

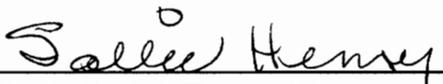
**MEASUREMENT OF THE EFFECTS OF REUSING
C++ CLASSES ON OBJECT-ORIENTED
SOFTWARE DEVELOPMENT**

by
Mark Richard Lattanzi

**Dissertation submitted to the Faculty of Virginia Tech in partial
fulfillment of the requirements for the degree of**

**DOCTOR OF PHILOSOPHY
in
Computer Science**

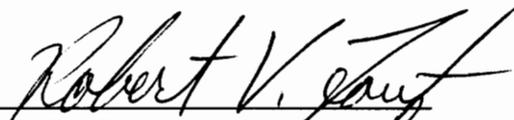
APPROVED:



Dr. Sallie M. Henry, Chairperson



Dr. Osman Balci



Dr. Robert V. Foutz



Dr. Dennis G. Kafura



Dr. Robert C. Williges

**March 1995
Blacksburg, Virginia**

Keywords: Software Reuse, Software Metrics, Object-oriented, C++ Classes

C.2

LD

5655

V856

1995

L368

C.2

**MEASUREMENT OF THE EFFECTS OF REUSING
C++ CLASSES
ON
OBJECT-ORIENTED SOFTWARE DEVELOPMENT**

by

Mark Richard Lattanzi

(ABSTRACT)

This research models the effects of software reuse on object-oriented software development, in particular, the reuse of C++ classes. Two types of reuse (with and without modification) are compared. The common traits of programmers who tend to reuse are identified, and some object-oriented software metrics are correlated with the inherent reusability of a C++ class. These issues are important because software reuse has been shown to increase productivity within the software development process.

This research effort describes three experiments. The first characterizes the effects of reusing C++ classes on object-oriented software development using nine development process indicators. The second experiment uses ten similar process indicators to differentiate the effects of writing C++ classes from scratch versus reusing them without modification versus inheriting new classes from existing ones. The last experiment correlates some object-oriented metrics with the expert opinions of the reusability of C++ classes.

This research has shown that the black box reuse (reuse without modification) of C++ classes is beneficial to object-oriented software development in many ways. Development time is reduced and system reliability increases. For abstract data type C++ classes, a set of fifteen skills and experiences are shown to be prominent in frequent class reusers. Lastly, a set of object-oriented metrics is used to predict C++ class reusability. All of these results can be used to increase programmer productivity when developing C++ software systems.

--

This work was supported in part by the National Science Foundation Grant #527524: *Measurement of Software Reusability in the Object Oriented Paradigm*, to Virginia Tech.

Acknowledgments

Four years of my life were used to complete this document. In that time, many people helped me with the technical aspects of this research and with maintaining my sanity. My deepest thanks go out to each and every person who helped make this research possible.

Dr. Sallie Henry, my advisor and my friend. None of this would have happened without her guidance and support.

My committee: **Dr. Osman Balci**, **Dr. Dennis Kafura**, **Dr. Robert Williges**, and **Dr. Robert Foutz**, all who provided me with invaluable technical assistance and advice about a myriad of topics.

My parents: **Henry and Jay Lattanzi**, who will always be there for me in sunny Florida.

Chris Moore, my fiancée, who put up with me more than anyone during this endeavor.

Todd Stevens, for listening to me, giving much needed advice, and putting up with me through many things too numerous (or desirable) to mention.

My closest friends in the world: **Frank Owen** and **Phil English**, whose phone calls and visits always brightened my life.

The **graduate students** in the CS department that were my friends and sounding boards: Randy, Siva, Bo, and Austin to name a few.

The **People of the LUP**: Suha, Steve, Mona, John P., Kurt, John G., Hla, Jim, Laura, Sean, Brian, Jeff, Greg, Big Joe and a host of others.

Matt, Mike, Jenn, and **Gina**: no one could ask for better brothers and sisters-in-law.

Grandma Lattanzi for her love and support for all of my life.

Shawn, who was here at the beginning and will always be in my heart.

Caroline, for a taking care of me in the summer of 1994.

The **sixty CS students** in two Software Engineering classes that I put through Hell.

And lastly, the dogs of my life:

Allie, the big white dog whose hopping clouds now,

Lizzie, the Chaser of Light #2 and my AT companion,

Kiera, the 3rd Chaser of Light, and the dog of my dreams.

Thanks for ALL of your help.

Table of Contents

Chapter	Page
1. Introduction	1
1.1. The Object-Oriented Paradigm, Reusability, and Software Metrics	1
1.2. Research Goals	3
1.3. Research Results	5
1.4. Research Approach	7
2. Motivation for Research	9
2.1. Software Reusability	9
2.1.1. Subdividing Reusability	11
2.1.2. Problems with Software Reuse	12
2.2. The Object-Oriented Paradigm	13
2.3. Software Metrics	15
2.3.1. Object-Oriented Metrics	15
2.3.2. Potential Reusability Metrics	15
2.4. Research into Software Reuse	16
2.4.1. Research Abroad	16
2.4.2. Research at Virginia Tech	19
2.4.2.1. Description of Research	19
2.4.2.2. The Metrics Generator	20
2.5. Summary	21
3. Definitions for the Research Effort	22
3.1. Definition of Terms	22
3.2. Statistical Terms	23
3.3. Object-Oriented Metrics	25
3.4. The Research Experiments	28
3.4.1. Experiment One	30
3.4.2. Experiment Two	32
3.4.3. Experiment Three	33
3.5. Summary	34
4. Effects of Software Reuse in the Object-Oriented Paradigm	35
4.1. Design and Setup	36
4.2. Experimental Data	37

4.3. Results	41
4.4. Analysis	49
4.4.1. Total Design Time	52
4.4.2. Total Implementation (Coding) Time	54
4.4.3. Total Library Time	56
4.4.4. Total Integration Time	58
4.4.5. Total Time for Whole Project Development	59
4.4.6. Number of Integration Errors	61
4.4.7. Number of Integration Compiles	63
4.4.8. Number of Integration Runs	64
4.4.9. Number of Errors in the Final System	66
4.5. Summary / Conclusions	67
4.5.1. Research Question One	69
4.5.2. Research Question Two	70
4.5.3. Research Question Three	71
4.6. Final Comments	72
5. Types of Software Reuse and Software Development	73
5.1. Design and Setup	74
5.2. Experimental Data	77
5.3. Results	78
5.4. Analysis	79
5.4.1. Analysis Data	82
5.4.2. Design Time	83
5.4.3. Time Examining / Learning the Library	84
5.4.4. Implementation (Coding) Time	85
5.4.5. Debugging Time	85
5.4.6. Total Development Time	86
5.4.7. Number of Compiles	87
5.4.8. Number of Runs	87
5.4.9. Number of Compile-Time Errors	87
5.4.10. Number of Logic Errors	88
5.4.11. Number of Run-Time Errors	88
5.5. Summary / Conclusions	89
6. Programmer Experience and Software Reuse	91

6.1. Experimental Data	91
6.2. Results	98
6.2.1. The Chi-Square Test	98
6.2.2. Chi-Square Tests on the Experimental Data	100
6.2.3. Results of the Chi-Square Tests	101
6.3. Analysis of the Chi-Square Data	101
6.4. Conclusions	106
7. Reusability and Characteristics of C++ Classes	108
7.1. Opinions on Reusability	109
7.2. Characteristics of Reusable C++ Classes	110
7.3. Experiment Three: Design and Results	112
7.4. Experiment Three: Analysis	117
7.5. Summary / Conclusions	123
8. Conclusions	125
8.1. Goal One	125
8.2. Goal Two	126
8.3. Goal Three	127
8.4. Goal Four	128
8.5. Implications of Research	130
8.5. Future Work	131
9. References	133
Appendices	147
A. Programmer Skill Areas	147
B. Questionnaires for Experiment One	150
C. Experiment One Raw Data	160
D. Experiment Two Assignmewnt Sheets	163
E. Experiment Two Data Collection Sheet	168
F. Experiment Two Pseudo-Code Solution	170
G. Experiment Two Raw Data	172
H. 18 Chi-Square Frequency Tables	174
I. Experiment Three: C++ Class Reusability Evaluation Form	184
J. Informed Consent Form For Research Involving Human Subjects	186
Vita	190

List of Figures

Figure	Title	Page
2.1	Research in Software Reusability	10
4.1	Design Time Model	54
4.2	Implementation (Coding) Time Model	56
4.3	Library Time Model	58
4.4	Integration Time Model	59
4.5	Total Development Time Model	61
4.6	Integration Errors Model	63
4.7	Integration Compiles Model	64
4.8	Integration Runs Model	65
4.9	Final System Errors Model	67
6.1	A 2x3 Chi-Square Explained	99
7.1	Some Characteristics of a Reusable C++ Class (Plus Metrics)	111

List of Tables

Table	Title	Page
4.1	Data Sets Collected During Experiment One	39
4.2	Development Process Indicators (Dependent Variables) for the Nine Projects	42
4.3	Reuse Data for the Nine Projects of Experiment One	46
4.4	Size Metrics for the Nine Projects	48
4.5	Object-Oriented Complexity Metrics for the Nine Projects	49
4.6	Summary of the Regression Models for the Nine Process Indicators	68
5.1	Means and Variances for the Three Experimental Groups	75
5.2	Statistical Test Values Showing the Three Experimental Groups Being Equal	76
5.3	Ten Process Indicators for Experiment Two	78
5.4	Development Data for Experiment Two - Means and Variances	79
5.5	Statistical Tests on the Ten Process Indicators	83
5.6	Summary of the Means Relationships for the Ten Process Indicators	89
6.1	Original Data Set of Programmer Skills and Which are Eliminated	93
6.2	Programmer Skills Data for the Programming Languages Skills	95
6.3	Programmer Skills Data for the Software Engineering Skills	96
6.4	Programmer Skills Data for the Reusability Skills	97
6.5	A 2x3 Chi Square for UNIX Experience Versus Level of Reuse	100
6.6	Results of the Chi Square Tests for the Eighteen Programmer Aptitudes and Level of Reuse	102
7.1	Percentages of Programmer Opinions for What Make a Class Reusable	109
7.2	The Fifteen Classes, Their Rank, and Averaged Reusability Scores	114
7.3	Metrics Values for the Fifteen Classes in Experiment Three	116
7.4	Averages for the Three Reuse Groups for the Fifteen Classes	117
7.5	Correlation Coefficients for Five Indicators Versus the Fifteen Metrics	119
C	Experiment One Raw Project Data	160-162
G	Experiment Two Raw Project Data	173
H	Chi-Square Frequency Tables for Programmer Experiences	175-183

Chapter 1

■ Introduction

The state of the software industry is again reaching crisis proportions. Statistics predict a world-wide shortage of programmers some time this decade [CoxB90, Horo84, Jone84, Stan84]. Programs are growing larger and more complex. The current programmer population is not adequate to meet the demands of the software users of the world. One partial solution to this problem is through software reuse. Studies [Jone84, Meye87a] have shown that around 15% of a new program is unique code; the other 85% is composed of common functions and routines that have been written many times before. By reusing existing source code, software developers can become more productive and hopefully generate a higher overall quality product. Reusing existing software has multiple benefits. Less testing is required on the reused routines. Maintenance efforts drop off substantially for the reused code, and the productivity of the software developers increases.

1.1 The Object-Oriented Paradigm, Reusability, and Software Metrics

One of the emerging technologies of the 1990's is the object-oriented paradigm: designing programs based on the concrete objects being manipulated in the system. A key idea of object-oriented languages is to encapsulate the objects (data structures) with the routines that act on them. Furthermore, complex objects can be developed by merging simpler objects. Both of these concepts lend themselves nicely to software reuse. As libraries of objects are constructed, programmers are able to safely reuse large sections of code instead of single procedures. This idea was originally proposed in an early paper by McIllroy entitled *Mass Produced Software Components* [McIl69]. Since then, researchers have been investigating how to reuse software effectively. Unfortunately, progress has been slow. McIllroy's vision has not yet come true.

Part of the reason for this failure is a lack of industry coding standards, a lack of repeatable processes, and a lack of a set of measures with which to gauge improvement. Software Engineering is concerned with creating software metrics. DeMarco states "You can't control what you can't measure." [DeMa82], a quotation based on advice from Lord Kelvin. One of the problems in software reuse is the lack of valid measures (standards) to use that capture the various traits of reusability. Part of this research focuses on creating and partially validating some software metrics that measure the reusability of a piece of object-oriented code.

Software reuse is a key to increasing productivity within the software development process. By reusing existing software, time and effort are saved in the development and maintenance phases of a software product [Lewi92]. The effects of reuse on software development are being accessed in this research. In particular, how can the effects of reusability be quantified?, How does encouraging software reuse affect programmer productivity and final system quality?, and What common traits or experiences exist in programmers who reuse well?

This research effort is centered on a relatively new area of Software Engineering: the measurement of object-oriented software (C++ classes) with a focus on software reuse. C++ is an object-oriented programming language designed by Stroustrup [Stro86, Stro88].

The object-oriented paradigm has been shown to promote reuse in some cases [Lewi92]. Object-oriented libraries (databases) are being introduced throughout the Computer Science industry. NeXtStep Computer [Next95] uses a large interface library of classes as the basis for its object-oriented software development environment. GNU [GNU95] developed a standard library for its UNIX compiler. Microsoft's [Micr95] development kit (an object library) is shipped with Microsoft's C++ compiler. Borland International [Borl93] ships a C++ object library with Borland C++, version 4.0. Smalltalk [Gold84] is

based on reusable libraries of objects and is hardly usable without them. COOL [Afsh93] is a public domain library of C++ classes. Classix [Empa90] and NIH [Gorl90] are two third-party libraries of C++ and Smalltalk objects.

However, there are few, if any, metrics that attempt to measure the reuse potential of one of the classes in these libraries (or even the reuse potential of a function in a procedural language library). This research first investigates the effects that software reuse has on the software development process and the final product. Additionally, two different types of reuse are investigated. Next, the characteristics of programmers who tend to reuse are identified, and lastly, the aspects of software that promote reusability are identified in an empirically based hierarchy. This hierarchy characterizes the reusability of C++ classes by identifying the measurable traits of C++ classes. Some current software metrics are then used with this hierarchy to see if they can predict the reusability of a C++ class.

The remainder of this chapter explains the four major goals of this research effort and their connections to software reusability, the object-oriented paradigm, and software metrics.

1.2 Research Goals

The goals of this research involve measuring reusability of object-oriented software and determining the effect that reuse has on the software development life cycle. The specific goals of this research are described below.

GOAL ONE is to characterize the effects of reusing C++ classes on the software development process.

One of the major goals of this research is to examine how reusing C++ classes affects the development of software. Nine process indicators are used to ascertain the impact that reuse in the object-oriented paradigm has on the software development process. The effects on each of these indicators are modeled using multiple linear regression model. An

empirical study (Experiment One) is performed in academia to accomplish this goal. Chapter 4 addresses this goal in detail.

GOAL TWO is to measure the effects of the different types of C++ class reuse (black box reuse and white box reuse¹) on the software development process.

The second goal of this research focuses on the type of reuse that is occurring and its effect on the software development process. Lewis [Lewi92] did some preliminary work in this area proving several important results concerning software reuse. One is that reusing C++ classes improves productivity. This new research refines and quantifies this result (i.e., What type of reuse of C++ classes is the most beneficial and how beneficial is it?). A second experiment (Experiment Two) is conducted to achieve this goal. It also occurs in an academic setting. Chapter 5 addresses this goal.

GOAL THREE is to determine what programmer characteristics influence or predict the level of reuse performed.

Goal three pertains to the human side of software reuse. Tracz [Trac88a, Trac88b] and others [Free87, Hoop91, Lewi92, Wood87] point out that software reuse is not just a technical problem. People are reluctant to reuse. Some programmers are reluctant to reuse other people's code. Some programmers are good at writing reusable code; others are good at reusing existing code. To meet this goal, data on the programmers involved in the first experiment is gathered and data on the code that each programmer writes or reuses is also recorded. An analysis is performed relating the traits of the programmers with the level of reuse attained by each of them. Chapter 6 addresses this goal.

GOAL FOUR is to identify some of aspects that make a C++ class reusable, and to assign metrics to some of these aspects to measure the reusability of a C++ class.

¹ Black box reuse refers to reusing code from software libraries without modifying it. White box reuse implies that the reused code is modified before being reused. Inheritance is a form of white box reuse.

Goal four of this research identifies some of the major aspects of C++ classes that make them reusable. The idea is to divide the broad concept of class reusability into smaller, more directly measurable parts for which metrics can be created. The idea of dividing broad concepts into constituent (measurable) parts can be found in [Balc93, McCa78]. Some aspects of class reusability are concrete and can be objectively measured (code complexity) and others require subjective measurement (code readability). Under this goal, the major characteristics of reusable classes are identified. The reuse potential of a set of C++ classes is obtained. Metrics are then gathered on this set of classes to ascertain if any object-oriented metrics can be used to determine the reusability of a C++ class. The third experiment (Experiment Three) of this research is conducted with industry experts on industry software (C++ classes from current C++ libraries). Chapter 7 addresses the third experiment and this research goal.

Summarizing, this research examines the concept of software reuse in the object-oriented paradigm and the effect that reusing C++ classes has on the software development process. Furthermore, this research decomposes C++ class reusability into measurable traits that can be used to quantify the problems involved in developing and using reusable software in the object-oriented paradigm, and to provide some tools with which to combat these problems.

1.3 Research Results

Some tangible results accompany each of the four major goals of this research effort. Goal one investigates the software development process. The result of this investigation is a **set of nine multiple linear regression models** that characterize the effects of reusing C++ classes on the software development process. The software development process is characterized by a set of nine indicators. Each of the nine models explains the trend in one of the process indicators. The nine indicators listed below each measure some facet of the

software development process from the design stage through the coding and integration stages to the submission of the final product.

The nine process indicators are:

1. Total Design Time (hours)
2. Total Coding Time (hours)
3. Total Library Time (hours)
4. Total Integration Time (hours)
5. Total Development Time (hours)
6. Total Number of Major Integration Errors
7. Total Number of Integration Compiles
8. Total Number of Integration Runs
9. Total Number of Final Errors

These nine indicators are explained in detail in Chapter 4. Explanations for each of the nine models (and their constituent terms) are provided in Chapter 4 as well.

Goal two delves into the differing effects of black box reuse and white box reuse on the software development process. The **relative rankings** of writing from scratch, reusing without modification (black box), and reusing with modification (white box) are given for a set of process indicators very similar to those outlined under goal one.

Goal three's result is a **list of programmer characteristics and experiences** and which ones correlate with ability (or likelihood) to reuse. Each of the subjects in Experiment One fills out an extensive background questionnaire, which is correlated to the level of reuse performed during the experiment.

From goal four, the result is a **list** of some of the **measurable aspects of C++ class reusability**. This hierarchical list is based on the opinions of the subjects in the first two experiments. This list is used to guide the selection of some software metrics to measure

the reusability of a set of C++ classes. **Dependencies** between some **object-oriented metrics** and five subjectively determined **reusability indicators** are made. The reusability indicators are obtained from the third experiment's poll of industry experts.

In this section, the results for each of research goals have been outlined. The next section explains the research method and approach to attaining these four goals.

1.4 Research Approach

Achieving these four goals of this research is accomplished through three major sources. The basis for this research comes from an extensive literature review detailed in a following chapter.

This research effort begins with the motivation for this research and a review of the literature covering general reuse and reusability in the object-oriented paradigm in particular. A hierarchy of the various research sub-areas in the field of reusability is presented along with the various papers and articles that pertain to each topic.

The next step of this research is to perform two experiments in academia to gather some empirical data about how reusing C++ classes affects the software development process and some opinions about software reusability and what makes a C++ class reusable. The first of these experiments is used to evaluate the effects of software reuse on the development process. During this experiment, some data is collected to aid the attainment of goals three and four. In particular, programmer characteristics and experiences are gathered so correlations can be made against the amount of reuse performed by each programmer. Also, programmer opinions are gathered on what the characteristics of reusable C++ classes are.

Experiment Two deals with two kinds of software reuse (black box reuse and white box reuse) and how each affects software development. A set of programmers is given a task

and asked to perform one kind of reuse or the other. Development data is collected as they work. The relative benefits of the different types of reuse are obtained for a set of software development process indicators.

Lastly, Experiment Three is an opinion poll of object-oriented paradigm (industry) experts. This poll gathers data on a set of C++ classes about their relative reusability. Five indicators are gathered: the class complexity, the class organization, the class ease of use, the class documentation, and the class completeness of functionality. These indicators are correlated to some object-oriented metrics, so that the reusability of a C++ class can be predicted, before it is actually reused or added to a software library.

The literature review and these three experiments form the bulk of this research. It is detailed in the coming chapters.

Chapter 2

■ Motivation for Research

At a high level, the motivation for this research stems from the need for software engineers to be more productive and more reliable when writing software systems. The reuse of software helps programmers achieve these goals. Since the object-oriented paradigm is designed with a focus on software reuse [Meye87a], it also lends itself to increasing programmer productivity. But, it can be difficult to perceive these benefits [Barn91, Boll90, Hend93, LimW94]. Experimentation and analysis are needed to understand how software reuse should be performed and what benefits can be achieved through it. Software metrics provide the means to measure quantitatively the process of developing (and reusing) software. Therefore, this research explores some current issues in Software Engineering and elaborates upon them.

The remainder of this chapter explores the three major areas mentioned above: software reuse, the object-oriented paradigm, and software metrics in more detail. Current research findings are investigated and presented. Figure 2.1 depicts the field of software reusability and the various citations addressing some of the specified areas. The bold lines connecting the shadowed boxes depict the path to the niche in which this research effort lies. The following literature review forms the foundation upon which this new research is based.

2.1 Software Reusability

Software reuse has been defined in many ways. Biggerstaff [Bigg87] defines it as the reapplication of code or the use of libraries, routines, and objects. Tracz [Trac90b] defines it as reusing software that was designed to be reused. Therefore, he states that software salvaging is not software reuse. The reused software must have been designed for reuse. Below, the concept of software reuse is elaborated upon and some of the problems with reusing software are discussed.

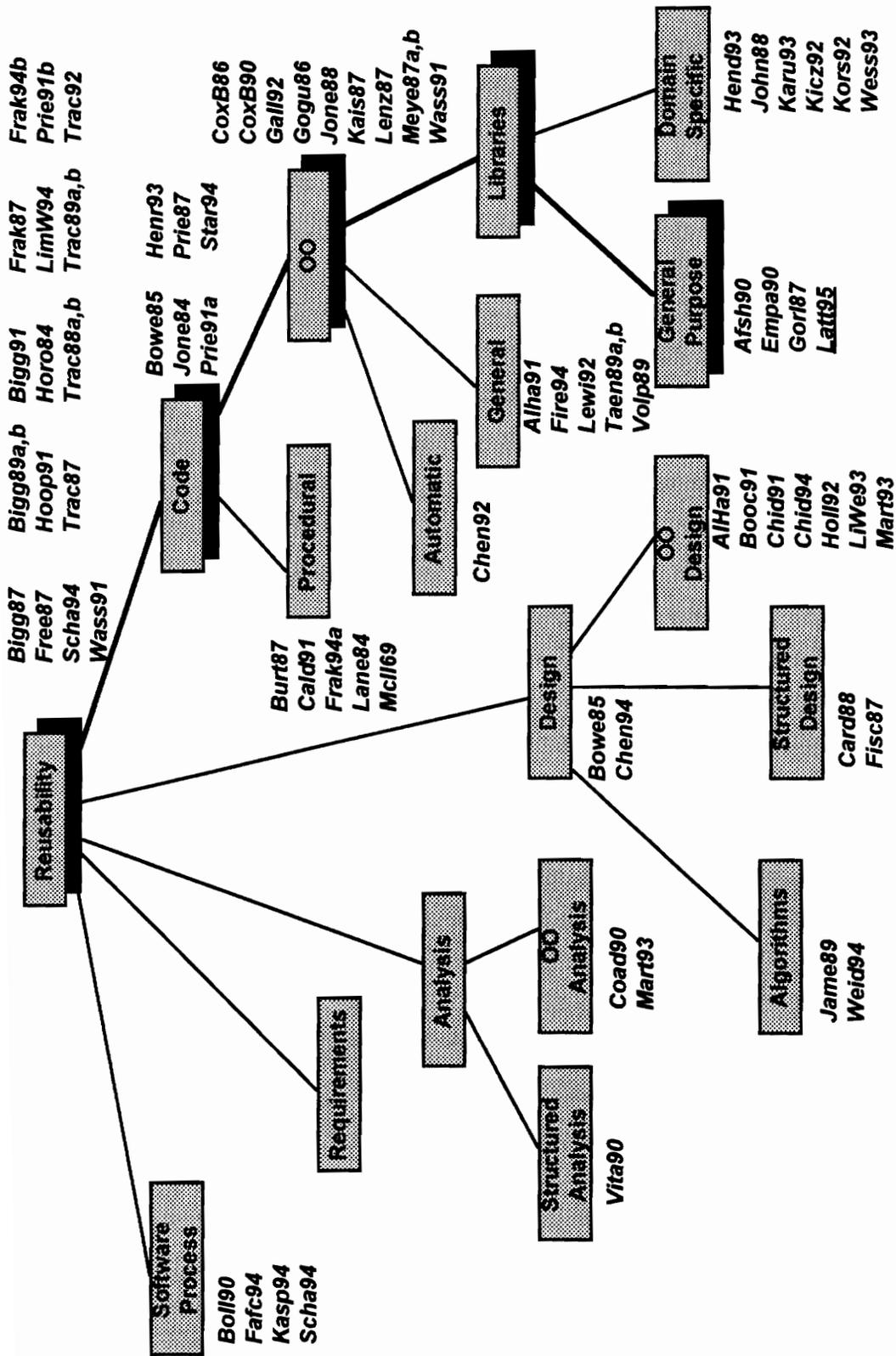


Figure 2.1. Research in Software Reusability

2.1.1 Subdividing Reusability

Software reuse is a broad topic that can be subdivided in a number of ways. Grumann [Grum88] divides software reuse into reuse of algorithms, reuse of components, and reuse of designs. Biggerstaff divides the field of reuse into two broad categories: generation and composition [Bigg87]. Reuse through generation involves code generators. Users input the required parameters and code generators output custom objects or even complete systems that can be used as needed. Much research is occurring in the field of reuse through generation [Bato94, Bigg89b, Guer94].

Code composition reuse refers to using libraries of components to build complex systems. Composition strategies are inherently easier to understand and use because there is no need to learn a new meta-generation language. The only prerequisite for building systems from reusable components is knowing what components exist in the libraries and how to access them correctly. Research in this arena includes [Burt87, Frak94a, Gall92, Gugu86, Jone88, Lenz87, Prie87, Prie91a]. This research effort focuses on compositional reuse.

Software reuse can also be divided by what is being reused: specifications, design documents, actual code, or some meta-language that describes code (such as C++ templates). Code reuse can be further subdivided into black box and white box reuse. Black box reuse implies reusing the component without modifying the internal code while white box reuse implies that the component is modified before being reused. This research explores code-level software reuse. It focuses on the reusability of object-oriented code components (specifically, general purpose C++ classes), and does not address the reuse of specification documents or design documents, although research in these areas has shown promise [Card88, Chen94, Lane84, Lieb88, Lieb89b, Neig84, Volp89, Weid94]. This research examines the distinction between black box and white box reuse and the differing effects on the software development process that these two types of reuse cause.

2.1.2 Problems with Software Reuse

Software reuse has been proposed and investigated since 1969 [McIl69]. Why hasn't it been integrated into the software community? Many [Free87, Meye87a, Meye87b, Prie91b, Trac87, Trac88b] have attempted to explain this discrepancy. Computer code is a specific custom-fit entity that does not lend itself to reuse well. As code component size grows, generality (and thus frequency of reuse) drops off rather sharply. Data representation is too varied to make reuse worthwhile. Top-down design has inhibited reuse by deferring the design of the detailed components of the system until the end. If the system was built in a bottom-up fashion, the initial components could be selected from pre-existing libraries of routines.

Meyer [Meye87a] identifies the root cause of the technical problems of reusing software as "too many variants". Although programmers write essentially the same code in system after system, it is not exactly the same. The amount of variable information is considerable ranging from data formats to hardware specifications.

Another problem is that the software industry is profit-based. It is not profitable to spend extra dollars to write reusable code only to post it for public access. Code reuse within one company is more feasible, but rarely implemented formally. Programmer attitudes can cause problems as well. Programmers are reluctant to reuse source code written by somebody else employed at some other company or even within their own company. Many programmers believe that they write the best software and attempting to reuse other programmers' source code invariably leads to problems [Trac88b, Wood87]

Reusable software must be easily retrievable or it will not be reused [Prie87]. If programmers can not easily find the module they need or do not know how to find it, they simply will not even look for it. There are many researchers working in this area of component classification and indexing. For more information into this sub-area of software reusability, refer to [Frak87, Prie87, Prie91a].

One of the key issues to improving software reuse is through industry standards [John88, Kern84]. Until the software community has developed and accepted some standard protocols for class interfaces and naming conventions, software reuse is virtually impossible. The benefit gained is offset by having to learn a new interface for every object (class) that is to be reused. Programmers will not do it.

The field of software development contains many myths and problems about software reuse [Reil87, Taen89a, Trac88b, Trac90a]. This research explores some of these common misconceptions and problem areas and adds empirical data and analysis to dispel or at least quantify some of the misconceptions and problems when dealing with software reuse.

2.2 The Object-Oriented Paradigm

This new research also focuses on the object-oriented paradigm because the paradigm has been shown to promote code-level software reuse [Lewi92]. Furthermore, the object-oriented paradigm has flourished in recent years. The OOPSLA² conference is one of the largest Computer Science conferences in the world. Research is proceeding in many sub-areas of object-oriented programming from reusability [Bigg89b, Bigg91, Fire94, Free87, Trac90a] to maintainability [Barn91, Basi90, LiWe93, Scha94, Wild92] to ease of coding [John88, Kicz92, Lieb89a, Lieb89b]. Object-oriented programming languages are built around the concept of viewing complex systems as collections of objects interacting with each other. An object (or class) is simply an encapsulated, highly cohesive module containing some data structures and a set of routines (methods) that are allowed to modify the data fields. These objects are meant to be highly reusable, and as standards emerge in the industry, highly maintainable. Once programmers learn what objects are readily available, they can piece together complex systems from previously built objects and become more productive.

² Object-Oriented Programming Languages, Systems, and Applications

Although object-oriented languages are claimed to aid in software reuse, the object-oriented paradigm is a new methodology fraught with unknown pitfalls and problems. The idea of using reusable objects to create complex systems leads to inefficiencies in the time and space domains for the final system. When programmers reuse a component or create a sub-class of a reusable object for reuse, the entire class becomes part of the binary code including a potentially large number of methods that never get invoked. Objects are designed to be general and reusable. Being general enough to allow reuse often results in execution speed inefficiencies as well [Taen89a, Taen89b].

Other problems exist when using the object-oriented paradigm. Specifically, the bulk of programmers today has learned to think and code procedurally. It is unclear whether these people can be effectively retrained to think in an object-oriented fashion [Beck89, WuCT93]. Furthermore, designing classes for inclusion into reusable libraries can be difficult [Barn91, John88, Kors92]. Deriving a new sub-class should be doable by reusers without necessarily knowing the internal structure (implementation) of the parent class. This task requires that the parent classes be well-thought out and designed for reusability. This additional programming effort has deterred much of the software industry from reaping the benefits of software reuse in the object-oriented paradigm.

Inheritance is a premise of object-oriented languages. But, class hierarchies can be complex and enigmatic. In order to reuse classes in deep hierarchies, software developers, potentially, have to understand the entire hierarchy [Arms94]. This understanding comes at a price that reduces the overall benefit of reusing software. This research quantifies this cost. Also, this research accesses the value of the inheritance construct to reusers by comparing the effects of reusing with and without modification (Chapter 5).

2.3 Software Metrics

Software metrics have existed since programmers started counting lines of code as a measure of program size and complexity [Cont86]. Since then, many software metrics have been developed to measure many different facets of the software development process [DeMa82, Henr92, Kasp94]. The remainder of this section presents the metrics upon which this research is based.

2.3.1 Object-Oriented Metrics

One of the new areas of focus within the object-oriented paradigm concerns design and code metrics for object-oriented languages [Barn94, Byar94, Cant94, Chid91, Chid94, LiWe93]. Procedural language metrics may not be accurate in the object-oriented paradigm and they have yet to be validated in it [Henr93a, Henr93b, LiWe93]. Many software companies have come to rely on metrics for feedback throughout the software development life cycle [Canf94, Joos94]. Part of this research effort is to create and/or modify some software metrics for use in the object-oriented paradigm. These new metrics focus on measuring C++ class reusability. Since reusability is a difficult trait to quantify, this research divides C++ class reusability into its more measurable aspects, which can be more directly measured. This idea of decomposing an abstract programming concept into simpler ones can be found in [Balc93, McCa78].

2.3.2 Potential Reusability Metrics

There are few metrics for use in the object-oriented paradigm. Chidamber and Kemerer [Chid91, Chid94] propose some complexity metrics that this research uses as a basis for measuring the reusability of a C++ class. Li and Henry [LiWe93] modify these and add a few new ones. Korson [Kors92] authored a paper on the characteristics of reusable classes. Some of the characteristics that he identifies are used in the measuring of reusability in this research. [Karu93] suggests some possible reuse metrics. Others [John88, Kors92, Trac88a, Trac90a] have attempted to measure reusability through subjective data (opinions of experts) with limited success.

2.4 Research into Software Reuse

Much research has been done in the field of reusability; some of it pertains to the object-oriented paradigm. This section first discusses research into software reuse and the object-oriented paradigm and then focuses on the research that have been done locally at Virginia Tech in these areas.

2.4.1 Research Abroad

The object-oriented paradigm is one of the most studied areas in Software Engineering today [Booc91, Coad90, CoxB86, DeCh93, McGr92, Meye87a, Meye87b, Wirf90]. Research in the area of reusability started with the now classic paper by McIllroy on software components in 1969. Not much research occurred in the field until the early 1980's, and one could say that the field of software reusability was still in its infancy. The software industry did not reuse well or frequently. Freeman [Free87] and Wegner [Wegn84] described the field of software reusability and predicted where the software industry was headed in terms of reuse. Jones [Jone84] reviewed the current research in the field. Few of the predictions were coming true. Programmers were not building software systems from libraries of reusable components.

When the library concept finally took off, a new problem was identified. These component libraries quickly grew to enormous size [Jone88, Liao93]. Prieto-Diaz [Prie87, Prie91a] developed a classification scheme to organize software libraries. Others [Burt87, Cald91, Frak87, Henn94] got involved in this area of software classification and searching techniques as well.

In 1987, Biggerstaff [Bigg87] reviewed the current research into software reuse and made some predictions about the direction of the field. Tracz [Trac88a, Trac88b] wrote a series of papers on misconceptions in software reuse and what to do about them. This new research disproves some of these misconceptions through empirical studies.

In July 1987, *IEEE Computer* published a special issue on software reuse. Reusability was finally being researched in earnest. Papers in this issue cover topics from reusable software libraries [Burt87] to programmer characteristics issues and software reuse [Wood87] to system construction from reusable components [Garg87, Lenz87, Kais87]. Researchers were investigating all type of reuse from specifications to designs to source code. Tracz [Trac87, Trac88a] believed that code reuse is too specific and would only result in modest productivity gains. Others [Holl92, Lieb88, Lieb89a, Meye92] agreed and turned towards design reuse and meta-languages to specify more general components.

In September 1994, *IEEE Software* also published a special issue on software reuse. This issues contains some of the successes of software reuse [Bato94, Frak94a, Joos94] and some methods for building software from reusable components under systems such as the NextStep Development Environment [Star94]. Some of the effects of reusing software at Hewlett-Packard are also discussed [Fafc94].

Many companies have instituted reuse programs. There have been many successes and failures. Frakes [Frak94b] describes some of the successes of reusing software. Many experience reports exist documenting the reuse of software in industry [Joos94, Lore91, Mili94, Star94].

With the advent of object-oriented languages, research into software reuse at the code level was renewed. Bertrand Meyer wrote a landmark paper: *Reusability: The Case for Object-Oriented Design* [Meye87a]. In it, he describes the process of software reusability and how the object-oriented paradigm lends itself well to designing reusable software. Meyer identifies overloading, genericity, and inheritance as the three fundamental traits of object-oriented languages that increase the reusability of the resulting software. Others in the field [Mica88, Morr87, AlHa91] agree with his statement. Some experts add data encapsulation as another significant trait although the term object-oriented implies this

idea. Experience reports exist for industries doing object-oriented reuse as well [Jett89, Wess93].

Korson and McGregor [Kors92] identify a set of twenty-three attributes of reusable software ranging from inheritance structure to completeness. Some of the attributes can be measured objectively and some are subjective and need to be further divided into their measurable aspects. A few of these attributes can only be measured subjectively (by experts). Korson and McGregor's research is based in the object-oriented paradigm as well. Johnson and Foote [John88] published a similar paper entitled *Designing Reusable Classes* which enumerates thirteen rules to follow to develop classes with a high reuse potential. Johnson's and Foote's rules are based on experience and subjective data. This research uses Korson and McGregor's attributes and Johnson and Foote's rules as one foundation for examining the reusability of C++ classes.

Gorlen [Gorl87] has developed an object-oriented class library in C++ under the UNIX operating system. His idea is that software developers need to produce some class collections similar to those found bundled with Smalltalk compilers before software reuse (in the C world) becomes a reality. Given the plethora of C++ libraries in the public domain today and the increased focus on software reuse, this prediction seems to be coming true.

Some researchers believe that reusable software will not be useful until programmers start designing software for reuse. One key step in doing this is called "Domain Analysis" [Trac92]. Domain Analysis is the process of intense examination of the domain of interest (e.g., management information system, control system, business application) [OCon94, Vita90]. The premise behind domain analysis is that reusable components are domain specific as opposed to general purpose. Greater reuse is obtained if components are designed with a particular domain in mind. Others [Lieb89b, Mart93, Neig84] have investigated this topic as well.

2.4.2 Research at Virginia Tech

At Virginia Tech, the Metrics group sponsored by Dr. Sallie Henry, has investigated many topics in Software Engineering relating to this research. Since 1984, many of her graduate students have performed research in or relating to the area of software metrics. Some of this work focuses on the object-oriented paradigm. Much of this work uses a tool developed by the Metrics group, the Metrics Generator. It is discussed at the end of this section.

2.4.2.1 Description of Research

Henry and Humphrey [Henr93a] performed a study comparing the procedural and object-oriented paradigms in terms of maintainability. They conducted an empirical study taking measurements on the subjects' performance on various predefined maintenance tasks. Henry and Humphrey show that building applications with object-oriented languages (like C++ or Objective C) results in final systems that are much more maintainable than systems constructed with procedural languages (like Pascal or C). This important result suggests the need for more research into the object-oriented paradigm.

Lewis and Henry [Lewi92] show several significant results regarding the object-oriented paradigm and software reusability. They conducted an experiment with Computer Science seniors to prove empirically some of the benefits of software reuse. Lewis and Henry show that the object-oriented paradigm has an affinity towards reuse. Subjects using the object-oriented languages reuse more often and "better". They also show that software reuse improves programmer productivity regardless of the language used. However, the productivity increase is greater in the object-oriented paradigm. Lewis defined productivity as a function of development time and several other measurements relating to the cost of developing the software system. See [Lewi92] for his complete definition of productivity. As with Humphrey's study, Lewis's work also shows that maintenance in the object-oriented paradigm is easier than in the procedural paradigm. This new research

supports some of these results and adds a few more quantitatively modeled effects of software reuse.

Li and Henry [LiWe93] developed some metrics (described in Chapter 3) specifically for object-oriented languages that predict maintainability from code and, in some cases, from design documents. The research is partially validated using two commercial systems written in Classic AdaTM (object-oriented Ada). The focus of the study is to bring together research in software metrics with research in the object-oriented paradigm. Specifically, Li and Henry investigated a suite of proposed object-oriented metrics [Chid91, Chid94], proposed several new object-oriented metrics, and partially validated the metrics using maintenance data collected from two commercial systems. Some of the metrics described in the next chapter are taken from this study.

Other relevant research includes several studies into code readability and documentation and how they relate to software complexity and maintainability [Gibb88, Towe92]. Gibbens showed correlations between in-line documentation (comments) and software maintainability. Towe compiled the current research dealing with correlating code readability and code documentation.

The results of these efforts serve as a foundation for this new research regarding the analysis of the complexity and readability of systems built from reusable object-oriented components.

2.4.2.2 The Metrics Generator

One of the chief tools used by the Virginia Tech Metrics group is the Software Metrics Generator, a program to generate metrics for large software systems. Many of the research efforts mentioned above have used and contributed to the Metrics Generator including [Chap90, Henr81, Henr88a, Henr88b, Henr89, Henr90, Henr91a, Henr91b, Henr93a, Henr93b, Henr93c, Latt93, LiWe93, Mayo89, Wake88]. Briefly, the Metrics Generator is a tool designed to gather source code metrics from large software systems.

The generator converts several different source languages (Ada, C, Pascal, C++) into a generic relational language for analysis. See [Henr88b] for more information regarding the intermediate language.

The generator outputs the three basic types of software metrics for procedural languages (code, structure, composite) and the set of new object-oriented metrics as well. The Metrics Generator is a highly portable tool that has proved to be invaluable when conducting studies of the type described above. For this research, the tool has been modified to parse C++ code and produce a set of object-oriented code metrics.

2.5 Summary

This chapter has explored the current research in the three areas of focus for this research: software reuse, the object-oriented paradigm, and software metrics. This research presents a decomposition of the concept of software reusability (of a C++ class) into more directly measurable parts. Previously validated metrics together with a few proposed metrics are then used to measure some of these parts.

Current research and problems in these areas are discussed above and some questions motivating this research effort are posed. The ongoing research effort at Virginia Tech is discussed and provides one of the foundations for this new work. The research presented in the remainder of the document addresses some of the problems and questions presented in this chapter. In particular, this new research quantifies some of the effects of software reuse on the software development process and how the concept of software reusability can be measured using software code metrics.

Chapter 3 defines the terms that pervade this research. It then describes the three experiments conducted in this research effort and defines some of the terms pertaining to each one.

Chapter 3

■ Definitions for the Research Effort

The goal of this chapter is to remove the ambiguities of the many terms presented throughout this research. Quality research must maintain accuracy, consistency, and unambiguity. With this in mind, this chapter defines some common technical terms used throughout this research. It also presents a high-level description of the three research experiments along with some specific terms necessary to understand the intricacies of each experiment.

3.1 Definition of Terms

Before the three experiments are described, some terms that pervade this document need to be defined. As each experiment is addressed, terms pertaining to them are defined. The global terms are:

1. A **software metric (measure)** is a function whose inputs are software (source code) and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given characteristic (such as size or complexity) [IEEE93]. Some common examples include McCabe's Cyclomatic Complexity [McCa76] and Halstead's Software Science metric [Hals77].
2. A C++ **class** is an implementation (in C++) of an abstract data type including its definitions of data structures, methods and interface [Sema93].
3. **Inheritance** is the language mechanism which allows the definition of one class (the derived class or sub-class) to include the attributes and methods of another class (the parent class or super-class). Inheritance enables the reuse of the code in a parent class by a number of derived classes [Sema93].
4. A **class hierarchy** is a characterization of the inheritance relationships among a set of classes. In a single-inheritance language, this structure is a directed acyclic graph (tree) [Sema93].

5. A **class library** is a collection of (sometimes) related C++ classes. The library contains the binary images of precompiled C++ classes and a set of header files for accessing the classes in the library [CoxB86].
6. **Software reuse**, for the purposes of this research effort, is compositional code-level reuse [Bigg89a]. In this research effort, the reuse of general purpose C++ classes such as abstract data type classes, is examined. Typically, these C++ classes are found in class libraries that have been designed and implemented for the purpose of being reused.
7. **Black box reuse** is the reuse of software component (classes) without modification [Prie93].
8. **White box reuse** is the reuse of software components (classes) by modification or adaptation [Prie93]. In the object-oriented paradigm, this is achieved by deriving a new class through the inheritance mechanism.
9. **Class coupling** is a description of the interdependence of classes. Two classes are tightly coupling if they access common data structures or rely on each other for services [Sema93].
10. **Class cohesion** is a measure of how well a class fits together [Somm92]. A highly cohesive class implements one logical entity (object).

These terms are commonly used in the field of Software Engineering and in the world of object-oriented software development. To avoid any ambiguity, these definitions have been presented. These terms are used throughout this research. An attempt is made to reiterate some of these definitions when deemed necessary for clarity and readability.

3.2 Statistical Terms

Because this research contains three empirical studies (experiments), many statistical methods and terms are used. To understand this research, the following brief definitions for some statistical terms and tests have been provided. All of these definitions are from [OttR93].

1. A **p-value** is a value representing the likelihood that the conclusion could have happened by chance. A p-value of 0.05 states that there is a 5% likelihood of finding a trend in the sample data when one does not really exist.
2. A **alpha level** is chosen p-value for a statistical test. When doing a statistical table look-up, an alpha level is selected to set the likelihood of arriving at a false conclusion. Typical values include 0.05 and 0.10.
3. A **test statistic** is a formula that computes a value that can be compared to values in a statistical table at several different alpha levels.
4. The **statistical level of significance** for a test is the calculated p-value or the selected alpha level. There are two methods for conducting a statistical test. One is to perform the test and calculate a p-value which reveals the likelihood of an error. The second is to select an alpha value, use it to look up the statistical level of significance (a number) in a statistical table, and compare the number to the calculated test statistic. If the calculated value is greater than the look-up value, then the conclusion is valid for the selected alpha level.
5. The **chi-square test of independence** is a statistical test to determine the independence between two categorical variables. The test results is a p-value signifying the strength of the dependency between the two variables.
6. A **T Test** is a statistical test based on a T distribution. A test statistic is created and used to calculate a value for the data set which can be compared to a statistical T distribution table at a variety of alpha levels.
7. An **F Test** is a statistical test based on a F distribution. A test statistic is created and used to calculate a value for the data set which can be compared to a statistical F distribution table at a variety of alpha levels. The test statistic is typically of the form of one variance divided by another.
8. **Tukey's Procedure** is a procedure used to compare the relative means of more than two sets of data at a given alpha level.
9. A **correlation coefficient** is a value that measures the strength of the relationship between two variables: the higher the correlation coefficient, the stronger the relationship.

10. An **independent variable** is a variable that can be controlled by the experimenter or is being used to explain the trend in some other facet of the experiment.
11. A **dependent variable** is the variable that is trying to be explained by the experiment. Typically, the dependent variable is explained by some linear combination of the independent variables.
12. A **regression model** is an equation relating a dependent variable to one or more independent variables.

This list of statistical terms represents the bulk of the statistical knowledge that needs to be understood. A few more terms are presented later that refer to the specific data of this research.

3.3 Object-Oriented Metrics

This section explains the various metrics that are presented and used throughout this research. These metrics come primarily from three sources. Chidamber and Kemerer [Chid91, Chid94] developed a set of six object-oriented metrics. Li and Henry [LiWe93] refined and partially validated these metrics and used them to create a set of ten object-oriented metrics. These object-oriented metrics are explained below. The remainder of the metrics presented below were developed during this research effort. For each of the metrics below, the source is given.

1. The **Number of Lines of Code (KLOC)** has been used extensively in Software Engineering [Cont86], so it is included for comparison purposes. The definition for the Lines of Code metric for this research is based on the number of semicolons found in the source code (C++ class).
2. **McCabe's Cyclomatic Complexity (MCC)** metric is designed for procedural languages. McCabe's metric is designed to measure the complexity and maintainability of a piece of code by counting the number of linearly independent paths through the code. This metric is based heavily on graph theory where sequential blocks of code represent nodes and decision statements represent the interconnections between the nodes. For a more complete description, see

[McCa76]. This research uses McCabe's metrics as a weight factor to measure the complexity of a C++ method.

3. The **Class Depth in the Inheritance Tree** (CDIT) is a measure of the exact position of the classes in the inheritance hierarchy [Chid91]. The deeper a class resides in the hierarchy, the more super-class properties the class can access. Intuitively, the larger (deeper) the depth of a class, the harder the class is to understand. The root of the class hierarchy has a depth of zero.
4. The **Depth of Inheritance Tree** (DIT) is the average depth of all of the classes in a class hierarchy or class library. It is the CDIT metrics averaged for an entire system of classes. When DIT is high, the system of classes has a deep (and narrow) class hierarchy. In terms of reusability, this makes classes from the system harder to reuse, because more classes must be understood in order to perform reuse.
5. The **Number of Children** (NOC) is a count of the number of sub-classes (children) directly derived from this class [Chid91]. The more children a class has, the more sub-classes that are affected when the parent class has to be changed. Therefore, as NOC increases, so does the software maintenance effort (software complexity).
6. The **Response for a Class** (RFC) metric measures the cardinality of the response set of a class [Chid91]. The response set of a class is defined to be all the local methods of the class and all the methods called by local methods of the class. Intuitively, the more methods a class has or can call, the more complex the class.
7. The **Lack of Cohesion of Methods** (LCM) metric measures the correlation among the methods and the local instance variables in a class [Chid91]. High cohesion indicates lower complexity. If LCM is large, the measured class lacks cohesion and is more complex and harder to maintain. LCM is measured by grouping the methods of the class into disjoint sets based on the local (class) variables that are modified by the methods. Methods modifying the same set of local variables are grouped into the same set.
8. The **Number of Local Methods** (NOM) metric can be viewed as an interface metric [Chid91]. NOM is a count of the number of local methods of a class (the

class's interface). The idea is that if the interface to a class is large, it is more complex.

9. The **Number of Semicolons (SZ1)** metric is based on the lines of code metric [LiWe93]. The more code that a class has, the more complex it tends to be. Although the lines of code metric has met with much resistance and criticism in the procedural paradigm, it is still widely used, so this metrics is included in this research.
10. The **Number of Methods plus Number of Attributes (SZ2)** size metric is defined to be the number of local methods in a class plus the number of local data attributes [LiWe93]. Again, the larger SZ2 gets, the more complex the given class is likely to be. This metric is suggested in [LiWe93] as an attempt to do better than just counting lines of code.
11. The **Weighted Method Complexity (WMC)** measures the static complexity of all the local methods of a given class and sums them [Chid91]. The static complexity measure can be any code metric. For this research, McCabe's Cyclomatic complexity is chosen as the base complexity metric. Clearly, classes with many methods or a few very complex methods are more complex than classes with simple methods or very few methods.
12. The **Message Passing Coupling (MPC)** metric is used to measure the complexity of message passing among classes [LiWe93]. Since the messages are defined by a class and used by objects of the class, the MPC metric gives an indication of how many messages are being passed among the objects of the classes. As the number of allowable messages increases, so does the coupling between this class and the rest of the system. Thus, the classes are more intertwined, and so, more complex and harder to modify and use.
13. The **Data Abstraction Coupling (DAC)** metric measures the number of instantiations of other classes within the given class [LiWe93]. This type of coupling is not caused by inheritance or the object-oriented paradigm. Generally speaking, any abstract data type with other abstract data types as members has data abstraction coupling. So, if a class has a local variable that is an instantiation (object) of another class, there is data abstraction coupling. Clearly, the higher the DAC, the more complex the data structures (classes) of the system.

14. The **Functions Per Class (FPC)** metric measures the average number of methods in a class. It is calculated by dividing the total number of methods in all the classes by the total number of classes in the system. As the FPC increases, the classes become more complete (hopefully), but also more complex and harder to maintain. It measures class hierarchies or whole systems rather than single classes. The metrics is the average NOM metric for a set of classes. It is created specifically by this research effort.
15. The **Lines Per Method (LPM)** metric measures the average size of all of the methods in a class or a class hierarchy. It is calculated by dividing the total lines in a class (or hierarchy) by the total number of methods in the class (or hierarchy). As LPM increases, the methods increase in size and therefore, are seen to be more complex. This metric is the WMC metric where the method weighting factor is simply the lines of code metric. This metrics is also created by this research effort.
16. The **Private Method Ratio (PriM)** metric measures the ratio between the number of private methods in a class to the total number of methods in the class. As PriM increases, there are more user-hidden methods. More calculations are occurring behind the scenes. As PriM increases, the complexity of the class increases.
17. The **Public Method Ratio (PubM)** metric is $(1 - \text{PriM})$ or the ratio of public/protected methods to the total number of methods in a class. As this metric increases, more of the class is visible and understandable to the programmer, and fewer operations are taking place in the background. As a C++ reusability measure, this metric increases as the evaluated reusability of the C++ class increases. Both of these last two metrics are created by this research effort.

All of this metrics can be calculated on a class hierarchy by calculating them for every class and averaging the values. This is how these metrics are used under goal one. Under goal four, these metrics are used at the individual class level.

3.4 The Research Experiments

Three experiments are performed within this research effort. The first two are conducted in senior-level Software Engineering courses on Computer Science students. The use of

students as subjects, while considered ill-advised by some, is justifiable in this case. Boehm-Davis [Boeh92] has shown that students are equal to their professional counterparts in many quantifiable areas of software development (programming). Weidenbeck [Weid86] and Holt [Holt87] corroborate this result. Although new skills and techniques are learned in an industry setting, a programmer's basic approach to software development is learned early in their education and professional careers.

Weidenbeck [Weid93] shows that complete beginners to programming develop some of the same skills as experts after just one programming languages course. Of course, Weidenbeck also finds areas of difference between complete novices and experts. However, McKeithen and Reitman [McKe81] show that intermediate programmers have most of the same skills as experts. This suggests that seniors in Computer Science may be nearly equivalent to industry professionals in terms of program comprehension and development.

This research is not attempting to prove that seniors in Computer Science are equivalent to industry programmer with years of experience and expertise. [Mody92] makes a case that programming, like many other disciplines, requires practice. However, for comparisons between similar experimental groups (e.g., all students), the cognitive abilities of student programmers are on the same level as their industry counterparts; students just take longer to complete the task at hand. Since the differences between experimental groups of student programmers are being measured, student programmers can be used. The use of students as subjects is also supported for comparison-type experiments (between experimental groups) by Brooks [Broo80].

Furthermore, due to the demands placed on the subjects of an empirical study, using industry programmers is usually not practical. The variability and outside influences that occur in software industry settings cannot be controlled and often results in false, or at

least, very weak, conclusions. Industry programmers rarely make good experiment subjects due to the difficulties involved in controlling the experimental variables.

Experiment Three of this research uses both students and industry programmers. The initial set of opinions on the reusable characteristics of a C++ class is generated from a student population, but the evaluation of the set of fifteen C++ classes is performed by programmers currently employed in the software development industry from several software companies.

The next three subsections describe the three experiments of this research effort respectively.

3.4.1 Experiment One

The first experiment conducted in this research provides data to achieve research goals one and three. A class of thirty Computer Science seniors develop ten systems in the object-oriented paradigm with the aid of a set of class libraries. Extensive data sets are gathered on the programmers and the software that they create. In particular, the amount and type of reused software for all ten of the projects is gathered in the first data set. The second data set is produced by the Metrics Generator (described in Chapter 2) which generates a series of source code metrics on each of the projects. These two data sets are used to model a series of nine software development process indicators to quantitatively characterize the effects of software reuse on the development process. This characterization occurs under goal one.

There are some terms that pertain to this experiment that need to be defined. The specific terms required to understand goal one are listed below. These three terms are defined for the purposes of this research effort; these explanations are not meant to be unique definitions.

1. A **software development process indicator** is a measure of some particular aspect of the software development process (e.g., the total design time). This research uses a set of process indicators to characterize the software process, so the effects of software reuse can be ascertained. The specific process indicators are explained in later chapters.
2. The **level of reuse** refers to the percentage of classes that a particular subject obtained from a C++ library and reused while developing software for this experiment. For Experiment One, there are two levels of reuse, black box reuse percentage and white box reuse percentage.
3. An **integration error (IE)** is a major occurs that occurs during the integration phase of the development process. When the various components (classes) of a system are implemented by many different programmers, there is often miscommunication about class definitions, method names, and other interactions that can cause integration problems. This is one of the process indicators used in Experiment One.

A third data set is gathered during Experiment One on the abilities of the subjects of the study. In particular, a portfolio is compiled of each of the subjects' experiences and aptitudes in some programming-related skills. Under goal three, this data set is correlated to the amount of reuse that each subject performs while working on the ten software systems. Some of the common terms used under goal three are:

1. A **programmer characteristic** for this research is defined to be any characteristic, experience, or skill that can be quantified about a programmer.
2. **Amount of reuse** refers to the number of classes that a particular subject obtained from a C++ library and reused while developing software for this experiment.

Again, these terms are defined for the scope of this research. These are not all-encompassing definitions. With these terms so defined, the descriptions and analyses of research goals one and three can be more easily understood.

3.4.2 Experiment Two

The second experiment of this research focuses on the type of reuse that is occurring and its effect on the software development process. Experiment Two also occurs in an academic setting. A group of twenty-four programmers are divided into three groups of eight programmers each. All three groups are given the same basic task: to design and implement a post-fix calculator. The first group, group A, writes all of their source code from scratch; group B reuses the underlying data structure (a stack) without modification (black box reuse), and group C derives a new underlying data structure class from a library class (via inheritance). Group C is the white box reuse group. While the subjects develop their systems, they record some development data (process indicators) similar to the ones used in the first experiment.

After all of the systems are completed successfully, the average values for the process indicators are examined searching for significant differences among the three groups. Terms necessary to understand this experiment and goal two are:

1. A **post-fix calculator** is a simple program that takes as input, post-fix expressions (expressions where the operator comes after the operands rather than in between them), and outputs the evaluation of the expression. This is the simple software system that is used in Experiment Two.
2. A **run-time error** (RTE) is an error that occurs while the system is executing. RTEs occur during the integration and testing phases [Somm92]. This is one of the process indicators for the first and second experiments.
3. A **logic error** (LE) is an error in the logic of the design or the source code. Logic errors can be found during any of the stages of the development process [Somm92]. This is a process indicator for Experiment Two.
4. A **compile-time error** (CTE) is an error generated by the compiler during the development of the software system [Somm92]. This is also a process indicator for Experiment Two.

3.4.3 Experiment Three

Experiment Three involves the identification of some of the characteristics of a C++ class that makes the class reusable. Opinions of the subjects of Experiment One are used as a starting point for this identification. A list of some of the measurable traits of a reusable C++ class is created. Next, a set of fifteen classes is given out to a group of fifteen industry experts. These experts judge the reusability of each of the classes based on the list of traits of a reusable class. Five reusability indicators are generated for each of the fifteen classes based on the subjective evaluations generated by these expert programmers.

Next, the object-oriented metrics presented under Experiment One are generated by the Metrics Generator for all fifteen classes of Experiment Three. The values of these metrics are then correlated to the five reusability indicators. Trends between the indicators and the metrics are analyzed.

The terms required to understand Experiment Three and goal four are listed below. These three terms are defined for the purposes of this research effort.

1. A **measurable aspect of software** is some facet of a broad ambiguous concept such as reusability that can be directly measured by using software metrics or some other quantified indicator.
2. The **reusability hierarchy** is a hierarchical decomposition of the broad concept of software reusability (of a C++ class) into more measurable traits such as complexity of the source code, etc. It can be viewed as a list of some of the measurable traits of a C++ class.
3. A **reusability indicator (evaluator)** is a subjective measure of the reusability of a C++ class. Five reusability indicators are used by the expert subjects of Experiment Three to gauge the relative reusability of a set of C++ classes.

These three definitions all play an integral part of the research presented under goal four. These terms are discussed in more detail in Chapter 7.

3.5 Summary

This chapter outlines the course of this research. The many terms of this research are rigorously defined. Each of the experiments of this research is outlined and the terms necessary for understanding this research effort are presented. The next four chapters, Chapters 4 through 7 describe each of the research goals of this effort respectively.

Chapter 4

■ Effects of Software Reuse in the Object-Oriented Paradigm

Reusing software has been shown to increase programmer productivity [Frak94b, Joos94, Lewi92]. Yet, clearly there must be tradeoffs, hidden costs, and maybe unrealized benefits. [Trac90b] showed that in order for any gain to be realized, software must be designed for reuse and reused at least two more times to receive any benefit from the process. Others have shown that reuse increases reliability and maintainability [LimW94, Scha94]. The object-oriented paradigm is designed to aid reusability of software. C++ classes are meant to be modular extensible units from which programs can be built.

The previous chapters outline some of the costs and benefits of reusing software and of using the object-oriented paradigm. This chapter focuses on goal four of this research: on some of the specific effects of reusing general purpose C++ classes such as abstract data type classes. Experiment One is conducted to gather empirical data on the effects of software reuse of C++ classes on software development in the object-oriented paradigm. In particular, the following questions are answered with respect to the object-oriented paradigm.

1. How does reusing C++ classes affect the total time required during the various phases of the software development life cycle (design, implementation, integration)?
2. How does reusing C++ classes affect the integration process (i.e., the number of compiles, runs, and major errors)?
3. How does reusing C++ classes affect the final system's reliability?

These questions are answered through a carefully designed and controlled experiment performed in an academic setting.

4.1 Design and Setup

This experiment involves a class of thirty Computer Science seniors. All of the participants voluntarily sign an Informed Consent form (Appendix J). The class is divided into ten teams of three programmers each. Each team produces a software system that uses various abstract data type classes. Some of these classes are obtained from class libraries and some are written from scratch. Measures are gathered on how well each project's development progressed and on the quality of the final systems.

This experiment is conducted in a senior-level Software Engineering course, which is divided into two parts. Part one consists of a lecture series discussing the principles and landmark research in the field of Software Engineering. For a complete description of the lectures, see [Henr83]. Part two consists of a lab in which the details of a class project are given and lectures are presented dealing directly with the project. Data for this research experiment comes from the class projects developed during the Software Engineering lab. Experiment One, described below, occurs over the course of a fifteen week class meeting five times a week for approximately one hour each meeting. Additionally, the programmers' average out of class time was just over seven hours per week.

The first part of this experiment is to teach the students object-oriented analysis and design. This is difficult, but solvable task in a semester course [Ante90, McKi93]. After that, the students are educated about the available reusable components in the C++ libraries which are given to them. The libraries provided to them are libraries of general purpose C++ classes such as stacks, queues, lists, random number generators, and strings. A library of graphics classes is also provided to allow the students to develop a graphic user interface for their projects without too much low-level programming.

Each team is required to design a medium size software system (2000-4000 lines of code). All the teams practice object-oriented design and coding in C++. Several libraries of reusable objects and routines are at the teams disposal, but the teams are not forced to use

any objects that they do not wish to use. Each team designs their own project and then "hires" other students to do the actual coding for them. Each team hires a minimum of five other students as coders. The design teams then integrate the various components into a working system. At this point (the integration phase), the design teams are allowed to write some connective code. It is important to remember that the systems are designed by each team of three, but written by five subjects not on the design team. This makes for a potentially difficult integration phase.

After the systems are finished, they are presented to the professor and graduate assistant for user testing and analysis. Summarizing, the steps of Experiment One are:

-
1. The class is taught the fundamentals of Software Engineering.
 2. The class is taught the principles of object-oriented analysis and design.
 3. The class is taught the fundamentals of software reuse.
 4. The class is educated about the C++ libraries of reusable components.
 5. The class is divided into ten groups of three students (designers) each.
 6. Object-oriented designs are created.
 7. Each design team hires five or more programmers.
 8. The programmers write / reuse code to fulfill design requirements.
 9. The design teams integrate the components into a final working system.
 10. The systems are extensively tested and evaluated by the instructors.
-

4.2 Experimental Data

Along the course of this experiment, four data sets are collected for statistical analysis. The first data set relates to the students. Thirty-one characteristics (described in Appendix

A) are gathered on each student through an initial questionnaire (Appendix B.1). Chapter 6 presents the analysis of this programmer data. Pertaining to this goal, this data set is used to divide the students into statistically equal design teams. Doing this equal division allows for valid project to project comparisons.

The second set of data consists of data gathered directly from the code developed for the projects. This data set is gathered using the Metrics Generator described in Chapter 2. Each of the projects is analyzed using current software complexity metrics to ensure that the projects are almost equal and to provide data about other possible causes of any trends that are found. The data generated from the project code can be classified in several distinct ways. One classification is to divide the code along project boundaries for a total of ten observations. A second scheme is to separate the code by programmer for a total of thirty observations. Since there are ten projects of an average of 3000 lines each, each programmer is the author of about $10 * 3000 / 30 = 1000$ lines of code.

The third data set involves the development statistics for each of the projects. Data such as total development time, integration time, and final evaluation (the grade) are gathered. These development statistics are correlated to final code quality, percent reusability, and final code complexity.

The last data set is obtained from the final questionnaire (Appendix B.6), which accesses each subject's opinions and exposure to the idea of software development and software reuse. This poll gathers opinions on how using the object-oriented paradigm aided (or hindered) the software development process and queries the programmers on the effectiveness of software (C++ class) reuse. The students are polled about the what make a C++ class reusable as well. This information is used to achieve goal four. These opinions are presented in **Chapter 7: Reusability and Characteristics of C++ Classes**.

After the empirical study has concluded, a statistical analysis is performed to answer the research questions relating to this goal. The table below shows the four general databases that are collected during this research experiment.

Table 4.1. Data Sets Collected During Experiment One

Number	Data Set
1	Programmer Data
2	Code Metrics Data
3	Development Data
4	Final Questionnaire Data

Each of the data sets consists of many pieces of data. Furthermore, some of the data sets can be viewed as having several different internal structures. For example, the code metrics data set can be examined at the project level and correlated with the development statistics for each project (goal one). Or, this data set can be divided along programmer boundaries to look for correlations between programmer data and reuse performed (goal three).

The data sets and their constituent data items are described below.

A. Programmer Data

This data set is described in greater detail in Appendix A. It consists of thirty-one skills and experience areas relating to programming in general, the object-oriented paradigm, and software reuse. The primary reason for gathering this data set is to achieve the third goal of this research (**Chapter 6: Programmer Experience and Software Reuse**).

B. Code metrics: (By Author, By Project) Gathered by the Metrics Generator

1. Source code size and complexity metrics (Lines of Code, McCabe's Cyclomatic complexity, some relatively new object-oriented metrics [Chid91, Chid94, LiWe93], and some new metrics created within this research)
2. Percent reuse for the system (number of classes reused / total number of classes in system)
3. Percent black box reuse (number of black box classes reused / total number of classes in system)
4. Percent white box reuse (number of white box classes reused / total number of classes in system)

This data set is described in greater detail below. Each of these data items is more rigorously defined.

C. Development Statistics: (By Project)

1. Time Spent for Design
2. Time Spent for Implementation (Coding)
3. Time Spent in C++ Libraries
4. Time Spent for Integration
5. Total Development Time
6. Number of Major Integration Errors
7. Number of Integration Compiles
8. Number of Integration Runs
9. Number of Errors in Final System

Again, this data set is elaborated upon later in this chapter.

D. Final Questionnaire: (By Programmer)

1. Opinions on software reusability and its effect on various phases of the life cycle
2. Opinions on the libraries of C++ classes
3. Opinions on difficulties of reuse (white box reuse, black box reuse, no reuse)
4. Opinions on what makes a C++ class reusable

The fourth data set provides for a subjective actual experience-based evaluation on software reuse in the object-oriented paradigm and its effects on software development. The data set is primarily discussed in Chapter 7.

The first and last data sets are gathered through two extensive questionnaires, one at the onset of the study (Appendix B.1) and one at its conclusion (Appendix B.6). The Metrics Generator collects all of the source code metrics found in the second data set. The difficult data set to gather is the third data set, the development data set. For these items, the experiment participants had to track their activities throughout the software life cycle. Honesty is emphasized and final course grades are not affected by the amount of development effort required. It is known that some students require longer than others to do the same tasks. A series of questionnaires and forms (Appendix B.2 through B.5) are provided to the subjects to track their project development.

4.3 Results

Although ten projects were begun, one design team failed to generate any working code. This is one of the pitfalls of conducting experiments in academia. The members of this design team had personality conflicts and lacked the expertise to resolve them. This project had no working code, so the project and its corresponding data were removed. The summarized raw data collected in each of the four data sets explained above is shown in Appendix C. The remainder of this presents and explains the variables (data items) that are needed for the analysis that follows.

The three research questions posed under this goal can be answered from this experiment's data sets. This research examines nine variables or development process indicators to answer these questions. Table 4.2 summarizes these nine indicators (the columns of the table) for the nine successful projects (the rows of the table) of this experiment. These nine indicators are correlated with the amount of reuse performed in each project and the

code-based metrics for the project. A multiple linear regression model is developed for each of the indicator variables.

Table 4.2. Development Process Indicators (Dependent Variables) for the Nine Projects

Project Number	Design Time (Hours)	Coding Time (Hours)	Library Time (Hours)	Integration Time (Hours)	Total Time (Hours)	Integration Errors (Number)	Integration Compiles (Number)	Integration Runs (Number)	Final Errors (Number)
1	65	226	45	132	468	9	485	420	3
2	72	301	11	180	564	7	578	493	7
3	86	240	20	241	587	8	1,085	951	10
4	94	232	24	250	600	15	1,231	1,108	12
5	71	211	29	216	527	13	591	544	5
6	81	235	31	193	540	8	749	628	6
7	82	217	18	239	556	12	652	604	5
8	64	204	65	156	489	5	527	580	2
9	90	239	19	223	571	8	981	1,090	6
Ave.	78	234	29	203	545	9.4	764	713	6.2
Std. Dev.	10	27	15.7	39	41	3	254	248	3

The three questions presented at the beginning of this chapter are now reviewed along with which process indicators (dependent variables) are used to answer each of the questions.

Question 1: How does reusing C++ classes affect the total time required during the various phases of the software development life cycle?

This first question is answered using the five time-based development process indicators listed below.

- ◆ **Total Design Time (hours)** is the total time spent by each three-person team to design the entire project.

- ◆ **Total Implementation (Coding) Time (hours)** is
the time spent by the hired coders to implement the source code for the project.
- ◆ **Total Library Time (hours)** is
the total time spent by the designers and the coders examining the C++ class libraries and their documentation.
- ◆ **Total Integration Time (hours)** is
the time spent integrating the components (classes) of the system that are written by the hired programmers. This indicator includes the testing time for the projects as well.
- ◆ **Total Time (hours)** is
the total time spent developing the project including design, coding, integration, and testing as well as any time spent learning C++ libraries. This variable is the sum of the previous four variables.

Question 2: How does reusing C++ classes affect the integration process?

The second question addresses the integration stage of the development process. The design teams received their modules back from the hired programmers and are now required to integrate the pieces into a working system. As they performed this task, they recorded the following process measures.

- ◆ **Total Number of Major Integration Errors** is
the number of major errors incurred while the design team was integrating the various components of the system. Errors like mismatched interfaces or class definitions fall into this category.

- ◆ **Total Number of Integration Compiles** is

the number of compiles performed by the three-person design team as they integrated their system.

- ◆ **Total Number of Integration Runs** is

the number of executions performed by the design team as they integrated their system.

The last question posed at the beginning of the chapter deals with the reliability of the final system. This question is answered using the last of the nine process measures.

Question 3: How does reusing C++ Classes affect the final system's reliability?

- ◆ **Total Number of Final Errors** is

the number of major errors found in the systems after they were submitted as finished products.

These nine variables are selected because they represent some of the areas of the software development process where reuse would logically make an impact. In the sections below, each of these variables is correlated to the percent of reuse occurring in each of the nine projects. The percent reuse is calculated as the percent of classes that are reused (either black box or white box) divided by the total number of classes in the system. This value is chosen because it is easily calculated and can be ported to other object-oriented languages readily.

For each of the nine projects, some size and code complexity data are gathered as well. This research experiment is designed to bring out the effects of software reuse while keeping the code size and complexity approximately constant. If this criterion is achieved, correlations involving code size and complexity are minimized and correlations involving amount of reuse should be more easily discovered.

For each of the nine completed projects, the C++ class reuse data is gathered. Table 4.3 presents this data. The columns of the table represent the data items gathered while the rows represent the nine projects of this experiment (observations). Columns B through E shown the absolute number of classes used by each project in several categories. Column B (*Number of Black Box Classes*) represents the number of classes in the system taken from the C++ class libraries and not modified (black box reuse). Column C (*Number of White Box Classes*) shows the number of classes in the project taken from the class libraries and modified before being reused (white box reuse). Typically, this modification was to use the library class as a parent class and derive a new class to use. Column D (*Number of Ind. Classes*) is the number of classes in the project written independently (from scratch), and column E (*Total Classes in the System*) is the sum of columns B, C, and D. Columns F, G, and H are percentages based on columns B, C, and D respectively.

The last two rows of Table 4.3 are the average and standard deviation values for each of the columns. They are provided for completeness and are not used in the analysis.

Table 4.3. Reuse Data for the Nine Projects of Experiment One

A	B	C	D	E	F	G	H
<i>Project Number</i>	<i>Number of Black Box Classes</i>	<i>Number of White Box Classes</i>	<i>Number of Ind. Classes</i>	<i>Total Classes in System</i>	<i>Black Box Reuse %</i>	<i>White Box Reuse %</i>	<i>Independ. Class %</i>
PNUM	BBC	WBC	IDC	TC	BBP	WBP	IDP
1	15	3	16	34	44.1	8.8	47.1
2	6	1	23	30	20	3.3	76.7
3	5	2	39	46	10.9	4.3	84.8
4	3	6	27	36	8.3	16.7	75
5	7	2	23	28	25	7.1	82.1
6	10	1	31	42	23.8	2.4	73.8
7	6	3	19	28	21.4	10.7	67.9
8	12	0	30	42	28.6	0	71.4
9	6	2	23	31	19.4	6.5	74.2
Average	7.8	2.2	25.7	35.2	22.4	6.7	72.5
Std Dev.	3.8	1.7	6.9	6.7	10.4	5	10.9

Two of the columns from Table 4.3, *Black Box Reuse Percentage* and *White Box Reuse Percentage* represent two of the fifteen independent variables used to develop the nine process indicator models. The remainder of the columns in Table 4.3 are linear combinations these two columns and are not discussed further.

More formally, the two independent variables selected from Table 4.3 are:

1. **Black Box Class Percentage (BBP)** is the number of classes leveraged directly from an object-oriented library and not changed in any way before being reused in the system. This variable is the *Number of Black Box Classes / Total Number of Classes in the System*.
2. **White Box Class Percentage (WBP)** is the number of classes based on a class (derived from) found in one of the C++ libraries. Typically, this means inheritance is performed. This variable is the *Number of White Box Classes / Total Number of Classes in the System*.

Along with these reuse percentages, some source code metrics are gathered on each of the projects to check for correlations between development data and project code complexity. These data items are divided into two groups: size metrics and complexity metrics. All of these metrics are defined in Chapter 3. The metrics are used as independent variables in the analysis to account for any difference in the projects in terms of system size or complexity. The experimental design is meant to minimize these effects; however, there are some size and complexity differences among the nine projects. Using these metrics as independent variables accounts for these small differences. The six gathered project size metrics are:

1. Total Classes in the System (TC)
2. Average Lines of Code Per Class (LPC)
3. Average Functions Per Class (FPC)
4. Average Lines of Code Per Method (LPM)
5. Average Size One (SZ1- Semicolons Per Class)
6. Average Size Two (SZ2 - Local Data + Number of Methods)

Table 4.4 presents these six size metrics for the nine projects. All of these metrics are automatically gathered by the Metrics Generator. Column B represents the total classes in the system written by the subjects of this experiment. Therefore, the number of black box classes has been removed. Columns C through G are five size metrics that have been defined in Chapter 3. The third row of Table 4.4 represents the code or acronym used to refer to each of these metrics. Again, the averages and standard deviations for these variables are shown for completeness.

Table 4.4. Size Metrics for the Nine Projects

A	B	C	D	E	F	G
Project Number	Total Classes (Excl. BB)	Ave. LOC / Class	Ave. Functions / Class	Ave. LOC / Method	Ave. SZ1 / Class	Ave. SZ2 / Class
PNUM	TCS	LPC	FPC	LPM	SZ1	SZ2
1	19	180	7.26	24.8	154	13.3
2	24	76	7.25	10.5	81	10.2
3	41	61	3.76	16.3	53	5.2
4	33	57	6.76	8.4	42	9.6
5	25	157	4.36	36.1	156	7.9
6	32	136	5.72	23.8	126	13.3
7	22	124	10.95	11.3	115	16.7
8	30	68	6.6	10.3	94	8.8
9	25	109	5.32	20.4	111	4.8
Average	27.9	108	6.4	18	103	10
Std. Dev.	6.8	44.9	2.1	9.2	40.2	3.9

The last set of independent variables gathered on the nine projects are the object-oriented complexity metrics based on the research of Chidamber and Kemerer [Chid91, Chid94] and Li and Henry [LiWe93]. These complexity metrics are also defined in Chapter 3. Table 4.5 shows the project values for these seven metrics. All of these metrics are automatically generated by the Metrics Generator developed for this research. Again, the third row represents the acronym used to refer to that particular complexity metric.

Table 4.5. Object-Oriented Complexity Metrics for the Nine Projects

A	B	C	D	E	F	G	H
Project Number	Ave. Number of Children	McCabe Weighted Methods Per Class	Ave. Depth in Inheritance Tree	Ave. Response for a Class	Ave. Data Abstract. Coupling	Ave. Message Passing Coupling	Ave. Lack of Cohesion Methods
PNUM	NOC	WMC	DIT	RFC	DAC	MPC	LCM
1	0.2	22.7	0.6	44.4	0.2	4.2	1.5
2	0.4	24.5	0.4	61.3	0.2	3.7	2.3
3	0.4	15	0.5	30.4	0.1	2.3	1.4
4	0.5	17.1	0.9	22.9	0	3.5	1.2
5	0.1	37.9	0.5	42.1	0.1	5	2
6	0.1	32.8	0.3	65.7	0.1	5.6	1.1
7	0.1	43.7	0.6	72.5	0	13.1	1.3
8	0.5	18.5	1.1	35.3	0.1	1.3	1.1
9	0.3	45.8	0.4	33.5	0	10.5	1.5
Average	0.3	28.6	0.6	45.3	0.1	5.5	1.5
Std. Dev.	0.2	11.7	0.3	17.3	0.1	3.9	0.4

The fifteen measures selected from the preceding three tables (Table 4.3 - two reuse variables, Table 4.4 - six size metrics variables, and Table 4.5 - seven complexity metrics variables) are the independent variables that are to be correlated to the nine software development process indicators presented earlier. A model is created to explain each of the nine process indicators in terms of the fifteen variables selected.

4.4 Analysis

With this gathered data, models are found that explain the nine dependent variables. These models are simple or multiple linear regression models. Each model quantitatively

captures the relationship between one of the dependent variables and one or more of the independent variables. For example, the equation

$$\mathbf{[Number\ of\ Integration\ Runs] = 1141 - 19.1 * [BBP]}$$

is a simple model that correlates the *Number of Integration Runs* performed during the project integration with the *Percentage of Black Box Reuse* realized by the project. It states that for every percentage point decrease in black box reuse, the *Total Number of Integration Runs* increases by 19.1. The initial constant (1141) establishes a baseline (intercept) for the data set. For the purposes of this research, the initial constants are presented, but ignored. For a detailed explanation of simple and multiple linear regression, please consult [OttR93].

For this analysis, models (regressions) that explain the trend in the dependent variable in terms of the two reuse variables (*Black Box Reuse Percentage* and *White Box Reuse Percentage*) are desired. This research experiment is designed to eliminate project differences such as size and complexity as much as possible. If the trends in the dependent variables can be explained using the size and complexity metrics for the projects, then this experiment was not designed properly or the trends in the development process can be explained by project differences other than amount of code reuse. Clearly, the nine projects are not completely equal, so the project size and complexity metric variables are expected to explain some percentage of the trends in the process measures, but not significant amounts.

For each of the nine dependent variables, a multiple linear regression is performed using all of the independent variables adding them according to the forward selection process (stepwise regression). As each variable is added, the model is examined and the process stops when enough of the trend in the dependent variable is explained or when none of the remaining variables explain a significant amount of the trend beyond what the model

already explains. When no more variables can be entered, the selection process stops, and the final model is output.

The last step in choosing the model is to perform all possible combinations of the variables that are selected by the stepwise regression process and to select the best model from this set. This selection is performed by using measures of the "goodness of fit" for each model to determine how well each model captures the relationship between the process indicator and the selected independent variables. Two such measures are given with each of these developed models, the **R squared (R^2)** measure and the **adjusted R squared (AR^2)** measure. The R^2 value measures the percentage of the trend explained by the model. An R^2 value of 0.85 states that 85% of the trend in the dependent variable (the process indicator) is explained by the given set of independent variables (the model).

The adjusted R squared value³ can be interpreted as the relative "goodness" of the model in relation to other models not selected. The adjusted R^2 value is used because the R^2 value can always be improved by adding more terms to the model. For this research, R^2 values of 1.000 can always be achieved because the number of independent variables exceeds the number of data points. For this research, the "best" model is the simplest one with the highest adjusted R^2 value. For the purposes of this research, simple models have three or less terms.

Model selection is a difficult process. The adjusted R^2 value is used to gauge how good the model is, but it must be balanced by the model's simplicity. Models with seven or eight independent variables cannot be used with this nine data point sample. Simpler (less terms) models are sought.

³ The adjusted R^2 value is calculated with the following statistical formula:

$$AR^2 = 1 - (SSE / SST) * (total\ d.f. / error\ d.f.)\ where$$

SSE = Error Sum of Squares,

SST = Total Sum of Squares, and

d.f. = degrees of freedom.

R^2 and adjusted R^2 values over 0.90 are very significant. For experiments involving human subjects or other factors with high variability, R^2 values as low as 0.60 can be viewed as significant. The nine models presented below have R^2 values between 0.57 and 0.99.

After the best (and simplest) model is found for each of the process measures, the variables contained within the model are examined to see how much of the trend in the process measure each component of the model explains. The reuse independent variables should be explaining most of the trend for the process measures (dependent variables) with the size and complexity metrics variables playing minor roles.

The next nine subsections address each of the nine software development process indicators in turn.

4.4.1 Total Design Time

The most reasonable model generated by the selection process contains only two terms, *Black Box Reuse Percentage* (BBP) and *Average Message Passing Coupling* (MPC).

The model for the design time dependent variable is

$$\mathbf{DTIM = 91.6 - 0.83 * BBP + 0.97 * MPC}$$

Total Regression Explained (R^2)= 0.77

Adjusted $R^2 = 0.69$

BBP explains 84% of regression explained by the model.

MPC explains 16% of regression explained by the model.

This model explains 77% of the trend in the design time dependent variable (from the R^2 value). The *Black Box Reuse Percentage* (BBP) independent variable explains 84% of that.⁴ Clearly, the percent of black box reuse realized by each project affects the overall

⁴ This percentage is calculated using the Type III sum of squares for all of the variables in the model. More precisely, it is the percentage of the model regression explained by this variable assuming all of the other variables are already in the model.

design time. Since the *Black Box Reuse Percentage* variable has a negative coefficient (-0.83), as the amount of black box reuse declines, the total design time increases.

Project ideas are submitted to and approved by the professor of the course who is quite experienced at what types of projects can be completed in a fifteen week semester. This experiment is designed so that each of the ten projects is approximately the same size and complexity. Given this fact, a project that reuses more should have a reduced design time.

The other model variable is a class complexity measure, the *Average Message Passing Coupling* (MPC). As the amount of coupling among the classes increases (the coefficient of MPC is 0.98), the design time increases. As the classes become more intertwined, the effort required to design them goes up. However, it should be noted that this variable only accounts for 16% of the explained trend. The other independent variables are not in this model since none of them meet the entrance criteria explained above.

Figure 4.1 shows how well the actual design time and the design time obtained from this model correlate by plotting the actual design time versus the modeled design time. The black triangles on this figure represent the points (modeled design time, actual design time) for the nine projects. The diagonal line represents the model.

Figure 4.1 shows the goodness of fit for the design time model. It is a type of residual plot. The next eight models have similar figures depicting them.

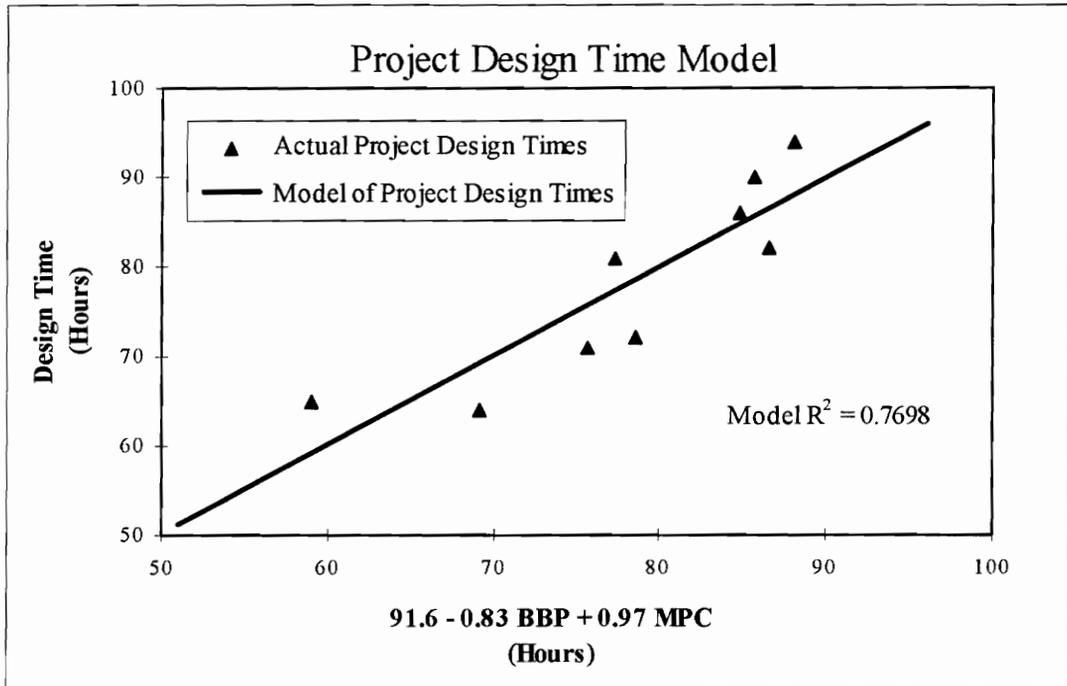


Figure 4.1. Design Time Model

4.4.2 Total Implementation (Coding) Time

For the amount of coding time, the best model is

$$\text{CTIM} = 163.7 + 4.2 * \text{SZ2} + 201.3 * \text{NOC} - 115.3 * \text{DIT} + 25.8 * \text{LCM}$$

Total Regression Explained (R^2)= 0.99

Adjusted $R^2 = 0.99$

SZ2 explains 15% of regression explained by the model.

NOC explains 41% of regression explained by the model.

DIT explains 36% of regression explained by the model.

LCM explains 8% of regression explained by the model.

This model has four terms that together explain 99% of the regression ($R^2 = 0.99$). This incredible fit can be partially attributed to the small number of data points and the fact that the model has four terms in it. However, the adjusted R^2 is also 0.99. This is a good model for this data set. Examining the model variables and their corresponding

coefficients, over a third of the regression is explained by the *Average Depth in the Inheritance Tree* (DIT). Its coefficient is -115.3 signifying that as the systems' class hierarchies become shallower, the coding time increases. For these nine systems, this is expected. If the systems' "reused" code is in the form of a class hierarchy, less code is needed to complete the system. A check on this is to examine the correlation between project size (KLOC) and the depth in tree (DIT) measure. These two measures have a correlation value of -0.52 which is significant at the 0.10 level. These two measures correlate negatively meaning that as the *Average Depth in the Inheritance Tree* (DIT) decreases, the *Total Lines of Code* (KLOC) in the project increases. The model above predicts this correlation.

The *Average Number of Children Per Class* (NOC) explains 41% of the trend in the coding time variable. A positive coefficient says that as the *Average Number of Children Per Class* increases, so does the overall coding time. This indicates that a narrow (and deep, from the DIT term) class hierarchy results in shorter implementation times for these software projects.

8% is explained by the *Lack of Cohesion of Methods* (LCM) metric. As the system classes become less cohesive (the lack of cohesion increases), the time to code the system (CTIM) also increases. As expected, a system made up of cohesive classes should reduce overall coding time.

The last term of the model explains about 15% of the coding time trend. With a positive coefficient, as the *Size Two* (SZ2) metrics increases, coding time increases. *Size Two* (SZ2) is a size metric measuring the total amount of data plus local methods per class. As the *Size Two* measure increases, the average class is larger and more complex. This term suggests that systems of a few large classes are harder to implement than a more modular system consisting of a many small classes. Figure 4.2 shows the actual and model implementation times for the nine projects.

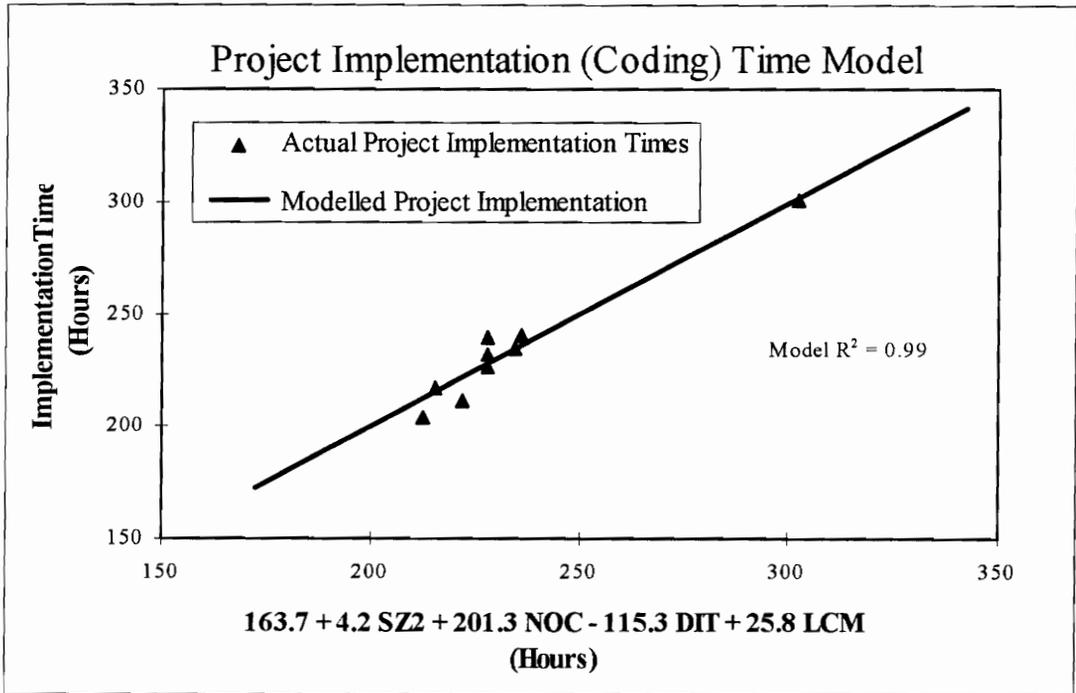


Figure 4.2. Implementation (Coding) Time Model

Interestingly, neither of the two reuse independent variables are in this model stating that the amount of reuse realized did not affect the amount of coding time for the projects. At first glance, this seems backwards. The projects with more reuse should have had less coding time. However, the projects did vary somewhat in size and complexity and the trend in the projects' sizes and complexities correlates better than the percentage of reuse to the amount of time spent implementing the project. The previous model and several subsequent ones better show where the time savings occurs when reuse is involved.

4.4.3 Total Library Time

Intuitively, the dependent variable, *Total Time Spent Examining / Learning the C++ Libraries* should have a model with both types of reuse in it. The selection process for

selecting the terms for the model generates the following model.

$$\text{LTIM} = -4.44 + 1.05 * \text{BBP} - 2.66 * \text{FPC} + 46.22 * \text{DIT}$$

Total Regression Explained (R^2)= 0.88

Adjusted R^2 = 0.81

BBP explains 41% of regression explained by the model.

FPC explains 49% of regression explained by the model.

DIT explains 10% of regression explained by the model.

Overall, the model explains 88% of the trend in the time spent in the C++ libraries. Only one of the two reuse variables appears in the model, *Black Box Reuse Percentage* (BBP). As the percent of black box reuse increases, so does the time spent in the C++ libraries. The *Depth in the Inheritance Tree* (DIT) term accounts for the other reuse variable, *White Box Reuse Percentage* (WBP). As average depth of a class increases (which includes the reused white box classes), the library time increases. Since most of the derived class have a depth of at least one and usually three or four, this explains why the *Depth in Inheritance Tree* (DIT) appears in the model and the *White Box Reuse Percentage* (WBP) term does not.

As the last term of the model, *Average Functions Per Class* (FPC) decreases, the total time spent in the C++ libraries goes up. One possible explanation for this negative correlation is that as the number of functions in a class decreases, there are more classes that could potentially interact with the classes in the class libraries. Since the projects are written by many programmers, potentially more programmers must spend time learning the C++ libraries. Figure 4.3 shows this model and how it compares against the actual library times.

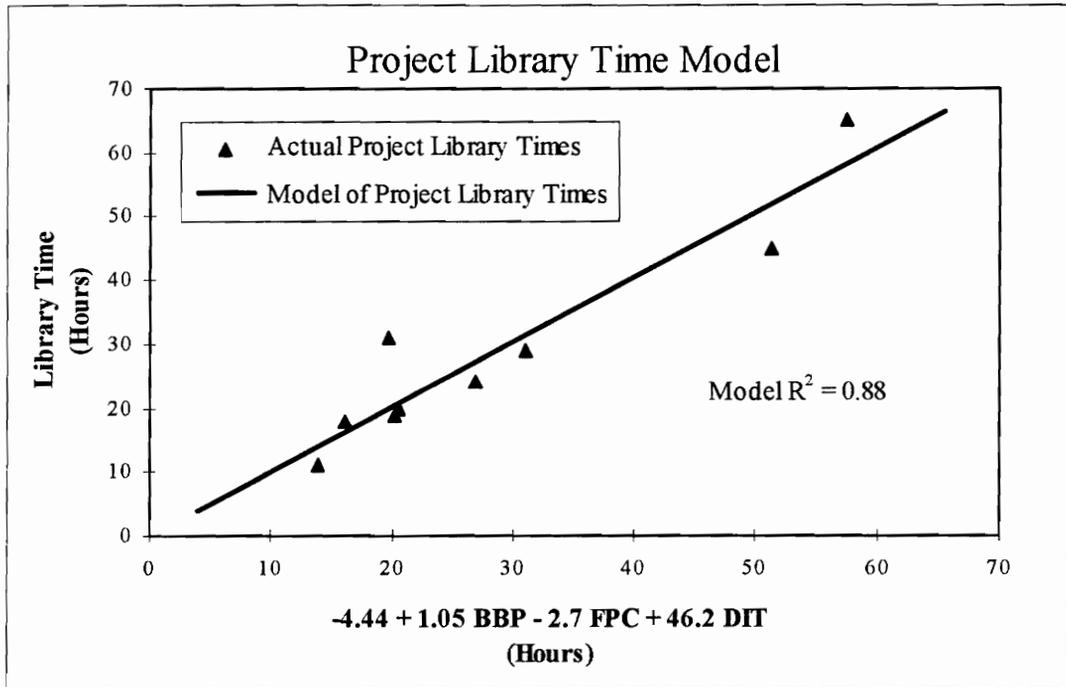


Figure 4.3. Library Time Model

4.4.4 Total Integration Time

The integration phase of the projects is likely to be affected by the complexity measures, the size measures, and also the reuse measures. Since all of the projects have classes from at least five programmers plus some number of library classes, the integration of the projects is a large task. The model explaining this trend is:

$$ITIM = 223.3 - 5.4 * BBP + 39.8 * DIT + 2.2 * MPC + 0.6 * LPC$$

Total Regression Explained (R^2)= 0.97

Adjusted $R^2 = 0.94$

BBP explains 76% of regression explained by the model.

DIT explains 6% of regression explained by the model.

MPC explains 4% of regression explained by the model.

LPC explains 14% of regression explained by the model.

Most of the trend in integration time is explained by the *Black Box Reuse Percentage* (BBP). As the amount of reuse increases, the integration time decreases. The library

classes are designed for reuse; therefore, they are easier to integrate into a system than classes not written with ease of (re)use in mind. Figure 4.4 shows the *Total Integration Time Model*.

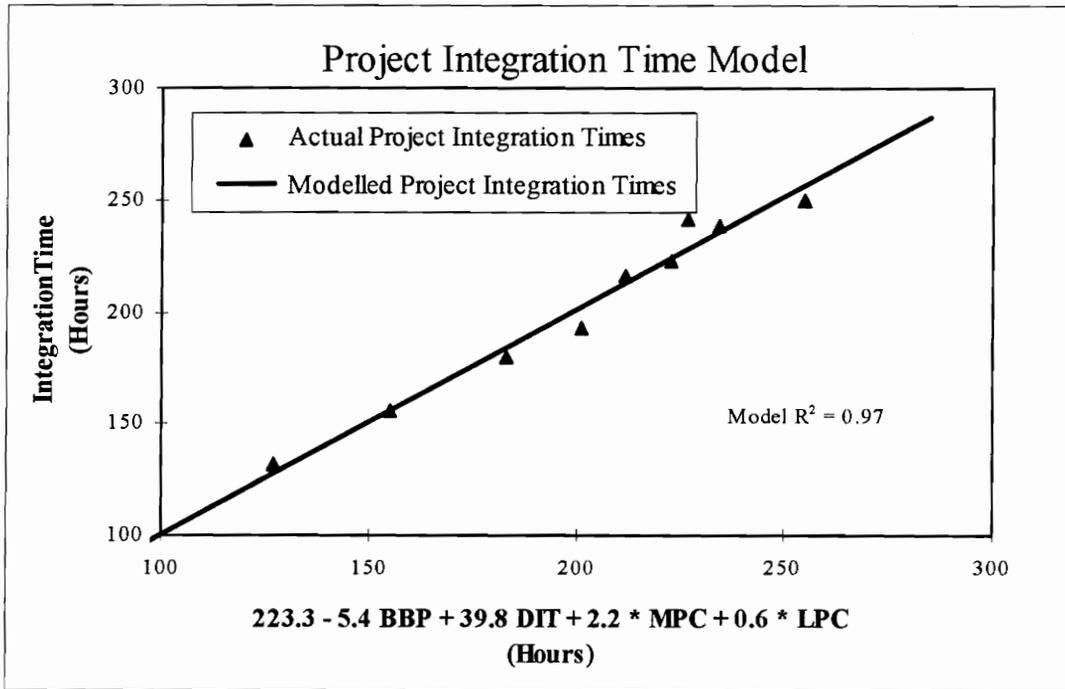


Figure 4.4. Integration Time Model

The next most significant term is the *Lines Per Class* (LPC) term, which explains 14% of the regression explained by the model. As the size of the classes increases, the difficulty in integrating them increases as well. Smaller classes tend to integrate more smoothly.

4.4.5 Total Time for Whole Project Development

This development process measure is merely the sum of the previous four. It is expected that the model for this indicator will have some of the terms from the previous four models. One model for this variable is to add up the previous four models; however, this gives a model with eight terms which is far too many for a data set containing only nine observations. So, using the forward selection process, a new model is found. Clearly, the

total development for a project should depend on the project size and complexity and reuse level. For this variable, the model turns out to be:

$$\text{TTIM} = 664.4 - 3.8 * \text{BBP} + 1.57 * \text{WBP} - 43.0 * \text{DIT}$$

Total Regression Explained (R^2) = 0.98

Adjusted R^2 = 0.97

BBP explains 89% of regression explained by the model.

WBP explains 4% of regression explained by the model.

DIT explains 7% of regression explained by the model.

The overriding term explaining the trend in the *Total Development Time* (TTIM) variable is the amount of black box reuse. As the amount of black box reuse increases, the total development time decreases. This is expected since the *Black Box Reuse Percentage* (BBP) term appears in three of the four models above. For the reasons given above, the amount of reuse lessens effort in the design phase and the integration phase. Since the total time encompasses these partial times, total development time should decrease as well.

The *White Box Reuse Percentage* (WBP) term of the model explains so little as to be insignificant. However, the coefficient of the white box percent reuse does stand out. Since it is positive, the more white box reuse that occurs increases the total development time. The next chapter delves deeper into the effects of the two types of reuse performed in this experiment.

The last term of the model, the *Average Depth in Inheritance Tree* (DIT), suggests that as the system class hierarchy grows deeper, the total development effort is reduced. Inheritance forces reuse of code since classes in the same hierarchy share code with their ancestors in the hierarchy. Furthermore, the overhead involved in writing more shallower classes should increase the total development effort. Figure 4.5 depicts the total development time model.

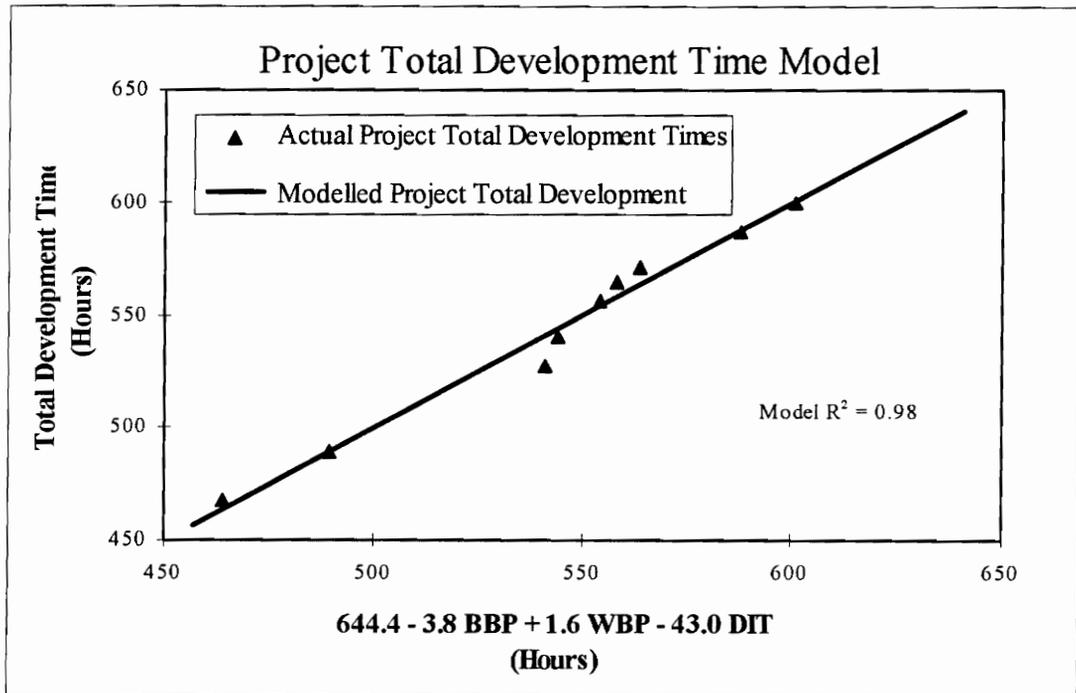


Figure 4.5. Total Development Time Model

4.4.6 Number of Integration Errors

The next three sub sections deal with the project integration phase. This phase is a large part of the project development for this experiment.

The number of major integration errors is counted. Major errors include mismatched interfaces, name conflicts, redesigned modules, and other integration problems. The model explaining this dependent variable is:

$$\mathbf{IERR = 5.9 - 0.15 * BBP + 0.47 * WBP + 0.03 * LPC}$$

Total Regression Explained (R²)= 0.88

Adjusted R² = 0.81

BBP explains 14% of regression explained by the model.

WBP explains 72% of regression explained by the model.

LPC explains 14% of regression explained by the model.

Most of the trend in integration errors is explained by the *White Box Reuse Percentage* (WBP) component, 72%. As *White Box Reuse Percentage* increases, so does the number of integration errors. Inheritance, although useful, can be damaging as well. Certainly, the subjects of this experiment are new to inheritance and object-oriented programming. With marginal experience, the inheritance construct proved too complex. The next chapter delves into this result in more detail.

The black box reuse component (BBP), explaining 14% negatively correlates with the number of integration errors. More reuse means fewer errors. Since these classes are unmodified and reliable, this term is expected. The library classes are easier to integrate because they have a known (good) reliability. Projects using a higher percentage of black box reuse classes suffer fewer integration errors.

The last term of the model, *Average Lines of Code Per Class* (LPC), also explains 14% of the trend that the model explains. As the lines per class increases, so do the integration errors. Larger classes create more integration errors due to their increased size and complexity. The actual and modeled integration errors are shown in Figure 4.6.

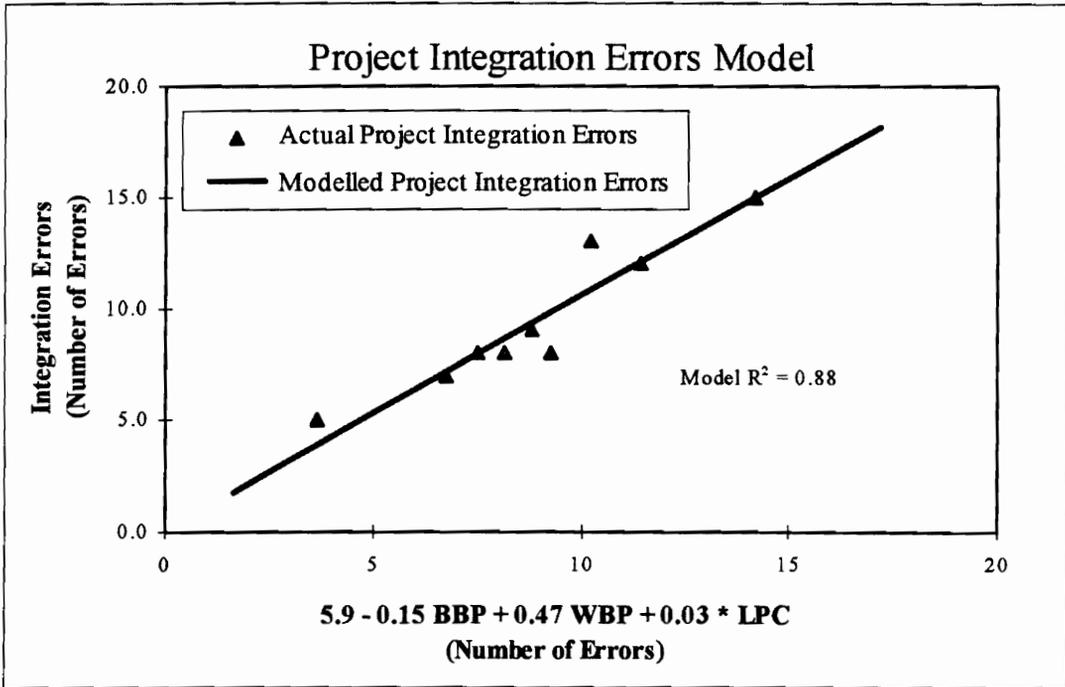


Figure 4.6. Integration Errors Model

4.4.7 Number of Integration Compiles

This dependent variable generated a model explaining the trend in the number of compiles performed during the integration phase. The model is:

$$\text{ICOM} = 1440 - 18.6 * \text{BBP} - 5.7 * \text{RFC}$$

Total Regression Explained (R^2) = 0.79

Adjusted $R^2 = 0.73$

BBP explains 80% of regression explained by the model.

RFC explains 20% of regression explained by the model.

The model explains 79% of the trend in the dependent variable and 80% of that is explained by the amount of black box reuse (BBP). As the *Percentage of Black Box Reuse* increases, the number of compiles performed during integration decreases. The more pre-compiled classes there are in the system, the less compiling that needs to be done.

The second term of the model, *Average Response for a Class (RFC)* has a negative coefficient. This needs some explaining. As the *Average Response for a Class (RFC)* decreases (less calls are made to non-local methods), then the number of separately compilable units increases which tends to increase the overall number of compiles performed during the integration of the system. If the whole system were one module, fewer compiles would be required, because all of the compiler errors could be found at one time. Figure 4.7 depicts the *Total Number of Integration Compiles* model.

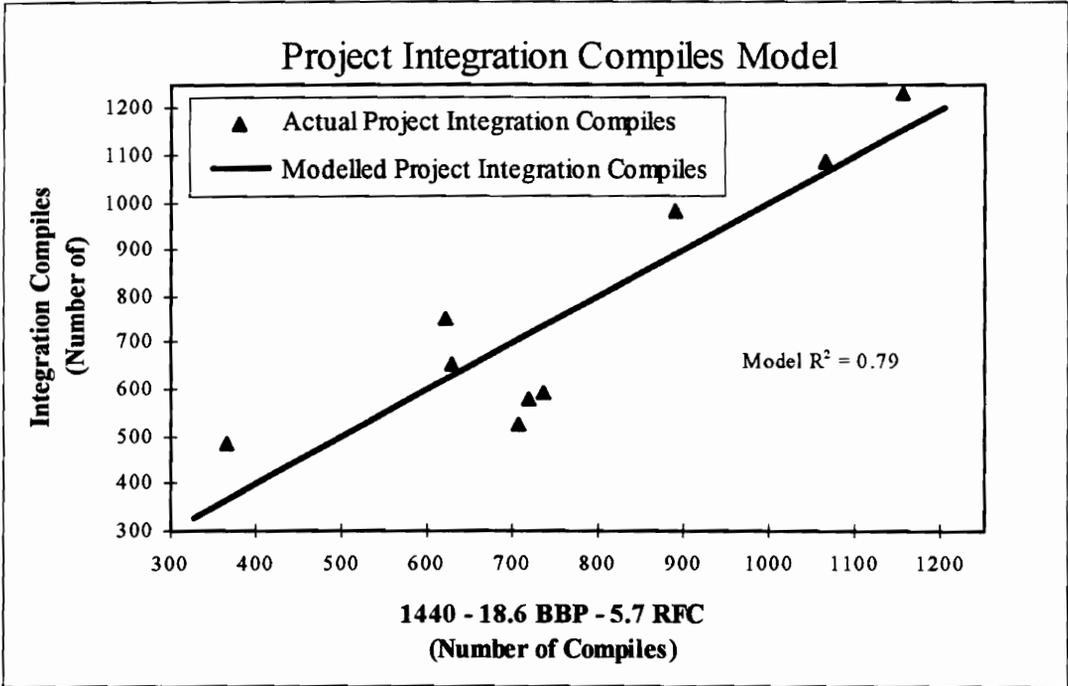


Figure 4.7. Integration Compiles Model

4.4.8 Number of Integration Runs

The next measure of the integration phase of the software development process is the number of runs performed to get the final system working completely. The model

describing this dependent variable is:

$$\text{IRUN} = 1141.3 - 19.1 * \text{BBP}$$

Total Regression Explained (R^2)= 0.57

Adjusted $R^2 = 0.51$

BBP explains 100% of regression explained by the model.

This model is mediocre at best, at explaining the trend in the *Number of Integration Runs* performed. The only term in the model is the percentage of black box reuse (BBP) achieved and the total regression explained is only 57%. Clearly, the number of integration runs cannot be modeled very well using these independent variables. The *Number of Integration Runs* model is shown in Figure 4.8.

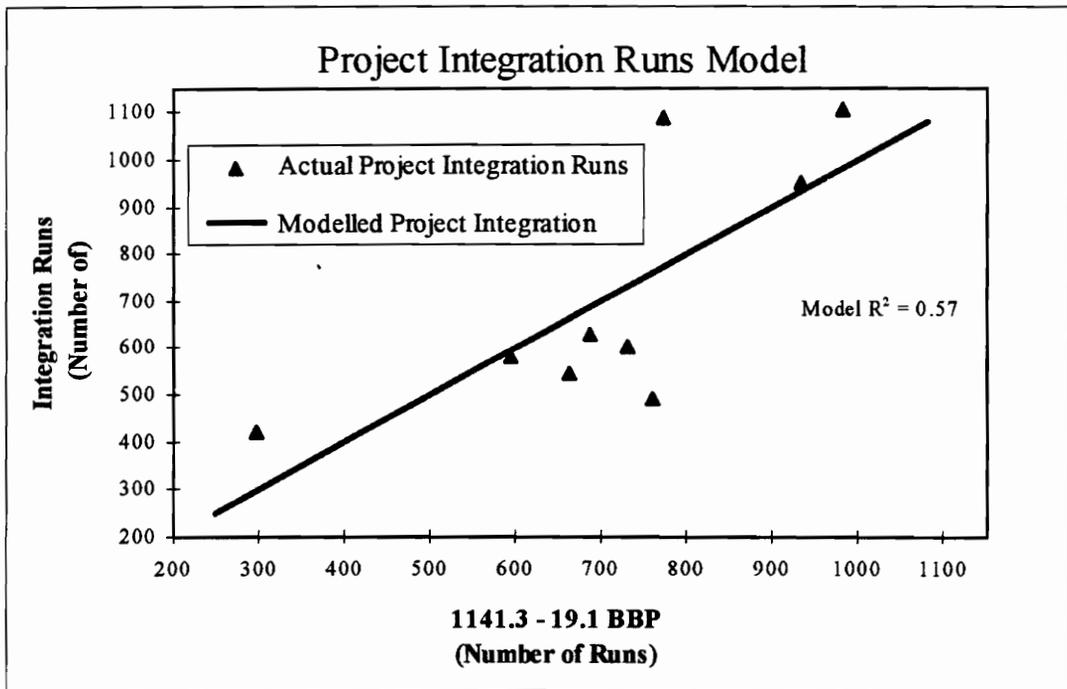


Figure 4.8. Integration Runs Model

4.4.9 Number of Errors in the Final System

The last measure of the projects is the *Total Number of Final Errors* left in the system after it is finished. The completed systems are evaluated and the number of significant errors left in them are counted. Significant errors include missing functionality, run-time errors, and incorrect operation. The *Number of Final System Errors* measures the reliability of the system. The model for this measure is:

$$\mathbf{FERR = 9.5 - 0.3 * BBP + 0.29 * WBP + 18.0 * DAC}$$

Total Regression Explained (R^2)= 0.92

Adjusted $R^2 = 0.88$

BBP explains 71% of regression explained by the model.

WBP explains 17% of regression explained by the model.

DAC explains 12% of regression explained by the model.

92% of the final system error trend is explained by this model. 71% of that is explained by the *Black Box Reuse Percentage* (BBP) achieved. As the black box reuse drops, the number of errors rises. Since the library classes are reliable, this is expected. More of the system's class interactions are suspect if the *Black Box Reuse Percentage* is low.

The second parameter, *White Box Reuse Percentage* (WBP) is positively correlated. As the subjects increased their *White Box Reuse Percentage* (WBP), the system reliability drops. Given that the systems had to be developed in a limited amount of time, it seems likely that the library classes could not be understood enough to modify without introducing errors.

The third term of the model, *Data Abstraction Coupling* (DAC) explains 12% of the trend explained by the model. As the system class coupling increases, the system final reliability drops. This is a well-established principle in Software Engineering. Reduced coupling leads to more error-free systems. The model for the *Number of Final System Errors* (system reliability) is shown below in Figure 4.9.

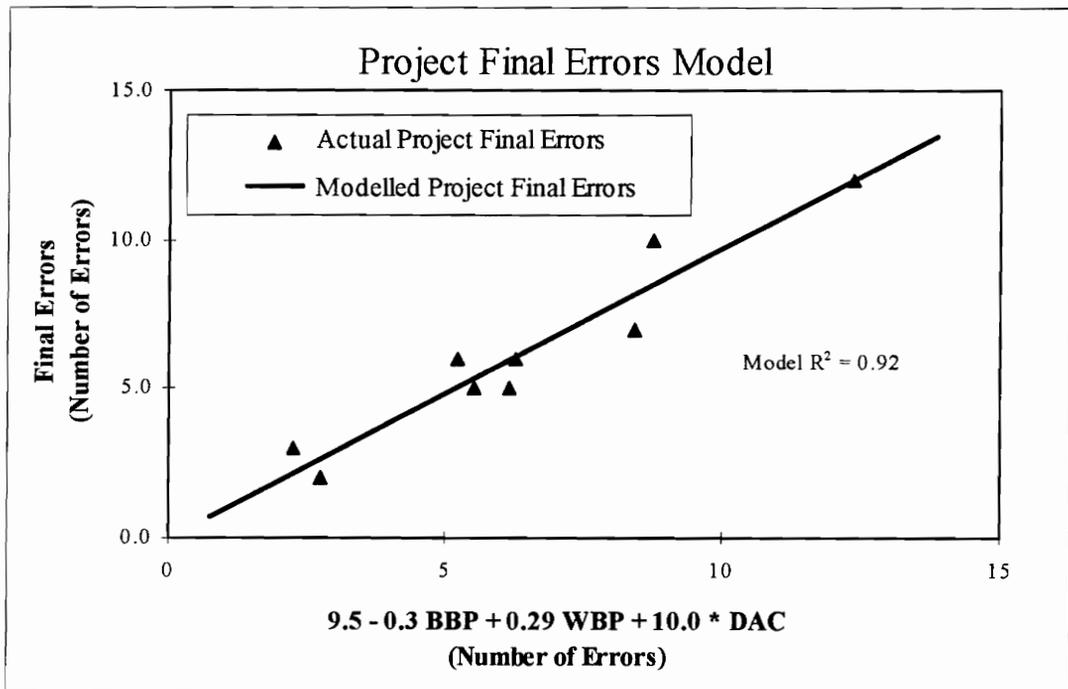


Figure 4.9. Final System Errors Model

4.5 Summary / Conclusions

Summarizing all this results into Table 4.6, for each of the dependent variables, the "best" model is given along with the R^2 value for that model.

Table 4.6. Summary of the Regression Models for the Nine Process Indicators

Dependent Variable	Model (Without Intercept)	R ²
<i>Design Time</i>	-0.83 BBP + 0.97 MPC	0.77
<i>Implementation Time</i>	4.2 SZ1 + 201.3 NOC - 115.3 DIT + 25.8 LCM	0.99
<i>Library Time</i>	1.05 BBP - 2.66 FPC + 46.2 DIT	0.88
<i>Integration Time</i>	-5.4 BBP + 39.8 DIT + 2.2 MPC + 0.6 LPC	0.97
<i>Total Development Time</i>	-3.8 BBP + 1.57 WBP - 43.0 DIT	0.98
<i>Integration Errors</i>	-0.15 BBP + 0.47 WBP + 0.03 LPC	0.88
<i>Integration Compiles</i>	-18.6 BBP - 5.7 RFC	0.79
<i>Integration Runs</i>	-19.1 * BBP	0.57
<i>Final System Errors</i>	-0.3 BBP + 0.29 WBP + 18.0 DAC	0.92

Note that the *Black Box Reuse Percentage* (BBP) variable appears in eight of the nine models, *White Box Reuse Percentage* (WBP) appears in half of the models, and the *Average Depth in the Inheritance Tree* (DIT) appears in half of the models. No other variable appears in more than two models. It seems that the development process is most affected by the amount of reuse performed and the structure of the underlying class hierarchy.

In the models where the *Black Box Reuse Percentage* (BBP) term appears, it explains a minimum of 41% of the trend in the dependent variable (that the model explains). Clearly, the amount of black box reuse achieved is the overriding indicator of the changes in the software development process for these nine indicators. Most of the size and complexity measures do not appear in the models or explain very little of the regression with the exception of the *Total Implementation Time* measure. The size and complexity metrics do not appear in the models very prevalently, because this experiment is designed to eliminate this type of project differences and focus on reuse level.

In the remainder of this chapter, the three research questions that are initially posed are reiterated and answered with the results of Experiment One.

4.5.1 Research Question One

The first question deals with the affect of reuse on the development time required during the initial phases of the software life cycle. The specific question is:

- ◆ How does reusing C++ classes affect the total time required during the various phases of the software development life cycle (design, implementation, library, integration, total)?

The first five process indicators: *Design Time* (DTIM), *Implementation Time* (CTIM), *Library Time* (LTIM), *Integration Time* (ITIM), and *Total Development Time* (TTIM) and their models from Table 4.6 are used to answer this question. From the models, the following conclusions about software reuse can be drawn:

- ◆ As the percentage of **black box reuse increases**,
 - the design time decreases
 - the integration time decreases
 - the total development time decreases.
 - the library time increases

Black box reuse causes shorter development times in all the phases except *Time Spent in Class Libraries* which obviously must go up.

- ◆ As the percent of **white box reuse increases**,
 - the total development time increases.

This unintuitive result stems from the fact that white box reuse (inheritance) is a difficult concept not completely mastered by the subjects of this experiment. Consequently, incorrect inheritance causes more problems than solutions and results in decreased productivity. Another explanation is that when inheriting from unfamiliar classes (such as a library classes), the additional effort required to understand the library offsets the gain realized from not having to write the class from scratch. However, if many classes are reused from the same library, this effect should diminish or disappear altogether.

4.5.2 Research Question Two

The second research question focuses on the integration phase of the software development process. For this experiment, the integration phase is a substantial part of the development effort. The research question is:

- ◆ How does reusing C++ classes affect the integration process?

Using the models for *Total Integration Time* (ITIM), *Number of Integration Errors* (IERR), *Number of Integration Compiles* (ICOM), and *Number of Integration Runs* (IRUN), the following conclusions can be made regarding the effects of software reuse.

- ◆ As the percentage of **black box reuse increases**,
 - total integration time decreases,
 - the number of integration errors decreases,
 - the number of integration compiles decreases, and
 - the number of integration runs decreases.

Clearly, the more black box reuse achieved for a software system, the easier the system is to integrate. Given that the classes in the libraries are useful and reliable, the integration effort should decline since a percentage of the modules are known to be working correctly already.

- ◆ As the percentage of **white box reuse increases**,
 - the number of integration errors increases.

This is an interesting conclusion stating that more reuse, if it is white box reuse, causes the number of major errors occurring during the integration phase to increase as well. Clearly, inheritance, although useful as a design tool, is not as useful for deriving new classes from library classes. One explanation for this result is that the subjects are not experts at inheritance yet, and may not make good decisions about when and how to perform it. Another explanation is that when inheriting from an unfamiliar class (such as a library class), the additional effort required to understand the library is not invested until the integration phase when problems with the derived class start occurring.

4.5.3 Research Question Three

The last question deals with the final system reliability measured by the *Number of Errors Found* in the submitted system (FERR). The research question is:

- ◆ How does reusing C++ classes affect the final system's reliability?

The conclusions drawn are:

- ◆ As the percentage of **black box reuse increases**,
 - the final system reliability increases (errors decrease).

Once again, as a larger percentage of the system is composed of reliable library classes that have not been modified, the whole system is more stable and less error prone.

- ◆ As the percentage of **white box reuse increases**,
 - the final system reliability decreases (errors increase).

Again, the problems with inheritance are observed. If inheritance is misused due to a lack of understanding of the parent (library) class or a lack of proficiency with inheritance, the result is increased effort and decreased final system reliability.

- ◆ As the **class coupling increases**,
 - the final system reliability decreases (errors increase).

This model also shows that as the *Data Abstraction Coupling* (DAC) among the classes in the final system increases, the final system reliability decreases. Again, this result corroborates the current feelings in the Software Engineering and software development communities.

4.6 Final Comments

Under this goal, the effects of software reuse of C++ classes on the software development process have been explored. Some interesting results regarding the varying effects of black box reuse versus white box reuse (inheritance) have been observed. Chapter 5 delves into this issue with an empirical study geared at examining these differing effects and providing some more insight into the effects that reuse of C++ classes has on software development.

Chapter 5

■ Types of Software Reuse and Software Development

The previous chapter investigates the effects of reusing C++ classes on the software development process. It concludes that several factors (measures) of the software development process are significantly affected by the percent of reuse achieved. Some of these measures include the *Total Development Time*, the final system reliability (*Number of Final Errors in the System*), and a few measures of the integration stage including *Number of Compiles*, *Number of Runs*, and *Number of Major Integration Errors*. Experiment One is designed to examine and quantify the effects of reusing C++ classes on the development process.

The goal of this chapter (goal two of this research) is to differentiate the effects of different types of reuse (black box and white box) from no reuse at all in the software development process. The specific questions that are the focus of this chapter are:

1. How do writing C++ classes from scratch, performing black box reuse, and performing white box reuse (via inheritance) compare to each other in terms of their impact on the software development process?
2. Which type of reuse of C++ classes is the most beneficial?
3. In what areas of the software development process are these benefits of reusing C++ classes realized?

Experiment Two examines the software development process when writing a small software system, either from scratch or when reusing the primary data structure. The reuse category is further divided into two groups: black box reuse and white box reuse in order to examine the effects of these two types of software reuse on software development. Some of the same measures used to characterize the software process during Experiment One are used during Experiment Two. Some of these measures have to be modified because of the slightly different nature of the system being developed. The ten process measures for this experiment are presented later in this chapter.

5.1 Design and Setup

This experiment involves twenty-four Computer Science seniors working in a controlled (observed) environment. The students are divided into three groups of eight programmers each. Each group writes a simple post-fix calculation program. Group A writes all their source code from scratch. Group B is required to reuse the primary data structure of the system (a stack). A library of classes, including a stack class, is provided. Group B represents the Black Box Reuse group. Group C is the White Box Reuse group. The subjects in this last group must use inheritance to reuse. They are required to derive a specialized stack class, which handles post-fix operations, from the stack class provided to them in the C++ library.

An initial data set is gathered on each of the subjects; it is used to divide them into three equal groups of eight programmers each. *Overall Quality Credit Average (QCA)*, *Computer Science Quality Credit Average (CS QCA)*, and *Total Number of Computer Science Credits taken (CS CREDITS)* are used to evaluate the aptitude of each subject. A point value is assigned to each person by taking these three gathered facts and weighting and summing them according to the following formula:

$$\text{Points} = 10 * \text{QCA} + 20 * \text{CSQCA} + 70 * \text{CSCREDITS} / 10$$

These three traits are used because they provide an indication of the programming skill of the subjects and are also readily available for the subjects of this experiment. The weights in the formula are chosen because, for this research effort, programming experience is more important than the final grade evaluation. Therefore, the number of credits is weighted at 70% while the combined QCA values receive only 30%. These percentages are chosen because the number of CS credits passed is a better indicator of programming experience than final grade point average. Furthermore, Computer Science course grades are deemed twice as important as the grades obtained in all university courses.

Table 5.1 shows the three group means and variances for each of the three subject evaluation factors and for the computed point value. This data set is used to show that the three groups are approximately equivalent for the purposes of this experiment.

Table 5.1. Means and Variances for the Three Experimental Groups

Description	QCA	CS QCA	CS Credits	Point Value
Group A Mean	2.81	3.08	32.3	315
Group B Mean	2.59	2.75	34.3	321
Group C Mean	2.76	3.04	32.6	317
Group A Variance	0.31	0.28	35.9	2,101
Group B Variance	0.23	0.31	31.1	2,048
Group C Variance	0.25	0.17	25.1	772

Once the initial data set is gathered, the three groups are formed by calculating a point value for each student, sorting the students by their point values, and then splitting the students into eight sets of three students each. Groups A, B, and C are constructed by assigning one person from each of the eight sets to a group. This is a complete block design for constructing equivalent groups.

Two statistical tests are used to further confirm that the three research groups are equivalent. A Hartley F test on the variances shows that all three groups have indistinguishable variances for all of the items in Table 5.1. Hartley's test is computed by dividing the variance of the group with the maximum variance by the variance of the group with the minimum variance. If this value is larger than a value looked up in a statistical table, then the variance are not the same and must be treated separately. For this experiment, the table value is 6.94. (obtained for a 0.05 significance level, three experimental groups, and eight data points per group). See [OttR93, Walp85] for more information and for the complete table.

The first two rows of Table 5.2 show the maximum group variance and the minimum group variance. Row three shows the four F test values obtained by dividing the maximum variance by the minimum variance. All four of these test values are less than 6.94 so the assumption that all the groups have the same variance is valid. Since this is true, an F test on the means is performed. This F test requires that the *variance between the three groups* be divided by the *variance within the three groups*. The *variance between the groups* measures the amount of variability that each group's mean has about the entire sample mean (all twenty-four observations) while the *variance within the groups* measures the amount of variability that a particular observation has about its own group mean. Rows four and five show the *variance between the groups* and the *variance within the groups* for the three selected subject traits and the weighted point value.

Table 5.2. Statistical Test Values Showing The Three Experimental Groups Being Equal

	QCA	CS QCA	CS Credits	Point Value
Maximum Group Variance	0.31	0.31	35.9	2,101
Minimum Group Variance	0.23	0.17	25.1	772
F Test on Group Variances	1.31	1.88	1.43	2.72
Variance Between Groups	0.11	0.25	9	59
Variance Within Groups	0.26	0.25	30.7	1,640
F Test on Group Means	0.43	1	0.29	0.04

The last row of Table 5.2 shows the four F test values computed for each of the four data items. As with the previous F test, a value is looked up in a statistical table and if these computed F test values are greater than the table value, then the difference between the means is significant. For these F tests, the table value is 3.47 (from a table for a 0.05 significance level, three groups, and eight subjects per group). For the complete table, see [OttR93]. Again, all four test scores (Table 5.2, Last Row) are less than the statistical value (3.47), so there is no distinguishable differences among the group means. The fact that a complete block design is performed and that this analysis finds no significant

differences allows the three groups to be treated as equally skilled in Computer Science knowledge with regard to these four categories.

Every subject in all three groups of programmers is given the same programming problem: to implement a basic post-fix notation calculator. The program must read in a series of post-fix strings from a data file, use a stack to process the expressions, and output the final answers to the screen. The three problem statements differ only in the requirement for reuse. Group A writes the whole program from scratch. They are required to write a stack class. Group B uses the stack class from the provided C++ library. Group C inherits a new stack class based on the one in the C++ library. Appendix D shows the three assignment statements that are used in this experiment. Appendix F shows the solutions to the assignments in high level pseudo-code

The subjects design and implement a solution to the problem in a supervised setting. As each subject works, ten software development process measures are recorded every fifteen minutes. The experiment proctor stops the subjects at fifteen minute intervals and asks them to record their development data on the provided form (Appendix E). By monitoring this experiment and the data collection, the accuracy of the gathered data is ensured.

5.2 Experimental Data

The initial data set of this experiment (QCA, CS QCA, CS CREDITS, POINTS) is gathered, so the twenty-four subjects can be evenly divided into three groups. The remainder of the data of this experiment is gathered during the development process. Each of the subjects is given a data collection form (Appendix E) to be used as the experiment is conducted. This developmental data set is based on some of the dependent variables from the previous chapter. It is chosen because it partially characterizes the software development process. The ten specific process indicators gathered for Experiment Two are shown in Table 5.3.

Table 5.3. Ten Process Indicators for Experiment Two

Description of Process Indicator	Indicator Acronym
Total Time Spent Designing the Problem Solution and the Program. (Hours)	DTIM
Total Time Spent Examining or Learning the Provided C++ Library. Only for Groups B and C. (Hours)	LTIM
Total Time Spent Coding the Program Solution. (Hours)	CTIM
Total Time Spent Debugging the Program Solution. (Hours)	DBTIM
Total Development Time. Sum of measures 1-4. (Hours)	TTIM
Total Number of Compiles.	TCOM
Total Number of Runs.	TRUN
Total Number of Compile-Time Errors. (Errors Caught By the Compiler)	CTE
Total Number of Logic Errors. (Errors in the Design or Code Logic or Flow of Control)	LE
Total Number of Run-Time Errors. (Errors Occurring While the Program is Executing)	RTE

The subjects record this data as they work. Once a correct solution is completed and checked, the subject (programmer) is finished. After all twenty-four programmers complete the experiment, differences in the process indicators among the three groups are sought using a series of F tests similar to the type presented in Section 5.1.

5.3 Results

For each of the ten process measures gathered, the mean and variance is computed for each of the three experimental groups. Table 5.4 presents the means and variances for the three groups for each of these ten indicators. The raw data from which these statistics are computed can be found in Appendix G.

Table 5.4. Development Data for Experiment Two -- Means and Variances

Development Process Measure	Group A Mean	Group B Mean	Group C Mean	Group A Variance	Group B Variance	Group C Variance
<i>Total Time Spent Designing the Program Solution</i>	16.6	14.4	18.6	38	31.7	241.7
<i>Total Time Spent in Library (Groups B, C)</i>	0	10.3	18.3	0	50.2	46.8
<i>Total Time Spent Coding the Program Solution</i>	39	19.5	34.8	238.3	197.7	302.8
<i>Total Time Spent Debugging the Program Solution</i>	42.5	20.9	44	451.1	196.1	302.3
<i>Total Development Time (Sum of measures 1-4)</i>	98.1	65	115.6	521	1,100	1,203
<i>Total Number of Compiles</i>	23.9	15	16.5	177.6	169.7	118.9
<i>Total Number of Runs</i>	14.4	8.3	13.1	30.8	30.8	25.8
<i>Total Number of Compile-Time Errors</i>	46.3	14.4	42	1,410	143.1	1,395
<i>Total Number of Logic Errors</i>	11.1	4.6	10	38.1	30.8	10.3
<i>Total Number of Run-Time Errors</i>	4.1	1.1	1.2	7	3.3	4.3

The last step is to analyze each set of three means for all ten of the process indicators looking for statistically significant differences among the three groups of subjects.

5.4 Analysis

This section uses the same two F tests described in Section 5.1 to analyze this data. If the means are found to differ, a third test called Tukey's procedure is used to establish the exact relationship among the three means. For Tukey's procedure, an F test value is calculated for each pair of experimental groups (AB, BC, AC) and a statistical significance value is calculated for the given data. The relationship among the three pairwise values

and the statistical cutoff value determines the exact relationship among the three means. For this research, all of the F tests are performed at an $\alpha = 0.05$ level.

The first F test (Hartley's F test) is used to make sure that for each of the ten process indicators, the sample variances of the three experimental groups are approximately equivalent. Hartley's test creates an F test statistic by dividing the maximum group variance by the minimum group variance. If the result is greater than a statistical table value⁵, the variances are assumed to be equal and a second F test on the sample means can be performed to determine if the group means differ significantly. If a significant difference is found among the group means, Tukey's procedure characterizes the exact relationship of the each of the group means to each other. All of these tests are performed at an $\alpha = 0.05$ confidence level meaning that there is only a 5% chance of getting this exact data set when the means are the same. This is a standard confidence level for most statistical research procedures.

If the group variances are found to be different for any of the ten indicators, then the second F test on the means is ill-advised. This occurs for two of the process measures: *Design Time (DTIM)* and *Number of Compile-Time Errors (CTE)*. For these two measures, no further analysis is performed on the original data, because this experiment is designed under the assumption that these variances will be the same. Several factors can explain why the variances are not equal, including that the variances for these measures really are different, an unforeseen bias in the experimental design, or just plain bad luck. Without further experimentation, there is no way to know. Rather than include this data and make suspect conclusions, these two measures are transformed using a power transformation to equalize the variances. Making the variances for the transformed data equal allows the F test on the means to be performed.

⁵ The value can be found in a F max. / min. percentage point table using $t = 3$ groups, degrees of freedom = 21, and $\alpha = 0.05$. See [OttR93] for more details.

For all ten indicators, if, after the test on equality of the variances is passed, the second F test value exceeds 3.47⁶, there is a significant difference among the three means. The value 3.47 comes out of a F distribution statistical table for this F test at a 0.05 confidence level using three sample groups with eight observations per sample. The complete table can be found in [OttR93]. For the measures whose means are significantly different, Tukey's procedure is used to characterize the exact relationship among the three group means. Tukey's procedure compares all three means in a pairwise fashion while maintaining the 0.05 significance level.

For three groups, there are three pairwise comparisons: group A to group B, group A to group C, and group B to group C. If all three pairs differ, then all three means differ. If only two of the pairs are found to differ, then two of the means (groups) are statistical the same and the third mean (group) is either larger or smaller (depending on the actual means) than the other two means. If only one pair of means is different, then a weak conclusion can be drawn about them. If no pairs are found to be different, then no statement can be made about the three means. They are statistically the same value. Therefore, no difference exists among the three groups for that indicator. Given the three means, Tukey's procedure is as follows:

1. Calculate the Tukey significance level using $q(0.05, 2, 21)$ which is found a statistical table. See [OttR93].
2. Calculate the Tukey W_{XY} value for each of the three group pairs ($XY = AB, BC,$ and $AC.$)
3. Compare each of W_{AB} , W_{BC} , and W_{AC} to the Tukey significance value.
4. If the W value is greater than or equal to the significance value, then a significant difference exists between the pair of means being examined.
5. Based on the three comparisons, determine the relationship that exists among the three means.

⁶ The value can be found in a F distribution table using degrees of freedom 1 = 2, degrees of freedom 2 = 21, and $\alpha = 0.05$.

For all ten process indicators, this analysis is performed and conclusions are drawn.

5.4.1 Analysis Data

The analysis below requires the statistical test values for each of these process measures, namely, the Hartley F test value for the group variances, the F test value for the means, and the Tukey W statistic for each of the three pairs of means (AB, BC, AC). Table 5.5 presents these values for all ten process indicators. The number in parentheses in each cell represents the statistical value that the test statistic must exceed for there to be a significant difference. Cells in bold are the ones showing a statistically significant difference. The next nine subsections discuss this table. Blanks in the Table 5.5 are for tests that are not performed for one reason or another. Explanations are given in the subsections below.

Table 5.5. Statistical Tests on the Ten Process Indicators. The top number in each cell is the test value. The bottom number is the table value for statistical significance at $\alpha = 0.05$.

Development Measure	F Test on Variances	F Test on Means	Tukey for Pair AB W_{AB}	Tukey for Pair BC W_{BC}	Tukey for Pair AC W_{AC}
<i>Total Time Spent Designing the Program Solution</i>	7.63 (6.94)				
<i>Total Time Spent Designing the Program Solution (Square Root)</i>	4.70 (6.94)	0.23 (3.47)			
<i>Total Time Spent in Library (Groups B, C)</i>	1.07 (4.99)	5.28 (4.32)			
<i>Total Time Spent Coding the Program Solution</i>	1.53 (6.94)	3.42 (3.47)	19.50 (16.31)	15.25 (16.31)	4.25 (16.31)
<i>Total Time Spent Debugging the Program Solution</i>	2.30 (6.94)	4.23 (3.47)	21.63 (18.49)	23.13 (18.49)	1.50 (18.49)
<i>Total Development Time (Sum of measures 1-4)</i>	1.94 (6.94)	5.43 (3.47)	33.13 (32.45)	50.63 (32.45)	17.50 (32.45)
<i>Total Number of Compiles</i>	1.49 (6.94)	1.16 (3.47)			
<i>Total Number of Runs</i>	1.19 (6.94)	2.87 (3.47)			
<i>Total Number of Compile-Time Errors</i>	9.85 (6.94)				
<i>Total Number of Compile-Time Errors (Square Root)</i>	1.75 (6.94)	3.52 (3.47)	3.13 (2.66)	2.69 (2.66)	0.44 (2.66)
<i>Total Number of Logic Errors</i>	3.71 (6.94)	3.65 (3.47)	6.50 (5.34)	5.38 (5.34)	1.13 (5.34)
<i>Total Number of Run-Time Errors</i>	2.14 (6.94)	4.88 (3.47)	3.00 (2.29)	0.04 (2.29)	2.96 (2.29)

5.4.2 Design Time

The first measure of the development process is the design time required for the problem solution. As explained earlier, since the computed F test on the variances is 7.63 which is larger than 6.94, significant differences exist among the three experimental group variances. Performing a square root transformation on the data compresses the data for

re-analysis. Doing this causes the variances among the three group means to be indistinguishable. Now, an F test on the group means (square roots) can be performed. However, as can be seen from the second row of Table 5.5, there is no significant differences among the group means. At this point, there is little more that can be done. In terms of design time, this analysis has shown that the type of reuse performed has no impact on the design time required.

5.4.3 Time Examining / Learning the Library

This measure is somewhat special in that only groups B and C are analyzed. Group A implemented their solutions from scratch, so this indicator has no meaning for that group. For groups B and C, the two reuse groups, the variances are found to be statistically the same (Row three of Table 5.5: $1.07 < 4.99$). Note that for only two groups, the statistical cut off value is not 6.94, but rather 4.99. Since the variances are equal, an F test on the means is showing that the means are also different. In this case, Tukey's procedure need not be performed as there are only two groups (1 pair). The final relationship is therefore that the Black Box Reuse group (B) mean (10.3) is less than the White Box Reuse group (C) mean (18.3) in terms of *Total Library Time*. In other words, when reusing classes from a C++ library, if they are used as parent classes for new derived classes, more time is required to examine and learn the library than if the library classes are used without modification. This is expected, since to derive a new class requires that the implementation of the parent (library) class be fully understood. To simply reuse a library class without modification, only a portion of the implementation (the interface) needs to be examined. One of the costs of white box reuse is extra time in the software class libraries. In fact, in this small experiment, the White Box Reuse group (C) library time mean (18.3) is nearly twice (1.8 times) as large as the Black Box Reuse group (B) mean (10.3).

5.4.4 Implementation (Coding) Time

For this measure, no significant difference is found among the group variances ($1.53 < 6.94$). However, no significant differences are found among the group means either ($3.42 < 3.47$). Since this value is so close, the $\alpha = 0.05$ significance level is relaxed and Tukey's procedure is applied revealing that the mean for the Black Box Reuse group (B) is in fact, less than the Scratch group (A) mean. However, no difference is found between pair AC or pair BC. The weak conclusion is that the Black Box Reuse group (B) outperformed the Scratch group (A), but no difference can be detected between the Black Box Reuse group (B) and the White Box Reuse group (C) nor between the Scratch group (A) and the White Box Reuse group (C).

Examining the means, the Black Box Reuse group (B) mean is only 19.5 while the other two means are 39.0 and 34.8. The Scratch group (A) has to write their own stack class which accounts for the larger coding time mean. The White Box Reuse group (C), although given a stack class, had to enhance a portion of it, thus contributing to their extra implementation time.

5.4.5 Debugging Time

Under this measure, the group variances are again statistically the same (Table 5.5, Column 2, Debug Time Indicator: $2.3 < 6.94$). The F test on the means shows that a difference exists ($4.23 > 3.47$) and Tukey's procedure clarifies the exact relationship. Looking at Table 5.5, W_{AB} is 21.63 which is greater than 18.94. This implies that the Scratch group (A) mean (42.5) is significantly greater than the Black Box Reuse group (B) mean (20.9). $W_{BC} = 23.13$ which is greater than 18.94, so the Black Box Reuse group (B) mean (20.9) is less than the White Box Reuse group (C) mean (44.0), and $W_{AC} = 1.5$ which is less than 18.94, which implies that means A and C are too close together to distinguish. Concluding, for debugging time, the Black Box Reuse group (B) spent less time than the other two groups which are too close together to distinguish from each other.

The expected result is that the Black Box Reuse group (B) should spent less time debugging their programs because they use a reliable data structure class for the foundation of their program while the other two groups are required to either write their own data structure or modify an existing one. Although the White Box Reuse group (C) also has a reliable foundation, the group must deal with the inheritance construct which accounts for some additional complexity that leads to increased debugging time.

5.4.6 Total Development Time

Experiment One showed that increased reuse decreases total development time. This measure helps validate the previous experiment. The variances are statistically the same ($1.94 < 6.94$), and the means are significantly different ($5.43 > 3.47$). Tukey's procedure gives the minimum significance value at 32.45 and the three pair wise values at 33.13, 50.63, and 17.50 for pairs AB, BC, and AC respectively. Since the Black Box Reuse group (B) mean (65.0) is lower than the other two (Scratch group (A) mean = 98.1, White Box Reuse group (C) mean = 115.6), we can conclude that the Black Box Reuse group (B) took less average total time than the other two, but we cannot distinguish groups A and C.

The results for this measure correspond to the previous one, debugging time. The data separates the Black Box Reuse group (B) from the other two, but fails to separate writing from scratch (Group A) from the White Box Reuse group (C). Clearly, the additional design, implementation, and debugging time cause the White Box Reuse group (C) mean to inflate out to the Scratch group (A) level. Using the means, we can see that the Black Box Reuse group (B) took $65.0 / 115.6 = 57\%$ of the time, almost twice as fast as the White Box Reuse group (C) and nearly $65.0 / 98.0 = 30\%$ faster than the from Scratch group (A). Interestingly, the White Box Reuse group (C) performed the poorest, although not statistically different from the Scratch group (A). Inheritance may not be worth the benefits achieved in the realm of C++ class reuse.

5.4.7 Number of Compiles

From Table 5.5, it is evident that the variances of the three groups are statistically equal ($1.49 < 6.94$). However, the means also show no significant difference among the three groups ($1.16 < 3.47$). Apparently, the number of compiles accrued during development does not significantly differ regardless of reuse level. Intuitively, the Black Box Reuse group (B) should have had fewer compiles since their overall development time is shorter. Two explanations exist. The first is that the black box reusers compiled much more often during initial development and coding since they already had some working code (the foundation class). Or, a second explanation is that some flaw in the experimental design caused this lack of a trend. It seems clear that more research should be done on this measure.

5.4.8 Number of Runs

The number of runs measures fall into the same category as the number of compiles. Table 5.5 shows that the variances and the means are statistically equal ($1.19 < 6.94$ for variance and $2.87 < 3.47$ for the means). This makes sense since the number of trial executions tends to parallel the number of runs quite closely. Again, perhaps more research needs to be performed.

5.4.9 Number of Compile-Time Errors

As with the design time measure, the variances of the three groups for this measure are not equivalent ($9.85 > 6.94$) implying that an F test on the means can not be done. If the variances of the three groups are examined (see Table 5.4), the Black Box Reuse group (B) variance (143) is significantly less than the other two (1410, 1393). For whatever reason, the subjects within the Black Box Reuse group (B) all made about the same number of errors (lowering the variance for group B), thus nullifying the equal variance assumption. Again, this data set can be transformed using the square root function which equalizes the variances. Now, an F test is performed on the square roots of the means showing a statistical difference between the square root of the group B mean and the

square roots of the group A and C means. However, groups A and C can not be distinguished from each other.

Concluding, the Black Box Reuse group (B) incurred fewer compile-time errors than the other two groups. One explanation is the stability and known reliability of the reused library class. The Scratch group (A) had to write their own stack class which then had to be debugged, and the White Box Reuse group (C) had to use inheritance to derive a new class which also causes more code and more complex code to have to be implemented and debugged.

This result is not as strong as some of the others due to the square root transformation of the data.

5.4.10 Number of Logic Errors

This measure reveals the same trend as some of the previous ones: the Black Box Reuse group (B) has fewer logic errors than the other two groups which are indistinguishable. Clearly, a solid unchanging foundation (the library data structure class) starts the programmer out in the right direction.

5.4.11 Number of Run-Time Errors

The last measure has an interesting trend. The variances are equal (Table 5.5, last row: $2.14 < 6.94$) and the means differ (Table 5.5, last row: $4.88 > 3.47$). However, examining the Tukey values, the two reuse groups (B and C) are indistinguishable and both outperform the Scratch group (A). The obvious conclusion is that when the underlying data structure is (or is based on) a stable class, integration problems are reduced. This trend was seen in Experiment One as well.

5.5 Summary / Conclusions

Summarizing the ten measures above, Table 5.6 presents each of the measures and the corresponding relationship among the three group means.

Table 5.6. Summary of Means Relationships for the Ten Process Indicators

Development Process Indicator	Relationship of Means
<i>Total Time Spent Designing the Program Solution</i>	Variances Differ. No Relationship.
<i>Total Time Spent Designing the Program Solution (Square Root)</i>	Scratch = Black Box = White Box
<i>Total Time Spent in Library (Groups B, C)</i>	Black Box < White Box Scratch not applicable
<i>Total Time Spent Coding the Program Solution</i>	Black Box ≤ Scratch Black Box = White Box Scratch = White Box
<i>Total Time Spent Debugging the Program Solution</i>	Black Box < White Box = Scratch
<i>Total Development Time (Sum of measures 1-4)</i>	Black Box < White Box = Scratch
<i>Total Number of Compiles</i>	Scratch = Black Box = White Box
<i>Total Number of Runs</i>	Scratch = Black Box = White Box
<i>Total Number of Compile-Time Errors</i>	Variances Differ. No Relationship.
<i>Total Number of Compile-Time Errors (Square Root)</i>	Black Box < White Box = Scratch
<i>Total Number of Logic Errors</i>	Black Box < White Box = Scratch
<i>Total Number of Run-Time Errors</i>	Black Box = White Box < Scratch

Examining Table 5.6, it is clear that black box reuse positively affects the software development process in several areas including *Time Spent in C++ Libraries*, *Debugging Time*, *Total Development Time*, *Compile-Time*, *Logic*, and *Run-Time Error Totals*. In all of these instances, the Black Box Reuse group (B) significantly reduced their development

effort over the other two groups. Furthermore, the number of errors occurring in the various phases of the software development life cycle decline when the underlying data structure is obtained from a C++ class library.

Experiment One demonstrates that C++ class reuse is a worthwhile beneficial endeavor. This experiment points towards limiting C++ class reuse to simple black box components and not attempting to reuse with modification (inheritance). In most instances (except *Number of Run-Time Errors*), white box reuse turns out to be no better than writing from scratch.

These results need further explanation. In the object-oriented paradigm, inheritance is touted as a great strength, supporting information hiding and modular, hierarchical code. This is not being challenged. Libraries of classes need an inheritance structure to provide an underlying organization for the classes. Programs need to use inheritance to reduce duplication of code and to achieve modular, well-designed programs. However, for the general C++ classes examined in this research, it is apparent that reuse should be limited to black box reuse. Modifying the functionality of a C++ class from a class library (using inheritance) provides no benefit over writing the class from scratch. Obviously, for very complex classes or architecture (machine) specific classes (such as graphics libraries), inheritance would save much development effort by preventing the reinvention of the wheel with every new piece of software.

Chapter 6

■ Programmer Experience and Software Reuse

Recall from Chapter 1, the third goal of this research focuses on the characteristics and common experiences of programmers that have a tendency to reuse. Since reuse has been shown to increase programmer productivity [Frak94b, Lewi92] as well as the reliability [Scha94] of the final software system, it is desirable to hire programmers with good "reuse potential". This chapter answers the question:

- ◆ What are some of the characteristics of programmers who reuse C++ classes?

In this case, characteristics are defined to mean the amount of experience and aptitude that the programmer has in a given set of content areas. By identifying these areas of expertise, programmers can be better trained to reuse C++ software. Furthermore, programmers who are likely to be good reusers can be identified by testing or measuring their aptitude in the skills that correlate with the amount of reuse achieved.

6.1 Experimental Data

Three major characteristic or skill categories are defined: **general computer science and programming concepts**, **software engineering concepts**, and **reusability concepts**. Each of these major areas is divided into more measurable skills like amount of experience/aptitude with the C programming language, amount of experience/aptitude using high-level designs, and amount of experience/aptitude using software libraries. In all, experience and aptitude data are gathered on thirty-one different skills in these three major categories.

The data set collected on the thirty Computer Science seniors who participated in Experiment One is analyzed for interdependencies between common experiences of the programmers and the amount of reuse that each programmer performed during

Experiment One. Appendix B.1 shows the **Programmer Data** questionnaire that each programmer completed before beginning the experiment. At the conclusion of Experiment One, the number of classes reused by each programmer is known. This datum is used to divide the thirty programmers into three groups based on their reuse level: high reuse (eight or more classes reused), medium reuse (four - seven classes reused), and low reuse (zero - three classes reused). These reuse levels divide the subjects into three groups of roughly equivalent size. Lastly, each of the experimental groups can be checked for dependencies to the programmer characteristic data.

For the thirty subjects who participated in Experiment One, sixteen of the thirty-one skills lacked sufficiently meaningful data for a statistical analysis. Many of the subjects had so little (or no) experience with these particular skills that a categorical correlation would have been futile. If 25% of the subjects (seven or more) have no experience in a particular skill, the skill is removed since not enough non-zero data exists to make any meaningful conclusion about the skill anyway. Removing these skills leaves fifteen fundamental skills. Additionally, three summary skills representing the total experience across each of the three major categories of skills (Programming Languages, Software Engineering, and Reusability) are computed bringing the number of skill areas to eighteen. Table 6.1 shows the thirty-one original skills and which of them were eliminated. The three skill category totals are shown as well.

The programmer skills are divided into three groups: one for each major category of skills. The first group, **Programming Language concepts** includes thirteen sub-areas pertaining to some of the many programming languages for computers like C and Pascal. Experience using various operating systems like UNIX and Microsoft Windows falls in this section as well. The second group, dealing with **Software Engineering concepts**, is divided into eleven skills. Examples in this section include experience using inheritance, encapsulation, and various software engineering tools (like debuggers, profilers, source code control systems, etc.). The last group, **Reusability concepts**, focuses on seven areas associated

with reusing software like the ability to modify other people's code and experience writing software libraries.

Table 6.1. Original Data Set of Programmer Skills and Which are Eliminated.

Skills To Be Analyzed	Skills To Be Eliminated
<i>C Experience</i>	<i>C++ Experience</i>
<i>Pascal Experience</i>	<i>Microsoft Windows Experience</i>
<i>Assembly Experience</i>	<i>X Windows Experience</i>
<i>FORTRAN Experience</i>	<i>UNIX Shell Script Experience</i>
<i>BASIC Experience</i>	<i>SmallTalk Experience</i>
<i>LISP / Prolog Experience</i>	<i>Ada Experience</i>
<i>UNIX Experience</i>	<i>Object-Oriented Design Experience</i>
<i>Encapsulation Experience</i>	<i>Polymorphism Experience</i>
<i>High-Level Design Experience</i>	<i>Inheritance Experience</i>
<i>Top-Down Design Experience</i>	<i>Source Code Debugger Experience</i>
<i>Bottom-Up Design Experience</i>	<i>Source Code Control System Experience</i>
<i>Procedural Paradigm Reuse Experience</i>	<i>Integrated Development Environment Experience</i>
<i>Experience Modifying Other's Code</i>	<i>Source Code Profiler Experience</i>
<i>Experience Reusing Other's Code</i>	<i>Object-Oriented Paradigm Reuse Experience</i>
<i>Experience Using Procedural Software Libraries</i>	<i>Experience Using Object-Oriented Class Libraries</i>
	<i>Experience Writing Software Libraries</i>
<i>Programming Language Experience Total</i>	
<i>Software Engineering Experience Total</i>	
<i>Reusability Experience Total</i>	

A list of all thirty-one skill areas, along with a brief description is shown in Appendix A. Each of the three groups includes a total experience skill which is simply the average of all

the areas for the category. This brings the total number of skill areas gathered to thirty-four.

A questionnaire (Appendix B.1) queries the programmers on how well each knows a particular skill. The programmers rate themselves in each of these areas in terms of experience/exposure (in months) and aptitude (on a scale of one to six). Tables 6.2, 6.3, and 6.4 show the data values for the thirty subjects and the thirty-four skill / experience categories for each of the three skill categories listed above.

The rows of these three tables are the subjects of the experiment while each column of the tables represents one of the skill areas. Table 6.2 consists of fourteen columns for the **Programming Language** skills. The next table, Table 6.3, contains twelve columns of **Software Engineering** skills, and Table 6.4 has eight columns for the **Reusability** skills. The last column in each of these tables is the dependent variable which represents the amount of reuse that the particular subject performed during the execution of Experiment One. In particular, the dependent variable counts the total number of C++ classes that each programmer reused during the execution of Experiment One.

The entries in the three tables are the months of experience in the particular skill weighted by the programmer's own assessment of their current aptitude for the skill in question. Experience ranged from zero to nearly one hundred months; aptitudes ranged from one (no aptitude) to six (complete understanding). The product of the two values gives the weighted experience value for each of the skills.

Table 6.2. Programmer Skills Data (Weighted Experience) for the Programming Languages Skills

Subject	Programming Language Concepts (Skills)													Programming Total	Total Classes Reused
	C++ / Objective C	C	Pascal	Assembly	Fortran	BASIC	X Windows	UNIX Shell Script	Smalltalk	LISP / Prolog	Ada	MS Windows	UNIX		
1	0	300	192	192	400	96	0	16	0	1	0	60	480	1737	0
2	0	420	96	24	8	0	0	5	0	0	0	0	240	794	1
3	0	180	300	80	120	1	4	5	0	1	1	300	240	1232	1
4	0	58	53	5	160	0	0	3	0	0	0	0	3	282	2
5	12	256	48	20	0	27	0	5	0	8	0	11	256	643	2
6	0	160	144	40	21	288	0	27	0	13	0	11	160	864	2
7	0	160	200	96	240	0	0	300	0	32	24	0	80	1132	3
8	0	360	600	96	120	0	0	180	6	1	0	720	180	2263	3
9	0	200	180	64	360	180	0	32	0	32	0	64	5	1117	3
10	5	120	5	13	20	40	0	0	0	21	0	96	300	621	3
11	0	500	160	5	0	83	0	40	0	21	0	1	400	1211	4
12	0	400	400	5	7	24	32	11	0	2	0	32	192	1105	4
13	0	240	48	21	21	37	0	96	0	16	0	128	144	752	4
14	0	80	125	0	32	21	0	0	0	11	0	1400	120	1789	5
15	0	216	192	144	64	288	0	128	0	7	0	288	480	1807	5
16	0	384	80	40	11	0	0	5	0	11	0	64	192	787	5
17	0	240	144	16	20	144	0	0	0	16	0	80	256	916	5
18	0	360	384	40	80	360	0	3	0	13	0	120	480	1840	6
19	0	120	24	13	32	16	0	48	0	0	0	7	400	660	6
20	0	384	216	20	3	0	0	1	0	3	0	0	85	712	7
21	0	360	480	48	0	32	0	8	0	16	0	0	600	1544	7
22	0	432	32	40	11	32	0	4	0	0	0	0	180	731	7
23	32	13	300	128	120	200	0	1	0	4	0	480	180	1459	8
24	0	160	40	40	20	3	0	3	0	3	0	240	64	572	8
25	0	480	120	48	3	21	0	5	0	5	0	8	480	1171	8
26	0	160	32	4	5	20	0	11	0	11	0	160	500	903	9
27	0	216	120	96	0	120	0	0	3	3	0	200	300	1057	9
28	0	480	120	80	32	120	0	320	0	0	0	72	320	1544	11
29	0	120	216	120	32	120	0	0	0	7	0	0	288	903	12
30	0	288	165	128	32	432	0	192	0	23	0	96	288	1645	12
Ave.	2	262	174	56	66	90	1	48	0	9	1	155	263	1126	5
Min.	0	13	5	0	0	0	0	0	0	0	0	0	3	282	0
Max.	32	500	600	192	400	432	32	320	6	32	24	1400	600	2263	12

Table 6.3. Programmer Skills Data (Weighted Experience) for the Software Engineering Skills.

Subject	Software Engineering Concepts (Skills)											Total Classes Reused	
	Object Oriented Design	Encapsulation	Polymorphism	Inheritance	High Level Design	Debugger	Source Code Control	Development Environment	Profiler	Top Down Design	Bottom Up Design		Software Engineering Total
1	0	256	0	0	20	0	0	0	0	600	0	876	0
2	0	3	0	0	1	0	0	0	0	160	80	244	1
3	0	16	3	0	13	11	0	0	0	240	0	283	1
4	0	8	8	1	21	48	0	0	0	213	213	513	2
5	4	0	11	0	13	48	0	32	12	160	0	280	2
6	0	48	0	11	21	5	0	0	0	40	20	145	2
7	0	108	0	0	8	0	0	64	0	50	0	230	3
8	12	80	0	0	144	60	0	0	0	360	10	666	3
9	0	144	0	16	11	72	0	24	0	192	32	491	3
10	0	96	16	16	72	144	6	480	8	96	108	1042	3
11	0	133	0	0	37	5	0	50	3	3	187	418	4
12	0	40	0	0	32	13	0	0	0	213	120	419	4
13	0	8	0	0	4	96	0	0	0	120	12	240	4
14	0	32	0	3	8	0	0	180	0	216	11	449	5
15	0	256	0	0	20	0	0	0	0	384	0	660	5
16	0	288	0	0	432	240	0	300	120	240	384	2004	5
17	0	0	0	0	5	0	0	0	0	300	240	545	5
18	0	320	0	0	48	8	0	80	0	360	16	832	6
19	0	0	0	0	40	5	0	16	0	144	0	205	6
20	0	120	36	128	90	432	0	0	0	200	36	1042	7
21	0	576	0	0	520	32	4	0	0	720	480	2332	7
22	0	120	0	0	72	48	0	24	0	120	144	528	7
23	12	32	32	32	0	72	0	320	8	0	0	508	8
24	0	0	0	0	5	21	0	160	0	600	0	787	8
25	0	180	0	0	0	53	0	0	0	180	144	557	8
26	0	140	0	0	240	80	8	0	0	400	60	928	9
27	0	24	16	3	48	5	0	0	0	288	144	528	9
28	0	160	1	0	128	60	3	20	0	420	192	984	11
29	0	160	0	100	153	0	0	0	0	220	160	793	12
30	0	200	0	0	576	53	0	80	0	400	576	1885	12
Ave.	1	118	4	10	93	54	1	61	5	255	112	714	5
Min.	0	0	0	0	0	0	0	0	0	0	0	145	0
Max.	12	576	36	128	576	432	8	480	120	720	576	2332	12

Table 6.4. Programmer Skills Data (Weighted Experience) for the Reusability Skills

Subject	Reusability Concepts (Skills)							Reusability Total	Total Classes Reused
	Procedural Reuse	Object Oriented Reuse	Modify Other's Code	Reusing Other's Code	Procedural Libs	Object Oriented Libs	Writing Software Libs		
1	160	0	96	96	107	0	0	459	0
2	0	0	0	4	0	0	0	4	1
3	0	0	53	53	0	0	0	107	1
4	0	0	120	107	0	0	0	227	2
5	5	0	11	3	0	0	11	29	2
6	48	0	11	11	87	0	0	156	2
7	0	0	67	67	0	0	0	133	3
8	80	0	80	60	40	0	13	273	3
9	192	0	72	72	15	0	0	351	3
10	96	0	36	36	27	0	4	199	3
11	233	0	11	11	0	0	0	255	4
12	96	0	0	0	120	0	0	216	4
13	6	0	10	4	24	0	0	44	4
14	40	0	0	0	32	0	0	72	5
15	192	0	96	200	240	0	0	728	5
16	216	0	216	216	156	0	32	836	5
17	300	0	240	240	219	0	0	999	5
18	48	0	40	40	0	0	0	128	6
19	240	0	80	27	0	0	20	367	6
20	250	0	200	200	200	0	0	850	7
21	600	0	240	320	128	0	0	1288	7
22	60	0	24	24	20	0	1	129	7
23	40	0	40	40	61	0	21	203	8
24	40	0	12	12	0	0	0	64	8
25	240	0	132	8	11	0	0	391	8
26	240	0	40	13	67	0	27	387	9
27	144	0	144	144	140	0	0	572	9
28	576	0	576	576	576	0	0	2304	11
29	200	0	132	200	192	0	6	730	12
30	576	0	576	576	165	0	32	1925	12
Ave.	164	0	112	112	88	0	6	481	5
Min.	0	0	0	0	0	0	0	4	0
Max.	600	0	576	576	576	0	32	2304	12

6.2 Results

The fifteen chosen skills (plus the three totals) are then related to the level of reuse that each of the programmers performed during Experiment One, namely, the number of classes that each reused. Do to the large variations among the individual programmer skill levels (partly, because the data is based on a self-evaluation), the attempt to correlate these skills to the amount of reuse using regression techniques is not recommended and, in fact, preliminary investigations using regression methods proved unsuccessful at finding any significant trends. However, using a chi-square test of independence on each skill produces some statistically significant results. The chi-square test analyzes a categorical data set for dependencies (trends) between two variables, in this case, a programmer skill area to the level of reuse achieved.

6.2.1 *The Chi-Square Test*

A chi-square test involves two variables, each divided into two or more categories (or levels). Since there are thirty subjects, a 2x3 chi-square test can be used to find significant trends or dependencies for each of the characteristics with the level of reuse performed. One variable is divided into two levels (the programmer skills) and the other variable is divided into three levels (the amount of reuse performed). The eighteen skills are each divided into a high group and a low group (providing the two columns for the chi-square) with the average providing the dividing point. In other words, everyone who rated themselves below the average is put into the low group. Everyone who rated themselves average or above is put into the high group. When a chi variable is divided into two levels, it is generally at the average to help meet some of the assumptions for using a chi-square test. For a more detailed explanation, please refer to [Walp85, OttR93]. Figure 6.1 shows a simple example of a chi-square frequency table.

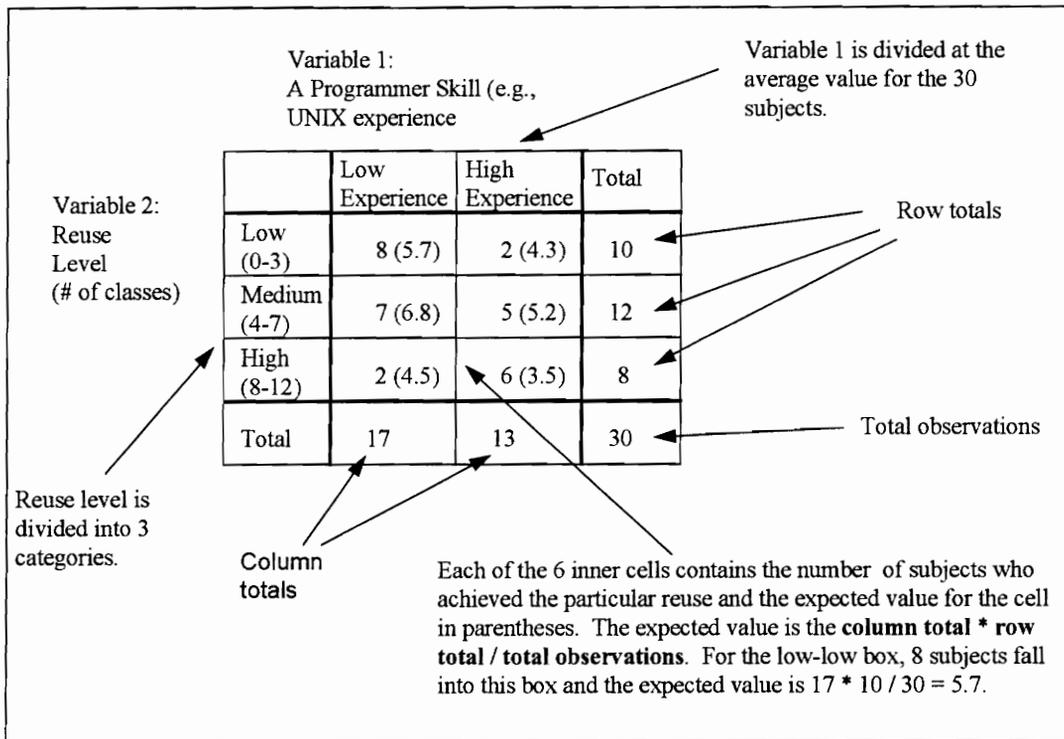


Figure 6.1. A 2x3 Chi-Square Explained.

The amount of reuse performed by each programmer provides the other dimension for the chi-square (three rows). The level of reuse is divided three ways: into low, moderate, and high reuse. Low reuse is defined as reusing zero to three classes; medium reuse is four to seven classes, and high reuse is eight or more classes reused. These divisions are created after the experiment is conducted. Among the thirty programmers, the maximum number of classes reused is twelve. The minimum is zero. Low reuse is defined to be the lowest third reuse levels of the subjects which is reusing zero classes, one class, two classes, or three classes. Moderate reuse is the next four levels (the middle third). So, reusing four, five, six, or seven classes is moderate reuse. High reuse is the remaining five levels from eight classes to twelve classes reused. Only one person reused twelve classes.

6.2.2 Chi-Square Tests on the Experimental Data

Once the two variables have been divided into levels, a chi-square frequency table can be constructed. Table 6.5 shows a 2x3 chi-square on the *UNIX experience* skill versus the level of reuse performed. The numbers in parenthesis represent the expected number of people for each box. The chi-square test relates the expected value to the actual value to determine if a dependency (a trend) between the two variables exists.

Table 6.5. A 2x3 Chi-Square for UNIX Experience Versus Level of Reuse Performed.

UNIX Experience vs. Level of Reuse	Number of Subjects With Below Average UNIX experience	Number of Subjects With Above Average UNIX experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	8 (5.67)	2 (4.33)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	7 (6.80)	5 (5.20)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	2 (4.53)	6 (3.47)	8
Totals	17	13	30

Given the chi-square frequency distribution for a particular skill (see Table 6.5 for the *UNIX experience* chi-square), the chi-square test of independence can be performed to determine if a trend in the data exists; that is, that the two variables are not independent. For the data in Table 6.5, the chi-square statistic calculates to $CHI = 5.5$ which results in a p-value of 0.06. A p-value is the significance level associated with the relationship between the two variables. The lower the p-value, the stronger the dependency between the two variables. For the *UNIX experience* characteristics, this frequency distribution occurs 6% of the time when no significant dependency exists. It is generally accepted that p-values at the 0.05 level or lower are statistically significant [OttR93]. However, for experiments involving human subjects and partially subjective data, many researchers [Walp85] use a 90% confidence interval (p-value = 0.10). Under this criterion, there

exists a significant dependency between the amount of *UNIX experience* the programmers have and the amount of reuse they each performed in Experiment One. For a more detailed explanation of how the chi-square statistic is calculated, please refer to [OttR93, Walp85].

6.2.3 Results of the Chi-Square Tests

Eighteen chi-square tests are performed -- one for each of the skills listed in the first column of Table 6.1 above. Table 6.6 shows the eighteen results (p-values) for the fifteen experience areas (skills) as well as three category totals. For the details of these tests (frequency tables and P-values), please refer to Appendix H, Tables H.1 through H.18. Table 6.6 also shows the minimum value, the maximum value, and the average value for each of the eighteen programmer categories for completeness. The average value represents the division point for the chi-square tests for that particular skill. The minimum and the maximum give a general idea about the range of the data. The last column of Table 6.6 is the p-value obtained from running a 2x3 chi-square test versus level of reuse on each of the eighteen programmer characteristics. Significant dependencies are shown in bold.

6.3 Analysis of the Chi-Square Data

The two characteristics that are highly dependent on level of reuse (p-value < 0.05) are the amount of reuse experience previously done (*Procedural Paradigm Reuse* and *Reusing Other's Code*). This result shows that programmers who have reused in the past tend to reuse more in the future. This fact suggests that programmers should learn to reuse early and should practice reuse in order to become better at reusing (or at least more likely to reuse).

Table 6.6. Results for Chi-Square Tests for the Eighteen Programmer Aptitudes and Level of Reuse.

Skill Number	Area of Experience / Aptitude	Average Experience Value	Low Value	High Value	P-value
1	<i>Procedural Paradigm Reuse</i>	164	0	600	0.03
2	<i>Reusing Other's Code</i>	112	0	576	0.04
3	<i>Assembly Language</i>	56	0	192	0.03
4	<i>UNIX Operating System</i>	268	3	600	0.06
5	<i>Modifying Other's Code</i>	112	0	576	0.06
6	<i>Using Software Libraries</i>	88	0	576	0.1
7	<i>Using Bottom-Up Design</i>	112	0	576	0.07
8	<i>Using High-Level Design</i>	93	0	576	0.11
9	<i>Encapsulation</i>	118	0	576	0.12
10	<i>C</i>	262	13	500	0.38
11	<i>Pascal</i>	174	5	600	0.55
12	<i>BASIC</i>	90	0	432	0.2
13	<i>Top-Down Design</i>	240	0	720	0.2
14	<i>LISP / Prolog</i>	9	0	32	0.33
15	<i>FORTRAN</i>	66	0	400	0.01
16	<i>Programming Total</i>	1,126	282	2,263	0.9
17	<i>Software Engineering Total</i>	714	145	2,332	0.02
18	<i>Reusability Total</i>	481	4	2,304	0.06

Another content area where experience is helpful was experience in *Assembly language programming* experience (p-value = 0.03). At first, this seems unintuitive; however, on reflection, a good (experienced) assembly programmer must be a good reuser. Assembly language, because of its extremely low level, forces programmers to use and reuse the code and macros written by others. Failure to do so would be like reinventing the wheel for every program. In Assembly language, this cost is prohibitively high.

Similar to the first two areas, *Modifying Other's Code* also is connected to level of reuse (p-value = 0.06) although the statistical significance is not at the desired 95% level (p-value = 0.05). Since subjective data is involved and human subjects, p-values at the 0.10 level and below signify a trend. In other words, a 90% confidence level for the resulting trend is acceptable. Given this fact, programmers who have much experience and aptitude at obtaining code written by other programmers and changing it to fit their own needs have a tendency to reuse code more often than programmers without this experience.

Experience with the *UNIX operating system* shows a dependency as well (p-value = 0.06). This is explained by the heavy modularity contained within UNIX. UNIX programmers and users know the advantages of reusing UNIX's small simple utilities like **grep** and **awk** and UNIX pipes to create larger, more complex commands. Furthermore, UNIX is heavily library-based. Anyone who has programmed in UNIX has used the standard libraries to access I/O devices, sockets, and many other aspects of the operating system.

Four other areas (*Using Software Libraries*, *Using Bottom-Up Design*, *Using High-Level Design*, and *Encapsulation Experience*) all have a significance level around the $p = 0.10$ level. *Using Software Libraries* is easily explained. This is simply a form of reuse. *Using High-Level Designs* can be seen as a first step towards reuse. If the whole system is designed before it is implemented, reuse possibilities are more apparent and more likely to be exploited. Experience *Using Bottom-Up Design* also creates reuse possibilities. In fact, the programmer may have some specific reusable components in mind during the design process. The last trait, *Encapsulation* experience can be helpful to a reuser. The more encapsulated one's design is, the more likely it is that other modular code (reusable components) can be easily integrated. It should be noted that all four of these trends are marginal at best. All four of these results occur around 10% of the time even if no dependency is present.

Interestingly, experience programming in *C*, *Pascal*, *BASIC*, and *Lisp/Prolog* did not show any dependency on the level of reuse. Several explanations include that *C* is a highly flexible language designed to make code writing quick and efficient, but not designed mainly for reuse. Although *C* is built on the idea of libraries, many *C* programmers view the libraries as a standard part of the language and since most compilers integrate these libraries seamlessly, many *C* programmers tend to forget of their existence. *Pascal* and *BASIC* are novice languages built more for learning tools than for practicing reusability. One reason to use these languages is to learn programming techniques. Unfortunately, that seldom includes the notion of software reuse. The two artificial intelligence languages, *Prolog* and *LISP*, are too recent to have many reuse possibilities. There are not enough repositories of reusable components for these languages. Furthermore, these languages are of a different variety than the first three (procedural) languages. *LISP* and *Prolog* are declarative languages designed for knowledge manipulation applications, not reusability.

Top-Down Design experience also does not show any dependency to reuse level either (p-value = 0.20). This is interesting since both experience with *High-Level Design Documents* and with *Bottom-Up Design* are dependent on the level of reuse performed. However, these dependencies are both marginally significant. Perhaps, there is a marginal dependency with *Top-Down Design* experience as well, but there is not enough data to show it. Firm conclusions can not be made without gathering more data on design experience and level of reuse.

It should be noted that no significant dependency is found for the above four language skills or for *Top-Down Design* experience. This does not say that no dependency exists. The only real conclusion that can be drawn is that this data set lacks the robustness to show a dependency between level of reuse and these five skills.

The last content area is FORTRAN knowledge. Interestingly enough, this shows a dependency on level of reuse in the opposite sense of all the previous ones. The more FORTRAN experience the programmer has, the less reuse he/she performed. Some explanations for this trend might be the extend to which FORTRAN varies from compiler to compiler and architecture to architecture. Reuse needs a stable environment to thrive. FORTRAN has not provided that. FORTRAN is also a language for mathematical programming. Although there are many libraries of FORTRAN routines in existence, they tend to be seamlessly integrated into the compiler (like C libraries) and FORTRAN programmers use them without realizing that they are reusing other programmer's code.

Three more chi-square tests of independence are performed on the total experience in the three skill categories. The first total, *Total Programming Language* experience (*C, Pascal, BASIC, Assembly, FORTRAN, and Lisp/Prolog*) shows no trend (p-value = 0.9). In fact, this data happens 90% of the time. This is expected since three of these five areas show no dependency with the level of reuse at all. This result, a lack of a dependency, suggests that experience in programming languages does not necessarily make for good reusers. Again, no significant result has been found. Only through further tests can a positive (firm) statement be made about *Programming Language* experience and tendency to reuse.

The second total, *Software Engineering* skills, is composed of *Using Bottom-Up Design, Using High-Level Designs, Encapsulation, and Top-Down Design*. A significant dependency is found (p-value = 0.02), which is expected. Programmers with a sound foundation of Software Engineering skills know the benefits of source code reuse and error-free programs.

The last total represents the four areas that can be viewed as direct *Reusability* Experience (*Procedural Paradigm Reuse, Modifying Other's Code, Reusing Other's Code, and Using Software Libraries*). Again, a significant dependency is found (p-value = 0.06). Again,

the notion of practicing software reuse is reinforced. Programmers that have learned to reuse tend to reuse more often now.

6.4 Conclusions

Goal three is to examine the characteristics and experiences of programmers who reuse often. This chapter has presented data on thirty programmers, their past experiences, and how much reuse each performed in a controlled environment. A set of eighteen programmer characteristics and experiences is compared with the level of reuse performed. These eighteen skills are divided into three categories: **Programming Language** skills, **Software Engineering** skills, and **Reusability** skills.

The question posed at the onset of this chapter is:

What are some of the characteristics of programmers who reuse C++ classes?

This experiment answers this question to some degree and suggests some areas for further investigation.

Summarizing the results presented above, the overwhelming, and perhaps obvious fact is that programmers who have reused before and who have had much of practice reusing, tend to reuse more now. The set of reusability skills and the reusability category skill total support this conclusion. This reuse practice must be in actually reusing software; just writing code does not improve one's ability (or likelihood) to reuse. The set of programming skills shows this fact. A well-experienced excellent C programmer does not necessarily make a good source code reuser.

Another significant result is that a sound foundation of Software Engineering principles can be found in the more frequent code reusers. The Software Engineering skill set supports this fact. Significant dependencies between amount of reuse and experience with

formal design methodologies and encapsulation are found in the analysis of Experiment One. Perhaps, and it is hoped that, following software engineering principles makes reuse possibilities more apparent and easier to perform.

To answer the research question directly, programmers with training and/or practice at reusing code and programmers with a sound foundation of software engineering principles under them tend to reuse more often than programmers lacking these skills.

The six skills that did not show a significant dependency with level of reuse (*C*, *Pascal*, *BASIC*, *Top-Down Design*, *LISP/Prolog*, and the *Programming Skills Total* -- see Table 6.3) all have the potential to be dependent given another, larger data set. For this data, no firm conclusions can be made about the skill areas that did not show a significant dependency. Clearly, the skills with marginally significant dependencies (like *Using High-Level Designs*) have higher probabilities for showing a dependency in a different data set, and these skills should be investigated further.

For the sixteen skills that were eliminated because of a lack of meaningful data, it is believed that some of these would show dependencies on the level of reuse (for example, *Experience Using C++*), but another experiment must be performed and more data collected before anything can be concluded about these skills. The subjects of this experiment are not experienced enough in many areas to provide an exhaustive list of the traits of programmers who tend to reuse.

Chapter 7

■ Reusability and Characteristics of C++ Classes

The last goal of this research focuses on the characteristics of reusable C++ classes and how they can be measured. Goal four centers around the idea that some software metrics can be used to gauge the reusability of C++ classes. Again, this research deals with general purpose C++ classes such as abstract data type classes. If the reusability of a C++ class can be measured, the measurement can be used to decide if a particular C++ class should be refined, rewritten, or added to a library of C++ classes for reuse by other programmers. The specific research questions guiding this chapter are:

1. What are some characteristics of C++ classes that can be used to gauge their reusability?
2. Are there software metrics that can be used to determine the reusability of a C++ class?

A lot of research has been done in the field of software reuse; some of it focuses on reusing designs [Chen94, Lane84]. Other focuses on meta-languages used to generate reusable pieces of code [Leib88, Neig84]. Chapter 2 provided more detail about research into software reusability. This chapter focuses on the source code aspects of C++ classes. The subjects in Experiment One were polled about what traits make a C++ class more reusable than another C++ class with similar functionality. These opinion polls aid the construction of a hierarchical set of characteristics to determine the reusability of a C++ class. This set of characteristics is used in the design of Experiment Three of this research. These characteristics are correlated to some common software metrics such as McCabe's Cyclomatic Complexity metric, the object-oriented metrics explained in Chapter 3, and some new metrics which measure the complexity of a C++ class interface.

Many of the identified characteristics have been addressed by other researchers in Software Engineering. Gibbens [Gibb88] and Towe [Towe92] discuss how documentation can be measured. They had only limited success. Barnes [Barn91] and Bollinger [Boll90] focus on the economics of reusing classes: the costs and benefits to the organization and the programmers performing the reuse. Li [LiWe93] uses some of the object-oriented metrics discussed in Chapter 3 to measure the complexity and maintainability of Classic Ada objects. This research focuses on measuring the reusability of C++ classes by examining the source code of the classes.

7.1 Opinions on Reusability

At the conclusion of the first academia experiment of this research, the thirty participants were asked what characteristics are the most important for a C++ class to be reusable. The top two responses were Ease of Use of the class (26%) and Documentation for the class (26%). Table 7.1 shows the percentages for the rest of the response categories.

Table 7.1. Percentages of Programmer Opinions for What Traits Make a Class Reusable

Number	Characteristic	Percent Mentioning
1	Documentation	26 %
2	Ease of Use	26 %
3	Generality of Functionality	18 %
4	Organization of the Source Code	12 %
5	Clarity of the Source Code	8 %
6	Completeness of the Functionality	6 %
7	Reliability	4 %

From these programmer opinions, a hierarchical set of reusability characteristics for a C++ class is created. This set is presented in the next section. Afterwards, the remainder of this chapter concentrates on the whether some source code metrics can be used to gauge the reusability of a C++ class.

7.2 Characteristics of Reusable C++ Classes

Several decompositions (hierarchies) centering on measuring software exist. Balci presents an extensive hierarchy decomposing many aspects of Computer Science into more measurable pieces including the concept of reusability [Balc93]. Korson and MacGregor [Kors92] examine some characteristics of reusable classes. Johnson and Foote [John88] delve into the traits of reusable class libraries and present some rules for creating reusable classes. [Chen92] presents a hierarchy for reuse. This section presents a partial decomposition of reusability derived primarily for the empirical data gathered in Experiment One.

Many of the identified traits of reusable classes are difficult or impossible to measure. Many are subjective. In light of these facts, this research concentrates on measuring the actual source code of the C++ class. Measuring the source code can shed light on the class's ease of use, complexity, reliability, functionality, and organization-- characteristics which have been cited by some of the above researchers as influencing reusability. Furthermore, many of these characteristics appear in the opinion poll presented above.

Figure 7.1 shows the set of the identified characteristics to measure the reusability of a C++ class. The set is hierarchical in nature. There are four broad areas of the identified: functionality, reliability, usability, and documentation. The functionality of a class certainly impacts its reusability. This research concentrates on abstract data type C++ classes. There are many other domains of application such as GUI classes and parallel processing classes, but to maintain a valid test bed, this research limits the reusable classes to abstract data types and general purpose low-level classes.

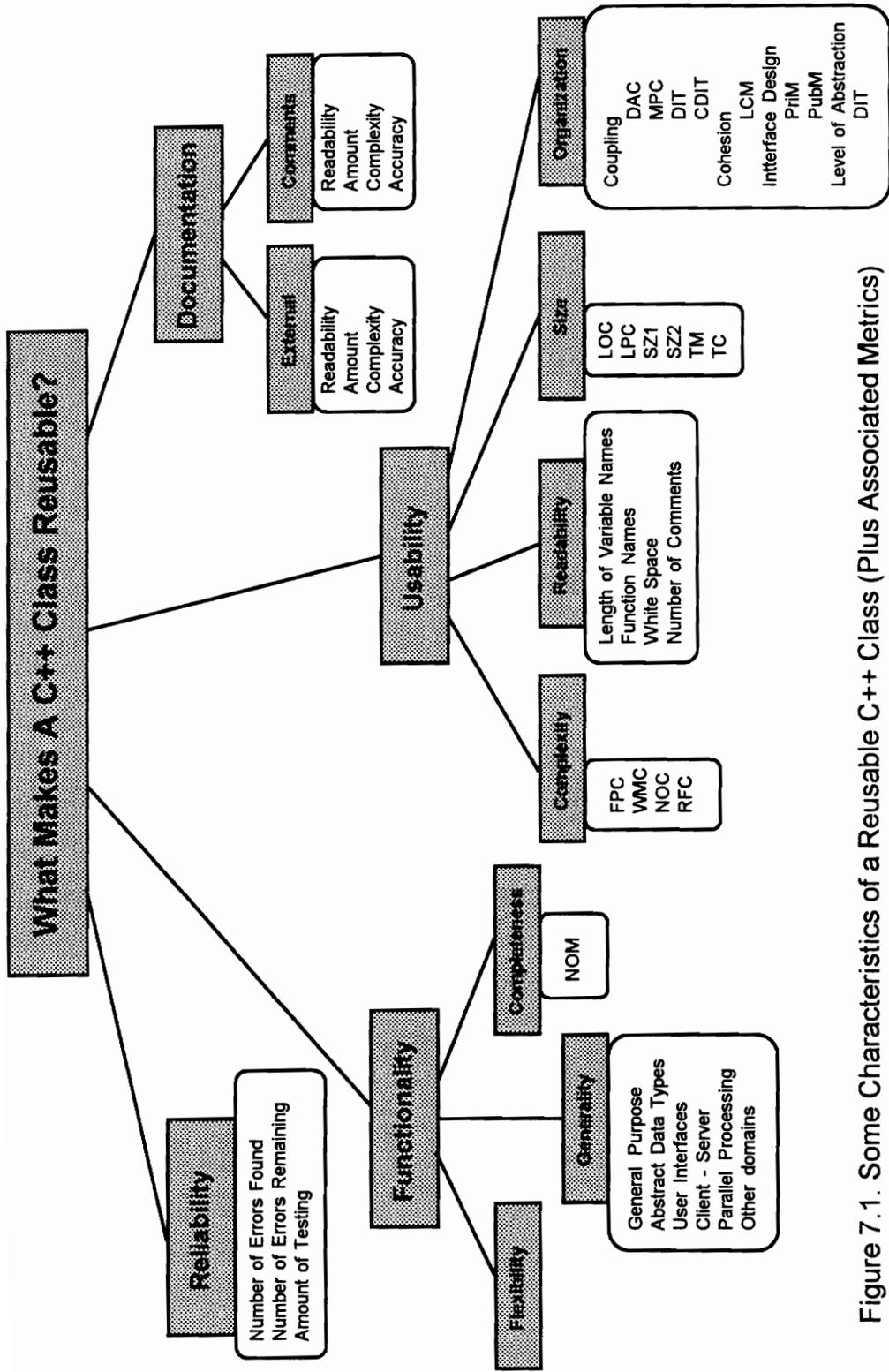


Figure 7.1. Some Characteristics of a Reusable C++ Class (Plus Associated Metrics)

The classes used in Experiment Three do not have any reliability data associated with them which is typically for C++ class libraries. For the purposes of this research, all library classes are assumed to be equally reliable.

The third major area on Figure 7.1 is documentation. Documentation is clearly one of the major aspects of a C++ class that affects its reusability. Clear and accurate documentation make a class easy to learn and understand. Unfortunately, documentation is difficult to measure (other than the amount) and studies have shown that without some subjective evaluation or artificial intelligence-style processing, measuring the quality of documentation has not yet been achieved [Gibb88, Towe92].

The last major division, class usability, is where this research concentrates. The opinion poll conducted during Experiment One and presented at the beginning of this chapter motivates this concentration. The ease of use of a class is a common response to the question "What makes a class reusable?". Experiment Three attempts to measure how easy a C++ class is to use.

The usability of a C++ class is divided into four more aspects of a class: complexity, size, readability, and organization. All of these except class readability are addressed below. Software metrics exist to measure many aspects of software including size and complexity. Perhaps, some of these code metrics can be correlated the reusability of a class as well. Figure 7.1 shows some software metrics (three-letter acronyms in rounded cornered boxes) that can be used to measure some of these traits. The metrics investigated in this chapter focus on the object-oriented metrics described in Chapter 3.

7.3 Experiment Three: Design and Results

A set of fifteen C++ classes, whose reusability potential has been determined by polling some industry experts, is used as a test bed to correlate the reusability of a C++ class with some of its measurable traits. Five sets of classes are given out to fifteen industry experts.

Each set contains three different (functionally equivalent) representations of a C++ class. The five class types are a random number generator class, a regular expression class, a stack class, a singly linked list class, and a string class. Since each set has three classes, there are fifteen total classes being examined by each subject.

The five sets of classes are a random number generator class set consisting of three stand-alone classes that generate random numbers on command, a regular expression class set which handles all type of regular expressions and pattern-matching functions, a stack and a list class set which provides the basic functionality associated with these two data structures, and a string class set provides an abstract data type class for dealing with the manipulation of dynamic arrays of characters.

Some of these fifteen classes are independent classes; that is, they have no parent class or ancestor classes, such as the random number generator classes. Other classes are at some level in a class hierarchy. If this is the case, then the entire hierarchy down to the selected class is provided to the experts for examination and evaluation. For example, one of the selected stack classes is based on a list class which is based on a collection class. All three of these classes are given to the experiment subjects for examination. Both of these situations are common in the C++ class libraries of today. Sometimes, the classes are independent (Depth in Inheritance Tree = 0), and other times, they can have two or three ancestors in the library's class hierarchy.

The experts rank each set of classes according to how reusable they perceive the classes to be. Each of the five sets of classes is treated separately. The experts examine all three of the string class implementations and decide which is the most reusable and which is the least reusable leaving one class between the two extremes. To aid the subjects in their evaluations, an evaluation form (Appendix I) is provided that forces the experiment subjects to think about the reusability of the classes in five areas: complexity, ease of use, organization, completeness of functionality, and documentation. These five indicators

stem from the opinion poll conducted during Experiment One. They are used because they allow the industry experts to more accurately rate each class in terms of its reusability.

After all fifteen subjects have ranked the classes, the average rank for each class is calculated. This is shown in Table 7.2, column two. Each class set has a rank order from one to three. Next, the five indicator average values are calculated from the evaluation forms that the fifteen subjects of the experiment fill out. These averages are shown in the last five columns of Table 7.2.

Table 7.2. The Fifteen Classes, Their Rank, and Averaged Reusability Scores

Class Name	Rank in Set	Complexity Score	Organization Score	Ease of Use Score	Functionality Completeness Score	Documentation Score
Random1	3	5.5	4	5.5	4.5	3.3
Random2	2	8.3	4.7	6.7	4.7	2.7
Random3	1	5.4	7.6	8.4	9.4	8.8
Regex1	2	8	7.2	7.6	4.8	4.4
Regex2	3	2.6	2.9	2.6	5.1	2.4
Regex3	1	7.8	8	8.3	8	8.6
Stack1	3	4	4.6	3.4	7.4	3.5
Stack2	1	9.5	8.5	9.8	7.8	4.5
Stack3	2	6	7	6	7	3.7
String1	2	5.5	6.5	6	9	4.8
String2	3	4.2	3.8	4.4	6	3.4
String3	1	6	7.1	8.6	8.9	8.9
List1	3	4.8	4.8	4.6	6.4	3.5
List2	1	8	8.3	7.1	7.4	4.1
List3	2	5.3	4.7	5.3	6	2.9

The rank column shows the rank that the class achieved in its set of three. The rank is decided upon by the number of experts who rated this class as the best. The next five columns show the average rating (from 1 to 10) of the five most often cited characteristics of a reusable class. Respectively, they are the class complexity, the class ease of use, the class organization, the class completeness of functionality, and the class documentation. In all five cases, zero is the lowest score and ten is the highest, so for the complexity indicator, a score of one is very complex and a score of ten is very simple.

The next step is to compute the values of some software metrics for the five class sets. If the classes have an associated hierarchy, the classes within the hierarchy are also analyzed and average class metrics are computed. These metrics, obtained for the fifteen classes using the Metrics Generator, are the essentially same ones described in Chapter 3 and used in Chapter 4. Three new metrics are used in this study. The first of the three new metrics is the ratio of private methods in the class to the total number of methods in the class. This should give an indication of how much processing is going on behind the scenes in a particular class. The correlated counterpart to this metric is also used: the ratio of public / protected methods to the total number of methods in the class. The third metric being used is named DIT: *Average Depth in Inheritance Tree*. Since some of these classes have an associated hierarchy, the average depth of the hierarchy can be examined (DIT) or just the depth of the selected class (CDIT). For easy of reference, the metrics used and their acronyms are presented briefly again.

- 1) **PriM** - Private Methods / Total Methods in Class
- 2) **PubM** - Public Methods / Total Methods in Class
- 3) **CDIT** - Class Depth in Inheritance Tree
- 4) **TC** - Total Classes in Hierarchy
- 5) **TM** - Total Methods in Hierarchy
- 6) **SZ1** - Average Size One (Semicolons) for Class Hierarchy
- 7) **SZ2** - Average Size Two (Local Data + Methods) for Class Hierarchy

- 8) **DIT** - Average Depth in Inheritance Tree for Class Hierarchy
- 9) **FPC** - Average Functions Per Class for Class Hierarchy
- 10) **NOC** - Average Number of Children for Class Hierarchy
- 11) **WMC** - Average (McCabe) Weighted Methods Per Class for Class Hierarchy
- 12) **RFC** - Average Response for A Class for Class Hierarchy
- 13) **DAC** - Average Data Abstraction Coupling for Class Hierarchy
- 14) **MPC** - Average Message Passing Coupling for Class Hierarchy
- 15) **LCM** - Average Lack of Cohesion of Methods for Class Hierarchy

Table 7.3 shows the values for this set of object-oriented metrics for the fifteen classes of Experiment Three.

Table 7.3. Metrics Values for the Fifteen Classes in Experiment Three

Class Name	P r i M	P u b M	C D I T	T C	T M	S Z 1	S Z 2	D I T	F P C	N O C	W M C	R F C	D A C	M P C	L C M
Random1	0	1	0	1	5	42	7	0	5	0	9	11	0	3	1
Random2	0	1	1	1	14	50	16	1	14	0	23	24	0	4	1
Random3	0.5	0.5	0	1	12	117	23	0	12	0	31	21	0	2	1
Regex1	0.14	0.86	0	1	7	52	9	0	7	0	28	20	1	0	1
Regex2	0.3	0.7	1	1	25	148	31	1	25	0	44	69	1	28	2
Regex3	0	1	0	1	11	113	21	0	11	0	41	44	0	1	1
Stack1	1	0	2	9	79	26	10.6	1	8.8	0.57	11	14	0.4	4	1
Stack2	0	1	1	1	12	26	13	1	12	0	13	22	0	5	1
Stack3	0	1	1	4	87	62	25.3	0.5	21.8	0.5	29	48	0.3	12	1
String1	0.19	0.81	0	2	111	249	59	0	55.5	0	104	218	1	134	1.5
String2	0.04	0.96	1	3	44	115	19	0.7	14.7	0	25	36	0.7	2	1.3
String3	0.06	0.94	0	2	87	316	51.5	0	43.5	0	141	116	1	39	2.5
List1	0.08	0.92	3	8	67	24	9	0.9	7.1	0.5	10	11	0.5	2.5	1
List2	0.03	0.97	0	4	43	45	12.3	0.5	20.8	0.5	23	21	0	6	1
List3	0.07	0.93	0	3	76	75	29.7	0.3	25.3	0.34	35	56	0.3	14	1.3

These metrics have all been explained in Chapter 3. Now that the evaluations of the reusability of the fifteen classes has been presented and the metrics values for each of the classes, the two data sets can be analyzed to determine if the reusability of a C++ class can be predicted from some source code metrics such as those presented in Table 7.3.

7.4 Experiment Three: Analysis

Some preliminary analysis is done on the data sets to determine if any correlations exist. Since each set of three classes is ranked by the experts, three groups of classes can be formed: the most reusable class in each set, the least reusable class in each set, and the class between the two extremes. The three groups can be analyzed for statistically different variances and means as done in Chapter 6. Table 7.4 shows the means for all of the software metrics for the five class sets. The last row of the table states whether or not there is a statistical difference among the three group means. Again, Tukey's procedure [OttR93] is used to determine these answers.

Table 7.4. Averages for the Three Reuse Groups for the Fifteen Classes

Class Name	P r i M	P u b M	C D I T	T C	T M	S Z 1	S Z 2	D I T	F P C	N O C	W M C	R F C	D A C	M P C	L C M
Most Reusable (Rank 1)	0.3	0.7	1.2	4.8	58.2	77.4	19.8	0.8	16.2	0.3	29.9	37.3	0.6	8.1	1.3
Partially Reusable (Rank 2)	0.2	0.9	0.2	2	60.4	157	33.2	0.1	27.6	0.1	52	82.6	0.5	16.1	1.4
Least Reusable (Rank 3)	0	1	0	1.6	17.4	57.2	14.3	0.5	13	0.1	22.1	26.2	0.2	3.2	1
Means Differ?	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No

Interestingly enough, for all fifteen of the software metrics, there are no statistical differences in the group means. It seems the nebulous quality of the concept of reusability cannot be measured by examining only the source code. This is not unexpected. Examining the hierarchy of the traits of reusability presented above, the actual source code of the class is only a small part of what needs to be measured to characterize the reusability of a C++ class. Since measuring all of these attributes is beyond the scope of this research, these fifteen metrics are investigated with a less rigorous statistical procedure, correlation coefficients.

If these fifteen software metrics (Table 7.3) are correlated to the five subjective expert measures (Table 7.2), the correlations can be examined to generate rules about the source code of a reusable C++ class. In particular, as a C++ class implementation becomes more reusable, does its metric value (say, Size One) increase or decrease?

The first step in this analysis is to generate the matrix of correlation coefficients for the five indicators versus the fifteen software metrics. Table 7.5 shows these correlation coefficients.

Table 7.5. Correlation Coefficients for Five Indicators Versus the Fifteen Metrics

Metric Name	<i>Complexity Indicator</i>	<i>Organization Indicator</i>	<i>Ease of Use Indicator</i>	<i>Functionality Indicator</i>	<i>Documentation Indicator</i>
PriM	-0.46	-0.14	-0.34	0.28	0.03
PubM	0.46	0.14	0.34	-0.28	-0.03
CDIT	-0.47	-0.33	-0.57	-0.06	-0.32
TC	-0.32	-0.16	-0.47	0.12	-0.29
TM	-0.3	0.04	-0.21	0.47	-0.06
SZ1	-0.15	0.2	0.29	0.56	0.54
SZ2	-0.18	0.16	0.17	0.56	0.35
DIT	-0.37	-0.61	-0.72	-0.4	-0.66
FPC	-0.11	0.17	0.11	0.5	0.19
NOC	-0.1	0.05	-0.32	0.07	-0.35
WMC	-0.19	0.12	0.11	0.45	0.27
RFC	-0.13	0.13	0.08	0.48	0.2
DAC	-0.39	-0.2	-0.25	0.02	-0.06
MPC	-0.24	0.03	-0.02	0.41	0.11
LCM	-0.37	-0.15	-0.04	0.22	0.23

The first three columns of Table 7.5 (Complexity, Organization, and Ease of Use) fall under the Usability category of the reusability hierarchy, so they are examined together. Columns four and five are treated separately. Examining the rows of Table 7.5, the fifteen metrics are divided into four groups. The first group (PriM, PubM, CDIT) are metrics pertaining only to the selected class being reused and not to the class hierarchy containing the class (if any). The next group (TC, TM) are two size metrics for the entire class hierarchy (the library's class hierarchy). The third group (SZ1, SZ2) are averaged size metrics for the whole class hierarchy, and the last group is the set of complexity metrics for the entire class hierarchy.

The next step in this analysis is to determine a test statistic. For the purposes of this exploratory research, an alpha level of $\alpha = 0.10$ is used. The $\alpha = 0.05$ level is relaxed because this research is interested in candidate reusability metrics. Further experimentation and analysis needs to be performed on any metrics identified as possible "reusability indicators".

Since there are fifteen classes, the degrees of freedom for a T Test statistic is thirteen (number of observations minus two). At an alpha level of $\alpha = 0.10$, the T statistic for significance is 1.77 (obtained from a T distribution table) [OttR93]. Using the formula,

$$T_{13,0.10} = R * \text{sqrt} (13 / (1 - R^2))$$

where $T_{13,0.10}$ = the T Test statistic (1.77) and
R = the correlation coefficient of significance,

and solving for the significance level R, R is calculated to be 0.44. Interpreted, this means that if any of the correlation coefficients in Table 7.5 are 0.44 or greater or -0.44 or less, then there is a statistically significant correlation between the two variables in question.

Using this criterion, several of the metrics correlate with the first three columns of Table 7.3, the Usability indicators. In particular, the metrics that correlate are PubM, PriM, CDIT, TC, and DIT. DAC and LCM are close to the significance level and are also examined.

In all of these cases except percentage of public methods, as the metric decreases, the evaluated score in all three columns increases. There is a negative correlation between the metrics and the Usability indicators.

Taken singly, as the *Percentage of Private Methods* (PriM) decreases (which means the *Percentage of Public Methods* (PubM) increases), the class is evaluated to be more reusable. The *Percent of Private Methods* (PriM) correlates at the -0.46 level which is significant under accepted statistical norms. Intuitively, a class with a lot of private

methods has a lot of action behind the scenes which may make the class harder to understand and therefore, harder to use.

As the class's depth in the library's class hierarchy decreases (CDIT), the reusability increases. Intuitively, the shallower the class is in the class hierarchy, the fewer classes that must be understood to reuse the class. Similarly, for the entire hierarchy, as the *Average Depth in the Inheritance Tree* (DIT) decreases, the reusability of the selected class increases.

Although less code must be understood to reuse the class, this result needs to be taken in moderation. A library that is one large class hierarchy may be easier to use if many classes from the library are to be reused. However, if only one class is needed from the entire hierarchy, the less coupled the library classes are, the less the reusing programmer needs to understand, and the easier it is to reuse a class from the library.

The next metric, *Total Number of Classes* (TC) behaves like the DIT metric. As the *Total Number of Classes* (TC) in the hierarchy decreases, the reusability increases. This is true if only one class is being reused. Again, the "understanding the library" overhead costs overlap if more than one class is used from the library hierarchy.

The next metric is the *Data Abstraction Coupling* (DAC) which marginally correlates to the perceived class complexity indicator. According the Software Engineering principles, as coupling decreases, code quality increases [Cont86, Somm92]. This trend applies here. As the coupling decreases, the class becomes more reusable. Again, less code needs to be understood to use the class.

The last metric that marginally correlates with these three reuse evaluators is the *Lack of Cohesion of Methods* (LCM). As the lack of cohesion decreases, the class is more reusable. Put another way, as the cohesion within the class increases, so does the class's

reusability. Again, Software Engineering predicts this [Cont86, Somm92]. More cohesive modules are easier to write, debug, use, and maintain.

The next column of Table 7.3 is the reusability indicator for the completeness of the functionality of the class. The correlation coefficients are shown in Table 7.5, column four. This evaluator is treated separately because it addresses a different part of the reuse hierarchy, that of functionality. More complete classes are more reusable. However, it seems that as the completeness of the class increases, so does the class's complexity. This conflicts with the complexity indicator for reusability.

Examining column four of Table 7.5, several of the metrics seem to correlate with the completeness of the class. In particular, the size and complexity metrics: *Total Classes in the Library* (TC), *Total Methods in the Hierarchy* (TM), *Size One* (SZ1), *Size Two* (SZ2), *Functions Per Class* (FPC), and *Weighted Methods Per Class* (WMC). All five of these metrics are measuring the relative size of the selected class to some degree. As the size of the class increases, so do these metrics, and so does the class's completeness evaluator. Unfortunately, complexity also tends to increase with size.

Three other metrics correlate with the completeness evaluator: *Response for a Class* (RFC), *Message Passing Coupling* (MPC), and *Depth in the Inheritance Tree* (DIT). *Response for a Class* measures class complexity by measuring the number of local methods (size) plus the number of non local function calls. As this metric increases, so the class's completeness. Because the number of local methods is part of this metric, more complete class score high because they tend to have more methods than incomplete classes.

Message Passing Coupling (MPC) also correlates positively. This is unintuitive. As the coupling increases, the reusability should decrease. However, this metric relates to the number of calls a class makes to other classes. It tends to follow the *Response for a Class* (RFC) metric which increases with class size.

The fifth column of Table 7.5 is the documentation indicator. Since all of these metrics are code metrics, any correlations with this evaluator are purely coincidental and need not be investigated. However, if the documentation of this class set were to be examined, this indicator could be used to draw some statistical inferences.

7.5 Summary / Conclusions

Summarizing, a hierarchical decomposition of some of the characteristics of a reusable C++ class is constructed based on the experiences and opinions of the subjects in Experiment One and on several existing hierarchies in current literature. Next, fifteen experts (industry C++ programmers) use this set of characteristics to gauge the relative reusability of a set of fifteen C++ classes against five indicators. The class set is also analyzed by the Metrics Generator providing fifteen different software metrics for each class. Finally, the reusability indicators are correlated with the software metrics to create some rules about reusable C++ classes.

From the correlation analysis, a series of rules can be generated about the reusability of C++ classes. These rules are based on the premise that as the ease of use of a class increases, so does its reusability. This result comes from the opinions of the subjects in Experiment One. Furthermore, as the completeness of functionality of a class increases, so does its reusability. Using this result, the following statements can be made about the correlations of the fifteen software metrics and the fifteen C++ classes.

The reusability of a C++ class increases as:

1. the depth in the inheritance tree of the class (CDIT) decreases.
2. the average data abstraction coupling of the hierarchy (DAC) decreases.
3. the total number of classes in the class hierarchy (TC) decreases.
4. the ratio of public methods to total methods (PubM) increases.
5. the ratio of private methods to total methods (PubM) decreases.
6. the total number of methods in the class hierarchy (TM) decreases

7. the average depth in the inheritance tree (DIT) of the hierarchy decreases
8. the lack of cohesion of methods (LCM) of the hierarchy decreases

A secondary measure of reusability is the completeness of the class. As the class is more functionally complete, it is more reusable. Therefore reusability increases as the size of the class increases (TC, SZ1, SZ2, FPC, WMC). However, this result contradicts some of the above guidelines since as size increases, complexity increases. Since, the class usability rates higher as an indicator of reusability, the guidelines that increase ease of use take precedence over this result on functional completeness. More complete classes, although more reusable, suffer from increased complexity and become more difficult to use and reuse.

This analysis represents only a start of validation on the reusability characteristics and the metrics. Metrics need to be assigned or created for each of the identified traits in Figure 7.1 and combined to form a composite reusability trait. This is left as an area of future research.

Chapter 8

■ Conclusions

This research has explored several facets of the area of software reusability in the object-oriented paradigm from its effect on the software development process to the traits of programmers who tend to reuse. Two different types of C++ class reuse have been investigated: black box reuse and white box reuse. Their relative benefits have been ascertained. Models have been developed to characterize the effects that C++ class reuse has on both the software development process and the final product. Lastly, some traits of reusable C++ classes have been identified and some candidate metrics have been suggested for measuring the inherent reusability of C++ classes.

This research effort is divided into four major goals. Each of these goals and their major conclusions are summarized in the remainder of this chapter. Lastly, some of the implications of this research and areas of future work are examined.

8.1 Goal One

The focus of Chapter 4 is on Experiment One and the first goal of this research.

GOAL ONE is to characterize the effects of reusing C++ classes on the software development process.

Nine process indicators are used to ascertain the impact that reuse in the object-oriented paradigm has on the software development process. The effects on each of these indicators are modeled using multiple linear regression model. Experiment One is performed to realize this goal. The major results from Experiment One are summarized below.

1. The amount of black box reuse achieved is the overriding indicator of the changes in the software development process for these nine process indicators. Most of the size and complexity measures have little significance in the nine process indicator models.
2. Increasing the amount of black box reuse decreases the development times in almost all the phases of the software life cycle; overall integration effort is reduced, and the final system reliability improves.
3. Increasing the amount of white box reuse causes the total development time to increase.
4. Inheritance, though a useful design tool, causes more integration problems and lowers final system reliability if the parent classes are from class libraries rather than just written from scratch.
5. As the coupling among the classes increases, the final system reliability decreases.

All five of these results are obtained from the analysis of the nine models developed for the software process indicators.

Under this goal, the effects of C++ class reuse on the software development process have been explored. Some interesting results regarding the varying effects of black box reuse versus white box reuse (inheritance) have been observed. Black box reuse comes out very favorably, and white box reuse very poorly. Reuse using inheritance (white box reuse) is a difficult concept. In fact, one could argue that it is a type of library design. Deriving new classes from existing library classes is expanding the library in order to reuse. The level of difficulty involved in doing this may have been too much for the subjects of Experiment One. Most of the subjects had little to no experience using inheritance.

The libraries chosen for use in Experiment One may have also biased the results towards black box reuse. Perhaps, if a different set of C++ class libraries are chosen that are designed to be easily expanded (white box reuse), these results may have turned out differently.

8.2 Goal Two

A second experiment (Experiment Two) is performed to characterize the effects of two types of C++ class reuse on software development. Chapter 5 addressed the second goal of this research.

GOAL TWO is to measure the effects of the different types of C++ class reuse (black box reuse and white box reuse) on the software development process.

Experiment Two shows that black box reuse positively affects the software development process in several areas including *Time Spent in C++ Libraries*, *Debugging Time*, *Total Development Time*, and *Compile-Time, Logic, and Run-Time Error Totals*. Furthermore, the benefits achieved are greater than performing white box reuse or writing all of the source code from scratch. The major conclusions under this goal are shown below.

1. Performing black box reuse causes greater benefits to the software development process than white box reuse in both the time spent in the various life cycle phases and on the number of errors encountered during development.
2. White box reuse of the underlying data structure of a program is no better than writing the data structure from scratch in terms of the benefits realized during the software development process.

Experiment One demonstrates that C++ class reuse is a worthwhile beneficial endeavor. This experiment points towards limiting reuse to simple black box components and not attempting to reuse with modification. For almost all of the process indicators used in Experiment Two, white box reuse showed no benefit over writing the class from scratch.

8.3 Goal Three

The sixth chapter of this document focuses on the programmers and the concept of software reuse. Goal three of this research effort pertains to the human side of software reuse.

GOAL THREE is to determine what programmer characteristics influence or predict the level of reuse performed.

Thirty-one programmer characteristics are investigated for their dependencies on the level of reuse achieved by the thirty subjects in Experiment One. From the analysis, the following conclusions are made:

1. Programmers who have reused before and who have had much practice reusing software, tend to reuse more now.
2. Reuse practice must be in reusing software; just writing code does not improve one's ability (or likelihood) to reuse. Experience in general programming skills and programming languages does affect the level of reuse for this set of observations.
3. A sound foundation of Software Engineering principles can be found in the more frequent code reusers.

These conclusions are based on the experiences of a class of Computer Science seniors. Some of them have industry experience, but most are new to the object-oriented paradigm. These conclusions certainly apply to novice industry programmers, but may not generalize well to more experienced software developers, who have years of practice and experience developing object-oriented software.

8.4 Goal Four

The last goal of this research effort is an exploratory goal looking for some reusability code metrics. Experiment Three is conducted to achieve goal four.

GOAL FOUR is to identify some of aspects that make a C++ class reusable, and to assign metrics to some of these aspects to measure the reusability of a C++ class.

Figure 7.1 shows the list of some of the traits found in a reusable C++ class. This set of traits is based on the opinions and experiences of the subjects involved in this research as well as some hierarchies of measurable traits of reusability found in the current literature.

The conclusions under goal four are a set of general guidelines about the behavior of some object-oriented software metrics and the reusability of a C++ class.

The reusability of a C++ class increases as:

1. the depth in the inheritance tree of the class (CDIT) decreases.
2. the average data abstraction coupling of the hierarchy (DAC) decreases.
3. the total number of classes in the class hierarchy (TC) decreases.
4. the ratio of public methods to total methods (PubM) increases.
5. the ratio of private methods to total methods (PubM) decreases.
6. the total number of methods in the class hierarchy (TM) decreases
7. the average depth in the inheritance tree (DIT) of the hierarchy decreases
8. the lack of cohesion of methods (LCM) of the hierarchy decreases

All of these guidelines apply to general purpose C++ classes such as abstract data type classes. They are only guidelines because the reusability of a C++ class can not be gauged by just examining the source code. Figure 7.1 illustrates this fact. There are other factors that must be measured like the class documentation and reliability to accurately determine the reusability of C++ classes.

8.5 Implications of Research

There are many implications that can be derived from this research. Goals one and two of this research indicate that black box reuse is clearly superior to white box reuse (inheritance). Given this fact, C++ libraries should contain a variety of larger, more complete classes since if the class is missing some necessary functionality, a new class must be derived which eliminates most of the reuse savings. Alternately, C++ class libraries could be designed with expansion in mind. In essence, the libraries could be optimized for white box reuse. This research indicates that current C++ libraries are primarily optimized for black box reuse.

Goal three relates to the skills found in reusers. This research points towards educating programmers in Software Engineering skills and in reuse skills. Looking at the first two goals, it is clear that new programmers need to be taught the tradeoffs between black box and white box reuse. The extra effort required to find the perfect class may be well worth it if white box reuse can be avoided. Certainly, the aspects of library design should be mentioned since white box reuse can be thought of a library design and expansion. Programmers need to be taught what a reusable class looks like (goal four), and how to write one. Goal three identifies some of the many skills that should be taught to new C++ class reusers to facilitate frequent reuse. Goals two and four indicate what new reusers need to be taught in order to reuse effectively.

Goal four brings out some interesting facets pertaining to software reusability. There are competing factors in a reusable class. On one hand, the class should be simple and easy to understand and reuse. On the other hand, larger, more complete classes have a greater likelihood of satisfying the reusers needs and being reused without modification. Reusers need to be aware of these issues when designing C++ classes and libraries.

8.6 Future Work

This research effort has encountered several areas that should be investigated further. One possible conclusion from the first goal is that impact of concepts like software reuse on the software process can not be adequately gauged by examining the final product alone (i.e., by using source code metrics). Investigations into some process metrics are suggested.

The second goal of this research investigates the differing effects of two types of software reuse on the software development process. The effects on the resulting product should also be examined. Experiment Two did not have large enough programs to permit any metrics to be generated on the subjects' class hierarchies. When a single class is reused, black box reuse is superior. But, what are the results when classes are reused from the same library? The overhead costs will overlap and this will affect the overall productivity.

The implication of goals one and two is that black box reuse increasing productivity and white box reuse does not. However, if C++ libraries were designed better (for expandability), perhaps white box reuse will show more promise. The issue of C++ library design needs to be investigated more thoroughly. Should C++ libraries be optimized for black box reuse to take advantage of the greater productivity gains? This would likely result in larger libraries and larger classes within them which creates organization and search problems. Conversely, should C++ libraries be geared towards expandability and white box reuse in order to keep their size down and promote reuse through inheritance? This issue merits further investigation.

Under the third goal, sixteen skills of the original thirty-one skills were eliminated because of a lack of meaningful data. It is believed that some of these skills would show dependencies on the level of reuse (for example, *Experience Using C++*), but another experiment must be performed and more data collected before anything can be concluded about these skills. The subjects of this research experiment were not experienced enough in many areas to provide an exhaustive list of the traits of programmers who tend to reuse.

The majority of the future work comes out of the exploratory research under goal four. The analysis in Chapter 7 represents only a start of an effort to measure the reusability of a C++ class. Metrics need to be assigned or created for each of the identified traits in Figure 7.1 and combined to form a composite reusability metric. Programmers with a substantial amount of industry experience should be used as the subjects for further experimentation. In addition to identifying reusable classes, identifying reusable class libraries needs to be investigated. Clearly, a reusable class library contains more than just a set of reusable components. The library structure (inheritance tree) needs to be investigated as well. What does the structure of a reusable library of C++ classes look like? Should the libraries contain deep narrow hierarchies or shallow, broad ones? Deep hierarchies are more connected and if multiple classes are reused, the average overhead costs should decrease. However, deeper classes are less reusable (goal four). How can a library be optimized for black box reuse or for white box reuse? Are these two goals mutually exclusive? Clearly, more research is required before truly reusable libraries of C++ classes can be created effectively.

This research has laid a foundation for creating and using reusable C++ classes. It has shown that programmers need to be educated into the various aspects of software reuse, and has provided some ideas on what this education should look like. The next step for this research is to implement reuse education and to start developing reusable C++ classes and learning how to construct reusable class libraries from them.

Chapter 9

References

- [Afsh90] Afshar, J., JCOOL Version 0.1: Software Library of C++ Objects, Texas Instruments, Austin, TX, 1990.
- [AlHa91] Al-Haddad, H.M., K.M. George, and M.H. Samadzadeh, "Approaches to Reusability in C++ and Eiffel," *Journal of Object-Oriented Programming*, Vol. 4, No. 5, September 1991, pp. 34-45.
- [Ante90] Antebi, M., "Issues in Teaching C++," *Journal of Object-Oriented Programming*, Vol. 3, No. 6, November/December 1990, pp. 11-21.
- [Arms94] Armstrong, J.M. and R.J. Mitchell, "Uses and Abuses of Inheritance," *Software Engineering Journal*, Vol. 9, No. 1, January 1994, pp. 19-26.
- [Balc93] Balci, O., D. DeVaux, and R.E. Nance, "Measurement and Evaluation of Complex Navy System Designs," *Proceedings of the 1993 Complex Systems Engineering Synthesis and Assessment Technology Workshop*, NSWC, Calverton, MD, July 1993, pp. 126-140.
- [Barn91] Barnes, B. and T. Bollinger, "Making Reuse Cost Effective," *IEEE Software*, Vol. 8, No. 1, January 1991, pp. 13-24.
- [Barn94] Barnes, G.M. and B.R. Swim, "Inheriting Software Metrics," *Journal of Object-Oriented Programming*, Vol. 6, No. 7, November 1994, p. 27.
- [Basi90] Basili, V., "Viewing Maintenance as Reuse-Oriented Software Development," *IEEE Software*, Vol. 7, No. 1, January 1990, pp. 19-25.
- [Bato94] Batory, D., V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The GenVoca Model of Software-System Generators," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 89-94.
- [Beck89] Beck, K. and W. Cunningham, "A Laboratory for Object-Oriented Thinking," *SIGPLAN Notices*, Vol. 24, No. 10, October 1989, pp. 1-6.

- [Bigg87] Biggerstaff, T. and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, Vol. 4, No. 2, March 1987, pp. 41-49.
- [Bigg89a] Biggerstaff, T. and C. Richter, "Introduction to Reusability," In *Software Reusability Volume I*, Biggerstaff, T. and A.J. Perlis, Eds., Addison-Wesley, New York, NY, 1989, pp. 1-17.
- [Bigg89b] Biggerstaff, T. and A.J. Perlis, Eds., *Software Reusability Volume I*, Addison-Wesley, New York, NY, 1989.
- [Bigg91] Biggerstaff, T. and A.J. Perlis, Eds., *Software Reusability Volume II*, Addison-Wesley, New York, NY, 1991.
- [Boeh92] Boehm-Davis, D., R.W. Holt, and A.C. Shultz, "The Role of Program Structure in Software Maintenance," *International Journal of Man-Machine Studies*, Vol. 36, No. 1, January 1992, pp. 21-63.
- [Boll90] Bollinger, T. and S. Pfleeger, "The Economics of Software Reuse," *Proceedings of the 1990 8th Annual National Conference on Ada Technology*, Atlanta, GA, March 1990, pp. 436-447.
- [Booc91] Booch, G., *Object-Oriented Design*, Benjamin/Cummings, Redwood City, CA, 1991.
- [Bor195] Borland International Corporation, Borland 4.0 C++ Compiler Object Library, Scots Valley, CA, 1995.
- [Bowe85] Bowen, T., "Design Considerations for Reusable Software," *Proceedings of the 1985 IEEE 9th Annual International Computer Software and Applications Conference (COMPSAC)*, Chicago, IL, October 1985, p. 203.
- [Broo80] Brooks, R., "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," *Communications of the ACM*, Vol. 23, No. 4, April 1980, pp. 207-213.
- [Burt87] Burton, B. A., R. W. Aragon, S.A. Bailey, K.D. Koehler, and L.A. Mayes, "The Reusable Software Library," *IEEE Software*, Vol. 4, No. 4, July 1987, pp. 25-33.

- [Byar94] Byard, C., "Software Beans: Class Metrics and the Mismeasure of Software," *Journal of Object-Oriented Programming*, Vol. 7, No. 5, September 1994, pp. 32-34.
- [Cald91] Caldiera, G. and V. Basili, "Identifying and Qualifying Reusable Components," *IEEE Computer*, Vol. 24, No. 2, February 1991, pp. 61-70.
- [Canf94] Canfora, G., A. Cimitile, and M. Munro, "RE²: Reverse Engineering and Reuse Re-engineering," *Journal of Software Maintenance: Research and Practice*, Vol. 6, No. 2, March/April 1994, pp. 53-72.
- [Cant94] Cant, S. N., B. Henderson-Sellers, and D.R. Jeffrey, "Application of Cognitive Complexity Metrics to Object-Oriented Programs," *Journal of Object-Oriented Programming*, Vol. 7, No. 5, September 1994, pp. 52-63.
- [Card88] Card, D.N. and W.W. Agresti, "Measuring Software Design Components," *Journal of Systems and Software*, Vol. 8, No. 3, June 1988, pp. 185-197.
- [Chap90] Chappell, B., S.M. Henry, and K. Mayo, "Measurement of Ada Throughout the Software Development Life Cycle," *Proceedings of the 1990 8th Annual National Conference on Ada Technology*, Atlanta, GA, March 1990, pp. 525-532.
- [Chen92] Cheng, J., "Parameterized Specifications for Software Reuse," *ACM SIGSoft Software Engineering Notes*, Vol. 17, No. 4, October 1992, pp. 53-59.
- [Chen94] Chen, D.J. and D.T.K. Chen, "An Experimental Study of Using Reusable Software Design Frameworks to Achieve Software Reuse," *Journal of Object-Oriented Programming*, Vol. 7, No. 2, May 1994, pp. 56-67.
- [Chid91] Chidamber, S.R. and C. Kemerer, "Towards a Metrics Suite for Object-Oriented Design," *SIGPLAN Notices*, Vol. 26, No. 11, November 1991, pp. 197-211.
- [Chid94] Chidamber, S.R. and C. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, June 1994, pp. 476-493.
- [Coad90] Coad, P. and E. Yourdan, *Object-Oriented Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1990.

- [Cont86] Conte, S.D., H.E. Dunsmore, and V.Y. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, CA, 1986.
- [CoxB86] Cox, B. and A. Novobiliski, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, MA, 1986.
- [CoxB90] Cox, B., "Planning the Software Revolution," *IEEE Software*, Vol. 7, No. 6, November 1990, pp. 25-33.
- [DeCh93] De Champeaux, D., D. Lea, and P. Faure, *Object-Oriented System Development*, Addison-Wesley, Reading, MA, 1993.
- [DeMa82] DeMarco, T., *Controlling Software Projects*, Prentice Hall, Englewood Cliffs, NJ, 1982.
- [Empa90] Empathy Incorporated, *Classix Object-Oriented Library of C++ Classes*, New York, NY, 1990.
- [Fafc94] Fafchamps, D., "Organizational Factors and Reuse," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 31-41.
- [Fire94] Firesmith, D.G., "Using Parameterized Classes to Achieve Reusability While Maintaining the Coupling of Application-Specific Objects," *Journal of Object-Oriented Programming*, Vol. 7, No. 3, July 1994, p. 41.
- [Fisc87] Fischer, G., A. Lemke, and C. Rathke, "From Design to Redesign," *Proceedings of the 1987 9th Annual International Conference on Software Engineering*, Monterey, CA, March 1987, pp. 369-376.
- [Frak87] Frakes, W. and B. Nejme, "Software Reuse Through Information Retrieval," *Proceedings of the 1987 20th Annual Hawaii International Conference on System Sciences*, Hawaii, January 1987, pp. 530-535.
- [Frak94a] Frakes, W.B. and T. Pole, "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Transactions on Software Engineering*, Vol. 20, No. 8, August 1994, pp. 617-630.
- [Frak94b] Frakes, W.B. and S. Isoda, "Success Factors of Systematic Reuse," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 15-19.

- [Free87] Freeman, P., Ed., *Tutorial: Software Reusability*, Computer Society Press of the IEEE, Washington DC, 1987.
- [Gall92] Gall, H. and R. Klash, "Reuse Engineering: Software Construction from Reusable Components," *Proceedings of the 1992 IEEE 16th Annual International Computer Software and Applications Conference (COMPSAC)*, Chicago, IL, September 1992, pp. 79-86.
- [Garg87] Gargaro, A. and T.L. Pappas, "Reusability Issues and Ada," *IEEE Software*, Vol. 4, No. 4, July 1987, pp. 43-51.
- [Gibb88] Gibbens, W., "The Relationships Among Commenting Style, Software Complexity, Metrics, and Software Maintainability," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, 1988.
- [GNU95] GNU Software Project, GNU Software Library for C++, Department of Computer Science, Massachusetts Institute of Technology, Boston, MA, 1995.
- [Gold84] Goldberg, A. and D. Robinson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1984.
- [Gogu86] Goguen, J., "Reusing and Interconnecting Software Components," *IEEE Computer*, Vol. 19, No. 2, February 1986, pp. 16-28.
- [Gorl87] Gorlen, K., "An Object-Oriented Class Library for C++ Programs," *Software Practice and Experience*, Vol. 17, No. 12, December 1987, pp. 899-922.
- [Gorl90] Gorlen, K., M.O. Sanford, and P.S. Plexico, *Data Abstraction and Object-Oriented Programming in C++*, John Wiley and Sons, New York, NY, 1990.
- [Grum88] Gruman, G., "Early Reuse Practice Lives Up to its Promise," *IEEE Software*, Vol. 5, No. 6, November 1988, pp. 87-91.
- [Guer94] Guerrieri, E., "Case Study: Digital's Application Generator," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 95-96.
- [Hals77] Halstead, M., *Elements of Software Science*, Elsevier North Holland, New York, NY, 1977.

- [Hend93] Henderson-Sellers, B., "The Economics of Reusing Library Classes," *Journal of Object-Oriented Programming*, Vol. 6, No. 4, July 1993, pp. 43-50.
- [Henn94] Henninger, S., "Using Iterative Refinement to Find Reusable Components," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 48-60.
- [Henr81] Henry, S.M. and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 5, September 1981, pp. 510-518.
- [Henr83] Henry, S.M., "A Project Oriented Course on Software Engineering," *Proceedings of the 1983 14th Annual ACM SIGCSE Symposium on Computer Science Education*, Orlando, FL, February 1983, pp. 57-61.
- [Henr88a] Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers," *Journal of Systems and Software*, Vol. 8, No. 3, January 1988, pp. 3-11.
- [Henr88b] Henry, S.M., "A Metric Tool for Predicting Source Code Quality from a PDL Design," *Proceedings of the 1988 Software Design Metrics Workshop*, Melbourne, FL, March 1988, pp. A22-A43.
- [Henr89] Henry, S.M. and R. Goff, "Complexity Measurement of a Graphical Programming Language," *Software Practice and Experience*, Vol. 19, No. 11, November 1989, pp. 1065-1088.
- [Henr90] Henry, S.M. and C. Selig, "Predicting Source Code Complexity at Design Time," *IEEE Software*, Vol. 7, No. 2, March 1990, pp. 36-44.
- [Henr91a] Henry, S.M. and R. Goff, "Comparison of a Graphical and a Textual Design Language Using Software Quality Metrics," *Journal of Systems and Software*, Vol. 14, No. 3, March 1991, pp. 133-146.
- [Henr91b] Henry, S.M. and S. Wake, "Predicting Maintainability with Software Quality Metrics," *Journal of Software Maintenance*, Vol. 3, No. 1, September 1991, pp. 129-143.
- [Henr92] Henry, J., S.M. Henry, D. Kafura, and L. Matheson, "Quantitative Assessment of the Software Maintenance Process," submitted for publication to *IEEE Transactions on Software Engineering* in May 1992.

- [Henr93a] Henry, S.M. and M. Humphrey, "Comparison of an Object-Oriented Programming Language to a Procedural Programming Language for Effectiveness in Program Maintenance," *Journal of Object-Oriented Programming*, Vol. 6, No. 3, June 1993, pp. 41-49.
- [Henr93b] Henry, S.M., M. R. Lattanzi, M. Tiffany, and W. Gong, "A Software Metrics Generation Tool for the Object-Oriented Paradigm," *Proceedings of the 1993 4th Annual International Conference on Applications of Software Measurement*, Orlando, FL, November 1993, pp. 600-627.
- [Henr93c] Henry, S.M. and M.R. Lattanzi, "Reusability Metrics for the Object-Oriented Paradigm," *Addendum to the Proceedings of the 1993 8th Annual Conference on Object-Oriented Systems, Languages, and Applications*, Washington, DC, October 1993, pp. 95.
- [Holl92] Holland, I., "Specifying Reusable Components Using Contracts," *Proceedings of the 1992 European Conference on Object-Oriented Programming*, Utrecht, The Netherlands, June 1992, pp. 287-308.
- [Holt87] Holt, R., D. Boehm-Davis, and A. Schultz, "Mental Representations of Programs for Student and Professional Programmers," Technical Report, Department of Psychology, George Mason University, Fairfax, VA, 1987.
- [Hoop91] Hooper, J.W. and R. Chester, *Software Reuse: Guidelines and Methods*, Plenum Press, New York, NY, 1991.
- [Horo84] Horowitz, E. and J. Munson, "An Expansive View of Reusable Software," In *Software Reusability Volume I*, Biggerstaff, T. and A.J. Perlis, Eds., Addison-Wesley, New York, NY, 1989, pp. 21-41.
- [IEEE93] Computer Society of the IEEE, *IEEE Standard for a Software Quality Metrics Methodology: IEEE Std. 1061-1992*, Computer Society Press of the IEEE, Washington, DC, 1993.
- [Jame89] Jameson, K., "A Model for the Reuse of Software Design Information," *Proceedings of the 1989 IEEE 11th Annual International Conference on Software Engineering*, Pittsburgh, PA, May 1989, pp. 205-216.

- [Jett89] Jette, C. and R. Smith, "Examples of Reusability in an Object-Oriented Programming Environment," In *Software Reusability Volume II*, Biggerstaff, T. and A.J. Perlis, Eds., Addison-Wesley, New York, NY, 1989, pp. 73-101.
- [John88] Johnson, R. and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, Vol. 1, No. 2, June/July 1988, pp. 22-35.
- [Jone84] Jones, T.C., "Reusability in Programming: A Survey of State of the Art," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 488-493.
- [Jone88] Jones, G. and R. Prieto-Diaz, "Building and Managing Software Libraries," *Proceedings of the 1988 IEEE 12th Annual International Computer Software and Applications Conference (COMPSAC)*, Chicago, IL, October 1988, pp. 228-236.
- [Joos94] Joos, R., "Software Reuse at Motorola," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 42-47.
- [Kais87] Kaiser, G. and D. Garlan, "Melding Software Systems from Reusable Building Blocks," *IEEE Software*, Vol. 4, No. 4, July 1987, pp. 17-24.
- [Karu93] Karunanithi, S. and J. Bieman, "Candidate Reuse Metrics from Reusable Building Blocks," *Proceedings of the 1993 1st Annual International Software Metrics Symposium*, Baltimore, MD, May 1993, pp. 120-128.
- [Kasp94] Kasperson, D., "For Reuse, Process and Product Both Count," *IEEE Software*, Vol. 11, No. 5, September 1994, p. 12.
- [Kern84] Kernighan, B., "The UNIX System and Software Reusability," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 513-518.
- [Kicz92] Kiczales, G. and J. Lamping, "Issues in the Design and Documentation of Class Libraries," *SIGPLAN Notices*, Vol. 27, No. 10, October 1992, pp. 435-451.
- [Kors92] Korson, T. and J. McGregor, "Technical Criteria for the Specification and Evaluation of Object-Oriented Libraries," *Software Engineering Journal*, Vol. 7, No. 2, March 1992, pp. 85-94.

- [Lane84] Lanergan, R. and C. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 498-501.
- [Latt93] Lattanzi, M. and S.M. Henry, "Object-Oriented Metrics: Generation and Application," *Proceedings of the 1993 Virginia Computer Users Conference*, Virginia Tech, Blacksburg, VA, October 1993, pp. 53-62.
- [Lenz87] Lenz, M., H. A. Schmid, and P.W. Wolf, "Software Reuse Through Building Blocks," *IEEE Software*, Vol. 4, No. 4, July 1987, pp. 34-42.
- [Lewi92] Lewis, J., S.M. Henry, D. Kafura, and R. Schulman, "On the Relationship Between the Object-Oriented Paradigm and Software Reuse: An Empirical Investigation," *Journal of Object-Oriented Programming*, Vol. 5, No. 4, July/August 1992, pp. 35-41.
- [Liao93] Liao, H. and F. Wang, "Software Reuse Based on a Large Object-Oriented Library," *ACM SIGSoft Software Engineering Notes*, Vol. 18, No. 1, January 1993, pp. 74-80.
- [Lieb88] Lieberherr, K., I. Holland, and A. Riel, "Object-Oriented Programming: An Objective Sense of Style," *SIGPLAN Notices*, Vol. 23, No. 11, September 1988, pp. 323-334.
- [Lieb89a] Lieberherr, K. and A. Riel, "Contributions to Teaching Object-Oriented Design and Programming," *SIGPLAN Notices*, Vol. 24, No. 10, October 1989, pp. 11-22.
- [Lieb89b] Lieberherr, K. and I. Holland, "Assuring Good Style for Object-Oriented Programming," *IEEE Software*, Vol. 6, No. 5, September 1989, pp. 38-49.
- [LimW94] Lim, W.C., "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 23-30.
- [LiWe93] Li, W., and S.M. Henry, "Object-Oriented Metrics Which Predict Maintainability," *Journal of Systems and Software*, Vol. 23, No. 2, November 1993, pp. 111-122.
- [Lore91] Lorenz, M., "Real World Reuse," *Journal of Object-Oriented Programming*, Vol. 4, No. 7, November/December 1991, pp. 35-39.

- [Mart93] Martin, J. and J. Odell, *Principles of Object-Oriented Analysis and Design*, Prentice Hall, Englewood Cliffs, NJ, 1993.
- [Mayo89] Mayo, K., "Improving Software Quality Through the Use of Interface Metrics," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, 1989.
- [McCa76] McCabe, T.J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, December 1976, pp. 308-320.
- [McCa78] McCall, J., "The Utility of Software Quality Metrics in Large Scale Software System Development," *Proceedings of the 1978 2nd Annual Software Life Cycle Management Workshop*, Washington DC, August 1978, pp. 191-194.
- [McGr92] McGregor, J.D. and D.A. Sykes, *Object-Oriented Software Development: Engineering Software for Reuse*, Van Nostrand Reinhold, New York, NY, 1992.
- [McKe81] McKeithen, K., J.S. Reitman, H.H. Rueter, and S.C. Hirtle, "Knowledge Organization and Skill Differences in Computer Programmers," *Cognitive Psychology*, Vol. 13, No. 3, July 1981, pp. 307-325.
- [McIl69] McIlroy, M.D., "Mass Produced Software Components," *Proceeding of the 1969 NATO Conference on Software Engineering*, New York, NY, September 1969, pp. 88-98.
- [McKi93] McKim, J., "Teaching Object-Oriented Programming and Design," *Journal of Object-Oriented Programming*, Vol. 6, No. 2, March/April 1993, pp. 32-39.
- [Meye87a] Meyer, B., "Reusability: The Case for Object-Oriented Design," *IEEE Software*, Vol. 4, No. 2, March 1987, pp. 50-64.
- [Meye87b] Meyer, B., "Eiffel: Programming for Reusability and Extensibility," *SIGPLAN Notices*, Vol. 22, No. 2, February 1987, pp. 85-94.
- [Meye92] Meyer, B., "Applying Design by Contract," *IEEE Computer*, Vol. 25, No. 10, October 1992, pp. 440-451.

- [Mica88] Micallef, J., "Encapsulation, Reusability, and Inheritance in Object-Oriented Programming Languages," *Journal of Object-Oriented Programming*, Vol. 1, No. 1, April/May 1988, pp. 12-35.
- [Micr95] Microsoft Corporation, Microsoft Windows Development Kit, Redmond, WA, 1995.
- [Mili94] Mili, H., J. Witt, R. Radi, W. Wang, K. Strickland, C. Boldyreff, J. Heger, W. Scherr, L. Olsen, and P. Elzer, "Practitioner and SoftClass: A Comparative Study of Two Software Reuse Research Projects," *Journal of Systems and Software*, Vol. 25, No. 2, May 1994, pp. 147-170.
- [Mody92] Mody, R.P., "Is Programming an Art?," *ACM SIGSoft Software Engineering Notes*, Vol. 17, No. 4, October 1992, pp. 19-21.
- [Morr87] Morrison, R., A. Brown, R. Carrick, and R. Connor, "Polymorphism, Persistence, and Software Reuse in a Strongly Typed Object-Oriented Environment," *Software Engineering Journal*, Vol. 2, No. 6, November 1987, pp. 199-204.
- [Neig84] Neighbors, J.M., "The DRACO Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 564-574.
- [Next95] NeXtStep Computer, Incorporated, NeXtStep Development Environment (OpenStep) Object Library, Redwood City, CA, 1995.
- [OCon94] O'Connor, C. Mansour, J. Turner-Harris, and G. Campbell, "Reuse in Command-and-Control Systems," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 70-79.
- [OttR93] Ott, R.L., *An Introduction to Statistical Methods and Data Analysis*, Wadsworth, Belmont, CA, 1993.
- [Prie87] Prieto-Diaz, R., "Classifying Software for Reusability," *IEEE Software*, Vol. 4, No. 1, January 1987, pp. 6-16.
- [Prie91a] Prieto-Diaz, R., "Implementation Faceted Classification for Software Reuse," *Communications of the ACM*, Vol. 34, No. 5, May 1991, pp. 88-97.

- [Prie91b] Prieto-Diaz, R., "Software Reuse Trends in the United States," *Proceedings of the 1991 IEEE 15th Annual International Computer Software and Applications Conference (COMPSAC)*, Kogakuin University, Tokyo, Japan, September 1991, pp. 6-7.
- [Prie93] Prieto-Diaz, R., "Status Report: Software Reusability," *IEEE Software*, Vol. 10, No. 3, May 1993, pp. 61-66.
- [Reil87] Reilly, A., "Roots of Reuse, Editorial," *IEEE Software*, Vol. 4, No. 1, January 1987, p. 4.
- [Scha94] Schach, S.R., "The Economic Impact of Software Reuse on Maintenance," *Software Maintenance: Research and Practice*, Vol. 6, No. 4, March/April 1994, pp. 185-196.
- [Sema93] Semaphore Incorporated, *Glossary of Object-Oriented Terminology*, Object Management Group, Inc., Framingham, MA, 1993.
- [Somm92] Sommerville, I., *Software Engineering, 4th Edition*, Addison-Wesley, Wokingham, England, 1992.
- [Stan84] Standish, T., "An Essay of Software Reuse," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 494-497.
- [Star94] Staringer, W., "Constructing Applications From Reusable Components," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 61-68.
- [Stro86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.
- [Stro88] Stroustrup, B., "What is Object-Oriented Programming?," *IEEE Software*, Vol. 5, No. 3, May 1988, pp. 10-20.
- [Taen89a] Taenzer, D., M. Ganti, and S. Podar, "Object-Oriented Software Reuse: The Yo-Yo Problem," *Journal of Object-Oriented Programming*, Vol. 2, No. 3, September/October 1989, pp. 30-35.
- [Taen89b] Taenzer, D., M. Ganti, and S. Podar, "Problems in Object-Oriented Software Reuse," *Proceedings of the 1989 European Conference on Object-Oriented Programming*, University of Nottingham, England, July 1989, pp. 25-38.

- [Towe92] Towe, B., "A Literature Search on Documentation Methods," M.S. Project, Department of Computer Science, Virginia Tech, Blacksburg, VA, 1992.
- [Trac87] Tracz, W., "Reusability Comes of Age," *IEEE Software*, Vol. 4, No. 4, July 1987, pp. 6-8.
- [Trac88a] Tracz, W., "Software Reuse Maxims," *ACM SIGSoft Software Engineering Notes*, Vol. 13, No. 4, October 1988, pp. 28-31.
- [Trac88b] Tracz, W., "Software Reuse Myths," *ACM SIGSoft Software Engineering Notes*, Vol. 13, No. 1, January 1988, pp. 17-21.
- [Trac90a] Tracz, W., Ed., *Tutorial: Software Reuse: Emerging Technology*, Computer Society Press of the IEEE, Washington DC, 1988.
- [Trac90b] Tracz, W., "Where Does Reuse Start?," *ACM SIGSoft Software Engineering Notes*, Vol. 15, No. 2, April 1990, pp. 42-46.
- [Trac92] Tracz, W., "Domain Analysis Working Group Report," *ACM SIGSoft Software Engineering Notes*, Vol. 17, No. 2, July 1992, pp. 27-33.
- [Vita90] Vitaletti, W. and E. Guerrieri, "Domain Analysis within the ISED RAPID Center," *Proceedings of the 1990 8th Annual Conference on Ada Technology*, Atlanta, GA, March 1990, pp. 460-470.
- [Volp89] Volpano, D. and R.B. Kieburtz, "The Templates Approach to Software Reuse," In *Software Reusability Volume I*, Biggerstaff, T. and A.J. Perlis, Eds., Addison-Wesley, New York, NY, 1989, pp. 247-255.
- [Wake88] Wake, S. and S.M. Henry, "A Model Based on Software Quality Factors Which Predict Maintainability," *Proceedings of the 1988 IEEE Conference of Software Maintenance*, Phoenix, AZ, October 1988, pp. 382-389.
- [Walp85] Walpole, R.E. and R.H. Myers, *Probability and Statistics for Engineers and Scientists, Third Edition*, MacMillan, New York, NY, 1985.
- [Wass91] Wasserman, A., "Object-Oriented Software Development: Issues in Reuse," *Journal of Object-Oriented Programming*, Vol. 4, No. 2, March 1991, pp. 55-57.

- [Wegn84] Wegner, P., "Capital Intensive Software Technology," *IEEE Software*, Vol. 1, No. 3, July 1984, pp. 7-32.
- [Wess93] Wessale, W., D. Reifer, D. Weller, "Large Project Experiences With Object-Oriented Methods and Reuse," *Journal of Systems and Software*, Vol. 23, No. 2, November 1993, pp. 151-161.
- [Weid86] Weidenbeck, S., "Processes in Computer Program Comprehension," In *Empirical Studies of Programmers*, Soloway, E. and S. Iyengar, Eds., Ablex, Norwood, NJ, 1986, pp. 144-158.
- [Weid93] Weidenbeck, S. and V. Fix, "Characteristics of the Mental Representations of Novice and Expert Programmers: An Empirical Study," *International Journal of Man-Machine Studies*, Vol. 39, No. 5, November 1993, pp. 793-812.
- [Weid94] Weide, B. W., William F. Ogden, and M. Sitaraman, "Recasting Algorithms to Encourage Reuse," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 80-88.
- [Wild92] Wilde, N. and R. Huitt, "Maintenance Support for Object-Oriented Programs," *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, December 1992, pp. 1038-1052.
- [Wirf90] Wirfs-Brock, R. and R. Johnson, "A Survey of Current Research in Object-Oriented Design," *Communications of the ACM*, Vol. 33, No. 9, September 1990, pp. 104-124.
- [Wood87] Woodfield, S., D. Embley, and D. Scott, "Can Programmers Reuse Software?," *IEEE Software*, Vol. 4, No. 4, July 1987, pp. 52-59.
- [WuCT93] Wu, C. T., "Teaching Object-Oriented Programming to Beginners," *Journal of Object-Oriented Programming*, Vol. 6, No. 1, March/April 1993, pp. 47-50.

Appendix A

Programmer Skill Areas

Programmer Skill Areas

Programming Language Skills

Each of the subjects rated themselves on their experience and aptitude in each of the following languages (skills). Experience is defined to be exposure time, formal study, or programming with one of these languages.

- C++
- C
- Pascal
- Assembly
- FORTRAN
- BASIC
- X Windows
- UNIX Shell Script
- SmallTalk
- LISP / Prolog
- Ada

The last two skills in this category are two operating systems. Experience for these skills corresponds to using a computer where the particular operating system is running as well as programming under that operating system.

- Microsoft Windows
- UNIX

The average programming skill for this subject is denoted by the skill named:

- Programming Language Average Skill

It is the average of all thirteen skills listed above.

Software Engineering Skills

The skills in this category represent software engineering techniques. Experience for these skills include time that the programmer has studied, used, or refined this tools and ideas.

- Object-Oriented Design
- Encapsulation
- Polymorphism
- Inheritance
- High-Level Design Documents
- Source Code Debugger
- Source Code Control Systems
- Integrated Development Environments
- Source Code Profilers
- Top-Down Design
- Bottom-Up Design

The average software engineering skill is denoted by the skill named:

- Software Engineering Experience Average

Reusability Skills

The last category relates directly to tasks and skills that involve reusability. Experience for this skill set is simply time spent doing or working with the following:

- Procedural Paradigm Reuse (of any form)
- Object-Oriented Paradigm Reuse
- Modifying Other Programmer's Source Code
- Reusing Other Programmer's Source Code As Is
- Procedural Software Libraries Use (excluding built-in libraries such as those found in most C compilers)
- Object-Oriented Class Libraries (excluding compiler libraries)
- Designing / Writing Software Libraries (including using tools to aid in this task)

Again, this category has a composite skill (the average of all the above skills).

- Reusability Experience Average

Appendix B

Questionnaires for Experiment One

- B.1 Initial Questionnaire**
- B.2 Design Questionnaire**
- B.3 Coding Questionnaire**
- B.4 Library Questionnaire**
- B.4 Coding Questionnaire**
- B.5 Integration Questionnaire**
- B.6 Final Questionnaire**

B.1 Experiment One Initial Questionnaire (Subject Experience and Aptitude)

Name: _____

For the following questions, give:

Experience: Total months of practice, usage, or study.

Aptitude Rating: Your perceived knowledge (mastery) of the subject

Experience

1 semester = 4 months

1 summer session = 1.5 months

1 year = 12 months

Aptitude

1 = None

2 = Recognize

3 = Define/Describe

4 = Understand Partly

5 = Understand Mostly

6 = Understand Fully

Programming / Computer Science Experience	Experience	Aptitude
Programming in C++ / Objective C		
Programming in C		
Programming in Pascal		
Programming in Assembly		
Programming in FORTRAN		
Programming in BASIC		
Programming in X Windows		
Programming in UNIX Shell script		
Programming in Smalltalk		
Programming in LISP or Prolog		
Programming in Ada		
Programming under Microsoft Windows		
Programming under or Using UNIX		

B.1 Experiment One Initial Questionnaire (Subject Experience and Aptitude)

Name: _____

For the following questions, give:

Experience: Total months of practice, usage, or study.

Aptitude Rating: Your perceived knowledge (mastery) of the subject

Experience

1 semester = 4 months

1 summer session = 1.5 months

1 year = 12 months

Aptitude

1 = None

2 = Recognize

3 = Define/Describe

4 = Understand Partly

5 = Understand Mostly

6 = Understand Fully

Software Engineering Concepts	Experience	Aptitude
Using the Object-Oriented Paradigm		
Encapsulation		
Polymorphism		
Inheritance		
High-level design documents		
A source code debugger		
A source code control system		
An integrated development environment		
An execution profiler		
Top-down design		
Bottom-up design		

B.1 Experiment One Initial Questionnaire (Subject Experience and Aptitude)

Name: _____

For the following questions, give:

Experience: Total months of practice, usage, or study.

Aptitude Rating: Your perceived knowledge (mastery) of the subject

Experience

1 semester = 4 months

1 summer session = 1.5 months

1 year = 12 months

Aptitude

1 = None

2 = Recognize

3 = Define/Describe

4 = Understand Partly

5 = Understand Mostly

6 = Understand Fully

Reusability Concepts	Experience	Aptitude
Reusing in the procedural paradigm		
Reusing in object-oriented paradigm		
Modifying other people's code		
Reusing other people's code		
Using non built-in procedural SW libraries		
Using object-oriented software libraries		
Writing software libraries		

B.2 Experiment One Design Questionnaire

Hours Spent

Design Activity	Mon	Tues	Wed	Thur	Fri	Sat	Sun
Thinking about Design of Project							
Time Writing Design Documents							
Time Learning / Practicing C++							
Time Learning / Using Reusable Class Libs							
Time Learning / Wrestling With C++ Compiler							
Miscellaneous Computer Time(Prototyping, etc.)							
Total Time (Non overlapping Hours)							
Group Communication Time Specify # of People Present.							

B.3 Experiment One Coding Questionnaire

Class Development Data

Categories of Class Development (Row 1):

- 1) **Full Reuse:** Developed class was obtained from a class library
- 2) **Inherited Class:** Developed class inherited from a library class (Specify).
- 3) **White Box Reuse:** Developed class based on code from a library class (Specify).
- 4) **New Class:** Developed class independent of class libraries
- 5) **Other:** Based on other code, etc. (Describe).

Data Description	Class 1	Class 2	Class 3	Class 4
Coding for Group Number:				
Class Name				
Development Category				
# of compiles to develop class				
# of runs to develop class				
Time spent developing class				
Libraries Examined: Hours spent in each				

Significant Run-Time Errors While Developing Class

Causes of Errors (Column 3):

- 1) Lack of C++ knowledge
- 3) Incorrect specification
- 2) Misunderstood the specification
- 4) Other - Please specify

Class Name	Description of Error	Cause

Use back of sheet for additional error data.

B.4 Experiment One C++ Library Questionnaire

Library Name	Hours (By Class)	Opinions / Problems (Classes Examined / Code Quality / Documentation / Ease of Use)
GNU libg++.a		
JCOOL Data Structure Library		
Custom: gui.lib		
String Library (str.lib)		
Mouse Library		
Borland Class Library		
ObjEase Interface Library		
Others (Specify)		

B.5 Experiment One Integration Questionnaire

Date	Number of Compiles	Number of Runs	Hours Spent	Classes / Modules Integrating
Nov 18				
Nov 19				
Nov 20				
Nov 21				
Nov 22				
Nov 23				
Nov 24				
Nov 25				
Nov 26				
Nov 27				
Nov 28				
Nov 29				
Nov 30				
Dec 1				
Dec 2				
Dec 3				
Dec 4				
Dec 5				
Dec 6				
Dec 7				
Dec 8				

B.6 Experiment One Final Questionnaire

Please start your answer for each of the three section on a new sheet of paper.

1. The Object-Oriented Paradigm

- a) Do you like working in the object-oriented paradigm? Why or why not?
- b) Does the object-oriented paradigm result in "better" code than the procedural paradigm? Why or why not? Describe what's better (or worse) about it.

2. Class Libraries

- a) Were the libraries helpful or did they cause you more work? How? Why?
- b) How could the libraries be improved
- c) Which class library was the most useful? Why?
- d) Which one was the least useful? Why?
- e) In your opinion, what one trait makes a class reusable (excluding content - a stack is obviously more useful than a 6x6 matrix multiplier)?
- f) What are your opinions on inheritance? Is it useful? Why or why not?

3) Integration

- a) Describe the difficulties encountered during your system integration.
- b) What could you as a designer have done to make integration easier?

4) Reuse Opinions

For these questions, use a 0-10 scale.

0 = Don't care, 1 = Not important, 10 = Extremely important.

How willing are you to reuse code (1-10)? _____

How important is it to write more reusable code (1-10)? _____

How important is it to write maintainable (easy to change) code (1-10)? _____

Is reusing others code worthwhile? (Yes/No): _____

Appendix C
Experiment One Raw Project Data

Table C.2. Experiment One Raw Project Data

Group	DTIM	CTIM	LTIM	ITIM	TTIM	IERR	ICOM	IRUN	FERR	Reusability Measures										
										R	R	R	R	R	R	R	R	R	R	R
											Number of Black Box Classes	Number of White Box Classes	Number of Ind. Classes	Total Classes Reused	TC	BBP	WBP	TRP	IDP	
1	65.0	226	45.0	132	468	9.0	485	420	3.0	Final Errors	15	3	16	18	34	44.1	8.8	52.9	47.1	
2	72.0	301	11.0	180	564	7.0	578	493	7.0		6	1	23	7	30	20.0	3.3	23.3	76.7	
3	86.0	240	20.0	241	587	8.0	1085	951	10.0		5	2	39	7	46	10.9	4.3	15.2	84.8	
4	94.0	232	24.0	250	600	15.0	1231	1108	12.0		3	6	27	9	36	8.3	16.7	25.0	75.0	
5	71.0	211	29.0	216	527	13.0	591	544	5.0		7	2	23	9	28	25.0	7.1	32.1	82.1	
6	81.0	235	31.0	193	540	8.0	749	628	6.0		10	1	31	11	42	23.8	2.4	26.2	73.8	
7	82.0	217	18.0	239	556	12.0	652	604	5.0		6	3	19	9	28	21.4	10.7	32.1	67.9	
8	64.0	204	65.0	156	489	5.0	527	580	2.0		12	0	30	12	42	28.6	0.0	28.6	71.4	
9	90.0	239	19.0	223	571	8.0	981	1090	6.0		6	2	23	8	31	19.4	6.5	25.8	74.2	
Ave.	78.0	234	29.1	203	545	9.4	764	713	6.2		7.8	2.2	25.7	10.0	35.2	22.4	6.7	29.0	72.5	
Dev.	10.0	27.0	15.7	39.0	41.0	3.0	254	248	3.0		3.8	1.7	6.9	3.4	6.7	10.4	5.0	10.3	10.9	

Table C.2. Experiment One Raw Project Data

Whole System Metrics				Class Based Metrics													
S	S	S	C	S	S	S	S	S	S	S	C	C	C	C	C	C	C
TCS	KLOC	TNOM	MCC	TNSIZ	TKSIZ	LPC	FPC	LPM	SZ1	SZ2	NOC	WMC	DIT	RFC	DAC	MPC	LCM
I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
Total Classes (Excl. BB)	KLOC	Total NOM	McCabe	SIZE1 * Classes	SIZE2 * Classes	Ave. LOC / Class	Ave. Functions / Class	Ave LOC / Method	Ave. SIZE1	Ave. SIZE2	Ave. NOC	WMC - McCabe	Ave DIT	Ave. RFC	Ave. DAC	Ave. MPC	Ave. LCOM
19	3.4	138	431	2924	252	180.1	7.26	24.8	154	13.3	0.2	22.7	0.6	44.4	0.2	4.2	1.5
24	1.8	174	588	1949	245	76.2	7.25	10.5	81.2	10.2	0.4	24.5	0.4	61.3	0.2	3.7	2.3
41	2.5	154	613	2158	213	61.2	3.76	16.3	52.6	5.2	0.4	15.0	0.5	30.4	0.1	2.3	1.4
33	1.9	223	563	1395	315	56.5	6.76	8.4	42.3	9.6	0.5	17.1	0.9	22.9	0.0	3.5	1.2
25	3.9	109	948	3910	197	157.4	4.36	36.1	156	7.9	0.1	37.9	0.5	42.1	0.1	5.0	2.0
32	4.4	183	1048	4016	426	136.4	5.72	23.8	126	13.3	0.1	32.8	0.3	65.7	0.1	5.6	1.1
22	2.7	241	962	2521	367	123.9	10.95	11.3	115	16.7	0.1	43.7	0.6	72.5	0.0	13.1	1.3
30	2.0	198	554	2811	263	67.8	6.60	10.3	93.7	8.8	0.5	18.5	1.1	35.3	0.1	1.3	1.1
25	2.7	133	1144	2775	120	108.7	5.32	20.4	111	4.8	0.3	45.8	0.4	33.5	0.0	10.5	1.5
27.9	2.8	173	761	2718	266	108	6.4	18.0	103	10.0	0.3	28.6	0.6	45.3	0.1	5.5	1.5
6.8	0.9	43.5	261.5	855.5	92.2	44.9	2.1	9.2	40.2	3.9	0.2	11.7	0.3	17.3	0.1	3.9	0.4

Appendix D
Experiment Two Assignment Sheets

Experiment Two General Assignment Description

- 1) The class will be divided into 3 equal sets of 10 people each using the **initially gathered data** on each person.
- 2) Each set will write the specified program following the rules for that set.
- 3) Rules for each set.
 - Group A (Scratch):
No code reuse will be allowed.
 - Group B (Black box reuse):
A library of useful classes will be provided. At least one class from the library must be reused in the problem solution.
 - Group C (White box reuse):
A library of useful classes will be provided. At least one class from the library must be used as a parent for a new derived class in the problem solution.
- 4) The problem to be solved is to implement a post fix calculator.
- 5) The library will contain at least a stack class.
- 6) Each student will have two hours to write the program. During this supervised time, each student will collect the following data:
 1. development time spent in specified areas
 2. number of compiles
 3. number of compile-time errors
 4. number of runs
 5. number of run-time errors
 6. number of logical / programming errors

Experiment Two Assignment Sheet (Group A)

Problem Description

Write a C++ program to implement a stack-based postfix notation calculator for single character numbers (1-9) and the four basic operations (+, -, *, /). The idea of a postfix calculator is simple. As numbers are read in, they are put on to a stack. When an operator is encountered, the top two numbers are removed from the stack, the operation is performed, and the result is put back on to the stack.

Input

The input will be a file called *eqn.dat*. It will contain one postfix expression per line. The expressions will contain only the numbers 1 through 9 and only the operations addition (+), subtraction (-), multiplication (*), and integer division (/). The letter 'E' will denote the end of the expression. The last line of the file will be an 'X'. There are no spaces in the file. A typical input file will look like:

```
----- Top of File -----  
34+E  
897*+E  
25+44-37-*E  
94+284/59-*+/E  
X  
----- End of File -----
```

This example file represents the expressions:

1. $(3 + 4)$
2. $8 + (9 * 7)$
3. $((2 + 5) - ((4 - 4) * (3 - 7)))$
4. $(9 + 4) / (2 + (8 / 4) * (5 - 9))$

Output

The output will be the answers to the postfix expressions, 1 answer per line. The output for the above input file would look like:

```
----- Top of File -----  
7  
71  
7  
-2  
----- End of File -----
```

Experiment Two Assignment Sheet (Group B)

Problem Description

Write a C++ program to implement a stack-based postfix notation calculator for single character numbers (1-9) and the four basic operations (+, -, *, /). The idea of a postfix calculator is simple. As numbers are read in, there are put on to a stack. When an operator is encountered, the top two numbers are removed from the stack, the operation is performed, and the result is put back on to the stack.

A library of C++ classes is provided for your use. You must use, at least, the stack class from this library.

Input

The input will be a file called *eqn.dat*. It will contain one postfix expression per line. The expressions will contain only the numbers 1 through 9 and only the operations addition (+), subtraction (-), multiplication (*), and integer division (/). The letter 'E' will denote the end of the expression. The last line of the file will be an 'X'. There are no spaces in the file. A typical input file will look like:

```
----- Top of File -----  
34+E  
897*+E  
25+44-37-*--E  
94+284/59-*+/E  
X  
----- End of File -----
```

This example file represents the expressions:

1. $(3 + 4)$
2. $8 + (9 * 7)$
3. $((2 + 5) - ((4 - 4) * (3 - 7)))$
4. $(9 + 4) / (2 + (8 / 4) * (5 - 9))$

Output

The output will be the answers to the postfix expressions, 1 answer per line. The output for the above input file would look like:

```
----- Top of File -----  
7  
71  
7  
-2  
----- End of File -----
```

Experiment Two Assignment Sheet (Group C)

Problem Description

Write a C++ program to implement a stack-based postfix notation calculator for single character numbers (1-9) and the four basic operations (+, -, *, /). The idea of a postfix calculator is simple. As numbers are read in, they are put on to a stack. When an operator is encountered, the top two numbers are removed from the stack, the operation is performed, and the result is put back on to the stack.

A library of C++ classes is provided for your use. Derive a new class, the **pfstack** from the **stack** class in this library. Your derived class should have a modified *push* routine that does postfix pushes. For example, when an operator is pushed on to the stack, the new *push* routine automatically performs the operation on the top two stack elements and leaves the result on the stack.

Input

The input will be a file called *eqn.dat*. It will contain one postfix expression per line. The expressions will contain only the numbers 1 through 9 and only the operations addition (+), subtraction (-), multiplication (*), and integer division (/). The letter 'E' will denote the end of the expression. The last line of the file will be an 'X'. There are no spaces in the file. A typical input file will look like:

```
----- Top of File -----  
34+E  
897*+E  
25+44-37-*+E  
94+284/59-*+/E  
X  
----- End of File -----
```

This example file represents the expressions:

1. $(3 + 4)$
2. $8 + (9 * 7)$
3. $((2 + 5) - ((4 - 4) * (3 - 7)))$
4. $(9 + 4) / (2 + (8 / 4) * (5 - 9))$

Output

The output will be the answers to the postfix expressions, 1 answer per line. The output for the above input file would look like:

```
----- Top of File -----  
7  
71  
7  
-2  
----- End of File -----
```

Appendix E
Experiment Two Data Collection Sheet

Experiment Two Data Collection Sheet

Name: _____

Group: A B C

(Circle One. Your group in on the top of the assignment sheet.)

Use 'X' marks in for the first four time statistics. Use tick marks to record the rest of the statistics.

Fill out each column as you work (every 15 minutes) so you don't forget anything. Please be as accurate as possible. The experiment monitor will remind you.

Statistic	Total	0 - 15	15 - 30	30-45	45 - 1h	0 - 15	15 - 30	30 - 45	45- 2h
Analysis/Design Time									
In Library Time									
Coding Time									
Debug / Test Time									
Number of Compiles									
Number of Compiler Errors (Syntax, etc.)									
Number of Logical / Programming Errors									
Number of Runs									
Number of Runtime Errors (seg. fault, etc.)									

Appendix F
Experiment Two Pseudo-Code Solutions

Pseudo-Code Solution for Scratch / Black Box Problem

```
read ch
while ch != 'X'
    while ch != 'E'
        if ch in ['+', '-', '*', '/']           // if input is an operator
            ch2 = pop stack
            ch1 = pop stack
            result = ch1 operator(ch) ch2      // perform operation
            push result on stack
        else
            push ch on stack                   // else input is a digit
        end if
        read ch
    end while
    read ch                                   // read past the 'E' symbol
end while
print (pop stack)                            // print out the final answer
```

Pseudo-Code Solution for White Box Problem

```
read ch
while ch != 'X'
    while ch != 'E'
        push ch on stack                       // new push does operation if necessary
        read ch
    end while
    read ch                                   // read past the 'E' symbol
end while
print (pop stack)                            // print out the final answer
```

```
pfstack : public stack
{ void push (char ch); }
```

```
pfstack::push (char ch)
{
if ch in operators
    ch2 = this.pop()
    ch1 = this.pop()
    result = ch1 operator ch2
    stack::push (result)
else
    stack::push (ch)
}
```

Appendix G
Experiment Two Raw Data

Table G.1. Experiment Two Raw Project Data

Subject Num.	QCA	CS QCA	CS Cred.	Points (QCA * 20 + CSQ*40+CREQ*7)	Group Number	Design Time	Design Time (Sqrt)	Lib Time	Code Time	Debug Time	Total Time	Compiles	Runs	Compiler Errors	CTE (Sqrt)	Logic Errors	Runtime Errors
1	2.5	2.5	20	215	1	15	3.873	0	70	60	145	55	23	76	8.72	6	4
2	3.3	3.7	27	296	1	28	5.292	0	42	45	115	25	15	45	6.71	13	7
3	2.5	3.1	32	311	1	10	3.162	0	25	40	75	21	12	12	3.46	9	1
4	2.1	2.8	34	315	1	10	3.162	0	25	80	115	22	16	35	5.92	25	5
5	2.5	2.3	37	330	1	17	4.123	0	46	17	80	17	7	13	3.61	8	5
6	3.1	3.1	37	352	1	23	4.796	0	34	38	95	23	21	123	11.09	7	1
7	2.7	3.3	37	352	1	15	3.873	0	25	45	85	11	9	46	6.78	13	2
8	3.8	3.8	34	352	1	15	3.873	0	45	15	75	17	12	20	4.47	8	8
9	2.1	2.7	25	250	2	20	4.472	10	15	20	65	12	7	21	4.58	5	2
10	2.7	2.2	28	267	2	15	3.873	5	5	0	25	1	1	0	0.00	0	0
11	2.0	2.0	34	298	2	10	3.162	12	13	45	80	17	15	12	3.46	2	2
12	2.7	3.3	32	317	2	10	3.162	10	33	22	75	8	8	0	0.00	4	0
13	2.3	2.3	40	349	2	10	3.162	13	27	20	70	34	16	16	4.00	2	0
14	2.4	3.6	37	355	2	10	3.162	2	8	15	35	3	1	11	3.32	0	0
15	3.3	2.9	38	357	2	15	3.873	5	10	10	40	10	8	18	4.24	7	0
16	3.2	3.0	40	372	2	25	5	25	45	35	130	35	10	37	6.08	17	5
17	3.5	3.6	24	275	3	10	3.162	5	15	45	75	13	9	8	2.83	7	0
18	2.5	3.1	27	276	3	5	2.236	20	28	37	90	14	10	37	6.08	7	1
19	2.8	3.1	32	314	3	13	3.606	12	43	52	120	11	12	3	1.73	11	0
20	2.8	2.9	34	324	3	23	4.796	17	23	72	135	18	14	40	6.32	16	6
21	2.0	3.0	35	325	3	15	3.873	23	30	22	90	12	11	35	5.92	8	0
22	2.7	2.8	35	328	3	13	3.606	20	20	37	90	6	8	33	5.74	8	0
23	3.4	3.5	34	342	3	55	7.416	23	62	25	165	16	18	124	11.14	10	2
24	2.4	2.3	40	350	3	15	3.873	26	57	62	160	42	23	56	7.483	13	0

Appendix H
18 Chi-Square Frequency Tables
for
Programmer Experience and Level of Reuse

Appendix H. Chi-Square Tables (Cont.)

Table H.1. Chi-Square for C Experience versus Level of Reuse

C Experience vs. Level of Reuse	Number of Subjects With Below Average C experience	Number of Subjects With Above Average C experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	7 (5.67)	3 (4.33)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	5 (6.80)	7 (5.20)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	5 (4.53)	3 (3.47)	8
Totals	17	13	30
Chi-Square Test Statistic = 1.93		P Value = 0.38	

Table H.2. Chi-Square for Pascal Experience versus Level of Reuse

Pascal Experience vs. Level of Reuse	Number of Subjects With Below Average Pascal experience	Number of Subjects With Above Average Pascal experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	5 (6.00)	5 (4.00)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	7 (7.20)	5 (4.80)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	6 (4.80)	2 (3.20)	8
Totals	18	12	30
Chi-Square Test Statistic = 1.18		P Value = 0.55	

Appendix H. Chi-Square Tables (Cont.)

Table H.3. Chi-Square for Assembly Experience versus Level of Reuse

Assembly Experience vs. Level of Reuse	Number of Subjects With Below Average Assembly experience	Number of Subjects With Above Average Assembly experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	5 (6.33)	5 (3.67)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	11 (7.60)	1 (4.40)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	3 (5.07)	5 (2.93)	8
Totals	19	11	30
Chi-Square Test Statistic = 7.21		P Value = 0.03	

Table H.4. Chi-Square for FORTRAN Experience versus Level of Reuse

FORTRAN Experience vs. Level of Reuse	Number of Subjects With Below Average FORTRAN experience	Number of Subjects With Above Average FORTRAN experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	4 (7.33)	6 (2.67)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	11 (8.80)	1 (3.20)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	7 (5.87)	1 (2.13)	8
Totals	22	8	30
Chi-Square Test Statistic = 8.57		P Value = 0.01	

Appendix H. Chi-Square Tables (Cont.)

Table H.5. Chi-Square for BASIC Experience versus Level of Reuse

BASIC Experience vs. Level of Reuse	Number of Subjects With Below Average BASIC experience	Number of Subjects With Above Average BASIC experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	7 (6.33)	3 (3.67)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	9 (7.60)	3 (4.40)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	3 (5.07)	5 (2.93)	8
Totals	19	11	30
Chi-Square Test Statistic = 3.19		P Value = 0.20	

Table H.6. Chi-Square for LISP / Prolog Experience versus Level of Reuse

LISP/Prolog Experience vs. Level of Reuse	Number of Subjects With Below Average LISP/Prolog experience	Number of Subjects With Above Average LISP/Prolog experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	6 (5.67)	4 (4.33)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	5 (6.80)	7 (5.20)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	6 (4.53)	2 (3.47)	8
Totals	17	13	30
Chi-Square Test Statistic = 2.24		P Value = 0.33	

Appendix H. Chi-Square Tables (Cont.)

Table H.7. Chi-Square for UNIX Experience versus Level of Reuse

UNIX Experience vs. Level of Reuse	Number of Subjects With Below Average UNIX experience	Number of Subjects With Above Average UNIX experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	8 (5.67)	2 (4.33)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	7 (6.80)	5 (5.20)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	2 (4.53)	6 (3.47)	8
Totals	17	13	30
Chi-Square Test Statistic = 5.50		P Value = 0.06	

Table H.8. Chi-Square for Encapsulation Experience versus Level of Reuse

Encapsulation Experience vs. Level of Reuse	Number of Subjects With Below Average Encapsulation experience	Number of Subjects With Above Average Encapsulation experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	8 (5.33)	2 (4.67)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	5 (6.40)	7 (5.60)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	3 (4.27)	5 (3.73)	8
Totals	16	14	30
Chi-Square Test Statistic = 4.32		P Value = 0.12	

Appendix H. Chi-Square Tables (Cont.)

Table H.9. Chi-Square for High-Level Design Experience versus Level of Reuse

High-Level Design Experience vs. Level of Reuse	Number of Subjects With Below Average HL Design experience	Number of Subjects With Above Average HL Design experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	9 (7.67)	1 (2.33)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	10 (9.20)	2 (2.80)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	4 (6.13)	4 (1.87)	8
Totals	23	7	30
Chi-Square Test Statistic = 4.47		P Value = 0.11	

Table H.10. Chi-Square for Top-Down Design Experience versus Level of Reuse

Top-Down Design Experience vs. Level of Reuse	Number of Subjects With Below Average TD Design experience	Number of Subjects With Above Average TD Design experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	8 (6.33)	2 (3.67)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	8 (7.60)	4 (4.40)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	3 (5.07)	5 (2.93)	8
Totals	19	11	30
Chi-Square Test Statistic = 3.55		P Value = 0.17	

Appendix H. Chi-Square Tables (Cont.)

Table H.11. Chi-Square for Bottom-Up Design Experience versus Level of Reuse

Bottom-Up Design Experience vs. Level of Reuse	Number of Subjects With Below Average BU Design experience	Number of Subjects With Above Average BU Design experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	9 (6.00)	1 (4.00)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	6 (7.20)	6 (4.80)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	3 (4.80)	5 (3.20)	8
Totals	18	12	30
Chi-Square Test Statistic = 5.94		P Value = 0.05	

Table H.12. Chi-Square for Procedural Paradigm Reuse Experience versus Level of Reuse

Procedural Reuse Experience vs. Level of Reuse	Number of Subjects With Below Average Proc. Reuse experience	Number of Subjects With Above Average Proc. Reuse experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	9 (5.67)	5 (4.33)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	5 (6.80)	7 (5.20)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	3 (4.53)	1 (3.47)	8
Totals	17	13	30
Chi-Square Test Statistic = 6.82		P Value = 0.03	

Appendix H. Chi-Square Tables (Cont.)

Table H.13. Chi-Square for Modifying Other's Code Experience versus Level of Reuse

Modify Code Experience vs. Level of Reuse	Number of Subjects With Below Average Modify Code experience	Number of Subjects With Above Average Modify Code experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	9 (6.67)	1 (3.33)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	8 (8.00)	4 (4.00)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	3 (5.33)	5 (2.67)	8
Totals	20	10	30
Chi-Square Test Statistic = 5.51		P Value = 0.06	

Table H.14. Chi-Square for Reusing Other's Src. Code Experience versus Level of Reuse

Reuse Other Code Experience vs. Level of Reuse	Number of Subjects With Below Average Reuse Other experience	Number of Subjects With Above Average Reuse Other experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	10 (7.00)	0 (3.00)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	7 (8.40)	5 (3.60)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	4 (5.60)	4 (2.40)	8
Totals	21	9	30
Chi-Square Test Statistic = 6.59		P Value = 0.04	

Appendix H. Chi-Square Tables (Cont.)

Table H.15. Chi-Square for Procedural Library Reuse Experience versus Level of Reuse

Procedural Libs Experience vs. Level of Reuse	Number of Subjects With Below Average Procedural Libs experience	Number of Subjects With Above Average Procedural Libs experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	9 (6.33)	1 (3.67)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	6 (7.60)	6 (4.40)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	4 (5.07)	4 (2.93)	8
Totals	19	11	30
Chi-Square Test Statistic = 4.59		P Value = 0.10	

Table H.16. Chi-Square for Programming Languages Experience versus Level of Reuse

Programming Language Experience vs. Level of Reuse	Number of Subjects With Below Average Prog. Lang. experience	Number of Subjects With Above Average Prog. Lang. experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	6 (5.67)	4 (4.33)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	7 (6.80)	5 (5.20)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	4 (4.53)	4 (3.47)	8
Totals	17	13	30
Chi-Square Test Statistic = 0.20		P Value = 0.90	

Appendix H. Chi-Square Tables (Cont.)

Table H.17. Chi-Square for Software Engineering Total Experience versus Level of Reuse

SW Engr. Total Experience vs. Level of Reuse	Number of Subjects With Below Average SWE Total experience	Number of Subjects With Above Average SWE Total experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	8 (5.59)	2 (3.41)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	8 (7.45)	4 (4.55)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	2 (4.97)	6 (3.03)	8
Totals	18	12	30
Chi-Square Test Statistic = 7.53		P Value = 0.02	

Table H.18. Chi-Square for Reusability Total Experience versus Level of Reuse

Reusability Total Experience vs. Level of Reuse	Number of Subjects With Below Average Reuse Total experience	Number of Subjects With Above Average Reuse Total experience	Totals
Number of Subjects Performing Low Reuse (0 - 3 classes)	10 (7.00)	0 (3.00)	10
Number of Subjects Performing Moderate Reuse (4 - 7 classes)	7 (8.40)	5 (3.60)	12
Number of Subjects Performing High Reuse (8 - 12 classes)	4 (5.60)	4 (2.40)	8
Totals	21	9	30
Chi-Square Test Statistic = 5.50		P Value = 0.06	

Appendix I
C++ Class Reusability Evaluation Form

<CLASSNAME> Class Comparison Form

1a) Which class are you **most** likely to reuse? (Circle one).

<CLASS1>

<CLASS2>

<CLASS3>

1b) Next, fill in the table below. In the second column, rate your class choice in terms of each of the characteristics listed in column 1 by circling a number from 1-5 rating how well you believe the class you picked satisfies the particular characteristic. Then, in the last column, rank (in order of goodness) the characteristics from 1 (best) to 5 (worst).

Characteristic	"Goodness" Rating					Rank		
Complexity of the Class	<i>(Complex)</i>	1	2	3	4	5	<i>(Simple)</i>	_____
Organization of Class	<i>(Poor)</i>	1	2	3	4	5	<i>(Excellent)</i>	_____
Ease of Use	<i>(Hard)</i>	1	2	3	4	5	<i>(Easy)</i>	_____
Functional Completeness	<i>(Poor)</i>	1	2	3	4	5	<i>(Excellent)</i>	_____
Documentation	<i>(Poor)</i>	1	2	3	4	5	<i>(Excellent)</i>	_____

2a) Which class are you **least** likely to reuse? (Circle one).

<CLASS1>

<CLASS2>

<CLASS3>

2b) Next, fill in the table below. In the second column, rate your class choice in terms of each of the characteristics listed in column 1 by circling a number from 1-5 rating how well you believe the class you picked satisfies the particular characteristic. Then, in the last column, rank (in order of goodness) the characteristics from 1 (best) to 5 (worst).

Characteristic	"Goodness" Rating					Rank		
Complexity of the Class	<i>(Complex)</i>	1	2	3	4	5	<i>(Simple)</i>	_____
Organization of Class	<i>(Poor)</i>	1	2	3	4	5	<i>(Excellent)</i>	_____
Ease of Use	<i>(Hard)</i>	1	2	3	4	5	<i>(Easy)</i>	_____
Functional Completeness	<i>(Poor)</i>	1	2	3	4	5	<i>(Excellent)</i>	_____
Documentation	<i>(Poor)</i>	1	2	3	4	5	<i>(Excellent)</i>	_____

Appendix J
Informed Consent Form for
Research Involving Human Subjects

VIRGINIA TECH
INFORMED CONSENT FORM

Department: Computer Science Department
Title of Project: Dissertation Experiments By Current CS Ph.D Students
Investigator(s): Dr. Sallie Henry, Mark Lattanzi

1) THE PURPOSE OF THIS RESEARCH/PROJECT

You are invited to participate in three studies about: the reusability of C++ class libs, the psychological dynamics of teams, and a usability study.

This study involves 30 additional students besides yourself.

2) PROCEDURES

The procedures to be used in this research are: to be explained for each of the three experiments, see attached document.

The time and conditions required for you to participate in this project are: enrollment in CS4704 lecture and lab.

The possible risks or discomforts to you as a participant may be: no hazardous procedures are involved.

The safeguards to be used to minimize your risk or discomfort are: none

3) BENEFITS OF THIS PROJECT

Your participation in this project will provide the following helpful information: Better knowledge of C++, the OOP, and team building skills.

4) EXTENT OF ANONYMITY AND CONFIDENTIALITY

The results of this study will be kept strictly confidential. At no time will the researchers release the results of the study to anyone other than individuals working on the project without your written consent. The information that you provide will have your name removed and only a subject number will identify you during analyses and any written reports of the research.

5) COMPENSATION

For your participation in this study, you will be compensated with: 50 points for final exam of CS4704.

6) FREEDOM TO WITHDRAW

You are free to withdraw from this study without penalty. If you chose to withdraw, a final exam for CS4704 will be provided for you worth 50 points.

7) APPROVAL OF RESEARCH

This research involving human subjects has been approved by: Dr. Sallie Henry and the Computer Science department.

8) SUBJECT'S RESPONSIBILITIES AND PERMISSION

I know of no reason I cannot participate in this study. I have the following responsibilities:

1) Enroll in CS 4704, Fall Semester

2) Honestly, to the best of your ability, fill out all data collection forms given to you and return them by the assigned deadlines.

I have read and understand the informed consent and conditions of this project. I have had all of my questions answered. I hereby acknowledge the above and give my consent for participation in this project.

Should I have any questions about this research or its conduct, I will contact

1) Mark Lattanzi, 562 McBryde, 231-6931

2) Dr. Sallie Henry, 634 McBryde, 231-7584

3) the Computer Science department, 562 McBryde, 231-6931

Signature: _____

Printed Name: _____

Vita

Mark Lattanzi was born in the small town of Milford, Connecticut. At the age of six, he was reluctantly dragged to sunny Florida, a tropical hell for someone who hates heat and humidity. Luckily, three years later, his parents (with him and his two brothers) moved to the Blue Ridge mountains - home. After a brief stay of thirteen years, Mark moved on to bigger mountains-- the glorious Rocky Mountains of Colorado. Although they might be taller, they didn't compare to good old Appalachia, so Mark returned to Virginia Tech and the Blue Ridge mountains of home in 1991 to obtain a Ph.D. It seemed the thing to do at the time.

Four years and many miles and friendships later, the new Dr. Mark appeared out the other side. Now what? Slowly, the childhood dreams are being conquered, but he still hasn't found himself. Hiking, biking, parachuting, bungee jumping, even a obtaining a Ph.D.: all fell short. The true Mark still lies hidden. But, the next challenge awaits.

With new companion Kiera (The Chaser of Light #3), the lightness of being is still being chased. The never-ending quest for self continues.

Long live the Chasers of Light- Mark, Lizzie, and now, Kiera.

I am here to chase the light,
The Unbearable Lightness of Being.
Life is surely a heavy plight,
But finding lightness may mean not seeing.

Written in an AT Shelter one starry night.

A handwritten signature in cursive script that reads "Mark Lattanzi". The signature is written in dark ink and is positioned to the right of the text "Written in an AT Shelter one starry night."