Factors Affecting the Programming Performance of Computer Science Students

by

John B. Raley

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Dr. Clifford A. Shaffer, Chairman

Dr. Sallie Henry

Mr. N. Dwight Barnette

October 11, 1996
Blacksburg, Virginia

Keywords: Programmer Performance, Application Frameworks, Software Metrics

Factors Affecting the Programming Performance of Computer Science Students

by

John B. Raley

Committee Chairman: Dr. Clifford A. Shaffer

Computer Science

(ABSTRACT)

Two studies of factors affecting the programming performance of first- and second year Computer Science students were conducted. In one study students used GIL, a simple application framework, for their programming assignments in a second-semester programming course. Improvements in student performance were realized. In the other study, students submitted detailed logs of how time was spent on projects, along with their programs. Software metrics were computed on the students' source code. Correlations between student performance and the log data and software metric data were sought. No significant indicators of performance were found, even with factors that are commonly expected to indicate performance. However, results from previous research concerning variations in individual programmer performance and relationships between software metrics were obtained.

ACKNOWLEDGEMENTS

I extend thanks to my advisor, Dr. Cliff Shaffer, for the many opportunities he provided. Many thanks also to Mr. Dwight Barnette for providing the forum for the research on using application frameworks. I am grateful to Dr. Sallie Henry for her guidance on using software metrics, as well as her patience and conversation.

While conducting this research, I was sustained by the love, support, and understanding of my wife, Sherry Raley; by the encouragement of Kathy Meacham; and by the memory of my grandmother, Jessie Davis Raley.

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1

Introduction

How quickly and correctly a programmer can produce software is affected by many factors. Some of these factors are external to the individual programmer, such as the design methodology and paradigm used, language used, and tools which are employed. Other factors affecting programmer performance are characteristics of the programmer, such as training, experience, temperament, and intelligence. The field of software engineering attempts to improve software quality, as reflected in its reliability and maintainability, and the productivity of software engineers and programmers.

One approach to improving software quality and programmer productivity is to provide the programmer with more powerful tools. In recent years, commercial software developers have begun to develop products with *application frameworks*. An application framework is "... a structure for building programs ... With a framework, the framework is in control. You usually start with a 'do nothing' program provided by the framework and add your own code. The framework calls your code at the appropriate time" [Mar96]. Application frameworks provide software engineers with several productivity advantages. First, an application framework gives the software engineer a ready-made structure for an application, sparing the engineer the task of inventing a design, and providing a standard which is common to everyone who uses the framework. Furthermore, application frameworks incorporate standard functionality, such as application initialization, event handling, and Graphical User Interface (GUI) handling, which is time-consuming and repetitive to implement in separate applications. Application frameworks are often integrated with a graphical tool for constructing a GUI, simplifying another aspect of software development.

A somewhat different way to improve software production is to focus on the practices of

the programmer. For example, software engineers are provided with design methodologies and process models to use when developing software. Also, programmers usually work with a set of stylistic rules for writing code. Structured programming, the practice of writing programs using only sequence, selection, and iteration for program flow control is an example of a rule set. Furthermore, guidelines for structuring programs are often employed. There is general agreement that programs should be written in *modules*, intuitive pieces of the program which interact with other modules as little as possible.

In the software industry, it is widely accepted that these practices lead to more efficient development of software. Students in Computer Science curricula are exposed to professional tools and programming practices, in part, to prepare them for employment in the software industry. However, it is also believed that by employing tools and practices used in professional environments students can complete their programming assignments more quickly and accurately; that is, professional software development methods can increase student performance.

This thesis presents two experiments concerning the relationship between programming methods and student programming performance. In one experiment, students in a second-semester introductory Computer Science course were provided with a simple application framework to use for an assignment. In the other experiment, students submitted detailed descriptions of the amount of time they spent on projects, and how their time was spent. Additionally, software metrics were computed on the students' programs. These data, along with student grades on the assignments, were used to test several hypotheses regarding student programming performance.

1.1 Previous Research Using Frameworks in CS Courses

A small amount of literature exists on using application frameworks in undergraduate Computer Science courses. Both commercial and academic frameworks have been used, typically in upper-level software engineering courses. The goals of using application frameworks

are consistant: increased student motivation; exposure to concepts required for commercial application framework development; and increased student programmer productivity, which allows more comprehensive projects. An overview of this literature is presented below. Additionally, two papers are described which detail using a cross-platform graphics library in a first-semester Computer Science course. Although the libraries were not application frameworks, many of the benefits of using a framework were realized, such as avoiding time-consuming low-level coding, and gaining experience with software reuse.

Laurie Werth incorporated MacApp, an application framework for the Macintosh, into a two-semester sequence of software engineering courses [Wer90]. Her primary objective of using MacApp was to enable students to be productive enough to write a system which is large enough to demonstrate specification, design, testing and maintainance issues. Students in her software engineering courses worked in teams of six or seven students, and produced tools to estimate software cost. Werth reported gains in student productivity and student motivation. Also, she noted that students had to adapt to MacApp's external control architecture. Werth's students successfully completed their projects, despite impediments such as long compile times and poor system documentation. She gave several important factors for successfully using MacApp (and these apply to other application frameworks): instructor proficiency in the framework and development environment, class time devoted to the framework and related topics, and previous student experience with the development environment.

Laszlo Szuecs used Borland's Object Windows Library (OWL) for group projects in an intermediate-level two-semester sequence of software engineering courses [Szu96]. Students in these courses typically had taken two semesters of introductory Computer Science courses, and were familiar with C++. An increase in student motivation and engagement, due to the exciting graphical nature of the projects, was reported. However, a large amount of class time was devoted to OWL in general, and to the student's projects in particular. Often, two of the four hours of class time each week were devoted to group work on the projects. Furthermore, the remaining class time was spent in a lab, with the instructor demonstrating

OWL development. This innovative course design achieved its goal of introducing students to an application framework; however, it would not be suitable for Computer Science courses in which students work individually on a term project while they attend lectures on other topics.

Giora Baram and Frank Friedman used a commercial integrated business package in a Computer Information Science course [BF88]. Students in the course are required to analyse, design, implement and demonstrate a software system which meets the needs of an actual business. The package includes modules for data-entry screens and menus, as well as database capabilities. A training video is provided with the package. Baram and Friedman report that using the business application package increased project quality and student productivity. Students were more satisfied with their work, since the business package enabled them to produce professional-looking projects. Also, students felt better prepared for employment.

Eric S. Roberts reported on how a simple graphics library was included in a first-semester, C-based Computer Science course [Rob95]. The library was not an application framework, and the programs typically did not present an interactive GUI. However, the issues which Roberts encountered are similar to those surrounding using application frameworks in introductory courses. Graphics were included in assignments to increase students' enthusiasm for their projects, as well as to introduce the concepts of parameter passing and stepwise refinement. The graphics library emphasized simplicity over feature-richness. Furthermore, the library was available for a variety of computing environments; thus, students who wanted to use the library were not committed to a single platform.

Koldo Gojenola, Tomás A. Pérez, and Arturo Jaime provided their students with libraries of Abstract Data Types (ADT) for use in assignments in a sequence of first-year Computer Science courses [GPJ95]. By allowing students to work with higher-level abstractions, they hoped that students would gain a better understanding of modular design and code reuse. In the first semester, students used ADT libraries provided by the professor for their programming assignments, which allowed for more interesting programming prob-

lems. Gojenola et. al. found that, when the ADTs were incorporated into assignments, the length and complexity of programs decreased significantly compared with those from previous courses. Furthermore, student evaluations of the course indicated that students were pleased with this approach. In the second semester, students implemented some of the ATDs used in the previous semester. Implementing previously used ATDs reinforced the concepts of modular design and code resue introduced previously.

Randy Pauch, Matthew Conway, and Robert DeLine used SUIT, an application framework, in undergraduate software engineering courses [PCD92]. SUIT is cross-platform, and is available for MS-DOS, Macintosh, and UNIX / X Window platforms. It is designed for simplicity, and thus does not incorporate as much functionality as commercial frameworks. SUIT mananges interface elements (called "objects" in the SUIT documentation) which include buttons, pulldown menus, and sliders. User manipulation of objects causes a programmer-defined C function to be called. SUIT interface specifications are stored in ASCII files. SUIT does not include a separate tool for building a user interface; instead, interface objects can be created and edited from within a running SUIT application.

The SUIT development team emphasized SUIT's accessability to novice programmers. To this end, a tutorial and on-line reference system was provided. Also, user feedback was obtained from various sources. A ten-page, paper-and-pencil tutorial for SUIT was developed. This tutorial required between 1.5 and 3 hours to complete, during which the users interacted with SUIT, according to the directions in the tutorial. The material in the on-line reference was taken from the 160-page SUIT reference manual. SUIT developers gathered feedback from a variety of sources. An electronic mail account was available, to which questions and comments could be sent. Furthermore, the SUIT team observed pairs of students working on the tutorial. Observers were able to determine which aspects of SUIT were confusing by listening to the students' conversations.

SUIT was used in undergraduate software engineering courses, in which projects were assigned which required a Graphical User Interface (GUI). Typically, college juniors were enrolled in these courses. Pauch et. al. report that students were able to use SUIT

productively after about 2.5 hours of tutorial experience. The courses in which SUIT was used have expanded their scope, and students were motivated by the professional look of their applications. Students continued to use SUIT for other projects after using it in their software engineering courses. However, students had great difficulty in accepting the external control imposed by the SUIT framework, in which SUIT's event handler calls programmer-provided functions. In students' previous experience, their functions had been on the execution path for the duration of the program.

It is clear that application frameworks can be used successfully in Computer Science curricula. However, most attempts to use frameworks have been undertaken in junior- and senior-level software engineering courses. These courses typically feature group projects. Furthermore, these courses often are offered in two-semester sequences, and can spare significant class time for instruction in using the framework. These conditions make upper level courses more amenable to including commercial frameworks than lower-level courses. Overall, application frameworks tend to achieve the goals of motivation, introduction to advanced ideas, and increased student productivity. It appears that quality of documentation, quality of instruction, and intructor proficiency with the framework are at least as important to using the framework successfully as the design of the framework itself.

1.2 Software Metrics

Quantitative software metrics provide a numerical measure of intuitive characteristics of software, such as "size" and "complexity." Software metrics considered in this study fall into three categories: code metrics, structure metrics, and hybrid metrics. Code metrics are a good measure of a module's or function's internal complexity. They are computed from a count of tokens in the source code for a module or function. Structure metrics measure an aspect of a module's or function's complexity that cannot be adequately captured by code metrics alone: its connectivity to other modules or functions. Henry and Kafura's Information Flow metric measures a function's complexity in terms of the number of flows

of information into and out of a function. Hybrid metrics measure both aspects of a module's or function's complexity (internal and contextual). The Henry-Kafura Complexity metric is a hybrid metric; it is given by the product of a code metric and the Information Flow metric.

Several of the more common software metrics have *thresholds* associated with them. The metric for a function should not greatly exceed the metric's threshold. Functions whose metrics exceed thresholds usually can be rewritten as a set of smaller functions.

There are many software metrics from which to choose. The metrics chosen for this study are widely used, and have well-known properties. Furthermore, the metrics used in this study are automatable; that is, they can be computed from source code alone, without additional human input. These metrics are defined for individual C functions. The metrics for an entire program are the sum of metrics for every function within the program.

1.2.1 Code Metrics

The simplest measure of a program's size is Lines of Code (LOC), the number of executable lines in the program's source files. Larger programs may be measured in thousands of Lines of Code (KLOC) or millions of Lines of Code (MLOC). Disagreement exists about exactly what constitutes a line of code. For example, some languages, such a C, allow multiple executable statements on a single line, and allow a single executable statement to span multiple lines. Also, it is unclear whether blank lines and comment lines should be included in the line count. In this study, a program's LOC measure is the sum of the number of lines in each function in the program. Blank lines and comment lines are included in this count. The threshold for the LOC of a function is 35.

Maurice Halstead proposed several software metrics [Hal77]. Halstead's software metrics derive from four measurable properties of a program:

η_1, number of distinct operators

η_2, number of distinct operands

N_1, total number of operators

N_2, total number of operands

A simple lexical analyser is sufficient to produce data from which Halstead metrics can be computed. In this study, four of Halstead's software metrics are considered: Length (N), Calculated Length ($\hat{N}$), Volume (V), and Effort (E).

The Length (N) of a function is the number of operands (N_1) plus the number of operators (N_2). Note that Length has an advantage over Lines of Code: a function's Length measure is not affected by programmer style, or by comments and blank lines.

Closely related to the Length metric is Calculated Length, denoted $\hat{N}$. Halstead found that a program which did not contain repeated sequences of symbols (operators and operands) would have a length of at most $\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$. The restriction on repeated sequences of symbols is reasonable, since a well-written program will not duplicate common computations. Furthermore, Halstead found high correlations between N and $\hat{N}$. If $\hat{N}$ is much smaller than N, then the function (or program) is likely to contain duplicated or unnecessary expressions.

A program's Volume (V) is the minimum number of bits needed to represent the program as a sequence of symbols. A program with η_1 distinct operators and η_2 distinct operands has a vocabulary of $\eta = \eta_1 + \eta_2$ elements; that is, the entire program uses η distinct symbols. A minimum of $\log_2 \eta$ bits are required to represent every symbol in the program. Thus, if N is the total number of operators and operands used in the program, the program can be represented with $N \log_2 \eta$ bits.

The last Halstead metric considered in this study is Effort (E). A program's Effort metric is meant to quantify the number of mental discriminations or amount of mental effort required to produce a program. This is derived from two other measures: Volume (discussed earlier), and Program Level. Stated briefly, Program Level measures the ratio between a given algorithm implementation, and the smallest possible implementation. The closer an implementation is to the minimum-size implementation, the higher the program

level. This matches the concept of "high-level" languages, in which complex ideas can be represented with short expressions. The Effort measure of a program (or procedure) is given by: V/L, where V is the Volume measure, and L is the Program Level measure.

Formally, L is defined as follows: $L = V^*/V$. In this equation, V^* is the *potential volume*, the volume of the most succinct expression of the algorithm, function or program. As this is almost never known, an approximation of L, $\hat{L} = 2/\eta_1 \times \eta_2/N_2$, is often used in place of L when computing the Effort metric. This study computes Effort using $\hat{L}$.

Another software metric considered in this study is McCabe's Cyclomatic Complexity [McC76]. Any program can be represented as a directed graph, in which nodes in the graph represent executable statements, or blocks of executable statements. An edge from node A to node B exists if the flow of control can proceed from node A to node B. When a program is considered as a graph G, the Cyclomatic Complexity of G, V(G), is given by $E - N + 2$, where E is the number of edges in the graph and N is the number of nodes. Generally, a function's Cyclomatic Complexity should not be greater than 14.

The Cyclomatic Complexity of a function can be computed quite easily; it is not necessary to construct a function's graph. A function's Cyclomatic Compexity is simply the count of the decision statements (such as `if`, `for`, and `while`), plus 1 [McC76].

Two versions of the Cyclomatic Complexity are used in this study: the standard Cyclomatic Complexity, and Extended Cyclomatic Complexity. Extended Cyclomatic Complexity counts decision predicates instead of decision statements [Mye77]. Thus, a decision statement containing N simple conditions adds one to its function's Cyclomatic Complexity, and N to its function's Extended Cyclomatic Complexity. This study considers both Cyclomatic Complexity and Extended Cyclomatic Complexity.

1.2.2 Structure Metrics and Hybrid Metrics

A somewhat different measure of a program's complexity is the Henry-Kafura Information Flow metric [HK81], a stucture metric. The Information Flow metric measures the connectivity of a module or function to the entire program. Each function is assigned two

values: fan-in and fan-out. A function's fan-in value represents a flow of information into the function; fan-out represents a flow of information out of the procedure. For any function F, the Information Flow metric C_F is given by (fan-in $\times$ fan-out)^2.

Henry and Kafura propose several different types of information flow. *Global flow* from A to B occurs when A writes to some global data structure D, and B reads from D. *Local flow* from A to B occurs when: A calls B, B calls A and A returns a value to B, or C calls A and B, passing an output from A into B. Local flow is *direct* if A calls B; otherwise, local flow is *indirect*. This study utilizes a simplified version of the Henry-Kafura Information Flow metric which considers only direct local information flow.

Henry's and Kafura's Complexity metric is a *hybrid metric*, and is derived from the Information Flow metric. The complexity HC_F is given by: length $\times$ (fan-in $\times$ fan-out)^2, where length may be Halstead's Length or McCabe's Cyclomatic Complexity. The Complexity metric considers both a module's or function's internal complexity, and its connectivity to the rest of the program. In this study, Halstead's Length metric is used to compute the Complexity metric.

1.2.3 Relationships Among Types of Metrics

Code metrics are computed from a count of tokens in source code. Due to this similarity, it can be said that the code metrics all measure a similar quality of a function. The structure metric presented above is not computed from a simple count of tokens; thus, it measures a quality of a function different from that measured by the code metrics.

Henry, Kafura, and Harris found that when a set of code metrics and the Information Flow Complexity metric was computed for the functions in a large piece of software (the UNIX operating system), the code metrics showed very high correlations among themselves, while the correlations between the code metrics and the Information Flow Complexity metric was relatively low [HKH81]. The code metrics and the Complexity metric were found to be *valid*, meaning that they were all good predictors of where errors occurred.

1.3 Research Into Individual Programmer Productivity

The field of software engineering could benefit from learning what makes individual programmers more productive. Many different design methodologies and coding practices have been proposed, in the attempt to enable programmers to do their work more quickly and accurately. Some contribute greatly to increased performance; others do not [Bro75, pp 128-137]. Furthermore, it is widely known from research as well as from anecdotal evidence that the productivity, or performance, of one software engineer can differ greatly from that of another. Knowing the factors which contribute to increased programmer performance could greatly benefit managers, improving staff selection and management practices.

However, considering its importance and potential benefits, there is little research on what factors improve a programmer's productivity, or account for the vast differences between individual programmers' performance. Several studies of programmer performance and productivity are summarized below. All but one of these studies are in industrial (as opposed to academic) environments, and use varying definitions of productivity and performance. The studies seek to relate performance to such factors as: experience, including the amount of time a subject has been programming and the subject's breadth of experience; coding practices; scores on paper-and-pencil tests; software metrics computed on the programs; and subjective evaluation of design effort.

H. Sackman, W. J. Erikson, and E. E. Grant conducted studies of individual industrial programmer performance [SEG68]. Their primary goal was to determine the degree to which a programmer's performance increases (if any) when the programmer works on an on-line system instead of a batch system. Additionally, they collected data on other factors which affect programmer performance.

Sackman et al. studied two groups of programmers. One group consisted of twelve programmers who were considered "experienced" – the average programmer in this group had seven years of experience. Members of the other group were classified as "programmer trainees."

Members of the "experienced" group were each given two problems to solve. One problem required the subjects to write a program which would accept as input an algebraic equation in a single independent variable. Then, given a value for the independent variable, the program was to evaluate the equation. For the other problem, the subjects wrote a program to find the (unique) path through a 20×20 maze, represented as a 400-element array containing information on the directions in which movement was allowed. Half of the group members solved the Algebra problem online and the Maze problem offline; the other half solved the Maze problem online and the Algebra problem offline. Programmers were allowed to refer to a published solution to the Maze problem.

Several measures of each subject's performance were employed, including coding time and debugging time. Also, each programmer took the Basic Programming Knowledge Test (BPKT), a paper-and-pencil test designed to test one's grasp of skills needed for programming. Coding time was defined as the number of hours spent on a program until the program compiled. Debugging time was considered to begin after a successful compile of the entire program, and end when the program processed a set of test data with no errors.

The "trainees" were also given two problems to solve. The first problem was to write a numerical sort routine. The second problem required a program which would find the arrangement of four cubes, which met a certain condition. Five of the trainees solved the Sort problem interactively and the Cube problem noninteractively. The remaining four solved the Cube problem interactively and the Sort problem noninteractively. Coding time was not recorded for the trainees; otherwise, the trainees were subject to the same performance measures as the experienced group.

This study found large differences in the performance of individual programmers. The ratio between the fastest and slowest times to code the Maze solution (all other factors, including type of system used, being the same) was 25:1; for the Algebra problem, this ratio was 16:1. The slowest debug time on the Algebra problem was 28 times more than the fastest. The slowest Maze debug time was 26 times greater than the fastest. The performance differences between the fastest and slowest trainees were almost as great. Sackman,

Erikson and Grant found that BPKT scores were a good predictor of performance of the trainees, but not the experienced programmers. They did not find any other predictors of individual programmer performance.

Sackman's, et al. report of individual differences of between 25:1 and 28:1 was criticized [Dic81]. These differences were obtained by comparing online programmers to offline programmers. This comparison was not mentioned in Sackman's et al. paper. However, other research has found performance differences of at least 20:1 ([Cur81], see below).

C. E. Walston and C. P. Felix describe a method of estimating programming productivity [WF77]. Walston's and Felix's work involved building a database from questionaires, which project managers submitted periodically. Questionaires include data such as: number of lines of source code delivered, pages of documentation, which programming languages were used, total effort (in man-months), and what "improved" programming techniques were employed (such as structured programming). Factors which are found to show high correlation with productivity, such as interface complexity, previous staff experience with computer operation or programming language, and ratio of staff size to duration are used in estimations. To estimate the amount of effort needed to complete a project, a manager completes a shortened version of the questionaire, and a "productivity index" is computed for the proposed project. However, the database contained no information about individual programmers; thus, it can provide no estimate of an individual's productivity based on the individual's characteristics.

Earl Chrysler studied COBOL programmers in an attempt to develop a reasonable method for estimating the amount of time which will be needed to complete a programming task [Chr78]. Chrysler considered both the characteristics of the programming task and, unlike Walston et al., also considered characteristics of the programmer. Furthermore, Chrysler's method was based on purely quantitative variables.

Chrysler's measure of productivity was the amount of time needed to develop a program to specifications, develop test data for the program, run tests on the program, debug the program, and provide the program's documentation.

Chrysler considered these factors in estimating a programmer's productivity:

1. Number of months of programming experience.
2. Number of months programming experience using COBOL.
3. Number of months of experience using the specific compiler to be used for the subject program.
4. Number of months of experience in programming business applications
5. Number of months experience in programming for the current employer
6. Age of the programmer (in months)
7. Years of formal education

Chrysler also considered properties of the subject program, such as number of input files, number of output files, and number of mathematical operations. However, these properties are somewhat specific to COBOL, and do not relate at all to the individual programmer.

Chrysler conducted his study at a data processing facility where new COBOL programs were created. The facility was chosen for several reasons: a time reporting system was in place, there was a wide range of staff experience, and the facility engaged in contract work, providing for a broad range of applications. Time and programmer data were available for 31 different programs at this facility.

The $\log_{10}$ of the programming time on each program was correlated with the $\log_{10}$ of each programmer characteristic listed above. Logarithms were used to remove "skew" effects which occur when a few data points are much larger than the rest. Chrysler found that programmer age correlated significantly (and negatively) with programming hours (at the .0005 level). The other programmer characteristics given above correlated significantly (and also negatively) at the .005 level of significance, with the exception of years of formal education, which correlated significantly only at the .05 level.

Sylvia B. Sheppard, Bill Curtis, Phil Milliman, and Tom Love conducted three experiments which tested the relationships between programmer performance and "good" programming practices such as structured programming, mnemonic variable names, and extensive comments [SCML79]. Furthermore, they also collected data on programmers'

characteristics using two tests and a questionaire. The definitions of performance were somewhat different from those used in other studies. In the first experiment, performance was considered to be program comprehension, the ability to recall a previously seen program. Comprehension is an important step in maintainance tasks. In the second experiment, performance was measured as the time required to make a certain modification to a program, and the accuracy of the modification. Performance in the third experiment was the time required to locate and correct a program bug. Programs were written in Fortran.

In the first experiment, subjects examined programs, then reproduced the programs as well as they could from memory. The programs were short (between 26 and 57 lines) and represented applications from three different areas: engineering, statistics, and "non-numerical" applications. Within these three program types were three programs; thus, a total of nine distinct programs were used in this experiment.

Versions of each program were written with one of three types of control structures: strict, natural, and convoluted. A strictly structured program used only three basic control constructs: sequence, selection, and iteration. Naturally structured programs exhibited limited deviation from strictly structured programs; multiple returns, "judicious" GOTO, and mid-loop forward exits from DO-loops. Convoluted programs featured unrestricted use of GOTO, mutliple returns, computed GOTO's, and forward and backward exits from DO loops.

Additionally, three levels of mnemonics for variable names were used. The highest level consisted of names chosen most frequently by non-participants. The next level were names chosen less frequently by non-participants. The lowest mnemonic level were names which were simply one or two randomly chosen alphanumeric characters.

From each of the nine programs (three from each of the three classes), nine versions were produced: one version for every unique combination of control structure type and mnemonic level. Thus, a total of 81 unique source files were used in this experiment. Each subject was given three programs, one at a time. Subjects were given 25 minutes to study a program, and were allowed to take notes. At the end of the 25 minute period, notes were collected

and each subject was allowed 20 minutes to write a functionally equivalent program from memory. No subject was given two programs with any common characteristic. In all, 36 programmers participated in this experiment; 9 programmers repeated work done by other participants.

Performance was judged by the number of statements in each program which were recalled correctly. Subjects' programs were not penalized for using variable names and statement numbers different from those used in the original program. Three judges scored each of the subject's programs; the judges' scores were found to correlate closely (.96, .96, and .94) with one another's. The average score was used as the performance measure.

The experimenters found that the type of control flow used had a statistically significant effect on performance (at the .05 level). Convoluted programs, as might be expected, were more difficult to comprehend. However, no statistically significant difference was noted between the strictly structured and naturally structured programs (though programmers performed slightly better on naturally structured programs). Mnemonic level of variable names had no significant effect on programmer performance.

The second experiment was designed to test the effects of control structure type and documentation type on programmer performance. Three programs from the first experiment were used. For each program, three modifications were defined. The three types of control structures used in the previous experiement were also used in this experiment. Also, three types of in-code documentation were used: global comments, in-line comments, and no comments. Global comments gave an overall description of the program and descriptions of its variables. In-line comments appeared throughout the program and gave specific information about sections of code.

As in experiment 1, the unique combinations of all variables resulted in 81 unique tasks. Each subject was given three programs (along with modification instructions) serially. In all, 36 programmers participated in this experiment. Subjects were allowed as much time as needed to complete the modifications. The amount of time spent on modifications was recorded.

Two factors were used to judge programmer performance: accuracy of modification and time required to complete the modification. Accuracy was judged against a predetermined program modification; an accuracy score was computed by comparing the subjects' modifications with the experimenter's modifications. (Sheppard et al. are not specific on this point.)

As in the first experiment, convoluted programs resulted in poorer programmer performance; no statistically significant difference in performance was noted between strictly structured programs and naturally structured programs. Also, inclusion of comments, both global and in-line, did not affect programmer performance significantly. The best predictor of performance was the extent to which a program required modification (more specificly, the number of new lines required), with longer modifications degrading performance.

Sheppard's, et al. third experiment tested the effects of different types of control structures, program length, and placement of bugs on debugging performance. Three programs, each containing several subroutines, were used in this experiment. The programs were presented in three different lengths: 27-75 lines, 100-150 lines, and 175-225 lines. Programs were shortened by deleting subroutines; subjects were to assume that missing subroutines worked correctly. Bugs were introduced into each configuration of each program. Bugs were of three types: computational, a sign change in an arithmetic expression; logical, an incorrect operator in an IF statement; and data, an incorrect index value. Effects of bugs were isolated to a small part of the program. Exactly one bug was introduced into each program.

For each program/bug configuration, three control flow types were used. Naturally structured flow was used in this experiment; the other control flow types, Fortran 77 and graph-structured, differed from those used in previous experiments. Fortran 77 control flow allows DO, DO-WHILE, and DO-UNTIL constructs, with no exits from loops; no GOTO statements, and a single return. Graph-structured control allows forward GOTO's and backward GOTO's only for expanded loops. Otherwise, graph-structure control and Fortran 77 control are the same.

Once again, there were 81 unique programming problems. Each subject was given three problems, one at a time. Subjects were given input files, source code, an example of correct output, and erroneous output, with differences between correct and erroneous output indicated. Since 54 programmers participated in this experiment, each debugging problem was attempted twice. Subjects were allowed 60 minutes on each program to find and correct the problem.

Performance was measured by the number of minutes required to locate and fix the error. The experimenters found that the size of the program had a "modest" effect on the time needed to debug it, while the deviations in control flow did not have a significant effect on debug time. The most significant factor affecting debugging time was simply the program and bug conbination. It is noted that bugs which appeared close to WRITE statements which they affected were found more quickly than other bugs.

Sheppard, et al. looked for programmer characteristics which had statistically significant effects on performance. Before each experiment, participants were given a pretest which resembled the tasks they would be asked to perform in the experiment. The pretest was the same for all participants, and was used as a measure of each participant's ability. In the first experiment, which was considered the most demanding, pretest scores accounted for only 17 percent of the variance in performance. The correlation between pretest scores and performance was less on experiments 2 and 3. This finding runs counter to the common conception that wide differences among individuals' performances exist, especially considering that subjects in the experiments had a wide range (from several months to 25 years) of experience.

All subjects submitted questionaires which reported the number of years of experience they had, as well as other programming experience. Additionally, participants in experiment 3 reported how many other programming languages they knew. Furthermore, they were presented with a checklist of Fortran concepts, and were asked to indicate concepts with which they were familiar. The experimenters found that number of years of experience did not correlate with programmer performance in any experiment. This finding is contrary to

Chrysler's finding that experience is a major factor in programmer performance. (Chrysler's performance measure was the number of hours required to complete a task; this measure differs from the measures used by Sheppard et al., though one would expect a high degree of correlation between the two measures.)

Sheppard et al. found that other experiential factors were better indicators of programmer performance. The performance of programmers with three or fewer years of experience correlated highly with scores on the "Fortran concept" test. This finding is similar to that of Sackman et al., who found that scores on the Basic Programmer Knowledge Test were a good predictor of the performance of trainees, but not of experienced programmers. Another good predictor of the performance of programmers with three of fewer years of experience was the number of programming languages known.

Sylvia Sheppard, Bill Curtis, Phil Milliman, M. A. Borst, and Tom Love conducted another set of experiments, similar to the ones described above, designed to test the relationship between software metrics and programmer performance [CSM+79b], [CSM79a]. Experiment 1 required the subjects to recall a program; in experiment 2 subjects modified a program, and in experiment 3 subjects debugged a program. As in their previous experiments, Sheppard et al. gave the subjects a series of programs, varied by application, length, control flow, type of commenting, and mnemonicity of variable name (in experiments 1 and 2) and bug type (in experiment 3). Performance measures on each task were the same as those used in previous experiments. Halstead's Effort metric (E), McCabe's Cyclomatic Complexity metric ($V(G)$), and Lines of Code (LOC) were computed on all programs used in this study.

Sheppard et al. found modest correlations between performance measures and software metrics in experiments 1 and 2. LOC was found to be the strongest predictor of performance. Larger LOC values degraded performance. The third experiment produced stronger results: E and $V(G)$ were found to correlate significantly (at the .001 level) with performance. The programs used in the first two experiments varied in length from 36 to 57 LOC. Program lengths in the third experiment ranged from 25 to 225 LOC, and Sheppard et al. credit the

stronger results obtained in the third experiment to the greater range of program lengths.

Tanya McGill and Simone Volet studied the relationship between the quality of a student's algorithm development and the quality of the program produced from the algorithm [MV95]. Eight pairs of students in an introductory Computer Science course developed an algorithm for a simple file-processing task, and implemented the algorithm in Pascal. Students were instructed to use one of the design methods described in the course, but they were free to design in any manner. The design method includes program specification; development of a top-level algorithm, structure chart, and detailed algorithm on paper; coding; and testing and debugging. The students' collaboration was videotaped; each algorithm was judged to be a Good Solution, an Incomplete Solution, or No Solution. Programs were judged Good, Satisfactory, or Majorly Flawed / Not Attempted. In general, teams with better designs produced better programs.

1.4 Research Into Factors Affecting Student Performance

Beyond a doubt, undergraduate students must spend time outside of class on studying, homework, and other coursework-related activities. However, there is some question about the relative importance of quantity of study time and quality or characteristics of study time to academic achievement. Since this study is concerned with the relationships between time spent on projects and performance on the project, research on the relationships between quantity and characteristics of study time and student performance, measured by GPA or grade in particular courses, is presented here briefly. The research presented below indicates that quality of study has a greater affect on student grades than quantity of study.

Howard Schuman, Edward Walsh, Camille Olson, and Barbara Etheridge conducted a series of studies designed to test the relationship between student effort and grades [SWOE85]. Student effort was defined as the quantity of time spent studying, as reported by the subjects. Grades considered included both overall Grade Point Average (GPA), and final grade for a particular course.

Schuman et al. conducted four major studies. In their first study, a random sample of undergraduate students was selected for interviewing. The students were asked how much time they "typically" spent studying on weekdays, and on weekends. The reasonability of student responses was indicated in several different ways: the amount of study time reported was highest for Natural Science students (matching the researcher's expectations), study time was significantly correlated to number of credit hours attempted, and detailed interviews of a small subset of the original subjects confirmed the validity of the reported study times. Subjects were also asked what percentage of regularly scheduled classes they attended. Study time was weakly correlated with overall GPA (r = .111). Class attendance was more strongly correlated with GPA (r = .276).

In their next study, Schuman et al. narrowed their focus to a single course, for which studying was considered important. Students in a general organic chemistry course were given a questionaire which asked for study times on the previous day, previous week, and previous weekend, and "typical" time for the entire semester. Students were also asked what percentage of classes they had attended. Once again, class attendance was found to be strongly correlated with final grade (r = .26), while study time showed virtually no correlation with final grade.

A different approach to measuring study time was taken in Schuman's et al. next studies. It was thought that by asking students to provide detailed accounts of all of their activities during the previous week, study time might be measured more accurately. Study time was computed from the subjects' accounts of how their time was spent. For one study, a random sample of undergraduates was selected for a single interview, during which the students were asked to give a detailed "time chart" of activities during the previous day. In another study, a random sample of students was selected; each student was interviewed three times over the semester. This alternate method of collecting study time data did not yield different results — study time showed almost no correlation with GPA.

Lester Hill conducted two studies which replicated Schuman's et al. results [Jr.90]. In one study, students in an introductory Sociology course were given a questionaire asking

about study time and high school GPA. Correlation among these data, final grade in the course, and class attendance were computed. In the other study, a random sample of students from an entire university was selected; subjects in the sample were asked to give the average number of hours spent studying on weekdays and during the weekend. Subjects in this study were also asked to estimate the percentage of scheduled classes they attended. In both studies, absences affected grades significantly, while study time did not, with one exception: study time over weekends was found to correlate strongly and positively with grades.

Donald McCausland and Nancy Stewart tested the relationship between study habits and GPA. First-semester college freshmen were given the Survey of Study Habits and Attitudes (SSHA) citemccausland. The SSHA includes a Study Habits scale, composed of two primary scales: Delay Avoidance and Work Methods. Delay Avoidance and Work Methods scores were significantly correlated with GPA at the .01 level. McCausland and Stewart conclude from this that the way in which a student studies has a major, measurable effect on the student's GPA.

Roy D. Goldman and David J. Hudson conducted a study of the relationships between academic abilities and "strategies" and major field, gender, and GPA [GH73]. A random sample of a freshman undergraduate class was given tests which measured various abilities as well as strategies. Goldman and Hudson found that two strategies in particular showed strong correlation with GPA: Planfulness (concern with punctuality and scheduling) and Formal Thinking (ability to think logically and mathematically).

James R. Leonard conducted a study which differs from studies presented above. His study investigated the effect of different instructional approaches in a first-semester Computer Science course on student performance [Leo91]. Leonard taught one group of students with a "bottom-up" approach to program design, in which individual modules are developed first, then integrated, or synthesized, into a complete program. The other group was taught the "top-down" approach to program design, in which an entire problem is broken into modules, and each module in turn is decomposed. Student performance was measured

with the total number of points accumulated in the course. The students who were taught top-down design performed significantly better than students who were taught bottom-up design. Furthermore, the standard deviation of points earned was smaller for the top-down group, indicating that top-down design may produce more consistant learning.

Chapter 2

Application Frameworks

In the software industry, software engineers are increasingly turning to application frameworks to speed the development process. Commercial application frameworks include Metrowerks PowerPlant for the Macintosh, Borland's Object Windows Library for MS Windows, and IBM's cross-platform Open Class Library. Because of their great importance in the commercial world, students in undergraduate Computer Science curricula must enter the job market prepared to work with application frameworks. In particular, students need to learn to work with systems which exhibit *external control*. Under external control, a programmer's code is not always on the execution path; instead, the system makes calls to the programmer's routines when they are needed. The idea of external control is perhaps the most difficult conceptual barrier to using application frameworks. Also, to use application frameworks with GUI's, students must learn to write programs which do not rely on a console-based user input model, but instead react to user input.

Student programming performance could benefit greatly from using application frameworks. The functionality which is provided in an application framework would save the students many hours of routine drudgery. Furthermore, a framework would provide the students with a ready-made structure for their programs, allowing the students to focus on the particular aspects of their programming tasks. Other benefits of using application frameworks in undergraduate courses include exposing students to GUI design and implementation. Students are highly motivated by the prospect of creating a professional-looking program. By using an application framework, students would get the feel of producing a "product," not just a program (see [Bro75, pp 4-6]).

This labor-saving aspect of application frameworks could benefit course instructors as

well. By giving students an application framework to use for class projects, instructors would be able to give students more complex projects without increasing the time demands on the students. For this reason, it is important to provide students with experience in using application frameworks in introductory courses. The sooner in the curriculum the students begin using frameworks, the more courses will be able to expand their scope because of increased student productivity.

Unfortunately, many commercial application frameworks are not suitable for inclusion in undergraduate Computer Science courses, especially first- and second-year courses. Most commercial frameworks are expensive; some cost thousands of dollars. The most common frameworks require strong C++ skills, which most first- and second-year Computer Science students have not developed. Moreover, almost all commercial systems require days, or weeks, of training before they can be used productively. Most Computer Science courses cannot devote this much class time to teaching an application framework. A particular application framework is often available for only one operating system, thereby confining the students to a single platform. A framework which is suitable for undergraduate Computer Science courses needs to be inexpensive, available in languanges which are taught early in the curriculum, easy to learn, and available for a variety of operating systems.

The GeoSim Interface Library (GIL) is a simple, cross-platform application framework developed at Virginia Tech. GIL was successfully used in Project GeoSim, a series of interactive simulations for introductory geography classes. GIL manages an application's GUI, providing a file format for storing interface specifications as well as a C function library for initializing and interacting with the interface. A programmer introduces functionality into a GIL application by providing *callback functions*, functions GIL calls when the user manipulates an element in the GUI. GIL is distributed freely over the World Wide Web (WWW), and is available for MS-DOS, Macintosh, and several varieties of UNIX / X Window.

Students in Introduction to Data Structures and Software Engineering (CS2) during the Spring, 1996 semester at Virginia Tech used GIL to complete an assignment which required a GUI. CS2 is the second semester of the two-semester sequence of introductory Computer

Science courses at Virginia Tech. CS2 is designed to teach basic data structures and software engineering techniques. It is the first Computer Science course in which students complete a medium-sized programming project. Aside from the inclusion of GIL CS2 in the Spring of 1996 differed from previous offerings in other ways: CS2 was taught in C, instead of Pascal; also, extensive course notes were made available on a WWW site for the first time.

The students' experience with GIL was evaluated in several ways. Students submitted log forms detailing how much time they spent on projects, and how their time was spent. Students also submitted anonymous surveys on which they evaluated GIL documentation and training materials. As they worked on the project using GIL, students sent electronic mail to an account which was set up to handle questions and comments about GIL. A newsgroup local to Virginia Tech was established for students to discuss the course. Finally, the course instructor's experience with this course and previous courses was considered.

2.1 GIL and the Graphical Interface Builder

The GeoSim Interface Library (GIL) is a simple *application framework*. GIL provides the programmer with a variety of standard Graphical User Interface (GUI) elements; a file format for specifying locations, dimensions, and other aspects of these elements; a means of associating interface elements with the programmer's functions; and a graphical environment for drawing. GIL is platform independent. Source code which does not contain platform-specific functionality will compile and run in any GIL-supported environment. GIL is implemented as a library of C functions. Currently, GIL is available for Macintosh, MS-DOS, MS Windows, and several varieties of X Window.

GIL was originally designed to support Project GeoSim, a series of interactive simulations for introductory geography classes. Project GeoSim developers needed an easy-to-use framework which allowed rapid prototyping and frequent changes to the user interface. Additionally, portability to a variety of computing environments was a major concern.

Every application framework is better suited for some tasks than for others. GIL is well-

suited for applications which have relatively static interfaces and simple controls. GIL is not as suitable for many commercial software tasks, including those with multiple document interfaces, and those which use platform-specific capabilities.

GIL is distributed as an archive containing an object code library of GIL functions, header files for GIL functions and data types, and the Graphical Interface Builder, a tool for developing GUI's for GIL. GIL is freely available via the World Wide Web (WWW) from URL `http://geosim.cs.vt.edu/gil.html`. The GIL Programmer's Manual and the Graphical Interface Builder User's Manual are also available from this location. Additionally, the WWW site features notes from two GIL tutorial presentations, and a Frequently Asked Questions (FAQ) page.

2.1.1 GIL Interface Elements

Primarily, GIL manages *interface elements*, items in the user interface which can be viewed and manipulated. GIL is responsible for the initialization and display of interface items. All elements in an interface are declared in an *interface file* (see below). Interface elements in GIL fall into two categories: *control* elements and *static* elements. Control elements respond directly to user actions. GIL control elements include: buttons, menus, drag areas, and scrolling lists. Control elements have programmer defined application functions associated with them; these functions are refered to as *callback functions* and are called by GIL when the user manipulates a control element. Static elements do not respond to user actions, and do not have application functions associated with them. Static elements in GIL include: windows, fields, labels, rectangles, and lines. The GIL interface elements are discussed below.

Windows Windows are named, rectangular regions of the screen within which other elements may be grouped. Windows are displayed as filled rectangles with a variable-width border. All GIL graphics functions implicitly send output to a particular window, the *current* window. Windows can be displayed or hidden at runtime. The interface elements

in a hidden window are not visible, and are not available to the user. Additionally, a window may be either *active* or *inactive*. The interface elements in an inactive window are displayed, but cannot be manipulated by the user. A window can be displayed as a *popup* window. A popup window deactivates all other windows until it is removed, or the other windows are explicitly activated from within the application.

Buttons Buttons are graphical elements which are "pressed" by depressing the mouse button. "Pressing" a button causes a function in the application program to be called. Buttons are drawn with a three-dimensional convex look when they are not pressed, and a concave look when pressed. Buttons can be either repeatable or nonrepeatable.

Menus Menus are lists of functions or program attributes from which a user may select. Items in the menu consist of text strings which describe the action or attribute associated with the item. Active items are available to the user. Inactive items are displayed, but the user cannot select them. Invisible items do not appear in the menu. Typically, a menu is assigned to a button; when the button is pressed, the menu appears.

Fields Fields are rectangular regions in which output is displayed. Fields do not have an inherent appearance or functionality; it is up to the application program to determine how a field will be used.

Drag Areas Drag areas are rectangular regions of the screen within which a user may perform mouse operations such as pointing, dragging, and clicking. Drag areas are displayed as filled rectangles. A drag area may be active, inactive, or invisible. Mouse activity of any kind in a drag area is reported to an application function. The function is notified of both the mouse coordinates and the mouse button state.

Scrolling Lists Scrolling lists provide a way to display a large number of text items (more than will fit in a window), or a number of text items which can change. Users can move

through the list by "dragging" a scroll bar, or clicking **'up'** and **'down'** buttons. Scrolling lists are composed of drag area and button elements.

GIL handles scrolling list display and updating, calling application functions for certain parameters such as the color of an item, or an item's text. The behaviors of the scrolling list – what occurs when the mouse button is pressed on an item, when the mouse moves over an item, or when the mouse leaves the list area – are implemented in application functions, which are called by the GIL list handler.

Labels, Rectangles, and Lines Labels are static text strings which are fixed in a particular position within a window, and are drawn automatically by GIL whenever the window which contains them is drawn. Rectangles and lines are static elements, drawn automatically by GIL whenever the window which contains them is drawn.

Interface Files and the Graphical Interface Builder An interface's color palette and elements are declared in a text file which conforms to the GIL interface file grammar. Interface declarations specify the parameters of an interface element: its dimensions, color, active status (in the case of windows, buttons, and dragareas), etc. When a GIL application starts, it parses its interface file and creates the interface elements. It is possible to generate interface files manually – this is how the interfaces for Project GeoSim were created. However, it is more expedient to create interfaces with the GIL Graphical Interface Builder, a GIL developer's tool, written for use in this study.

The Graphical Interface Builder allows a developer to "draw" interface elements, and see them exactly as they will appear from within the application. Stated differently, the Graphical Interface Builder is a What You See Is What You Get (WYSIWYG) editor for GIL interface files. The Graphical Interface Builder produces GIL interface files, and can modify existing interface files as well. With the Graphical Interface Builder, a developer can study the "look and feel" of an interface as the interface is created, instead of having to run the program and make incremental changes to a text file. Furthermore, the interface

files generated by the Graphical Interface Builder are free from the syntactical errors which inevitably occur in hand-coded text files.

The Visual Editor in the Graphical Interface Builder supports "drawing" interface elements. The Visual Editor contains a floating tool palette. To create an interface element, the user first selects the element type from the palette. Next, the user positions the mouse over one corner of the new element's location, presses the mouse button, move the mouse to the other corner and releases the mouse button. Immediately, the user sees a *property window*, a modal dialog box with fields for setting the new element's properties, such as active status (for windows, buttons, and drag areas) and color. Existing interface elements can be dragged to new locations; also, their properties can be modified in the property window.

The Graphical Interface Builder enables editing various interface parameters. The Graphical Interface Builder also allows the user to edit the *color palette*, the set of colors available for use in interface elements. Only one interface file can be open in the Graphical Interface Builder at any one time; however, the Graphical Interface Builder can edit multiple files during one execution.

2.1.2 The GIL Application Programmer Interface

The GIL programmer's tasks fall into three broad categories: (1) providing callback functions for GIL to call when interface elements are manipulated (callback functions may be quite simple, or may lead to sophisticated subsystems), (2) adjusting the properties of interface elements appropriately, (3) and drawing program-defined graphics in the GIL interface. The GIL object library and header files provide C functions and data types for these purposes.

Additionally, the GIL programmer is responsible for initializing the GIL interface, and starting the GIL *event handler*. The event handler translates user input from the mouse or keyboard into operations on GIL interface elements. GIL adjusts its interface elements appropriately (by changing a button's appearance when the mouse is pressed in the button,

for example), and calls programmer-defined callback functions if necessary. Once it has been started, the event handler typically retains control for the rest of the program execution.

Callback Functions

All control elements in the interface have callback functions associated with them, which are called from the GIL event loop when the the user manipulates the element. Callback functions are supplied by the application programmer, and determine the specific functionality of a control. All control elements specify callback function names in their interface declarations. Function names are bound to actual functions in the program through an array of structures. These structures contain a character string which gives the name of the function as it appears in the interface file, and a pointer to a function in the program to which the name is bound.

Callback functions for buttons accept a single parameter: a pointer to the name of the button which was pressed. Some buttons do not cause a callback function to execute. Instead, they create a popup menu. Each item in the popup menu has a callback which is invoked when the user selects the item. Menu item callbacks accept as parameters the name of the menu and the name of the item.

Mouse activity in a drag area is reported to the drag area callback. A drag area callback accepts three parameters: a drag status parameter and two integer coordinates. The drag status indicates the state of the mouse, including whether the mouse button has been pressed, released, or "clicked" (pressed and released before the mouse location changed).

Scrolling lists require callback functions for returning a pointer to each item's text, and setting each item's text attributes, such as color and size. Additionally, the *in function* is called when the mouse is inside the scrolling list. The *out function* is called once, after the mouse has move outside the scrolling list. The *click function* accepts a slot number as a parameter. It is called when the mouse is clicked on an item in the scrolling list.

In GIL, there are three types of callback functions which are not associated with a control element: *character handlers*, *independent functions*, and *redraw functions*. A character

handler is called any time the user types a character. Independent functions are eligible to be called on each pass through the event loop. The redraw function is called when part of the GIL interface is exposed by another platform level window. The redraw function is responsible for calling the GIL function that redraws the GIL interface, and for redrawing program-maintained graphics. A redraw function is necessary because GIL operates in multi-window environments such as Macintosh and X Window.

Manipulating Interface Elements

Often, an application's interface must change during the application's execution. For example, windows and controls may be displayed, hidden, activated or deactivated in response to user actions. GIL provides functions for adjusting the properties of interface elements during an application's execution.

Windows in GIL can be displayed or hidden using GIL functions. Also, a window's control elements can be temporarily disabled. A window can be displayed as a *popup*. Popup windows function similarly to modal dialogs – control elements in other windows are not available while the popup window is displayed. GIL provides functions for adjusting the properties of individual control elements. Buttons, drag areas, and menu items can be made active, inactive, or invisible. Active control elements can be manipulated by the user. Inactive elements are visible, but cannot be manipulated. Invisible elements are hidden from the user.

In addition to these capabilities, GIL also allows the application to change properties of interface elements, such as their dimensions, location, and text (in the case of buttons). These functions are used infrequently.

Displaying Graphics

In addition to managing interface elements, GIL provides an environment for programmer-defined graphics. The GIL graphical environment, like the rest of GIL, is platform-independent. An application's graphical output is consistant across all GIL-supported platforms.

GIL maintains a graphics state, which includes the current window, current color, text size, and line thickness. Graphical output in GIL is always directed to a particular window, the *current* window. The current window can be set from within the application. Coordinates in drawing commands are relative to the current window. Drawing is "clipped" to the current window; that is, drawing commands do not change the appearance of the area outside of the current window's interior. All drawing is done in the current color (except for pixel vector and image drawing; see below). Text is displayed in the current text size. Drawing commands use the current line thickness for the width of lines and curves. Current color, text size, and line thickness can be set with GIL functions.

Graphics commands are available in GIL for text display; point, line, rectangle, ellipse, and polygon drawing; pixel vector drawing; and image display. Optionally, text can be "clipped" to a rectangular region. Rectangles, ellispes, and polygons can be displayed as outlines, or as "filled" regions. GIL defines a raster image format based on *runlength encoding*. Runlength encoding is a compression technique. GIL images consist of arrays of positive integers stored in either 4, 8, 12, or 16 bits. The GIL image display functions call a programmer-supplied conversion function which accepts as an argument the integer value for a pixel, and returns the color in which the pixel should be drawn.

2.2 Using GIL in CS2

Students in CS2 were given a programming assignment which required them to use GIL. GIL was made available to them from a World Wide Web (WWW) site. Soon after the assignment was given, the author conducted two GIL tutorial presentations during CS2 class time. The first tutorial was an overview of the steps required to produce a simple GIL application; the second tutorial was more specialized, focusing on how to use the scrolling list capabilities in GIL. An electronic mail account was provided, where students could send questions, comments, bug reports, and complaints. Although students uncovered several problems in GIL and the Graphical Interface Builder, most technical problems were minor,

and were fixed promptly. When the assignment was due, students sumitted log forms detailing how their time was spent, survey forms on which the students evaluated their experience with GIL, and electronic copies of their source code.

2.2.1 Assignment

Students in CS2 were required to write a program for managing a record file. Input files could be in one of two formats: a standardized format known as ODB format, and a "raw" text format. Programs were required to read both formats, storing the records in a doubly-linked list. Upon command, student programs were to write the current record file to disk in ODB format. Student programs were required to handle more than one record file during one program run, but only a single file at any one time.

The records used in this assignment were modeled on optical scan test result data. Records contained the following fields: Social Security Number (SSN), Name (including a field for Last Name, and a field for first two initials), Score, T Score, Number of Questions, Group Number, Form Number, Number Omitted, and Seat Number. Student programs permitted adding records, deleting records, and changing the value of any field whose value did not depend on values in other fields. In addition to the test records, the files used in this assignment contained a record specifying data attributes of a test, including Number of Questions, Description, and Date. Student programs enabled editing these fields. Upon command, student programs updated the computed values in the records to reflect changes, and also computed the average score, and standard deviation of the test scores.

Students were given a specific Graphical User Interface (GUI) specification for this assignment. Records were displayed in a GIL scrolling list, and were "selected" for editing by clicking the mouse on the record. File and editing commands were available from GIL menus (activated by GIL buttons). Editing operations were performed in GIL popup windows (which disable other interface elements). Additionally, closing a file, quitting the program, and deleting a student were confirmed using an OK/Cancel dialog box. A Help menu was provided, with an option to display an informational screen, and a minimal help screen.

2.2.2 Tutorials

GIL is distributed with a sample application, Giltest, designed to exhibit the basic features of GIL. The source code and makefile for Giltest is provided in the GIL distribution. However, examining an example application cannot provide the same insights as seeing an application's construction first-hand, and does not permit asking questions of someone who is familiar with GIL. For these reasons, two in-class presentations on how to develop applications with GIL were conducted. The first presentation demonstrated how to create a simple, two-player TicTacToe game using GIL. The second tutorial showed how the scrolling list capabilities in GIL can be used to implement a simple checkbook record-keeping program. Source code, interface files, makefiles, and notes for tutorials were made available from the GIL World Wide Web (WWW) site.

The TicTacToe presentation was designed to illustrate the steps needed to create a basic GIL application which used standard GIL interface elements, including popup menus, windows, buttons, and drag areas (rectangular regions in which mouse operations may be performed). Also, TicTacToe demonstrated how to use some of the graphics capabilities in GIL, such as line drawing and text display.

During this tutorial, a two-player TicTacToe game was developed with GIL. Students were shown every step of the process needed to create the game. First, an interface for the game was constructed with the GIL Graphical Interface Builder. The interface was constructed as the students watched (a portable computer was brought to the class); thus, students were able to see firsthand how to use the Graphical Interface Builder. Also, students were shown "stepwise refinement" of a program, by stepping though a series of source code files, each of which added functionality to the application.

After the interface for TicTacToe had been created, students were shown a minimal source code file, containing only the functions and data items needed to run the TicTacToe application. Particularly, students were asked to note the `FuncNames` table, which is used to associate interface elements with functions in source code. Also, students were shown the

functions for initializing the GIL interface and starting the GIL event handler. The next source file demonstrated basic graphics operations by implementing a board display function and a status message. Board display required using the GIL line-drawing capabilities, and the status message used text-output functions. After this, actual game-playing was implemented. Students were walked through the code for translating a mouse click into a "move" (a board location), validating the move, updating the internal game state, and updating the board and status display. Finally, an "undo" command was added to the application. This demonstrated modification of the original game logic, as well as enabling and disabling a menu item.

Since the interface for the students' project was built around a scrolling list, scrolling list handling was the emphasis of the second tutorial, the Checkbook tutorial. A simple checkbook was implemented, in which users could store, modify, and delete records representing checks. Checks included check number, payee, and amount fields. Checks were displayed in a scrolling list. To add a check to the list, the user presses the **'add'** button. This displays a modal dialog box for setting the check number, payee, and amount fields in the new check. To edit a check, the user selects a check in the list by clicking the mouse on the check, then presses the **'edit'** button. The check dialog box appears with the check's field's current values. Both adding and editing are cancelable. To delete a check, the user selects the check to be deleted, and presses the **'delete'** button. In the check entry dialog box, the payee field was a "type-in" field; keystrokes appeared inside of the field. A type-in text entry field was demonstrated, since GIL does not include text-entry fields as an interface element, and the students neeeded text-entry fields in their programs. Also, this tutorial showed how to implement cancelable dialog boxes, another feature in the students' assignment. Again, students were shown how to use the Graphical Interface Builder to develop a GIL interface, and how to use stepwise refinement to complete a project.

Instead of starting with a new interface, the Checkbook tutorial began with a partial interface. Interface elements required for supporting scrolling lists, modal dialog boxes, and text entry were created in the Graphical Interface Builder. Next, students were shown

a source file which contained as much functionality as possible without implementing the features which were demonstrated. With this source code file as a starting point, the functions to initialize and control a scrolling list were demonstrated.

2.2.3 Help Requests

Since GIL and the Graphical Interface Builder were being used in a class for the first time, and since the class consisted of mostly inexperienced programmers, it was expected that students would benefit from individual help with using GIL. For this reason, an account was created for reporting problems with GIL. Students in CS2 used this account extensively, reporting bugs in GIL and the Graphical Interface Builder (see Section 2.2.4), asking questions pertaining to compiling and linking GIL programs, and occasionally requesting help with programming problems not directly related to GIL. The author answered the electronic mail that came to this account.

Another venue for detecting and responding to students' questions and complaints was the CS2 newgroup, `vt.class.cs1704`. This newsgroup stayed fairly active throughout the semester, with everything from specific questions about GIL and the Graphical Interface Builder to discussions of the quality of GIL and the appropriateness of GIL to CS2.

2.2.4 Bug Fixes and Updates

When the students in CS2 started using GIL, GIL had been in existance for over two years, and had been used extensively in Project GeoSim. Thus, GIL was considered resonably stable and reliable. Also, the GIL Programmer's Manual was in its second revision. However GIL was used in several new ways. GIL was ported to FreeBSD and Linux (two freely available UNIX operating systems for IBM PC-compatable computers) for the CS2 students. Furthermore, students were compiling their GIL programs with C++ compilers, which had not been attempted in the past. Most importantly, the CS2 students were the first large group of GIL users who were relatively inexperienced programmers, and who had not participated in the creation or maintainance of GIL. This combination of circumstances

exposed a few problems and weaknesses in GIL.

The first reported problem with GIL resulted from a portability problem with FreeBSD. In a timing routine, a platform-dependent assumption had been made about the number of clock "ticks" per second. This was fixed by using a platform-defined macro. This problem affected only FreeBSD users. Two incompatabilities with C++ were found and fixed. One problem arose because GIL's object code library was created with a C compiler; when GIL source code was compiled with a C++ compiler, the C++ compiler used mangled names for the functions in the GIL API, which could not be linked to the functions in the GIL object code library. This problem was fixed by placing all GIL prototypes in an `extern "C"` block when compiling with C++. The other problem involved the `bool` type. Originally, the GIL public header defined a `boolean` type as an integer, and defined `bool` as a macro which expanded to `boolean`. This caused problems when headers containing `bool` arguments were included in GIL source files and compiled with a C++ compiler, since the types of some of some function arguments in header files expanded to `int`, when they had previously been typed to `bool`. The solution was to make `bool` a type only when compiling under C, and to make `boolean` a macro which expands to `bool`. Another problem was located by a student who used a call in the GIL API which the GIL programmers tended not to use. This call made unwarrented assumptions about the program interface, which were removed. A problem with scrolling lists was discovered by students using list elements in ways which differed from the practices of GIL developers. A workaround was provided; however, the problem was not fixed until after the assignment was due.

Students reported other problems which were not inherent in GIL, but affected their work. One student reported losing all colors in his interface, seeing nothing but a black region in the GIL application window. The student was running Netscape Navigator, which left no colors available in the display's color palette. This is a standard problem for X Window users, but most of the students had not experienced it before. Another student was unable to link his program with the GIL object code library. This problem arose because the GIL object code library for Linux was compiled on a version different from the one he

was using. An especially vexing problem was attributed to the linker which is distributed with the FreeBSD operating system. The linker will replace standard library functions with programmer-defined functions, with no warnings. Students who wrote functions with the same names as standard library functions (such as `open`) noticed very strange behavior when their functions were called in place of the standard library functions.

The GIL Graphical Interface Builder (GIB) was written specifically for use in this project. The GIB had been completed three months before the CS2 students began GIL development, but efforts to recruit beta testers had been unsuccessful. The GIB developer tested the GIB as much as possible, even using the GIB to construct parts of its own interface. Unfortunately, when the GIB was released it still contained numerous bugs. These problems tended to undermine student confidence in GIL.

The first problem which the CS2 students noticed in the GIB was actually a GIL problem, which was ultimately traced to the timing difficulty noted above. However, this fault was especially vexing for the students because it caused the GIB to quit, destroying work which had been done on the interface. Other "crashes" in the GIB resulted from problems in the GIB itself. One of the GIB's modal dialog boxes enabled other interface elements; this resulted in an attempt to display the same dialog box twice, causing the GIB to exit immediately. Another problem resulted from not properly deallocating lists; eventually, GIL would reach its maximum number of lists, and shut down immediately. After the first two fatal errors were discovered, a **'Save'** button was added to the GIB's Visual Editor, so that users could save recent copies of their work.

Other problems with the GIB did not result in crashes, but confused the user. Drag areas, which support mouse activity, were made inactive by default. Many students were confused when their drag area functions weren't called. The property dialog for windows has a toggle button to set the the window's active status. At times, this button was not set correctly. In the palette editor, new colors did not appear in the palette display. Also, the initial dialog which accepted a file name for the interface would create an interface with no name (if the **'new'** button was pressed when the name field was empty). Most systems will

refuse to write a file whose name has length 0. The GIB did not check for reserved words in interface fields; thus, it was possible to create an interface file which neither GIL nor the GIB could parse.

Problems with GIL were reported to an electronic mail account for that purpose. Some students reported problems to the Project GeoSim account; others sent mail directly to the GIL developement team member. Additionally, the GIL developement team monitored the CS2 newsgroup for discussions of GIL problems. The GIL developement team attempted to address problems as quickly as possible. Fortunately, most problems with GIL and the GIB were simple, and could be fixed by modifying a few lines of source code. Problems were addressed soon after they were brought to the attention of the GIL developement team. In an attempt to remove problems from GIL as quickly as possible, five updates of GIL were released while students were working on their assignment. New versions were announced in the CS2 class, on the WWW sites for GIL and CS2, and on the CS2 newsgroup. New versions of GIL did not require the modification of previously written source code.

2.2.5 Log Forms, Project Survey, and Source Code Submission

To assist in assessing the suitability of GIL for introductory Computer Science classes, students provided the CS2 instructor and the GIL developer with three information sources: a log form indicating how much time was spent on the assignment, and how the time was spent; a survey of the students' opinions about various aspects of GIL, and an electronic copy of the source code and interface file for their programs.

The log forms which the students submitted gave information about how much time was spent on four aspects of the assignment: designing, coding, debugging, and manual/documentation study. Designing meant planning the solution to the assignment, especially paper-and-pencil work such as creating structure charts. Coding meant writing code, and debugging meant correcting compile and runtime errors. Manual study was the time spent reading the GIL and GIB documentation.

Students also submitted surveys which asked about their experience using GIL. Surveys

were anonymous, to encourage honesty. On the surveys, students were asked to rate the GIL manual and the GIB manual on thoroughness and clarity. Also, students rated the tutorials on appropriate complexity ("too simple", "appropriate", and "too complicated" were the possible responses) and helpfulness. Students were asked whether they created new source code files or modified tutorial files. Also, they were asked to estimate the percentage of their programming time they spent on GIL-specific tasks. Space was provided on the survey for any comments students wanted to make.

Archives containing the source code from the students' programs were sent to the GIL email account. Software metrics computed on thes programs, as well as the log form data, were included in the study of student performance.

2.3 Results

2.3.1 GIL Project Surveys

Students in CS2 submitted survey forms on which they evaluated the GIL instructional materials and provided comments on GIL. The GIL Programmer's Manual and the GIL Graphical Interface Builder User's Manual were rated on clarity and thoroughness. The GIL sample program, Giltest, and the two GIL tutorials presented in class, TicTacToe and Checkbook, were rated on appropriate complexity (too simple, appropriate, or too complicated), and on whether they were helpful or not. Additionally, students who had used the CURSES text interface package were asked to compare the usability of GIL to that of CURSES. Finally, students estimated the percentage of their time which was spent on "GIL-specific" coding and debugging. In all, 70 survey forms were collected.

On the survey forms, students rated the clarity and thoroughness of the GIL Programmer's Manual and the GIL Graphical Interface Builder User's Manual on a 1-5 scale, in which 1 meant not clear/not thorough, and 5 meant clear/thorough. The average clarity rating for the GIL Programmer's Manual was 2.78; the average thoroughness rating was 2.46. For the GIL builder User's Manual, the average clarity rating was 3.00 and the average

Table 2.1: Student Assessment of GIL Tutorials

	Too Simple	*Appropriate*	*Too Complicated*	*Helpful*
Giltest	24%	43%	33%	47%
TicTacToe	43%	45%	12%	26%
Checkbook	21%	55%	23%	75%

thoroughness rating was 2.57. Since all ratings fell at the midpoint of the scale or lower, it is clear the students felt that the GIL documentation could be improved.

A sample program, Giltest, is provided with GIL. Additionally, two programs were developed during CS2 class time: TicTacToe and Checkbook. The source code for Giltest consists of a single source file which is 1065 lines long. The tutorials were developed using a series of source files, to illustrate the concept of stepwise refinement. The completed TicTacToe source file was 296 lines, and the final Checkbook source file was 835 lines. Giltest and TicTacToe are designed to illustrate basic GIL concepts such as callbacks for interface elements and interface construction. Checkbook's purpose was to present GIL's scrolling list capabilities.

The students' assessment of the GIL tutorials is given in Table 2.1. Overall, students found the complexity of the tutorials appropriate. The purpose of the TicTacToe tutorial was essentially the same as that of the Giltest example: to introduce GIL programmers to the basic GIL concepts. All of the GIL concepts presented in TicTacToe were also demonstrated in Giltest. This may explain why TicTacToe was considered too simple and unhelpful by many students. The Checkbook tutorial was considered appropriately complex and helpful by the largest number of students. This is not suprising, since the student's assignment was similar in many ways to the Checkbook tutorial. Students were encouraged during the tutorial presentations to modify tutorial source files instead of writing their own source files from scratch; 70% of the students reported modifying tutorial files.

Of the 70 students who submitted surveys, only nine reported having used the CURSES text interface package. Of these, three considered GIL harder to use than CURSES, and six considered GIL and CURSES equally usable. This was disappointing, since GIL was

designed to require as little effort as possible from the programmer.

Students were asked to estimate the percentage of their time which was spent on GIL-specific coding and debugging. On average, students reported spending 69% of their time on GIL tasks. However, the students' assignment required them to add an interface to an existing project; thus, most of the interface-unrelated code had already been written. Students reported that, on average, seven hours of project time was spent reading GIL documentation. Approximately 10%, on average, of the students' project time was spent on documentation.

Space was provided on the survey for comments. Many students felt that more in-class instruction on GIL would have been helpful. Students indicated that the source code for the tutorials was not adequately commented. Several students commented that the idea of using an application framework for the course was good, but that GIL (or their assignment) was too difficult.

It is clear that, overall, the students felt that there was room for improvement in the GIL instructional materials. Although this was disappointing, it suggests that many of the diffulties encountered in using GIL in CS2 could be alleviated with better documetation and tutorials. The students' estimate of the percentage of their time which was spent on tasks directly related to GIL was also higher than had been hoped. On the positive side, students reported spending an average of only 10% of their project time reading documentation, which is low considering that the students had never seen GIL before this project.

2.3.2 Student Mail and Newsgroup Posts

An electronic mail account was provided for students in CS2, to which they could send questions about GIL as well as bug (or suspected bug) reports. This account was monitored by the author. Additionally, a newsgroup local to Virginia Tech was established for students in CS2. The majority of articles posted to this newsgroup were related to GIL. Both the email account and the newsgroup were heavily used: the email account received over 100 messages, and over 250 messages were posted to the newsgroup. The volume of mail

messages and newsgroup posts indicates both significant student involvement with GIL, and a great amount of student stress, brought on in part by the demands of using GIL for their assignments.

Electronic mail messages tended to focus on specific problems. Regretfully, students found a number of bugs in both GIL and the GIB. Initially, student bug reports tended to be vague. To address this problem, instructions for bug reporting were placed on the CS2 WWW site. Specifically, students were asked to include in all bug reports what system they were using, and how to reproduce the bug. This resulted in much more informative bug reports, which in turn allowed more timely updating of the GIB and the GIL library.

Most electronic mail messages were requests for help with GIL programming. Typically, students problems could be easily addressed with a one- or two-sentence explanation. Quick responses to student inquiries seemed to boost student confidence in GIL. Also, problems the students were having with separate compilation of multiple source files and object code libraries were addressed. Usually, the students experienced either multiple definition errors, due to their not declaring multi-file global variables `extern`; or undefined function errors, resulting from not including the GIL object code library when linking their programs. These types of problems were encountered in previous offerings of this course, and are not directly related to the use of GIL.

Only one complaint message came to the GIL account. The author of this message resented relying on the GIB for graded work. The GIB was in an early revision at the time, and had several problems. Another message to the GIL account included a number of suggestions for future versions of GIL, including the much-needed text entry capabilities. Later, this student, on his own initiative, provided a Perl script which allowed GIL to use fixed-sized fonts.

The newsgroup `vt.class.cs1704` is local to Virginia Tech, and is intended for unmoderated discussion of CS2. The author monitored the discussion on the newsgroup to determine which aspects of GIL were causing difficulties for the students. Also, the GIL development team member posted announcements to the newsgroup, and answered students' questions,

especially those relating to the GIL tutorials.

Many of the articles posted to the CS2 newsgroup were requests for help, similar to the requests which came to the electronic mail account. Often, other students answered these questions. The newsgroup also hosted discussions about the appropriateness of GIL in CS2. Most students seemed to favor using an application framework for the class project; one student refered to using GIL for his project as "a taste of real programming." However, students criticized particular aspects of the way GIL was included in CS2. Students felt that GIL should have been more reliable before they were expected to use it; that more classroom GIL instruction should have been given; and that their assignment was too large, and not suitable for GIL.

Judging from the electronic mail messages and newsgroup articles, it appears that the students made the conceptual transitions to external control and console-less programming. Only one article on the newsgroup asked whether the programmer needed directly to call any of the callback functions; this student's question was address by another student. Two articles were posted about character input and output in GIL. One student asked whether there were GIL functions which were equivalent to the standard library functions `printf` and `scanf`. (GIL has a routine to display text in a window; no equivalent to `scanf` is possible, since there is no concept of a console in GIL.)

Both the GIL electronic mail account and the CS2 newsgroup were vital to the success of GIL in CS2. These venues were a vital source of information about GIL, and the students' experience with GIL. Particularly, the students' discussions indicated enthusiasm for using application frameworks, despite criticism of GIL, and high stress levels. Furthermore, the elecronic mail account provided students in CS2 with a convenient way to receive guidance and feedback from the GIL development team. Timely GIL updates and fast responses to student questions greatly eased the students' frustrations and anxieties.

2.3.3 Instructor's Observations

The instructor for CS2 had taught the course which CS2 replaced in previous semesters. Thus, he was in a good position to judge the impact of GIL on the course. Despite the problems encountered, the CS2 instructor felt that GIL was successfully incorporated into CS2, and that the major goals of including GIL in the course were achieved. However, the instructor concluded that CS2 was not the most appropriate place to introduce application frameworks.

The CS2 instructor felt that the students in CS2 made significant progress with the concepts of external control and non-console programming. This agrees with observation of students' electronic mail and newsgroup messages. Furthermore, the instructor's conversations with students indicated that students were excited by the opportunity to use an application framework for their assignments. One student referred to CS2 as, "a chance to write real programs."

Including GIL in CS2 accomplished the goal of allowing larger assignments. Previously, the second-semester introductory course included assignments whose interfaces were implemented with the CURSES text interface library. Students were required to support pull-down menus, selection from a list of items, and simple windowing in CURSES. Student projects in the previous course tended to average over 2000 Lines of Code (LOC), much of which was dedicated to implementing the interface. Comparitively, the GIL assignments averaged 1400 LOC, while incorporating more functionality than previous assignments. The average grade on the assignment was 65%; this is no worse than previous classes.

The CS2 instructor found several factors which made GIL inappropriate for inclusion in CS2. CS2 already has a full syllabus. It is the first course in which students write large programs. Also, CS2 introduces students to abstract data types such as stacks, queues, and linked lists. Dynamic memory allocation is taught in CS2 as well. The number and scope of these topics makes introducing another topic very difficult. Furthermore, students in CS2 have widely varying backgrounds, especially in separate compilation of multiple source files

and object code libraries before using an application framework. Finally, although most students managed to complete their projects, anxiety was significantly greater in this class than in previous, comparable classes.

2.4 Discussion

Despite difficulties, GIL was successful in raising student performance in CS2. Students completed, with less source code, an assignment with GIL which was more complex than assignments given in previous comparable courses. Furthermore, students' spent time on the problem at hand, instead of spending time on tasks such as building an interface with CURSES. Students were exposed to issues related to the use of GUI's and application frameworks. Their anxiety and frustrations notwithstanding, students were enthusiastic about using an application framework. However, application frameworks will not be included in CS2 in the future, due to the number of topics in CS2 and the variance of CS2 student backgrounds.

Many technical problems were encountered during GIL's use in CS2. Despite the fact that it had been in use for over two years, GIL was used in new computing environments (FreeBSD and Linux), with C++ compilers for the first time, and by a completely different user base (first-year students, as opposed to graduate students). These circumstances exposed errors and weaknesses within GIL. The Graphical Interface Builder was relatively new and untested, though not from lack of effort to recruit beta testers. Under these conditions, problems with the Graphical Interface Builder were to be expected.

The semester before CS2 with GIL was taught, students in the class which was replaced by CS2 were recruited to use GIL for their last assignment. Only three students volunteered, and none completed the assignment with GIL. Testing GIL on a group of volunteers before requiring all students to use it was clearly the right course of action; however, incentives to volunteer as a GIL tester apparantly were not strong enough.

Student survey responses indicated that GIL instructional materials could be improved.

One method of locating parts of the documentation which need improvement would be to observe a student, or a pair of students, working on GIL, and noting the difficulties they encounter with the documentation. Overall, students indicated that they benefitted from the tutorial presentations, and felt that there should have been more in-class instruction on GIL and the Graphical Interface Builder.

Some students complained about the complexity of their assignment. It is true that their assignment required more functionality than previous assignments in comparable courses; however, the students did not have the burden of implementing interface elements. On average, the students in CS2 turned in smaller programs than students in previous classes. Overall, the assignment which required GIL was well-suited to GIL. However, one minor incompatability between the assignment's interface specification and GIL's capabilities caused several students some anxiety. Originally, the assignment called for a hierarchical menu. In GIL, only one menu may be active at a single time. This discrepency between assignment requirements and GIL's capabilities caused several students to spend many hours searching the GIL manuals for a way to implement this, and generated several newsgroup articles. After the CS2 professor was alerted to this problem, the hierarchical menu requirement was removed from the assignment. Extreme caution must be taken to ensure that all aspects of an assignment can be completed with the application framework.

There are other, less tractable problems with including GIL in CS2. As noted earlier, CS2 covers a lot of material, even without GIL. CS2 introduces students to various abstract data types, including stacks, queues, and linked lists. Dynamic memory allocation is covered in CS2. Additionally, basic software engineering techniques are taught in CS2, including top-down design and stepwise refinement. Process models, including the Waterfall Model and Spiral Model are introduced. CS2 is the first course in which students complete a medium-sized program (over 1000 LOC). Attempting to include GIL in CS2 adds to an already crowded syllabus.

The strength of the CS2 staff and the GIL development team was their close involvement with the CS2 students. Three primary means of communication between students and staff

were used: the World Wide Web, an electronic mail account, and a newsgroup devoted to CS2. Every effort was made to address students' questions promptly and accurately, and to fix problems with GIL and the Graphical Interface Builder as quickly as possible. Students responded very positively to this intensive involvement with GIL and CS2.

Students bring widely varying backgrounds to CS2. Students from many departments, including Computer Science, Computer Engineering, and Electrical Engineering enroll in CS2. Thus, some students are familiar with writing larger programs and separate compilation; others are not. Finally, although students did reasonably well with GIL, high stress levels among students were noted. These factors convinced the instructor that, before application frameworks could be included in CS2, the CS2 syllabus would have to be reduced, and students would need stronger backgrounds before taking CS2.

Chapter 3

Software Metrics, Work Habits, and Performance

In introductory Computer Science courses, students typically are given rules, guidelines, and methods for designing, writing, and correcting their programs. Some of these rules specify certain stylistic practices, such as subroutine headers and meaningful identifiers. Other rules specify coding practices like accessing local variables and parameters only, and always using structured programming. Compliance with these rules can be checked by inspecting the source code which produced the programs. Often, a portion of the project grade is based on compliance with these rules.

Apart from stylistic and syntactic rules, students are introduced to basic design principles and methods of software engineering. They are told, near the beginning of the course, to design solutions to programming problems before writing code. Usually, students are introduced to design methods such as flowcharts and structure charts. Students are taught *top-down design*, in which they conceptualize the major components of a program, then design each component separately. Related to this is the technique of *stepwise refinement*: writing function prototypes or skeletons before their details are provided [Sta94]. Students are also introduced to software engineering process models, such as the Waterfall Model and Spiral model.

Unlike stylistic and syntactic rules, students are almost never graded on design effort or quality. Instead, the incentive for a student to invest time in the design process is indirect: design time supposedly allows the student to finish projects more quickly (leaving more time for activities other than programming), and leads to a higher grade on the project. In other words, design effort should increase the students' performance. Computer Science professors believe this to be true, even if students often do not. However, research to study

the validity of this belief is lacking.

In introductory Computer Science courses, students are told to write *modular* programs, in which each procedure or function represents an intuitive piece of the entire program, and interacts with other parts of the program as little as possible. In particular, functions should not get too large; 35 to 50 Lines of Code (LOC) is generally accepted as the maximum length of a function. Though it is possible to penalize students for not following modularity guidelines, it is not known whether modularity affects any other measures of performance.

A study was conducted to discern factors which contribute to timely, correct completion of student assignments. In particular, the study focused on whether design effort and modularity was beneficial to student performance. Students in an introductory data structures course submitted source code from their assignments, along with log forms detailing how much time they spent on projects, and how their time was spent. Also, the programs and log forms from the Application Framework study were analysed. The software metrics computed on the students' programs were intended as objective measures of modularity. Students who adhered to acceptable per-function metrics were compared to those who did not. Also, grades on assignments were considered. Two aspects of performance were used in this study: time spent on the project, and grade received on the project. Since the ideal programmer produces correct programs in a timely manner, a third performance measure, the ratio of grade to time, was considered, as well.

The data collected for this study allowed several different hypotheses to be tested. First, it was hypothesized that a relationship could exist between the amount of time spent on a project and the grade received for the project. Variations of this hypothesis allowed for a relationship between the amount of time spent on a particular aspect of the project – designing, coding, or debugging – and project grade. Particularly, it was expected that design effort, measured as the number of hours spent on desing, or the proportion of time spent designing, would be associated with high performance. Also, it was considered that there might be a relationship between the proportion of time spent on one aspect of the project and the grade received for the project, or the total amount of time spent on the

project. For example, a student who spent a considerable amount of time designing a solution to the project might be expected to finish the project in a (relatively) short amount of time, and receive a high grade.

Various hypotheses concerning log data and software metrics were tested. Possibly, students with acceptable per-function software metrics would display higher design times, lower overall times, and a higher grade:time ratio than others. Also, it was thought that programs with high code or structure metrics would possibly receive lower grades, since they probably involved unnecessary effort, and were likely hard to correct. Similarly, it was thought that more time spent designing (or a greater proportion of time spent designing) would result in simpler, more straightforward solutions to the assignments, indicated by lower values for code size and complexity measures. Finally, higher debugging times (or a greater percentage of time spent debugging) were expected for large and/or complicated programs.

Additionally, the data collected for this study allowed previous research to be verified. Software engineering research has found that, while some software metrics correlate highly to others, other software metrics show a low correlation [HKH81]. This result was checked against the data collected for this study. Industrial research has found very large differences in the performance of individual programmers working under similar conditions. This finding was also verified in this study.

3.1 Methodology

To study the relationships between student work habits, grades, and software metrics, log forms and source code were collected from students in CS2604, Introduction to Data Structures. For three assignments in CS2604, students submitted log forms, on which they indicated how much time they spent on three aspects of the project: designing, coding, and debugging. Also, students submitted floppy diskettes containing their source code for metric analysis. Programs and log forms from the Application Framework study were also

analysed. The software metrics for each student, along with the student's log data and grade, were collected in an Excel spreadsheet for analysis.

3.1.1 Assignments

Log forms and source code were collected for three assignments. The first assignment required the students to implement a dynamic memory manager using a "first fit" allocation method. The students wrote programs which read memory allocation commands from a file, parsed these commands, and executed the commands on their memory managers. Commands included:

request ID *size* Request *size* units to be allocated in block ID. The memory manager must allocate space for its information fields, in addition to the *size* units available to the client.

return ID Return the block ID to the memory pool, combining with adjacent free blocks.

list List the contents of the memory pool from beginning to end. For each block, indicate whether the block is free or reserved. The list is written to standard output.

reset Reset the memory pool to a single free block.

The memory pool was an array of 25,000 `unsigned short` values.

This assignment was considered "moderately difficult" by the professor. Programs averaged 489 lines of code (LOC); the average grade on this assignment was 63 out of 75 possible points. The average time spent on this assignment was 22 hours.

For the next assignment, the students implemented a simple record retrieval system. A record was defined as two lines in an ASCII record file; thus, every other newline character marked the end of a record. Initially, the students' programs read the record file, storing the starting position of each record in an array. Next, the programs processed a binary request file. The request file consisted of two-byte unsigned short integer values. For each request in the request file, the students' programs printed the record corresponding to the request. Printing a record required looking up the record's position, seeking to that position in the record file, and reading and printing the record's two lines.

This assignment was considered "easy" by the professor, with the caveat that this was the first assignment in this course which used file input and output. Many students had problems with the mechanics of file access, especially students using MS-DOS, which distinguishes between ASCII and binary files. Programs averaged 213 lines of code (LOC). The average grade on this assignment was high: 73 out of 75 possible points. The average time needed to complete this assignment was 8 hours.

The last assignment required the students to implement a B+ tree of database records. Records consisted of a 4 byte key and a 60 byte data field. The data field contained ASCII characters. Record keys were stored in internal nodes of the B+ tree; record data were stored in leaf nodes. The programs processed a command file, which contained operations to perform on the B+ tree. The commands were:

insert *key data* Insert the record *key* with data *data* into the tree.

remove *key data* Remove the record *key* from the tree.

print *key* [*high_key*] Print the record *key*; or print all records with key values in the range *key* to *high_key*, if the parameter *high_key* is provided.

makenull Remove all records from the database, and initialize the database.

load Read the database file into the tree. The database file contains 64 byte records; the first 4 bytes give the record key, and the last 60 bytes give the record data.

This assignment was considered "difficult" by the professor. Programs averaged 979 lines of code (LOC). The average grade on this assignment was 94 out of 150 possible points. The reported average time spent on this project was 42 hours.

3.1.2 Log Forms and Grades

For each assignment, each student submitted a log form detailing how much time was spent on the project, and how time was spent (see Appendix A). For each day spent on the assignment, students indicated the date, the time they started working, when they finished working, and what activity they were doing. Activities included designing, coding, and debugging. Students were left to interpret for themselves what constituted each activity. When the log forms were submitted, the students totaled their activity times, and indicated

these totals on the log form. The Teaching Assistant wrote each student's grade on the student's log form. Log data was entered by hand into an Excel spreadsheet for comparison with other data.

The accuracy of the log data was a major concern. Little direct validation of the log data could be performed. Interviews with students indicated that log data was entered periodically (anywhere between every two or three days to a week), with some attention to accuracy. The log forms asked for a high level of detail (students indicated how long they spent on each activity, for each day they worked on an assignment). This required more in-depth recall from the students. Other researchers, such as Schuman et al. have obtained reasonably accurate data from students logging the amount of time students spend on various activities [SWOE85].

In general, on each project 20% of the project grade was based on programming and documentation style, and 80% was based on correctness. All of the projects used in this study required the students to implement database functionality. Typically, their programs were required to process various commands on their databases. The correctness grade was divided evenly among the different types of transactions. Implementing a transaction type correctly gave the student full credit for the transaction.

3.1.3 Source Code Collection

In addition to log forms, each student submitted a floppy disk containing source code for the project. Several problems arose from collecting source code from floppy disks. Damaged or unreadable floppies prevented source code collection in some cases. Often, experimentation was the only way to determine a floppy disk's format. Furthermore, students were not given naming conventions for their source files. Source files with ".c", ".cpp", and ".cc" extensions were common, and header files ended in ".h" and ".hpp".

3.1.4 Source Code Analysis

Source code from student projects was analysed with the UX-METRIC 4.0 for OSF/1 analysis tool. UX-METRIC is a product of SET Laboratories, Mulino, OR. UX-METRIC analyses both C and C++ code. Since there were three projects, each with at least 40 submissions, the metric analysis and collection process was automated as much as possible. UX-METRIC does not have a "batch" processing mode; it can only be invoked on a set of source files in a single directory. Thus, it was necessary to write shell scripts to invoke UX-METRIC, and compile the analysis results into a usable format.

First, a list of directories containing source code to be analysed was created. This was done by redirecting the output of the `ls` command to a file. A directory list was stored in each project directory. Next a shell script to invoke UX-METRIC, `metric1.sh`, was run. This shell script ran UX-METRIC on source files in every directory in the directory list. Since UX-METRIC required two auxiliary files to be in the current directory, the `metric1.sh` created soft links to these files before invoking UX-METRIC.

UX-METRIC produces several different output files, including: a human-readable Complexity Report; another version of the Complexity Report, in a machine-friendly format; and a calling information report. The Complexity reports give the following code metrics for each function in the source files: Lines of Code (LOC), Length (N), Volume (V), Effort (E), Cyclomatic Complexity (VG1), and Extended Cyclomatic Complexity (VG2). The calling information report indicates, for each function call in the project, the name of the caller and the name of the called function. This report was used to compute the Henry-Kafura Complexity metric for each function in each project.

After analysing a set of source code, UX-METRIC leaves its output files in the directory containing the source code. Another script, `collect1.sh`, was invoked to collect metric data from each student's directory and compile this data into two files: a Function Data file, containing software metrics on every function in every project; and a Student Data file, with software metrics for each student's entire project. `collect1.sh` invoked a small

program, `studata`, which wrote the data from the Complexity Report and calling information report to the Function Data and Student Data files (computing summary information before writing to Student Data files).

Once the results of source code analysis had been compiled into a single file, they were added to an Excel spreadsheet containing student names, grades and log data. Each student's data occupied one line in the student spreadsheet. Another spreadsheet containing information for each function in each project was compiled, so that software metrics for each function could be compared.

3.1.5 Comparison of Log Data, Grades, and Software Metrics

Several types of comparisons were conducted: software metrics were correlated with other software metrics; log data (including project grades) were correlated with other log data, and software metrics were correlated with log data.

On all projects, some students had to be removed from the database due to insufficient data. Some students had not submitted log forms; others did not have software metric data for a variety of reasons, including bad floppy diskettes and incomplete source files (a few source files appear to have been truncated). Every effort was made to include as many students as possible in all of the tests.

Upper-triangular correlation matrices were computed for metric-to-metric correlations, and log data-to-log data correlations. Also, a correlation matrix was computed for log data-to-metrics correlations. Correlation matrices were produced by writing a small program to print the matrix formulas to standard output. The output from this program was redirected to a text file. Then, the contents of this file were "pasted" into the spreadsheet.

Correlations were computed using Excel's "=correl" command. This command produces the standard correlation coefficient, usually denoted by RHO, and defined as the covariance of two arrays divided by the product of their standard deviations. Correlation coefficients fall in the range [-1, 1]. Correlation coefficients measure the degree of linear dependence which exists between the two arrays.

3.2 Results

All data collected in this study – grades, log data, and software metrics – were brought into Excel spreadsheets for analysis. Pearson correlations were computed for all pairs of observations in this study. Specifically, upper-triangular correlation matrices were computed which correlated log data to other log data and software metrics to other software metrics. Also, correlations between log data and software metrics were computed. Finally, correlations between project grades and both log data and software metrics were computed. Data from each project was considered separately.

A high absolute value of the coefficient of correlation between two sets of data indicates that the data vary linearly with each other, while a low coefficient of correlation shows that the data do not vary linearly with each other. For each correlation a null hyposthesis H_0 and an alternative hypothesis H_α are proposed. The null hypothesis is that the two data sets in question do not show statistically significant correlation. The alternative hypothesis is that the two data sets do exhibit statistically significant correlation. α is the probability that the alternative hypothesis will be accepted when, in fact, the null hypothesis is true. To test the null hypothesis, the test statistic used is $t = r\sqrt{\frac{N-2}{1-r^2}}$, where N is the cardinality of each data set and r is the Pearson correlation coefficient. Under the null hypothesis, t has an approximate T distribution with N degrees of freedom. The null hypothesis is rejected anytime $|t|$ exceeds the value for $t_{\frac{\alpha}{2}}$.

3.2.1 Relationships Among Log Data

Eight log data values were considered in this study. Three of them, design time, coding time, and debugging time, were taken directly from the students' log forms. Four other log data values were computed from the log forms: total time, percentage of time spent designing, percentange of time spent coding, and percentage of time spent debugging. Also, the grade:time ratio was computed from total time and grade.

Sixty-seven log forms were available from the Memory Manager assignment. Sixty-four

forms were obtained from the Record Retrieval assignment, and only fourty-eight were available from the B+ Tree asignment. Sixty-two log forms were obtained from the GUI assignment. Thus, for Memory Manager, Record Retrieval, and GUI data, if a correlation's test statistic is greater than 1.671, the two variables correlated significantly at the .10 level. A test statistic which is greater than 2.000 indicates that the variables correlate significantly with at the .05 level, and a test statistic larger than 2.66 indicates significance at the .01 level. For B+ Tree data, test statistics of 1.684, 2.021, and 2.704 are needed for significance levels .10, .05, and .01, respectively.

It is easy to compute the threshhold for a significant correlation, given the number of observations and the confidence desired. For the Memory Manager, Record Retrieval, and B+ Tree assignments, correlation coefficients which are at least .2249 yield test statistics which are greater than or equal to 1.671, the threshhold for a significance level of .10. Correlations of at least .2722 produce a significance level of .05 on the Memory Manager and Record Retrieval assignments, while correlations at least .3728 give significance levels of .01. B+ Tree data requires correlations of .2840, .3470, and .4881 for significance levels of .90, .95, and .99, respectively.

Tables 3.1, 3.2, 3.3, and 3.4 show the correlation coefficients between log data for each project. Design time correlated significantly to coding time at the .10 level on the Memory Manager assignment, at the .05 level on the Record Retrieval assignment, and at the .01 level on the GUI assignment. Design time did not correlate significantly to coding time on the B+ Tree assignment. Design time correlated significantly to debugging time at the .10 level on the Memory Manager assignment and at the .01 level on the Record Retrieval and B+ Tree assignments. Coding time correlated to debugging time at the .05 level on the Memory Manager and B+ Tree assignments, and at the .01 level on the Record Retrieval assignment. Debugging time did not correlate significantly with either design time or coding time on the GUI assignment. All correlations were positive, suggesting that large amounts of time spent on one aspect of a project (such as designing) will not result in time savings on other aspects. Rather, it seems that a student who spends a lot of time on one part of

Table 3.1: Correlations Between Log Data - Memory Manager

	Code	*Debug*	*Total*	*%Design*	*%Code*	*%Debug*	*G/T*
Design	0.243	0.260	0.579	0.732	−0.390	−0.125	−0.493
Code		0.302	0.734	−0.224	0.400	−0.249	−0.566
Debug			0.804	−0.227	−0.573	0.750	−0.615
Total				−0.012	−0.258	0.273	−0.783
%Design					−0.384	−0.322	−0.084
%Code						−0.750	0.291
%Debug							−0.238

Table 3.2: Correlations Between Log Data - Record Retrieval

	Code	*Debug*	*Total*	*%Design*	*%Code*	*%Debug*	*G/T*
Design	0.286	0.386	0.608	0.685	−0.441	−0.005	−0.546
Code		0.380	0.797	−0.189	0.271	−0.154	−0.600
Debug			0.817	−0.129	−0.562	0.673	−0.664
Total				0.009	−0.262	0.267	−0.801
%Design					−0.386	−0.275	−0.090
%Code						−0.780	0.320
%Debug							−0.272

Table 3.3: Correlations Between Log Data - B+ Tree

	Code	*Debug*	*Total*	*%Design*	*%Code*	*%Debug*	*G/T*
Design	0.223	0.637	0.766	0.615	−0.501	0.074	−0.503
Code		0.372	0.686	−0.387	0.496	−0.270	−0.504
Debug			0.881	0.004	−0.476	0.584	−0.449
Total				0.056	−0.206	0.204	−0.615
%Design					−0.598	−0.147	−0.186
%Code						−0.705	0.099
%Debug							0.043

a project will likely spend more time on other parts, as well.

For each student, the percentage of time spent on each aspect of the project as computed as follows: $p = \frac{tasktime}{totaltime}$, where $tasktime$ is the amount of time spent on a particular aspect of the project (designing, coding, debugging), and total time is the amount of time spent on the entire project. Virtually no correlation was found between percentage of time designing and total time. Furthermore, percentage of time designing showed no significant correlation with the grade:time ratio, indicating that spending a greater portion of project time on designing will neither reduce the total time spent on the project, nor enable one to work more efficiently (when "efficiency" is indicated by the grade:time ratio).

On all of the assignments, percentage of time coding correlated negatively with total time on the project. On the Memory Manager and Record Retrieval assignments, the correlations were significant at the .10 level; on the B+ Tree and GUI assignments, the correlations were not statistically significant. All correlations were negative; thus, students who spent a greater portion of their time coding tended to spend less time overall on the projects. Correlation between percentage of time spent debugging and total time were positive on all projects. The correlation was statistically significant at the .05 level on the Memory Manager assignment and the .10 level on the Record Retrieval assignment. The correlations were not statistically significant on the B+ Tree and GUI assignments.

Correlations computed between project grades and the log data described above are shown in Table 3.5. Two statistically significant correlations were found. On the Memory Manager assignment, grade correlated to percentage of time spent coding at the .05 level. On the B+ Tree assignment, grade correlated to percent time debugging at the .05 level. No other log data showed high correlation with project grade. Thus, it is doubtful that any implications can be drawn from the two statistically significant correlations.

Three measures of performance were considered in this study: project grade, time spent on project, and grade:time ratio. Using these three performance measures, performance variations as high as 18:1 were found. These variations are similar in magnitude to those noted in other studies. The performance data from each project are summarized in Ta-

Table 3.4: Correlations Between Log Data - GUI

	Code	*Debug*	*Total*	*%Design*	*%Code*	*%Debug*	*G/T*
Design	0.459	0.170	0.591	0.746	−0.106	−0.167	−0.397
Code		0.087	0.801	−0.058	0.504	−0.473	−0.563
Debug			0.642	−0.218	−0.728	0.792	−0.581
Total				−0.015	−0.087	0.090	−0.771
%Design					−0.130	−0.235	0.078
%Code						−0.933	0.198
%Debug							−0.223

Table 3.5: Correlations Between Log Data and Grades (all projects)

Project	*Design*	*Code*	*Debug*	*Total*	*%Design*	*%Code*	*%Debug*	*G/T*
MemoryManager	−0.185	0.078	−0.100	−0.078	−0.156	0.283	−0.178	0.322
RecordRetrieval	−0.090	−0.114	−0.202	−0.188	0.037	0.025	−0.051	0.193
B + Tree	−0.032	0.094	0.235	0.146	−0.328	−0.027	0.324	0.496
GUI	0.031	−0.006	−0.162	−0.087	0.055	0.133	−0.151	0.463

Table 3.6: Performance Differences

	Grade			*Time*			*G/T*		
Project	*Max*	*Min*	*Ratio*	*Max*	*Min*	*Ratio*	*Max*	*Min*	*Ratio*
MemoryManager	75	39	1.92	60	5.25	11.43	13.33	1	13.33
RecordRetrieval	75	42	1.79	23.25	2	11.63	37.5	2.63	14.26
B + Tree	150	40	3.75	161	9	17.89	6	.40	15
GUI	100	48	2.08	188	24	7.83	3.04	.27	11.26

ble 3.6.

3.2.2 Relationships Among Software Metrics

Previous research has found high correlations among code metrics, and substantially lower correlations between code metrics and the Henry-Kafura Information Flow metric [HKH81]. Data collected from this study, shown in Tables 3.7, 3.8, 3.9, and 3.10 confirm these previous results. Code, structure, and hybrid metrics were computed on 636 functions from the Memory Manager assignment, 280 functions from the Record Retrieval assignment, 1059 functions from the B+ Tree assignment, and 2817 functions from the GUI assignment. Since these functions were written by many different students (between 44 and 66, depending on the assignment), they cover a wide range of programming styles. Coefficients of correlation were significantly higher between code metrics than between code metrics and the hybrid metric HC_F. Furthermore, there is almost no correlation between code metrics and the structure metric C_F.

The correlations between the code metrics Length, Effort, and Cyclomatic Complexity and the hybrid metric HC_F are statistically significant at the .05 level (at least) on all projects. However, this does not mean that the code metrics and hybrid metric are in any sense interchangable. No coefficient of correlation between the hybrid metric and a code metric was above .5; thus, less than half of the variance in the hybrid metric is due to factors which determine the code metrics.

3.2.3 Relationships Between Log Data and Metrics

Log data and software metrics were collected from 66 students on the Memory Manager assignment, from 61 students on the Record Retrieval assignment, 37 students on the B+ Tree assignment, and only 24 on the GUI assignment. The drop in the number of students included is due to the need for both log data and software metric data. Students participating in the log data and metrics comparisons represent the intersection of students who provided log data with students who successfully submitted source code. Correlation

Table 3.7: Correlations Between Software Metrics - Memory Manager

	$\hat{N}$	$Volume$	$Effort$	$VG1$	$VG2$	LOC	C_F	HC_F
N	0.645	0.996	0.861	0.755	0.762	0.857	0.065	0.378
$\hat{N}$		0.687	0.349	0.768	0.699	0.677	0.127	0.288
$Volume$			0.853	0.780	0.784	0.865	0.070	0.383
$Effort$				0.581	0.624	0.693	0.031	0.341
$VG1$					0.940	0.802	0.182	0.383
$VG2$						0.767	0.165	0.404
LOC							0.086	0.350
C_F								0.799

Table 3.8: Correlations Between Software Metrics - Record Retrieval

	$\hat{N}$	$Volume$	$Effort$	$VG1$	$VG2$	LOC	C_F	HC_F
N	0.831	0.994	0.907	0.869	0.880	0.872	0.107	0.211
$\hat{N}$		0.869	0.604	0.788	0.764	0.761	0.024	0.103
$Volume$			0.894	0.877	0.878	0.862	0.094	0.197
$Effort$				0.739	0.755	0.729	0.086	0.176
$VG1$					0.986	0.819	0.136	0.227
$VG2$						0.833	0.136	0.230
LOC							0.119	0.215
C_F								0.934

Table 3.9: Correlations Between Software Metrics - B+ Tree

	$\hat{N}$	$Volume$	$Effort$	$VG1$	$VG2$	LOC	C_F	HC_F
N	0.771	0.998	0.856	0.883	0.861	0.907	0.208	0.358
$\hat{N}$		0.770	0.425	0.818	0.798	0.842	0.235	0.282
$Volume$			0.861	0.886	0.865	0.912	0.21	0.365
$Effort$				0.681	0.652	0.671	0.131	0.293
$VG1$					0.982	0.909	0.187	0.276
$VG2$						0.889	0.18	0.261
LOC							0.244	0.34
C_F								0.837

matrices from the assignments are shown in Tables 3.11, 3.12, 3.13, and 3.14.

Primarily, software metrics were collected to test the effects of modularity on student performance. The percentage of functions over the LOC threshold of 35 was used as the measure of how much each student deviated from good modularity. The percentage of functions over baseline did not correlate significantly to grade or time on any project. On the Record Retrieval assignment, percentage over baseline correlated positively with the grade:time ratio; it did not correlate significantly to the grade:time ratio on other assignments. Also, percentage over baseline did not correlate significantly to design time or debug time on any assignment.

It was hypothesized that there could be a significant, negative correlation between design time and complexity, as measured by code metrics or the hybrid metric. However, no statisticly significant correlations were found between design time and any software metric. Furthermore, correlations between design times and software metrics, though small, were positive on the Record Retrieval, B+ Tree, and GUI assignments, indicating that students who spent longer designing wrote larger programs. No statistically significant correlation was found between percentage of time designing and any metric. On the B+ Tree assignment, correlations between percentage of time designing and software metrics tended to be negative. This is most likely due to the high correlation (on this project) between time spent coding and complexity. Students who spent a lot of time coding on this project tended to produce more code, and a student who spent a great deal of time coding probably had a small percentage of time designing.

A common industrial measure of programmer productivity is the ratio of amount of code produced to the time taken to produce it. In light of this, it was surprising that correlations between coding time and code metrics, as well as total time and code metrics, were not stronger. On the Memory Manager and GUI assginments, no significant correlations were found between coding time, total time, or percentage of time coding and any code metrics. This held for the Record Retrieval assignment as well, with the exception of the correlation between total time and $\hat{N}$, and between total time and LOC. Both of these correlations

Table 3.10: Correlations Between Software Metrics - GUI

	$\hat{N}$	$Volume$	$Effort$	$VG1$	$VG2$	LOC	C_F	HC_F
N	0.908	0.993	0.866	0.767	0.749	0.886	0.058	0.265
$\hat{N}$		0.898	0.690	0.681	0.659	0.865	0.059	0.220
$Volume$			0.897	0.779	0.758	0.876	0.060	0.299
$Effort$				0.777	0.768	0.750	0.030	0.221
$VG1$					0.984	0.753	0.084	0.213
$VG2$						0.728	0.074	0.189
LOC							0.062	0.232
C_F								0.528

Table 3.11: Correlations Between Log Data and Software Metrics - Memory Manager

	N	$\hat{N}$	$Volume$	$Effort$	$VG1$	$VG2$	LOC	C_F	HC_F
$Design$	−0.163	−0.106	−0.185	−0.183	−0.067	−0.105	0.018	−0.123	−0.158
$Code$	−0.052	−0.128	−0.037	−0.033	0.032	0.090	0.121	−0.039	−0.079
$Debug$	0.062	−0.092	0.089	0.128	−0.013	0.016	0.080	−0.039	−0.048
$Total$	−0.040	−0.149	−0.024	0.002	−0.012	0.021	0.111	−0.080	−0.116
$\%Design$	−0.064	0.039	−0.096	−0.105	−0.009	−0.079	0.033	−0.135	−0.136
$\%Code$	−0.144	−0.078	−0.145	−0.154	−0.038	−0.008	−0.063	0.009	−0.016
$\%Debug$	0.195	0.053	0.218	0.235	0.046	0.064	0.042	0.087	0.114
$Gr/Time$	0.193	0.197	0.186	0.152	0.146	0.122	0.095	0.079	0.136

Table 3.12: Correlations Between Log Data and Software Metrics - Record Retrieval

	N	$\hat{N}$	$Volume$	$Effort$	$VG1$	$VG2$	LOC	C_F	HC_F
$Design$	0.179	0.225	0.180	0.147	0.229	0.223	0.197	0.115	0.142
$Code$	0.073	0.125	0.063	0.012	0.111	0.103	0.101	0.052	0.051
$Debug$	0.179	0.201	0.167	0.129	0.211	0.215	0.238	0.174	0.184
$Total$	0.179	0.231	0.168	0.113	0.229	0.226	0.230	0.149	0.161
$\%Design$	0.083	0.113	0.092	0.076	0.095	0.089	0.072	−0.008	0.023
$\%Code$	−0.100	−0.107	−0.099	−0.096	−0.103	−0.109	−0.136	−0.094	−0.100
$\%Debug$	0.045	0.031	0.038	0.047	0.040	0.050	0.089	0.103	0.087
$Gr/Time$	−0.122	−0.162	−0.104	−0.058	−0.184	−0.182	−0.183	−0.144	−0.122

Table 3.13: Correlations Between Log Data and Software Metrics - B+ Tree

	N	$\hat{N}$	$Volume$	$Effort$	$VG1$	$VG2$	LOC	C_F	HC_F
$Design$	0.130	0.010	0.116	0.040	0.019	0.015	0.137	−0.004	0.026
$Code$	0.363	0.309	0.341	0.230	0.314	0.321	0.425	0.148	0.204
$Debug$	0.304	0.064	0.316	0.261	0.092	0.085	0.117	−0.026	0.060
$Total$	0.338	0.157	0.331	0.235	0.175	0.173	0.269	0.043	0.119
$\%Design$	−0.270	−0.273	−0.278	−0.243	−0.280	−0.291	−0.183	−0.095	−0.145
$\%Code$	0.078	0.234	0.050	−0.049	0.205	0.214	0.233	0.104	0.120
$\%Debug$	0.113	−0.073	0.153	0.246	−0.033	−0.036	−0.141	−0.053	−0.034
$Gr/Time$	0.202	0.262	0.215	0.133	0.284	0.290	0.164	0.005	0.042

Table 3.14: Correlations Between Log Data and Software Metrics - GUI

	N	$\hat{N}$	$Volume$	$Effort$	$VG1$	$VG2$	LOC	C_F	HC_F
$Design$	0.308	0.324	0.3	0.195	0.297	0.264	0.198	0.24	−0.06
$Code$	−0.23	−0.18	−0.27	−0.26	−0.18	−0.21	−0.27	0.056	−0.15
$Debug$	0.146	0.208	0.109	−0.08	0.073	0.077	0.199	−0.14	−0.3
$Total$	−0.01	0.062	−0.06	−0.18	−0.02	−0.04	−0.03	−0.01	−0.28
$\%Design$	0.434	0.355	0.472	0.502	0.42	0.402	0.293	0.194	0.129
$\%Code$	−0.35	−0.37	−0.35	−0.18	−0.26	−0.28	−0.36	0.092	0.21
$\%Debug$	0.244	0.28	0.228	0.057	0.153	0.178	0.283	−0.14	−0.24
$Gr/Time$	−0.13	−0.18	−0.08	0.069	−0.07	−0.08	−0.02	0.128	0.409

Table 3.15: Correlations Between Grades and Software Metrics

$Project$	N	$\hat{N}$	$Volume$	$Effort$	$VG1$	$VG2$	LOC	C_F	HC_F
$MemoryManager$	0.200	0.218	0.224	0.103	0.175	0.171	0.184	0.015	0.048
$RecordRetrieval$	0.095	0.166	0.104	0.077	0.108	0.102	0.087	0.041	0.041
$B+Tree$	0.683	0.655	0.677	0.477	0.665	0.664	0.580	0.160	0.268
GUI	0.070	0.071	0.102	0.153	−0.062	−0.066	0.186	0.120	0.104

were significant at the .10 level. However, due to the number of correlations computed, these two significant correlations can be attributed to chance.

On the B+ Tree assignment, coding time correlated to Length (N) and LOC at the .05 level. Also, coding time correlated to Predicted Length ($\hat{N}$), Volume (V), Effort (E), and the McCabe Cyclomatic Complexity metrics ($VG1, VG2$) at the .10 level. Total time showed significant correlation with Length and Volume, at the .10 level. No significant correlations were found between percent time coding and any software metric. Also, no significant correlation was found between the grade:time ratio and any software metric (with a single, most likely random, exception).

Debugging time and percentage of time spent debugging showed few significant correlations to software metrics, except on the GUI assignment. Percentage of time spent debugging correlated significantly to Effort on the Memory Manager assignment at the .10 level. Debugging time correlated significantly to LOC on the Record Retrieval assignment at the .10 level. On the B+ Tree assignment, debugging time correlated significantly with Length at the .05 level, and with Volume at the .10 level. On the GUI assignment, percentage of time spent debugging correlated significantly with Length, Volume, Effort, and Cyclomatic Complexity at the .05 level.

Coefficients of correlation between project grades and software metrics are shown in Table 3.15. The difference in correlations between grades on the Memory Manager, Record Retrieval, and GUI assignments and the B+ Tree assignment is striking. On the Memory Manager and Record Retrieval assignments, no significant correlation exists between grade and any software metric. On the B+ Tree assignment, however, correlations between grade and all code metrics considered in this study were significant at the .01 level (except for

Effort, with which grade correlated significantly at the .05 level). The B+ Tree assignment was much more difficult than the Memory Manager and Record Retrieval assignments, and many students did not complete the B+ Tree assignment, as is indicated by their grades. Students who completed more of the project were likely to get higher grades; also, completeing more of the project increased their code metric values. The size of the B+ Tree assignment alone does not explain the high positive correlations between code metrics and project grade, since the GUI programs were, on average, larger than the B+ Tree programs.

3.3 Conclusions

Almost universally, students are taught certain approaches to solving programming problems, which are thought to be superior to the "code and fix" approach. Particularly, they are exposed to top-down design with stepwise refinement, a design method appropriate to small and medium sized programs. Students are advised to write "modular" programs, in which functions are reasonably short (35 lines is a common maximum function length). Usually, students are not graded on whether they adhere to this, or any, design method. Instead, it is expected that design effort and quality will indirectly benefit students, with better (more accurate) programs and reduced time spent on projects.

A study of student programmers in an introductory data structures course and an introductory Computer Science course was conducted, in an attempt to find factors which predict or contribute to student performance. Of particular interest were design effort and program modularity. Three performance measures were considered: project grade, time spent on project, and grade:time ratio. Students submitted log forms, detailing how much time they spent on particular aspects of the project: designing, coding, and debugging. Additionally, the students' source code for their project solutions was collected. Software metrics, including code metrics, structure metrics and hybrid metrics, were computed on the students' source code. Correlations were calculated among the log data, grade, and software metric data, in the effort to find predictors of student performance.

No factors which contribute to student programming performance were found in the log data or software metric data. However, wide ranges of performance levels were noted among students, agreeing with previous work regarding large variations in performance of individual programmers. Furthermore, previous studies on the relationships among software metrics was verified. This study, like a previous study, found that code metrics are very strongly correlated to each other, while hybrid metrics are not as strongly correlated to code metrics.

Chapter 4

Conclusions and Future Research

Computer Science students are exposed to tools and practices which are believed to enhance the performance of professional software engineers. It is believed that, by employing these tools and practices, the performance of student programmers (their ability to produce high-quality software in a timely manner) will benefit. This thesis has described two studies in which factors related to student programming performance were studied.

The purpose of the Application Framework study was to determine whether Computer Science students in introductory-level courses would be more productive programmers when using an application framework. Students in a second-semester introductory Computer Science course (CS2) were provided with a simple application framework, which they used to complete an assignment. The students using the framework produced a program which was more functional than those in previous offerings of the course. Additionally, it was clear from interactions with the students that they gained an understanding of the concepts of external control and console-less user interfaces.

The Software Metric study was designed to test whether design effort, program modularity, and program simplicity resulted in measurable performance gains. Students in a Data Structures course submitted log forms, on which they indicated how much time they spent on three portions of their programming projects: designing, coding, and debugging. Additionally, software metrics were computed on the students' programs. Two measures of design effort were used: number of hours spent designing, and percentage of the project time spent designing. Modularity was measured by the percentage of functions in a given assignment exceeding a threshold for the Lines of Code (LOC) metric. Performance was measured in three ways: project grade, amount of time spent on the project, and the ratio

between grade and time. The software metrics were used as measures of program complexity. None of the design effort, modularity, and complexity measures were found to correlate significantly with any performance measure. Furthermore, no significant indicators of student performance were found, other than the unsuprising result that, on large projects, students who complete more of the project tend to receive higher grades. However, results from previous research on the relationships among software metrics [HKH81] and on the variations in performance among individual programmers [SEG68], [Cur81] were obtained in this study.

Many valuable lessons were learned from incorporating GIL into CS2. When application frameworks are included in Computer Science courses, they must be thoroughly reliable. Problems within the framework or its tools will reduce student confidence in the framework. Also, documentation and instructional materials must be complete and accessible to the students. Assignments must be carefully designed to fit the capabilities of the framework. In-class presentations on how to use the framework can benefit the students, especially when the programs developed in these presentations are similar to the students' assignment. Most importantly, the course syllabus must permit the coverage of topics such as event-driven, reactive programming and external control.

A more thorough investigation of the performance benefits of application frameworks would be constructed as follows: Two groups of students would solve identical, or similar, programming problems. The experimental group would use the framwork, and the control group would use a lower-level set of tools. The performance measures of the two groups, such as time required to complete projects or number of defects found in projects, could be compared. This type of investigation could be carried out in two consecutive offerings of a course, or in two concurrent offerings of the same course.

The Software Metrics study did not find any significant effect of design effort or modularity on student performance. It is possible that designing a solution to a programming problem before writing the program is of no benefit. More plausibly, the conditions in this study were such that the benefits of design effort, design quality, and modularity were not

revealed.

In the Software Metrics study, design effort, as measured by reported hours spent designing, was considered. Design quality was not considered, since it would be much harder to measure. Studies of student performance have found that study time shows little or no relationship with college grades, while use of better study habits shows significant correlation with grades. It seems reasonable that a higher quality design would affect performance measures. However, assessing design quality would be a subjective process.

Inaccuracies in the reported times is also a possibility. Unfortunately, there are few ways to improve data collection which don't intrude on the students' work, and thereby distort the data with artificial circumstances. If students were provided with an automated tool for design, then design time could be monitored fairly accurately. The drawback of this approach is that students would be restricted in their design approach, and would not be typical of students outside of the study. Coding and debugging time could be automatically recorded by a specially-designed Integrated Development Environment (IDE). Such monitoring would be reasonably inobtrusive.

The nature of the assignments may also explain why design effort was not significantly related to performance measures. The assignments used in this study were small, single-purpose programs. While the largest asignment averaged close to 1000 LOC, assignments in upper level courses tend to be much larger. For example, an assignment from an operating systems course can approach 5000 LOC.

Apart from their size, two other factors differentiate the assignments used in the Software Metrics study from upper-level undergraduate projects and industrial software development. First, students worked alone on these projects. Second, these projects were not meant to be maintained or modified. Later in the curriculum, students are required to modify previous projects, and work in teams. It is likely that both design effort and design quality would greatly affect overall performance in these situations. If so, then it makes sense to teach good design methods in introductory courses, even though students in these courses can get by without good design skills.

Future studies of the benefits of design effort, design quality, and modularity should take place in courses in which there are large projects (at least 2500 LOC) assigned to teams of students, which require maintainance and modification over the semester. There is no convenient alternative to programmer log forms for measuring the amount of time spent on aspects of the project, although log forms could be collected weekly, or at each class meeting, with the hope that students would be able to recall amounts of time with greater accuracy. Calvin Selig and Sallie Henry have proposed a way to quantitatively assess design quality [SH90]; design quality metrics could supplement the design effort measure. Finally, source code should be collected electronicly for a software metric analysis of its modularity. Positive results from such a study would validate the belief that good design is well worth the effort.

Appendix A

Student Log Form

Name: ID#:
Computer System: Compiler:
Class: Project #:
bytes in final source files:

Did you use a debugger (Y/N)? _____ If yes, which one? _____

Below, list date and start/stop times for each work session.
Under "Activity", mark D for Design, C for Coding or B for deBugging.
List all activities appropriate for that work session.

Date	Start	Stop	Activity
			Total time for
			(D) Design: _____
			(C) Coding: _____
			(B) deBugging: _____
			Total: _____
			Total running time
			for the program: _____

REFERENCES

[BF88] Giora Baram and Frank F. Friedman. Should Commercial Software Packages Be Used in the Classroom? *SIGCSE Bulletin*, 20(3), 1988.

[Bro75] Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1975.

[Chr78] Earl Chrysler. Some Basic Determinants of Computer Programming Productivity. *Communications of the ACM*, 21(6), June 1978.

[CSM79a] Bill Curtis, Sylvia B. Sheppard, and Phil Milliman. Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics. *Proceedings of the Fourth International Conference on Software Engineering*, 1979.

[CSM+79b] Bill Curtis, Sylvia B. Sheppard, Phil Milliman, M. A. Borst, and Tom Love. Measuring the Psychological Complexity of Software Maintainance Tasks with the Halstead and McCabe Metrics. *IEEE Transactions on Software Engineering*, SE-5(2), 1979.

[Cur81] Bill Curtis. Substantiating Programmer Variability. *Proceedings of the IEEE*, 69(7), 1981.

[Dic81] Thomas E. Dickey. Programmer Variability. *Proceedings of the IEEE*, 69(7), 1981.

[GH73] Roy D. Goldman and David J. Hudson. A Multivariate Analysis of Academic Abilities and Strategies for Successful and Unsuccessful College Students in Different Major Fields. *Journal of Educational Psychology*, 65(3), 1973.

[GPJ95] Koldo Gojenola, Tomás A. Pérez, and Arturo Jaime. Integrative Assignments for CS1 and CS2 through Libraries of Abstract Data Types. *SIGCSE Bulletin*, 27(4), 1995.

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Elsevier Scientific Publishing Company, New York, New York, 1977.

[HK81] Sallie Henry and Dennis Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, SE-7(5), September 1981.

[HKH81] Sallie Henry, Dennis Kafura, and Kathy Harris. On the Relationships Among Three Software Metrics. *Performance Evaluation Review*, 10(1), Spring 1981.

[Jr.90] Lester Hill Jr. Effort and Reward in College: A Replication of Some Puzzling Findings. *Journal of Social Behavior and Personality*, 5(4), 1990.

[Leo91] James R. Leonard. Using a Software Engineering Approach to CS1: A Comparative Study of Student Performance. *SIGCSE Bulletin*, 23(4), 1991.

[Mar96] Dave Mark. From the Factory Floor (Interview with Greg Dow). *MacTech Magazine*, 12(4), 1996.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(1), December 1976.

[MV95] Tanya McGill and Simone Volet. An Investigation of the Relationship between Student Algorithm Quality and Program Quality. *SIGCSE Bulletin*, 27(2), 1995.

[Mye77] Glenford J. Myers. An Extension to the Cyclomatic Measure of Program Complexity. *ACM SIGPLAN Notices*, 12(10), October 1977.

[PCD92] Randy Pausch, Matthew Conway, and Robert DeLine. Lessons Learned from SUIT, the Simple User Interface Toolkit. *ACM Transactions on Information Systems*, 10(4), 1992.

[Rob95] Eric S. Roberts. A C-Based Graphics Library for CS1. *SIGCSE Bulletin*, 27(1), 1995.

[SCML79] Sylvia B. Sheppard, Bill Curtis, Phil Milliman, and Tom Love. Modern Coding Practices and Programmer Performance. *Computer*, 12(12), December 1979.

[SEG68] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory Experimental Studies Comparing Online and Offline Programming Performance. *Communications of the ACM*, 11(1), January 1968.

[SH90] Calvin Selig and Sallie Henry. Predicting source-code complexity at the design stage. *IEEE Software*, 7(2), 1990.

[Sta94] Thomas A. Standish. *Data Structures, Algorithms, and Software Principles*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[SWOE85] Howard Schuman, Edward Walsh, Camille Olson, and Barbara Etheridge. Effort and Reward: The Assumption that College Grades are Affected by Quantity of Study. *Social Forces*, 63(4), June 1985.

[Szu96] Laszlo Szuecs. Creating Windows Application Using Borland's OWL Classes. *SIGSCE Bulletin*, 28(1), 1996.

[Wer90] Laurie Honour Werth. Graphical Interfaces as Software Engineering Projects. *SIGSCE Bulletin*, 22(1), 1990.

[WF77] C. E. Walston and C. P. Felix. A Method of Programming Measurement and Estimation. *IBM Systems Journal*, 16(1), January 1977.

VITA

John Burl Raley

John Raley was born in Atlanta, GA. He graduated from Mars Hill College, Mars Hill, NC, with a BS degree in Mathematics. Currently, he is living in San Francisco, CA with his wife, Sherry Raley, and working for Taligent, an IBM subsidiary.