbers of files. However, neither tool can be used by itself to remove comments and blank lines from size counts.

Since a simple regular expression can match blank lines and at least most types of program comments, a slightly smarter version of `wc` is not terribly difficult to create. Tama Software[28] has produced a program which can count lines of code after removal of comments from several programming languages, including C, C++, Basic, and FORTRAN. Languages such as Perl[30] with extensive regular expression capabilities can also be used to create similar tools.

While regular expressions are sufficient for handling blank lines and (most) comments, they cannot be used to recognize more complex language structures such as classes and methods. For this, one needs some sort of parser. JavaCount[12] was an attempt at creating an ad-hoc parser for the Java language to recognize such structures while performing line counts of programs. JavaCount was successful in that it became fairly widely used, but suffered from incomplete coverage of the language and regularly erupting bug reports. JavaCount was also limited, naturally, to counting Java source files, and could not be extended in any way.

LOCC contains parsers generated by JavaCC[11] and has proven to be much more robust than JavaCount. It was designed for easy use and extension, and constrains size measures only by requiring them to be implemented as Java classes satisfying certain interfaces. Any size measurement which can be automatically collected can be supported by LOCC. Even metrics which do not appear to lend themselves to automated counting, like function points, can sometimes be adapted to an automated system, as was demonstrated by Banker in the study mentioned above.

Chapter 3

LOCC Operation

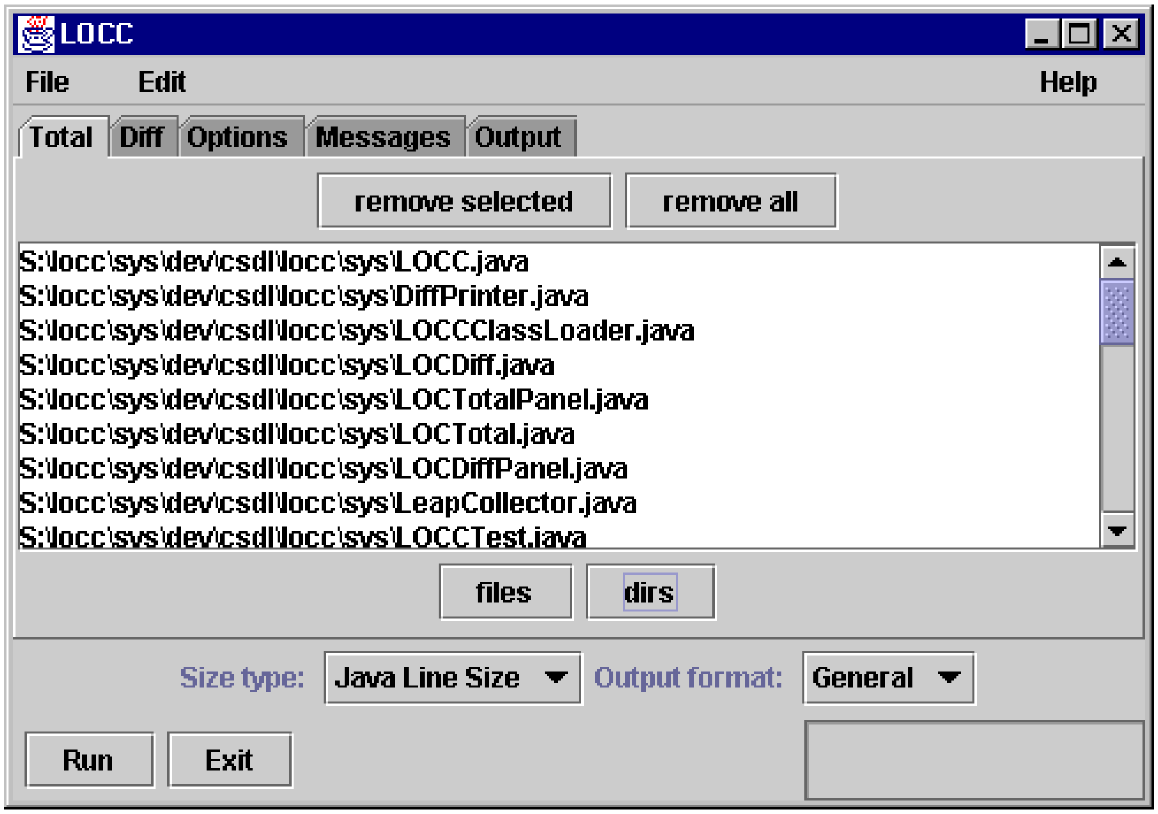
There are two ways of interacting with LOCC. A graphical user interface provides an easy way to specify large numbers of source files to count. A command line interface is useful if developers wish to automate their data collection process, e.g. by writing shell scripts to invoke size counting. The operation of LOCC is described in this Chapter, and is further described in the LOCC User's Guide[18].

3.1 Using the Graphical Interface

The graphical interface is intended to be the primary means of interacting with LOCC. The interface is composed of several "tabbed" panels in a larger window. The window contains the panels, as well as buttons to execute a size count or exit the system, menus which provide shortcuts to common actions, selectors to choose the size metric and output format, and a progress bar. The individual panels are described below.

3.1.1 Total Panel

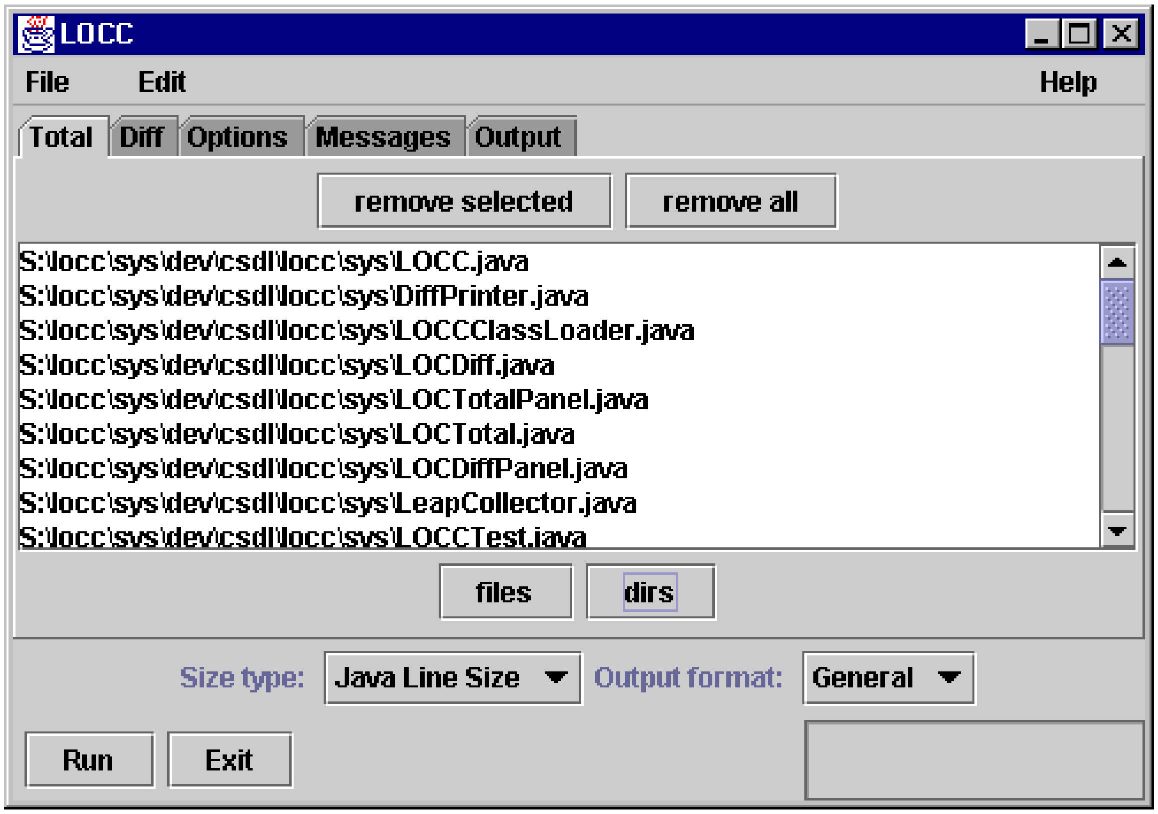
This panel, shown in Figure 3.1, provides methods by which the developer can add files to the list of files to be counted in total. The "files" button allows files to be added one at a time.



```
S:\locc\sys\dev\csdl\locc\sys\LOCC.java
S:\locc\sys\dev\csdl\locc\sys\DiffPrinter.java
S:\locc\sys\dev\csdl\locc\sys\LOCCClassLoader.java
S:\locc\sys\dev\csdl\locc\sys\LOCDiff.java
S:\locc\sys\dev\csdl\locc\sys\LOCTotalPanel.java
S:\locc\sys\dev\csdl\locc\sys\LOCTotal.java
S:\locc\sys\dev\csdl\locc\sys\LOCDiffPanel.java
S:\locc\sys\dev\csdl\locc\sys\LeapCollector.java
S:\locc\sys\dev\csdl\locc\sys\LOCCTest.java
```

Size type: Java Line Size
Output format: General

Figure 3.1: LOCC's total size counting panel



```
S:\locc\sys\dev\csdl\locc\sys\LOCC.java
S:\locc\sys\dev\csdl\locc\sys\DiffPrinter.java
S:\locc\sys\dev\csdl\locc\sys\LOCCClassLoader.java
S:\locc\sys\dev\csdl\locc\sys\LOCDiff.java
S:\locc\sys\dev\csdl\locc\sys\LOCTotalPanel.java
S:\locc\sys\dev\csdl\locc\sys\LOCTotal.java
S:\locc\sys\dev\csdl\locc\sys\LOCDiffPanel.java
S:\locc\sys\dev\csdl\locc\sys\LeapCollector.java
S:\locc\sys\dev\csdl\locc\sys\LOCCTest.java
```

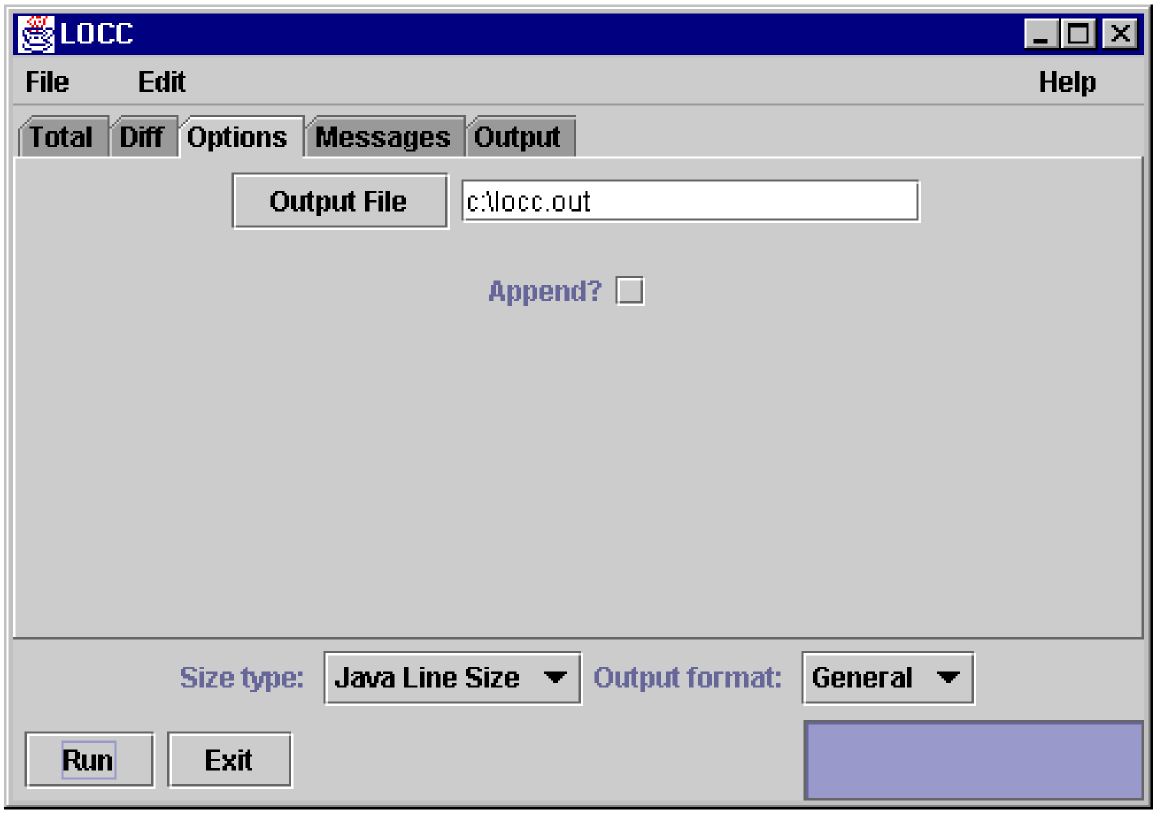
Size type: Java Line Size
Output format: General

Figure 3.2: LOCC's size difference counting panel

The "dirs" button allows an entire directory to be added, or those files in a directory matching a particular pattern. Additional buttons allow the user to selectively remove files from the list, or to clear the list entirely. Identical functionality can be accessed via the menu bar.

3.1.2 Diff Panel

This panel allows the user to specify the file pairs which will be compared in a difference operation. The user must specify both the old and new versions of the file. There are again two ways to do this. The "files" button will bring up a dialog which will allow the user to directly select the two files. The "dirs" button will bring up a dialog which will allow the user to specify two directories and a file extension. The directories containing the original file versions and the new file versions will be called the "old" and "new" directories, respectively.



c:\locc.out

Append? ☐

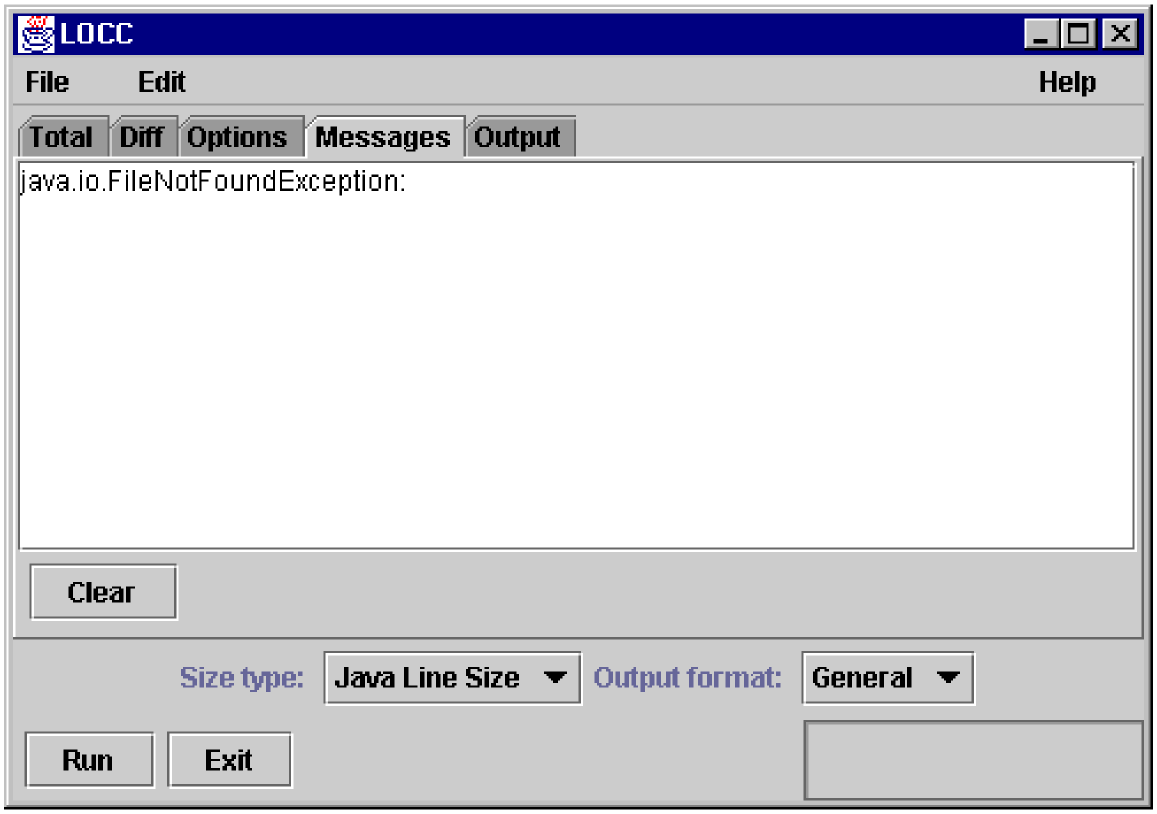
Size type: Java Line Size

Output format: General

Figure 3.3: LOCC's options panel

Processing proceeds as follows. All files in the old directory which match the chosen extension (all files ending in ".java", for instance) are collected in a list. Then a list of all files in the new directory is obtained and sorted. For each file in the old directory list, a search is made in the new directory. For each successful search, a pair of files is added to the list of pairs to be counted. Both the sort and the search are implemented in such a way that even large directories should be processed fairly quickly.

Like the total panel, the diff panel has buttons to remove file pairs, and to clear the list of file pairs.



```
java.io.FileNotFoundException:
```

Size type: Java Line Size

Output format: General

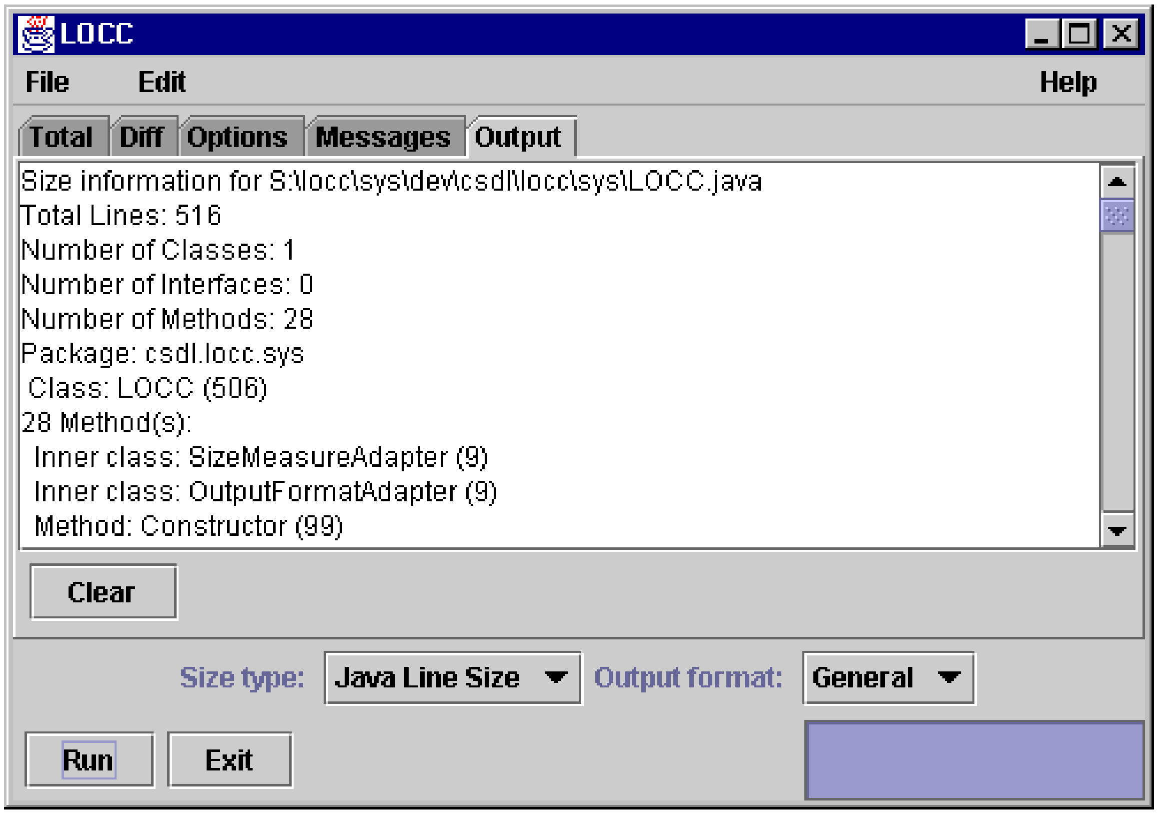
Figure 3.4: LOCC's message panel

3.1.3 Output Options

The output options panel allows the user to select a file in which to place the output of the counts. The file name can be typed directly into the text field, or the button can be pressed to activate a file chooser. A check box allows the user to select overwrite or append mode when writing to the file.

3.1.4 Messages

The message panel contains error and warning messages generated when producing the count. These error messages may be "procedural", e.g. the user specified an output file for which he does not have write permission, or "operational", e.g. some event occurred which prevented LOCC



```
Size information for S:\locc\sys\dev\csdl\locc\sys\LOCC.java
Total Lines: 516
Number of Classes: 1
Number of Interfaces: 0
Number of Methods: 28
Package: csdl.locc.sys
 Class: LOCC (506)
28 Method(s):
  Inner class: SizeMeasureAdapter (9)
  Inner class: OutputFormatAdapter (9)
  Method: Constructor (99)
```

Size type: Java Line Size   Output format: General

Figure 3.5: LOCC's output panel

from counting a file. The most common operational error is a parse error when using a metric which includes a parser. LOCC provides a simple mechanism for authors of size metrics to produce error messages which will be displayed in the message panel.

3.1.5 Output

The output panel captures output from the running size operation. Output may additionally be sent to a file, as specified in the output options panel.

3.1.6 Running LOCC

The user begins a session with LOCC by double clicking the mouse on the icon given to LOCC when it was installed. The LOCC GUI can also be invoked from a command line, with the command

```
java csdl.locc.sys.LOCC
```

Once LOCC is running, the user will add the files to be counted in total to the total panel, and the file pairs to be compared to the diff panel. If output to a file is desired, it should be specified in the output options panel. The size type is selected from the pull down list, and the output format from a corresponding list. The user then clicks the “Run” button, and the processing begins. Error conditions will produce a dialog directing the user to the message panel.

3.2 Using the command line interface

The command line interface is not intended to be the primary interface for users of LOCC, but is available, e.g. so that users who wish to use LOCC from within a script can do so. There are separate commands to invoke total and diff counting.

3.2.1 Total counting

The syntax for the command line for total counting is

```
java csdl.locc.sys.LOCTotal [options]
```

where `[options]` can be any of the following:

- `-sizetype` *metric* : this option tells LOCC which size metric to use. If *metric* is not specified, or if an invalid metric is given, a list of available size metrics will be printed.

- `-outformat` *format* : this option tells LOCC which output format to use for the given size metric. If no output format is given, or if an invalid output format is given, then available output formats for the given size metric will be listed.

- `-outfile` *file* `| -` : the `-outfile` option specifies the name of the file in which to send output. If a single dash (“`-`”) is given, then output is directed to standard output. If this option is not given, the default behavior is to produce a distinct output file for each input file, with name generated from the input file name with “.siz” appended.

- `-outdir` *dir* : this option, if given, directs LOCC to produce output files in the given directory.

- `-infiles` *file1 file2* … : this option specifies a list of files to count. This option can be given more than once with the result that the file names from all `-infiles` options are concatenated.

- `-indir` *dir ext* : the `-indir` option tells LOCC to count all the files in *dir* with file extension *ext*. A wildcard character is not needed, so the user would specify “`.java`” instead of “`*.java`”, which would be expanded by the shell anyhow.

3.2.2 Diff counting

The syntax for the command line for diff counting is

```
java csdl.locc.sys.LOCDiff [options]
```

where `[options]` can be any of the following:

- `-difftype` *metric* : *metric* is the name of the size metric to use in producing the diff. If this option is not given, or if an invalid metric is named, a list of valid metrics will be printed.

- `-outformat` *format* : this option tells LOCC which output format to use for the given size metric. If no output format is given, or if an invalid output format is given, then available output formats for the given size metric will be listed.
- `-new` *file* : *file* is the name of the new version of the file.
- `-old` *file* : *file* is the name of the old version of the file.
- `-outfile` (*file* `| -`) : the *-outfile* option specifies the name of the file in which to send output. If a single dash ("`-`") is given, then output is directed to standard output.

Notice that while LOCTotal can count any number of files in one invocation, LOCDiff can only process one file pair at a time.

Chapter 4

LOCC Architecture and Implementation

LOCC provides both a tool for size counting and a set of "shrink-wrapped" size metrics. This chapter will describe the design of the various components which comprise LOCC.

LOCC is implemented as a Java application. Presently, LOCC uses the 1.1 version of the Java language, and the 1.1.1 version of the Swing graphics library. A move to the Java2 platform is planned for the future. LOCC is composed of a total of 76 classes, with a total of 555 methods and 8116 lines of code. Figure 4.1 shows a high-level overview of the system.

The description of the architecture will proceed in three steps. First, I will describe the interfaces which must be supported by size counting code. Second, I will discuss the classes comprising the user interface. Finally, I will describe the code which implements the default size counting for the Java language. Comprehension of this chapter depends on a familiarity with the basics of the Java language and with object-oriented terminology in general.

4.1 Size counting interface

The size counting interfaces are the means by which "system" code, such as the command line parser and the GUI, invoke methods on code which provide size counting services. Since one of the primary goals of LOCC is modularity, it is important that these interfaces be general enough

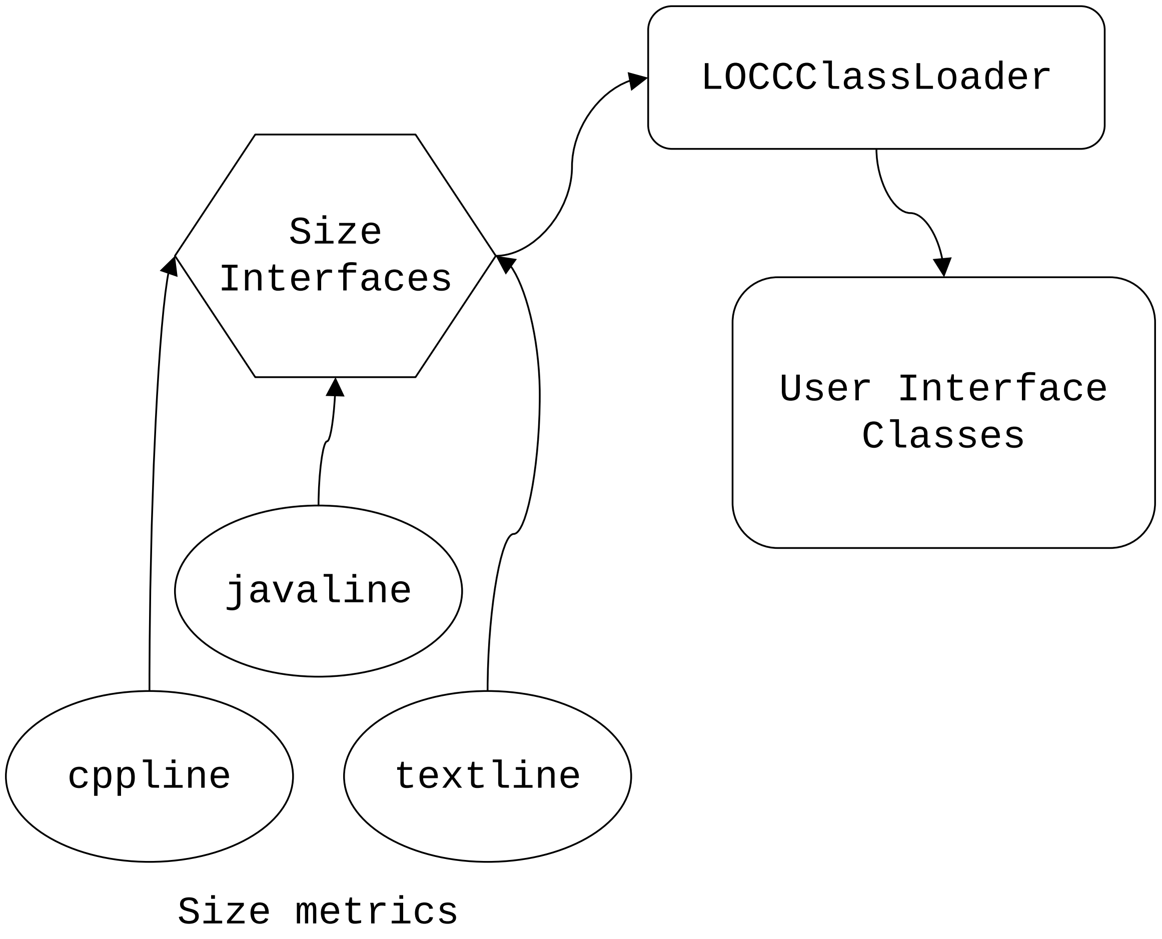


Figure 4.1: Overview of the LOCC architecture

to support any size counting procedure imaginable. A mechanism (which will be described in the user interface section below) by which size counting code can be loaded at run time implies that LOCC may not know at compile time which size counting modules will be used at run time.

4.1.1 The `SizeMeasure` interface

Figure 4.2 shows the relations between the four interfaces. The "top-level" interface is `SizeMeasure`, and this interface specifies the following methods:

- `String getName()`

  This method should return a name for the size measure. The name should be short, but descriptive. For example "Java lines of code".

- `String getCLIArg()`

  This method should return a `String` which will select the size measure from the command line. The `String` should be all lowercase with no spaces. An example is "`javaline`". No mechanism is in place for detecting when two or more size measures declare the same argument name.

- `String getDescription()`

  This method should return a description of the size metric. As the result of this method is meant to be used in "Help" operations, the message should be concise and clear.

- `OutputFormat[] getOutputFormats()`

  Each `SizeMeasure` is expected to provide its output in one or more formats. In practice, this usually means that output can be in a form suited for human consumption, or in a form meant to be used as the input to some other tool[21]. This method returns an array of objects implementing the `OutputFormat` interface, described below.

- Zero-argument constructor

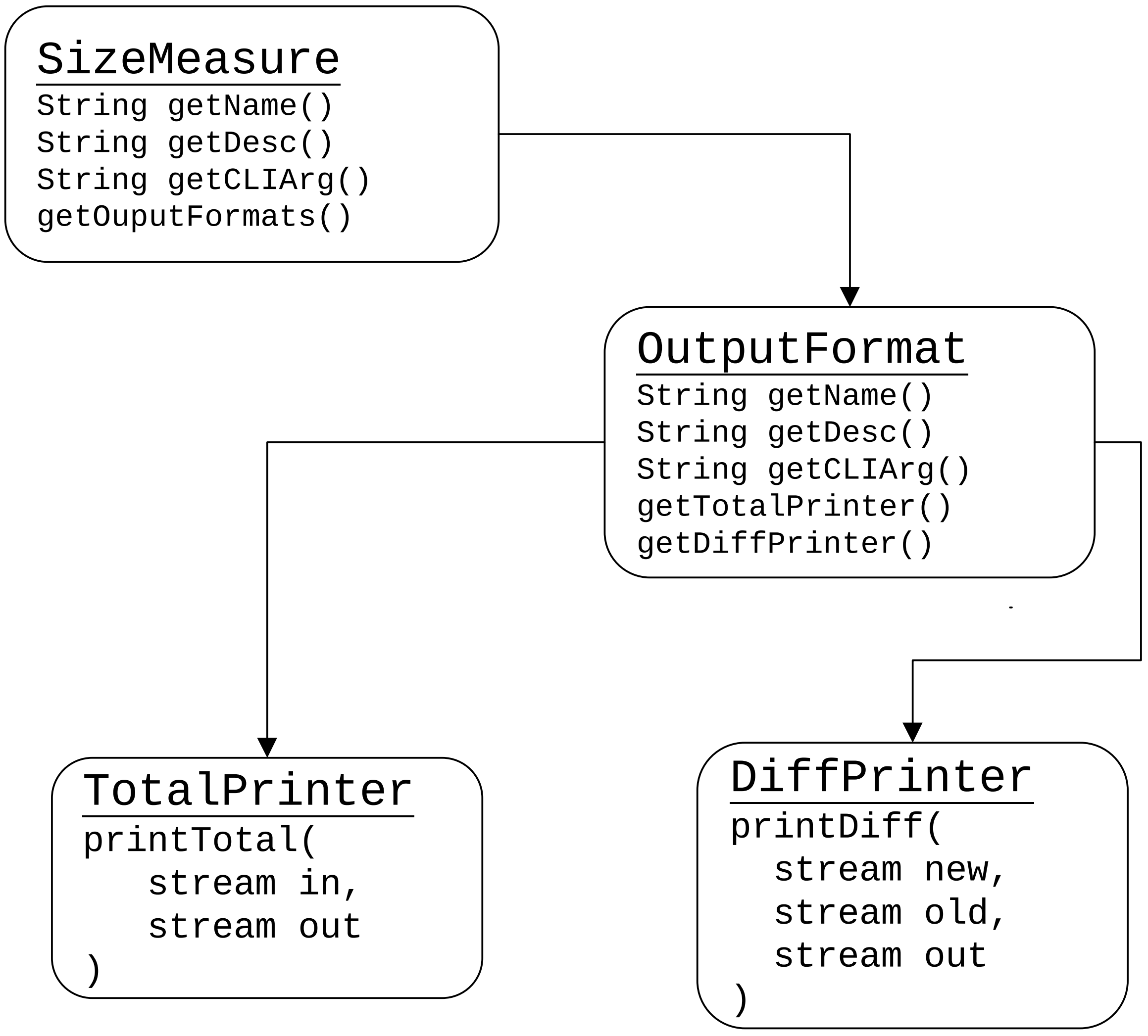


Figure 4.2: LOCC size counting interfaces

The `SizeMeasure` must also include a zero-argument constructor. The Java language does not support the specification of constructors in interface declarations, so this requirement is implicit. It is required because of the run-time code loading process described below.

4.1.2 The `OutputFormat` interface

The `OutputFormat` interface abstracts the behavior of code which produces a single size measure with a single output format. The methods in the interface are:

- `String getName()`

  Returns a `String` giving a short descriptive name for the output format, similar to the method in `SizeMeasure`.

- `String getCLIArg()`

  Returns a `String` used for selecting the output format at run time, similar to the method in `SizeMeasure`.

- `String getDescription()`

  Returns a description of the `OuputFormat`, similar to the method in `SizeMeasure`.

- `TotalPrinter getTotalPrinter()`

- `DiffPrinter getDiffPrinter()`

These methods return references to objects implementing the interfaces `TotalPrinter` and `DiffPrinter`, respectively. Methods on these objects are invoked to count code. There are several “administrative” methods in these interfaces which will not be described here. The two important methods, implemented by `TotalPrinter` and `DiffPrinter` respectively, are

- `void printTotal(InputStream in, PrintWriter out) throws IOException, Parse Exception`

-
  ```
  void printDiff(InputStream new, InputStream old,
                 PrintWriter out)
    throws IOException, ParseException
  ```

`printTotal` reads from its input and generates, by whatever means appropriate to the size measure and output format it supports, some report on the size of what it has read. In the process of generating the size of the object read, it may encounter some exceptional conditions. If an error occurs in either read or writing, the method can throw the standard Java exception `java.io.IOException`. If the size measure is attempting to parse the input and is unable to complete the parse, an exception `csdl.locc.sys.ParseException` may be thrown. Since `printTotal` and `printDiff` are finally invoked by code in the user interface classes they can use these exceptions to communicate information regarding the errors to the interface code, where the information can be displayed to the user.

Finally, note that there is no requirement that these interfaces be implemented by distinct classes. In some circumstances it may be simpler to have a single class implement more than one, or even all, of the interfaces. See Chapter 5 for an example.

4.2 User interface architecture

A simplified class hierarchy for the LOCC user interface classes is shown in Figure 4.3.

The class `LOCC` is responsible for initializing the system, loading size metrics, and displaying the main application window. When the `main` method of `LOCC` is called, it will create a new top-level window and add subclasses of the abstract class `LOCComponentPanel` to a tabbed panel.

`LOCComponentPanel` is an abstract base class which is extended by classes wishing to display a panel within LOCC. It arranges for event handling to be set up, and imposes a single size on all its subclasses.

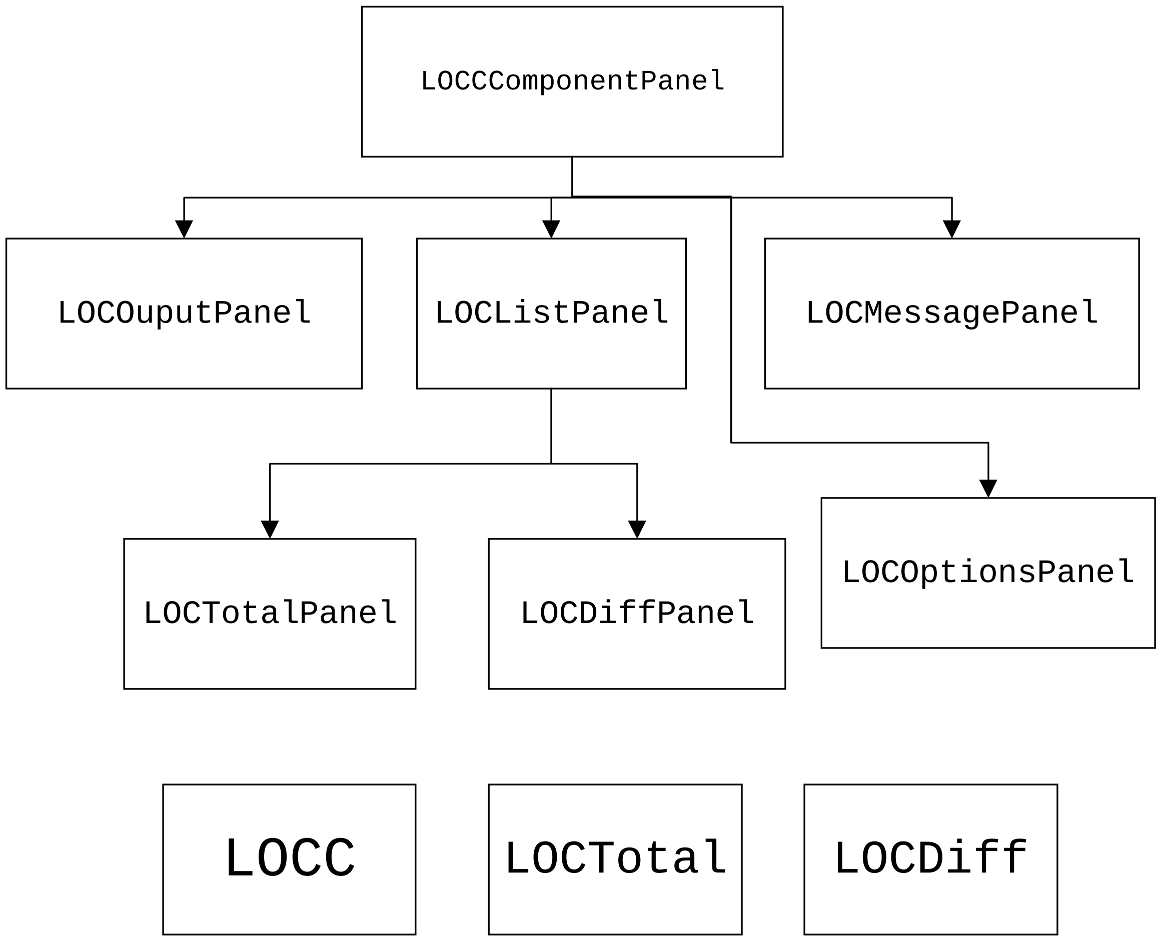


Figure 4.3: LOCC User interface classes

The classes `LOCTotalPanel`, `LOCDiffPanel`, `LOCOuptutPanel`, `LOCOption-sPanel` and `LOCMessagePanel` are all subclasses of `LOCComponentPanel`, and all define graphical screens in which input is accepted or output displayed. `LOCDiffPanel` and `LOCTo-talPanel` are also subclasses of `LOCListPanel`, which provides a list box which can display the files to be processed.

The classes `LOCTotal` and `LOCDiff` contain methods to handle command line invokation of total and difference counting, respecitvely. They request that the `LOCC` class load all size metrics but not display a GUI.

4.2.1 `LOCCClassLoader`

The `LOCCClassLoader` class provides the glue which joins the interface classes to the `SizeMeasure` classes. At system startup, LOCC looks for a properties file in certain defined locations. The properties file contains information about additional size metrics to be loaded at run time. The system creates an instance of `LOCCClassLoader` to locate and load these metrics. No mucking about with a CLASSPATH variable is required, and the size metric classes can be installed anywhere on the system where the user invoking LOCC has read privilege. For details on the format of the properties file, see Chapter 5.

4.3 Design of the Java LOC counter

The most fully realized size metric currently shipping with LOCC is a metric for counting Java source code. This metric was designed to suit the requirements of the Leap software development toolkit[21], which was itself inspired by Watts Humphrey's Personal Software Process[10]. Humphrey describes a regime for development effort prediction which depends upon the developer having collected data from past projects. One of the measurements required is program size. The developer is required to break past projects into "objects", which in an object-oriented language such as Java corresponds to classes, and to separate these objects into categories depending on their size. Size estimation for the current project proceeds by categorizing the objects in the current design using the developer's best guess as to which size category a particular object will fall into when it is finally coded. The size for the current, projected object is then taken from the average size of the objects in the same category from the historical data set.

Leap takes a similar approach, but generalizes the "object" categorization into multiple, hierarchical size categories. For example, using the terminology of Java, methods can be considered the atomic unit, with classes being composed of some number of methods, and packages in turn composed of classes. Leap is careful not to impose this particular set of hierarchies, and leaves it up to the developer to define the size hierarchy. While the PSP uses lines of code explicitly, Leap lets the developer use whatever unit he or she wishes. While Leap does not require it, the LOCC size counter for Java, which will be referred to as JAVALINE, does use lines of code as its basic unit. The primary reason for this is consistency with the PSP, and also for the general reasons described in Section 2.2.3, chief among them familiarity.

JAVALINE is a realization of a Leap size hierarchy for Java code. It proceeds as described above, by collecting data for the three hierarchical levels of methods, classes, and packages. The Java language complicates things slightly by allowing "inner" classes, which can be declared within classes, methods, or even simple blocks. These classes can themselves contain methods, which can contain classes, and so on.

JAVALINE take a practical approach to this complication, and others. Only top-level classes are counted in the "class" level of the size hierarchy. The reason is that the entities counted at that level are meant to be characteristic of a certain entity in a design. We are not so interested in the complexities of Java syntax as we are in the process of proceeding from design to implemented product. The idea of a "class" is meant to abstract a certain design entity which maps most closely onto Java top-level classes.

It could certainly be argued that some inner classes are themselves complicated beasts, and as such should be conferred the same status as top-level classes. While this may be true, JAVALINE takes the more simplistic view that *in general* this is not the case, and that an inner class should best be modeled as just another indistinguished piece of code.

JAVALINE is composed of two parts: the total size counter and the size difference counter. They will be described separately below.

4.3.1 Java Total Size

Total size counting for Java programs is fairly straightforward. JAVALINE includes a parser for version 1.1 of the Java programming language. The parser was generated using the JavaCC parser generator[11].

Parsing Java source files

A parse of a Java program proceeds as one would expect. The only special actions taken are to keep track of which lines in the input file contain Java code and which are empty or contain only comments. The JavaCC token manager makes this job fairly simple. The result of the parse is an abstract syntax tree which has been annotated with information to be used later. The information kept depends on the type of node in the tree. Nodes representing Java classes are annotated with the name of the class. Method nodes are annotated with the full signature of the method, to distinguish overloaded methods. If we are not interested in statement or expression level constructs, these can be excluded from the generated syntax tree.

```
package csdl.locc.java.parser;

public interface JavaLineParserVisitor
{
  public Object visit(CompilationUnit node, Object data);
  public Object visit(JavaClass node, Object data);
  public Object visit(InnerClass node, Object data);
  public Object visit(Interface node, Object data);
  public Object visit(Method node, Object data);
  public Object visit(AnnonClass node, Object data);
}
```

Figure 4.4: The JAVALINE Visitor interface

Applying the Visitor

JAVALINE utilizes the Visitor pattern[7] for traversing the generated parse tree. The reason for utilizing this pattern is partially the conceptual cleanliness it imposes, and partially because JavaCC can be directed to automatically generate much of the code to support the use of this pattern.

In a Visitor pattern, an object designated the Visitor is applied to the root node of a tree. The Visitor can examine the state and structure of the node, and can potentially recursively visit the children of the node. The pattern is expressed as a Java interface with one method, "`visit`", which is overloaded for each node type, as shown in Figure 4.4. Code wishing to visit the nodes in a parse tree need simply to supply a behavior for the visitor at each node type by implementing this interface. The advantage of the Visitor pattern is that the logic of the operation to be performed is kept in separate Visitor classes, instead of in the syntax tree nodes. Therefore, the tree nodes can contain only the information specific to the node, and different actions can be defined by providing different Visitors.

The total size counter in LOCC supports two output formats: output in a "general" human readable format, and output in a Leap data table for input into Leap. These formats are implemented as two Visitors. The `GeneralTotalVisitor`, for example, examines each node in a parse tree, producing human readable output as it traverses the tree.

4.3.2 Java Size Difference

The difference counter in the JAVALINE metric is the result of certain decisions described in this section. The choices made were motivated by the desire for the measured difference to correlate with the amount of time spent in moving from one program version to another.

JAVALINE makes the following choice regarding the diff problem:

- Deleting code takes zero time.
- Rearranging code takes zero time.
- Adding new code takes non-zero time.

One complaint that could be lodged against JAVALINE is that its is overly simplistic in its treatment of deleted code. Also, code added to a new version is given the same weight as the original code, when in fact much more thought (and therefore time) might have gone into the new code, since it may be constrained by the environment of the old code. A developer has fewer choices available when modifying existing code, which must continue to support existing interfaces, than when developing an entire product from the ground up. It might be reasonable to give added code a weight greater than unity.

It should be noted that while JAVALINE's diff counter does report differences in terms of syntactic units (new lines added to each method, class, and package) this is not a requirement of difference counting in general. In measuring differences, we are usually interested in how much a piece of code has changed, and not how much, say, each method changed on average. However, JAVALINE was designed to be used within the Leap toolkit[13], and Leap prefers its size measure-

```
public class aClass {
    int i, j;
    public String methodOne() {
        return "methodOne";
    }
    public String methodTwo() {
        return "methodTwo";
    }
}
```

Figure 4.5: Old program version

```
public class aClass {
    public String methodOne() {
        return "methodOne";
    }
    public String methodThree() {
        return "methodThree";
    }
    int i, j;
}
```

Figure 4.6: New program version

ments expressed in this manner. As will be described below, this causes some fairly significant problems.

Consider the program fragments in Figures 4.5 and 4.6. In going from the old version in Figure 4.5 to the new version in Figure 4.6 three transformations have been applied. First, a method has been deleted. Second, a new method has been added. Finally, the order of the declarations in the classes has been changed. When given these files as input, JAVALINE will arrive at a total “new” size of 3 lines.

Java size difference counting begins in the same way as the total count. An abstract syntax tree is generated for both the old and new versions of the program. Then a Visitor is applied to the “new” tree and is passed the old tree as an argument.

What counts as a difference?

The difference Visitors work by utilizing the following algorithm:

1. For the node currently being visited, look for a corresponding node in the old tree.
2. If no match is found in the old tree, then the code is new. Add its lines to the total count and continue.
3. If a match is found, first recursively visit the children of the node, looking for matches in the children of the old node. Then produce a textual difference count of the text in the old and new nodes.

The means of matching old and new nodes is worth looking at more closely. JAVALINE tries to do the best it can to make a sensible match. For class nodes, JAVALINE will look for a class with the same name. For method node, JAVALINE will look for a method with the same signature, that is, a method of the same name with the same number and type of arguments. A problem arises if the developer simply renames a class or method and makes no other changes. JAVALINE will then be unable to find a match in the old version and will consider the method or class to be completely new. This is probably the most serious fault in the method JAVALINE uses, and will be discussed further in Chapter 6.

When two pieces of code are finally compared, a specialized hash table is used to determine the number of line differences. The hash table uses the text of the line of code as a key, and as a value the number of times the key has been added. Multiple identical source lines are therefore counted correctly. Before being added to the hash table the line is stripped of leading and trailing whitespace, in an attempt to minimize the effects of simply reformatting code. Comments are also stripped before the code enters the hash table, so the difference count is insensitive to the addition or deletion of comments.

The question must naturally be asked, “Since this matching business can be fooled by simply renaming a method, why bother with it at all?” The answer is that JAVALINE was meant to be used in conjunction with the Leap toolkit, and the benefits of using an integrated toolkit such as Leap were felt to outweigh the possible complications of this size metric. It was also unknown when JAVALINE was designed how serious the problem was. If the problem occurred only rarely, then perhaps it would be enough to let JAVALINE fail and manually correct the results. As will be discussed further in Chapter 6, it appears that the problem is too serious to ignore. Chapter 6 will also discuss an alternative to JAVALINE which has been developed to alleviate the problem.

4.4 Other predefined size metrics

LOCC currently ships with several predefined size metrics. The JAVALINE metric has already been described. There is a metric for counting C++ code which operates in much the same way as JAVALINE. Because the parser for the C++ size metric was written without regard to current C++ standards, however, it should be considered less reliable at this time.

LOCC additionaly contains metric to count statements and expressions in Java source files. These metrics are fully functional, in that they contain methods to count both total size and size difference in terms of statements or expressions. However, when decisions had to be made in designing the difference counters, the easy solution was always chosen. Enhancing these metrics with more intelligent size difference measures is one of the proposed future directions for LOCC.

LOCC also contains a size metric for counting plain ASCII text files. There is nothing particularly interesting in the metric itself: it is basically a reimplementation of the UNIX `wc` and `diff` utilities. The ASCII text metric is interesting because is suggests a wider application domain for LOCC. Instead of being exclusively a tool for measuring the size of software products, LOCC can be utilized to measure the size of any work product. For example, one may wish to keep track of the size of a paper or proposal being composed. The same productivity measures and time estimation procedures used in software development can be used in these non-software development environments.

Chapter 5

Adding a size metric to LOCC

LOCC was designed with one goal clearly in mind: to make it as simple as possible to add size metrics to the tool. It was clear early in the development of LOCC that it would be foolish to think that one single size measure would be sufficient for all circumstances. Differences in development languages alone would most likely require a modification of the size metric being used. If the tool was to have any chance of being used, it would have to be flexible enough to support multiple size metrics.

Once a developer has decided on a new metric, his efforts should be focused on implementing that metric and not on the details of interaction with the user of the metric. It should be noted that these details are not as trivial as they may seem. It is common in computer science research to find products developed with the goal of illustrating some fine point of the research in question, but which take little care with actual usability issues. One may design the perfect program size metric, but making it simple to use is usually left "as an exercise for the reader". LOCC alleviates this problem by taking care of the boring details of the interfaces to the user and file system.

5.1 Deciding on a metric

The first step is naturally to figure out what is to be counted and how. This should be the most difficult step, and if it is, then LOCC can be considered a success. The only limitations imposed on the size metric by LOCC are that it be implemented as one or more Java classes and that the classes fulfill the size counting interfaces defined by LOCC. These interfaces are so general that they will not constrain the designer of new size metrics.

To illustrate the process of adding a new size metric to LOCC, I will present a simple size metric, one which counts the number of semicolons in a Java source file. This is not quite as useless as it might sound: since semicolons are used to separate statements in Java, the number of semicolons should be close to the number of statements. Two functions (methods, actually) must be defined: one for measuring total size and one for measuring size difference. In this case, total size will be the total number of semicolons in a source file. Size difference will be defined simply as the difference in the numbers of semicolons in the respective source files.

5.2 Write the Java code

The Java code which implements this count is shown in Figure 5.1. In most cases the size metric designer would probably spread the implementation out over several classes. To keep things compact here, I have compressed everything into a single top-level class and an enclosed inner class. The top level class, `JavaSemiCount` will be loaded by the `LOCCClassLoader` and incorporated into LOCC. The size metric will be constructed by a call to its zero-argument constructor, and LOCC will ask the metric for a description of itself by calling the `getName()`, `getCLIArg()`, and `getOutputFormats()` methods.

This size metric provides a single output format, represented by the inner class `Plain-OutputFormat`. Like the `SizeMeasure` of which it is a part, `PlainOutputFormat` provides methods which allow LOCC to ask about the format name, description, and command line

```
public class JavaSemiCount implements SizeMeasure {
    public String getName() { return "Java Semicolon Counter"; }
    public String getCLIArg() { return "javasemi"; }
    public OutputFormat[] getOutputFormats() {
        OutputFormat[] f = new OutputFormat[1];
        f[0] = new PlainOutputFormat();
        return f;
    }

    private static class PlainOutputFormat
      implements OutputFormat, TotalPrinter, DiffPrinter {
        private PrintWriter out;
        private String n1, n2;
        public String getName() { return "Plain output format"; }
        public String getCLIArg() { return "plain"; }
        public TotalPrinter getTotalPrinter() { return this; }
        public DiffPrinter getDiffPrinter() { return this; }
        public void printTotal(InputStream in)
          throws IOException {
            out.println(countSemis(in) +
                        " semicolons in file " + n1);
        }
        public void printDiff(InputStream oldS, InputStream newS)
          throws IOException {
            int diff = countSemis(newS) - countSemis(oldS);
            out.println("semi difference for files " + n1 + " and " +
                        n2 + " is " + diff);
        }
        private int countSemis(InputStream in)
          throws IOException {
            BufferedReader reader =
                new BufferedReader(new InputStreamReader(in));
            int c, count = 0;
            while ((c = reader.read()) != -1) {
                if (c == ';')
                    count++;
            }
            return count;
        }
    }
}
```

Figure 5.1: A simple size metric

```
!
! This is a properties file for LOCC
! The lines below would cause a size measure to be loaded
! from the class and directory given.
!

sizes : javasemi

javasemidir : /export/home/jdane/locc-ext
javasemiclass : JavaSemiCount
javasemipackage : javaSemi
```

Figure 5.2: An LOCC properties file

argument switch. `PlainOutputFormat` additionally implements the methods of both the `TotalPrinter` and `DiffPrinter` interfaces, so when it is asked to supply references to the objects which will ultimately execute the count, it simply returns references to itself. LOCC stores these references and calls `printTotal` and `printDiff` on them after opening streams on the input and output files.

5.3 Modify the LOCC preferences file

After writing and compiling the Java code, the next step is to direct LOCC to the code. This is done by modifying a preferences file. LOCC uses both a global and a user-level preferences file, so the modifications can be done at either level. The preferences file must look something like Figure 5.2.

The format of the file is a standard Java preferences file. Comments can begin with either the "`!`" or the "#" characters and extend to the end of line. Blank lines are ignored. Non-comment lines must be formatted either as *key : value* or *key = value*.

The "sizes" line is a list of short names of size metrics. The names are arbitrary, and are only used within the preferences file. For each size metric named on this line, three additional lines must appear. The keys for these lines are generated from the name of the size metric with the strings "dir", "class" and "package" appended. The "dir" value specifies the directory in the file system in which the classes reside. The "class" value gives the name of the main Java class which implements the `SizeMeasure` interface. The class must exist in a file in the directory specified in the "dir" value. The use of jar or zip files is not supported. The "package" value indicates the Java package in which the size code resides. Classes from sub-packages of this package are loaded in a manner similar to the system class loader, from directories beneath the top level directory.

5.4 Use the metric

After the preferences file has been modified, LOCC will be ready to use the new metric. Simply starting LOCC in the usual way will cause the new classes to be loaded and incorporated into LOCC. The entire process, from the design to the use of the size metric took 20 minutes. This particular size metric is, admittedly, rather simple. The important point is that almost all of that 20 minutes was used in actually thinking about how the size metric should work, and very little effort was wasted in thinking about extraneous issues.

Chapter 6

Evaluation

How well has LOCC lived up to the claims presented earlier in this thesis? There have been both successes and failures, but on balance LOCC seems to have lived up to its claims. LOCC has been evaluated using three methods. First, LOCC has been run on a large Java code base to confirm that it can calculate size data for a wide variety of source code. Selected output from LOCC's default JAVALINE metric has been checked against manually counted size data and has been found to be accurate. Second, current users of LOCC have been surveyed. Their answers, while qualitative, have indicated that LOCC is useful. Finally, I have tried to objectively evaluate the process of adding and extending size measures within LOCC. The responses from the user survey provided information as to what could be changed in the JAVALINE size metric, and I have attempted to evaluate the ease with which these changes were made.

6.1 Baseline functionality

LOCC has been used to measure the sizes of several systems developed at the CSDL, including the AWT library from Sun (99 classes, 1468 methods, 13080 lines), the Leap toolkit (308 classes, 2328 methods, 49082 lines) and LOCC itself (76 classes, 555 methods, 8116 lines). It has been used successfully in a graduate-level software engineering course.

A subset of the output from these uses of LOCC has been compared to manually counted size data. In all cases, the output from LOCC has been identical to the hand-counted cases.

6.2 User survey

All known users of LOCC were given a brief survey and asked for both their general impression of LOCC and specific problems they encountered while using it. I received five responses to the survey. The responses were mostly positive, with all respondents indicating that they had found LOCC useful. While the general attitude was favorable, there were some suggestions as to how LOCC could be improved.

The most commonly mentioned problem was with the JAVALINE size metric. The metric, as it was originally defined, made an effort to do a “structural difference” when comparing two versions of a program. That is, JAVALINE would try to find exactly how many lines had been added to each package, class, and method in a file. It did this by trying to match old and new versions of the package, class, or method, by using the algorithm described in Section 4.3.2. It was noted in that section that if the developer made a simple change to the source code that interfered with JAVALINE’s matching algorithm, such as renaming a class or adding a parameter to a method, then the class or method would be counted as new code. At the time of JAVALINE’s initial design, it was not know how much this would affect people.

After reviewing the responses of the user survey, it was clear that many people had encountered this problem. The students in the software engineering class especially, who were developing projects on their own and had little reason to maintain interface consistency across versions, had problems with this. The changes enacted to alleviate the problem are described below, in Section 6.3.

Several people mentioned features they wished existed in the user interface. One feature mentioned more than once was the ability to recursively measure files in a directory and its subdirectories. This feature is planned for the next revision of LOCC.

Despite the problems with JAVALINE, most respondents felt that LOCC has been successful, and had significantly helped them in collection of their process data.

6.3 Evaluation of LOCC extension

Probably the most interesting aspect of LOCC is how it allows, even encourages, experimentation with size metrics. As described in Chapter 5, adding a new size measure to LOCC is only as difficult as designing and implementing the metric itself. For a simple metric, this may be a matter of a few hours.

The situation with the JAVALINE metric is itself interesting. The problems involved in the syntax tree node matching algorithm were apparent, and yet it still seemed that a diff that was aware of the structure of the language shouldn't be thrown away. The Leap toolkit, for one thing, prefers that size measures, total and difference, be expressed in this way. Also, it can be worthwhile after working on modifying a program for some time to take a look at where the changes have been made. A diff which can distinguish changes made to one method from changes made to another can provide this.

Therefore, the JAVALINE metric was split in two. The original metric would be modified to try to be a bit smarter in its matching. The new metric, called SIMPLE-JAVALINE, was identical to the original in total size counting, but for size difference counting considered only file level differences. That is, a diff in SIMPLE-JAVALINE compares two files, without regard for program structure, returning the number of new or changed lines in the new file.

JAVALINE was improved by adding some logic to its matching algorithm. Matching would begin as in the original metric. After a failure to find a match, a special search for a matching

```
public class aClass {
    int i, j;
    public String methodOne() {
        return "methodOne";
    }
    public String methodTwo() {
        return "methodTwo";
    }
}
```

Figure 6.1: Old program version

```
public class aClass {
    int i, j;
    public String methodOne(int k) {
        return "methodOne";
    }
    public String methodTwo() {
        return "methodTwo";
    }
}
```

Figure 6.2: New program version

node would be undertaken. This search proceeded by sequentially examining each of a number of candidate nodes. The candidate nodes were those at the same level in the old syntax tree as the new node being matched.

For example, consider the program versions in Figures 6.1 and 6.2. Assume that the metric has proceeded to the point where it is counting the contents of `aClass`, and it is looking for a match in the old syntax tree for the method with signature `String methodOne(int)`. After failing to find a match, it will examine every child node of the old `aClass` node. For each of these nodes, a textual comparison will be made with the new `methodOne` method. If we find a node in the old tree which "looks like" the new method node, then we consider the two nodes a match. If more than one node in the old tree looks like the new node, we take the old node which looks most like the new one as the match. We say a node looks like another if the ratio of new and changed lines to total lines is less than 0.25. In other words, if fewer than 1/4 of the lines in a method are new or changed, we should be able to find a match.

Much of the code to implement the new and improved JAVALINE was taken without change from the original version.[1] In particular, the entire parser was reused. The process of making the changes and testing the new metric took 120 minutes. The construction of the SIMPLE-JAVALINE metric took 30 minutes. The fact that these changes could be made and incorporated into a working product so quickly is the most convincing evidence that LOCC has succeeded in its goals.

[1]Only 12 lines were added. If we counted deleted lines, the count would have been higher.

Chapter 7

Future Directions

There are a number of paths which could be followed to improve LOCC. The most obvious expansion path is the addition of new language support. We have seen how the inclusion of a parser in a size metric can lead to useful information. However, building a parser for a complex language is not a trivial task. Parser generators such as JavaCC go a long ways toward making the job easier, but there is still a non-trivial amount of work left to the implementor.

Once a parser has been constructed for a language, it is a simple matter to construct size metrics for the language. The effort spent on the parser can be leveraged to provide any number of size metrics, each expressing a different view of the "size" of the code. Experimentation may lead to the identification of one of these metrics as being superior to the others. Alternatively, multiple metric could continue to be used to give a fuller description of program size than could be obtained with a single metric.

One particular application of a parser could be the definition of a function point mapping for the language. For example, in the Java programming language we may decide that `TextField`s correspond to Albrecht's concept of a user input, and we could use the parser to determine the number of `TextField`s denoted in the source. The parser would have available the inheritance hierarchy, so that subclasses would also be counted correctly.

More research is needed to imbue the statement and expression based metrics in LOCC with more intelligence. These metrics perform perfectly when applied to the total size of a Java source file, but make rather simple decisions when it comes to size difference measuring. The current solution is to take the difference in total size between two program units as the size difference. While this is a reasonable first approximation, it seems that a more precise way of expressing the magnitude of the difference between two parse trees is needed.

Another possible extension to LOCC would be the ability to have a central repository of size metrics which can be accessed over a network. LOCC's dynamic code loading is currently restricted to loading code located on local disk resources. Extending LOCC to load from a network stream or URL would be straightforward.

Bibliography

[1] Allan J. Albrecht and Jr. John E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(6):693–647, November 1983.

[2] Rajiv D. Banker, Robert J. Kaufman, Charles Wright, and Dani Zweig. Automating output size and reuse metrics in a repository-based computer-aided software engineering (CASE) environment. *IEEE Transactions on Software Engineering*, 20(3):169–186, March 1994.

[3] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[4] Anne M. Disney and Philip M. Johnson. Investigating data quality problems in the PSP. Technical Report CSDL-98-04, Collaborative Software Development Laboratory, University of Hawaii, November 1998.

[5] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company, second edition, 1997.

[6] Sean Furey. Why we should use function points. *IEEE Software*, 14(2):28–30, March 1997.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[8] Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice-Hall, 1987.

[9] Tracy Hall and Norman Fenton. Implementing effective software metrics programs. *IEEE Software*, 14(2):55–64, March 1997.

[10] Watts S. Humphrey. *A Discipline for Software Engineering*. SEI Series in Software Engineering. Addison Wesley, 1995.

[11] Javacc web site. `<http://www.metamata.com/javacc>`.

[12] Javacount web site. `<http://csdl.ics.hawaii.edu/Tools/JavaCount/JavaCount.html>`.

[13] Philip M. Johnson. Project Leap: Lightweight, empirical, anti-measurement dysfunction, and portable software developer improvement. Technical report, University of Hawaii, Department of Information and Computer Science, October 1997.

[14] Barbara Kitchenham. The problem with function points. *IEEE Software*, 14(2):29–31, March 1997.

[15] Luiz A. Laranjeira. Software size estimation of object-oriented systems. *IEEE Transactions on Software Engineering*, 16(5):510–522, May 1990.

[16] J.L. Lassez, D. van der Knijff, and J. Shepherd. A critical examiniation of software science. *Journal of Systems and Software*, 2(2):105–112, June 1981.

[17] Anany V. Levitin. How to measure software size, and how not to. In *Proceedings, COMPSAC*, pages 314–318, 1986.

[18] Locc user's guide. `<http://csdl.ics.hawaii.edu/Tools/LOCC/LOCC.html>`.

[19] Mark Lorenz. *Object-Oriented Software Development*. Prentice Hall, 1993.

[20] K. H. Moller and D. J. Paulish. *Software Metrics: A Practitioners Guide to Improved Product Development*. Chapman and Hall, 1993.

[21] Carleton Moore. Project Leap: Addressing measurement dysfunction in review. In *Proceedings of the Eighth International Conference on Human-Computer Interaction*, 1999.

[22] UNIX Manual Pages. *diff*.

[23] UNIX Manual Pages. *wc*.

[24] Shari Lawrence Pfleeger. Measuring reuse: A cautionary tale. *IEEE Software*, 13(4):118–127, July 1996.

[25] Shari Lawrence Pfleeger, Ross Jeffery, Bill Curtis, and Barbara Kitchenham. Status report on software measurement. *IEEE Software*, 14(2):33–43, March 1997.

[26] Lawrence H. Putnam and Ware Myers. *Measures for Excellence*. Prentice Hall, 1992.

[27] Walker Royce. Pragmatic software metrics for iterative development. In *International Conference on Software Engineering*, page 585. ACM SIGSOFT and IEEE TCSE, 1997.

[28] Tama software. `<http://www.tamasoft.co.jp>`.

[29] Robert C. Tausworthe. Information models of software productivity: Limits on productivity growth. *Journal of Systems Software*, pages 185–201, 1992.

[30] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., second edition, September 1996.

[31] C.E. Walston and C.P. Felix. A method of programming measurement and estimation. *IBM Systems Journal*, 16(1):54–73, 1977.

[32] S.N. Woodfield, V.Y. Shen, and H.E. Dunsmore. A study of several metrics for programming effort. *Journal of Systems and Software*, 2(2):97–103, June 1981.