1

of/de

1

PM-1 3 1/4"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

METRO

THRU COUNTY ROAD 5 BURNINGWELL NEW BRUNSWICK, NJ 08837 USA
TEL: 609-426-7007 FAX: 609-426-7007 TOLL: 800-550-0000

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Towards Metrics for Object Oriented Languages

by

Kevin Glenn Haaland

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science**

**School of Computer Science
Carleton University
Ottawa, Ontario
April 15,1992**

**copyright
1992, Kevin Glenn Haaland**

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-75970-4

Canada

**The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of this thesis.**

**Towards Metrics for Object Oriented
Languages**

submitted by

Kevin Glenn Haaland, BCS

**in partial fulfilment of the requirements for
the degree of Master of Computer Science**

Thesis Supervisor

Director, School of Computer Science

**Carleton University
April 15, 1992**

Abstract

Software metrics are an important component of the software development process, considerable work has been done in the development and application of metrics for traditional programming languages; this thesis presents the results of developing metrics for object oriented programming languages. Three categories of metrics were defined: Static, Runtime and Evolutionary. For each category, software was written to automatically gather the metrics. A diverse collection of projects was analyzed and the results are presented. The results obtained confirm many of the purported benefits of object oriented languages, provide several important insights into the process and characteristics of object oriented software development and form a foundation for future work in the area.

Acknowledgements

Thank you to my advisors John Pugh and Dave Thomas. Their guidance, and helpful suggestions are greatly appreciated. I would also like to thank Brian Barry and Jeff McAffer for being available to discuss this thesis and for their helpful suggestions.

I would also like to acknowledge the contributions of the organizations that supplied the data for this thesis: OTI, Jorg Sack, Digitalk and ParcPlace and those that did not want their names revealed. Thanks to John Duimovich for his expertise in Smalltalk's process model and Greg Adams for parsing. A special thank you, to the readers of early drafts of this thesis, Mike Wilson, Thai Nguyen, Brian Barry, and Arne Haaland.

Finally, I would like to thank my wife, Linda, who has endured the hardships of being married to a student. Without her, none of this would have been possible. Thank you.

Trademarks

ENVY is a registered trademark of Object Technology International Inc.

Smalltalk/V is a registered trademark of Digitalk Inc.

Objectworks is a registered trademark of ParcPlace Systems Inc.

Other products may be trademarks of their respective companies.

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Software Engineering	2
1.3	Software Metrics	3
1.4	Object Oriented Programming	3
2	Related Work	6
2.1	Introduction	6
2.2	Role of Software Metrics	6
2.3	Software Metrics	8
2.3.1	Assessing Candidate Metrics	8
2.3.2	Measurement Theory	9
2.3.3	Categorizing Metrics	10
2.4	Previous Work	11
2.4.1	Size Metrics	12
2.4.2	Quality Metrics	14
2.4.3	Complexity Metrics	14
2.4.4	Cohesion Metrics	15
2.4.5	Data Metrics	15
2.4.6	OOP Metrics	16
2.5	Summary	18
3	OOP Metrics	19
3.1	Introduction	19
3.2	Definition of OOP Metrics	21
3.3	Collecting the Metrics	21
3.4	Static Metrics	23
3.4.1	Method Metrics	23
3.4.2	Class Metrics	26
3.4.3	Application Metrics	28
3.4.4	Class Edition Metrics	30
3.5	Runtime Metrics	32
3.6	Evolutionary Metrics	33
3.7	Summary	34

Table of Contents

4	Applying the Metrics	35
4.1	Introduction	35
4.2	The Projects	35
4.3	Static	40
4.3.1	Method Metrics	40
4.3.2	Class Metrics	42
4.3.3	Class Edition Metrics	46
4.3.4	Application Metrics	48
4.3.5	Taking a Closer Look	50
4.3.6	Programmer Productivity	58
4.3.7	Cost Models	59
4.4	Applying Runtime Metrics	60
4.5	Evolutionary Metrics	61
4.6	Summary	63
5	Conclusions and Future Work	65
5.1	Conclusion	65
5.2	Future Research Topics	67
5.3	The Last Word	68
6	References	70
	Appendix A: Definitions	72
	Appendix B: Correlation Tables	76

1 Introduction

1.1 Motivation

The computer is the most useful and powerful tool ever developed. This is remarkable given its relatively short history. Beginning in the early 1940s, programmable computing devices have improved exponentially, both in reliability and capacity. Today, computers affect the lives of everyone and range in size from small embedded controllers to large mainframes. These improvements have enabled the development of complex systems that would have been impossible only a few years ago. However, the development of such systems is both difficult and costly. As a result, a disturbing number of new software systems do not meet the user's expectations, contain errors, are more expensive than anticipated and are often delivered late. [Demarco 82]

Since the late sixties a considerable amount of research has been directed at producing quality software, on time and on budget. The combination of software metrics and Object Oriented Programming (OOP) languages is a promising area of software engineering research, yet it has received relatively little attention. Before proceeding with a detailed analysis of OOP and software metrics, we will describe the goals of software engineering research in general and software metrics, in particular.

1.2 Software Engineering

Software engineering research is a generic term that applies to the techniques used to improve the software development process. The goal of software engineering research is to elevate the development of software systems from an art to an engineering discipline. The desire to move from an artistic to an engineering perspective is driven by the fact that the development of quality software is critically important to our society. The goal of software engineering research is not to stifle the creativity required to produce quality software, its purpose is to ensure that quality software is consistently developed on time and on budget. The prerequisite of a consistent software development process is a good understanding of the factors that affect its production.

One of the fundamental requirements of any engineering discipline is the ability to measure the factors which influence the end product. Measurements, coupled with the scientific method, have been used for hundreds of years by scientists and engineers. For example, given the materials and design of a bridge, it is possible to compute its ability to withstand loads. The ability to measure the structural members, combined with a model explaining its behaviour, enables the engineer to have confidence that the bridge will meet the design requirements. Over time models invariably undergo changes to reflect new information. For example, a static stress analysis is not sufficient to ensure that a bridge will perform properly under all conditions.

1.3 Software Metrics

The usefulness of applying measurement techniques to improve the software development process was realized in the late 60s [Rubey and Hartwick, 68]. Since then considerable research has been directed at developing models that capture the essential attributes of the process. Significant progress has been made, but the current state of software development suggests that there is much work to be done.

Software metrics is the area of software engineering that applies the techniques of measurement and model building to improve the software development process. However, metrics are only part of the answer. They cannot verify that a program satisfies the design requirements, nor can they automatically detect errors in the code.

1.4 Object Oriented Programming

OOP languages such as Smalltalk [Goldberg and Robson 83] and C++ [Stroustrup 86] are beginning to be accepted outside the research community [DeNatale, Irby, Lalonde, Leathers and Phillips 90]. However, if OOP is to achieve wide-spread acceptance, tools and techniques to manage the development of complex computer systems must be available. Software metrics are an important component of an effectively managed development process. [Humphrey 91, pg 301]

Software systems written in an object oriented language consist of a collection of

collaborating objects, which are abstractions created by the programmer. In a pure object oriented language like Smalltalk, everything is an object and all computation in the system is performed by sending messages to objects. In addition to the ability to define objects, an object oriented language typically contains other features not normally found in procedural languages such as C and Fortran. These features include: inheritance, polymorphism, automatic storage reclamation and incremental compilation.

OOP languages support rapid prototyping, which enables large systems to be grown instead of built. The growth of a large system in an object oriented language is usually described by the spiral model [Boehm and Papaccio, 90], where repeated iterations of analysis, design, development, and testing are performed to arrive at the finished product. Each iteration of the spiral model takes a fraction of the time used by the waterfall model [Boehm 81]. After each iteration a prototype of the final system is available for examination by the customer and serves as a test bed for the programmer to gain a better understanding of the final product.

Moving from the research lab into mainstream development imposes additional requirements on any software technology. The development of commercial applications demands a level of quality that is normally not required in the research lab. In addition, the development must meet strict schedules and budgets. Since the application of object oriented technology to the development of real world applications is relatively new, not much work has been done on software metrics for object oriented languages. The work presented here contributes to the effective use of software metrics for these languages.

This research has focused on developing a set of measures specifically targeted at the development of large software systems written in an object oriented language. The previous work relevant to this research is presented in Chapter 2. Chapter 3 describes the work performed in this thesis. Several software projects were analyzed using the measures that were developed and Chapter 4 summarizes the results. The final chapter summarizes the significant contributions of this work and suggests areas for future research.

2 Related Work

2.1 Introduction

This chapter begins with a discussion of the problems encountered developing large computer systems and the role that software metrics can play in improving the situation. An evaluation criteria for assessing candidate metrics follows. The third section of this chapter is a brief summary of existing research.

2.2 Role of Software Metrics

The failure to consistently produce quality software on time and within budget is almost universal. There are many reasons and Brooks identifies several of them in his paper, "No Silver Bullet - Essence and Accidents of Software Development". A study which examined software developed for the United States government produced some startling results. Only three percent of all systems were delivered on time and were used without modification, another two percent required minor modifications. The remaining ninety five percent were either late, required major modifications or were never delivered! All software developers are faced with the challenges: to improve the development process so that large software systems perform to the user's satisfaction and to deliver them on time and within budget.

Software metrics is an area of software engineering research that continues to receive

a great deal of interest in the industrial community. The goals are to be able to manage the process and to determine in advance a realistic estimate of the cost of developing software systems. Since software systems are becoming larger and more complex, the need to manage the process is becoming even more important and can mean the difference between success and failure.

A fundamental requirement for controlling a process is the ability to measure it [Humphrey 91, pg. 3]. Only then is it possible to quantitatively predict quality and cost. And until statistical process control has been implemented, it is impossible to have an organized strategy for improving the software development process [Humphrey 91, pg 5]. When the process is controlled, it is possible to carry out a cost benefit analysis before the system is written. In addition, predictive metrics of software quality will enable management to become proactive in the delivery of quality software. Instead of waiting until the system is released to fix problems, it will be possible to identify problem areas before it is too late.

Improving the software development process requires that everyone not only works harder, but smarter. Unless measurements are taken that capture the state of the process, it is impossible to know with any certainty if changes have had any affect, positive or negative. Measurements of software development should help to understand and improve the process. If not, they are a waste of time and money. Measurements must add value to the software development process. Since it is impractical and too costly to gather every measurable attribute of the process, the measures must be carefully chosen.

2.3 Software Metrics

Software metrics research is based on a combination of measurement theory, statistics and common sense. Measurement theory defines the set of valid operations that can be performed, and statistical analysis is used to extract useful information from the raw data. Common sense is equally important; without it, the entire effort may not be applicable to real world problems.

2.3.1 Assessing Candidate Metrics

A considerable amount of research has been directed at identifying measures that capture the significant factors affecting quality and cost. When potential measures are examined, several factors must be kept in mind [Conte, Dunsemore, Shen, 86]. Perhaps the most important is the metric's *validity*; i.e., does the metric actually measure what it claims to measure? For example, the number of bytes of a compiled method accurately captures its size, but it may not express the difficulty that a programmer will have to understand its semantics.

A metric's *simplicity* is also important. The simpler the metric, the easier it is to collect, analyze and interpret; e.g., a single measure is much easier to work with than a measure with several attributes.

The *robustness* of a measure reflects its tolerance to changes in factors that do not affect the process being measured; e.g., changes in the person doing the measurement

will affect subjective measures, but will not affect objective measures obtained by a data gathering program.

Other factors which describes a measure are its *prescriptiveness* and its ability to be *analyzed*. A prescriptive metric will suggest an improvement strategy; e.g., the obvious improvement strategy suggested by counting the number of defects in a program, is to achieve zero defects. A metric that can be automatically analyzed by standard statistical analysis programs is preferable because it is difficult to extract useful information from raw data.

2.3.2 Measurement Theory

Metrics assign values to measurable attributes of the software development process. When these values are analyzed, care must be taken to ensure that the analysis is valid. Measurement theory defines four types of measurement scales: nominal, ordinal, interval and ratio. The *nominal* scale categorizes observations into groups. The only valid analysis permitted on these measurements is one of group membership. A person's sex, race, and religion are all examples of nominal measures.

The *ordinal* measurement scales perform equality analysis and allows ranking of observations which are not valid for measures from the nominal scale. For example, a questionnaire that rates the performance of an individual using numbers from one through five, will allow observations to be ranked and tested for equality.

In addition to ranking and categorizing, measures from the *interval* scale support addition and subtraction. A measure on the interval scale must support all the operations defined by the nominal and ordinal scale, and the difference between successive values must be identical. For example, the difference between 20 °C and 40 °C is the same as the difference between 0 °C and 20 °C.

The most desirable measurement scale is the *ratio* scale. This scale permits division, which is meaningless for the other scales. For example, while 40 °C is numerically twice that of 20 °C it is not true that 40 °C is twice as hot as 20 °C. To be a member of the ratio scale there must be an absolute zero. Using the temperature example again, it is true that 40 degrees Kelvin is twice as hot as 20 degrees Kelvin.

2.3.3 Categorizing Metrics

A metric can be either *subjective* or *objective*. A subjective measure is affected by the person doing the measurement, while an objective measure is not. Asking programmers to estimate the difficulty of a programming task is a subjective measure because the answer depends on the individual. Conversely, the number of lines of code in a method depends only on the definition of a line of code, and not on the person measuring.

A *product* metric measures the output of a software development process. For example, the output of the design process could be a Class-Responsibility-Collaborator [Wirfs-Brock, Wilkerson, Wiener, 90] description of the system. A *process* metric captures how the product was produced. For example, the number of code reviews during the development.

A *dynamic* metric includes the concept of time; a *static* metric does not. The number of lines of code is a static metric. The number of times a method is executed is a dynamic metric.

2.4 Previous Work

The purpose of software metrics is to better understand the development process. For example, there is an intuitive relationship between the size of a program and the time required to develop it. If it were possible to accurately estimate the size of a program before it is written and if we knew the productivity of the programmers, more accurate estimates of time and cost would be possible.

Since large software systems continue to evolve, most of the cost of a software system is often incurred after its initial development. Ideally, the system should be designed so that it is easy to maintain and to develop further. Part of the requirements for easy maintenance is for the system to be understandable. Using Brook's terminology, every system has some essential complexity that is dependent on the problem that the system solves. The actual complexity of a program is the sum of its essential complexity plus any accidental complexity introduced by the programmer. Accidental complexity can be introduced by the tools used to write the program or by the programmer. Since unnecessary software complexity will make it more difficult to understand, it should be minimized.

Traditional languages such as Pascal and Fortran distinguish between the program and the data structures on which the program operates. In information processing applications it could be argued that the data is more important than the program. Therefore, measures of the data may yield more relevant information than the size of the program.

2.4.1 Size Metrics

The most widely known measure is the number of lines of code (LOC). Given a consistent definition, it is straightforward to count the number of lines of code. Depending on the particular situation, it may be necessary to distinguish lines of code as either new or reused, delivered or not delivered. However, there are several arguments against counting lines of code. The first one recognizes the ambiguity in the definition of a line of code. For example, are blank lines and embedded comments included? Are only new lines of code counted or does the metric include reused codes as well? The second shortcoming of simply counting lines of code is that there is no distinction between easy and hard lines of code. The final objection to the lines of code metric is that it is difficult to equate the lines of code used in different languages.

Recognizing that all lines are not equal, Halstead developed a collection of metrics called Software Science [Halstead 77]. The Software Science metrics are a collection of composite metrics that are based on counting tokens in the source code. These tokens are classified as either operators or operands. There are four primitive metrics in Software Science: n_1 , n_2 , N_1 , N_2 . The number of unique operators and operands are

n_1 , n_2 , respectively. N_1 , N_2 represent the total number of operators and operands. Based on his model of the programming process, Halstead proposed several formulas based on these four primitive metrics. Many of these formulas have been criticized for their questionable theoretical basis and their lack of empirical support. Nevertheless, Software Science is one of the most significant early contributions to software metrics research and has contributed much to the work that followed. For example, there is a strong correlation between n_1 , n_2 and other measures of program size.[Conte, Dunsmore, Shen 86, pg 87]

The size measures discussed so far all suffer from a common fault; it is impossible to estimate their value with any accuracy until the program is written. Typical business application programs share the same structure. They respond to user queries that cause data to be read and written. In situations like this, it is possible to use a metric called Function Points [Albrecht 79]. Function points are derived from the design elements such as files, user queries or reports and are combined to form a weighted total. Albrecht defined five function types: External Input, External Output, Logical Internal File, External Interface File and External Inquiry. The number derived for each user function type is weighted by its complexity. An unadjusted function point value is computed as the sum of the weighted function points. The value is then adjusted by an application environment complexity factor to arrive at the number of function points for the system. Using this value and the organizations productivity expressed in function points per person-month, more accurate estimates can be made. Function points have been successfully used in data processing applications, but they have had limited success in other areas, due to the difficulty of determining which objects to count.

Recent work by SPR¹ to extend function points to include algorithmic measures may make them more widely used.

2.4.2 Quality Metrics

In the long term, improving software quality is as important as improving the productivity of the process. Metrics that measure software quality often count the number of defects. This data may be used to estimate the number of bugs remaining in a program. Improvements made to the software development process can be evaluated, in part, by their affect on the quality of the software. The development of these metrics depends on the definition of quality. Depending on the particular situation, quality may be a combination of correctness, portability, maintainability, and efficiency.

2.4.3 Complexity Metrics

As mentioned previously, lines of code metrics do not differentiate between simple and complex lines. Complexity metrics attempt to overcome this limitation by measuring factors distinguishing simple and complex programs. There are two significant complexity metrics: Software Science and Cyclomatic Complexity [McCabe 76]. The Cyclomatic Complexity of a program is the number of linearly independent paths in a program. Cyclomatic Complexity was initially proposed as an indicator of the ability to test and maintain a program. The Cyclomatic Complexity of a program is similar to the cyclomatic number of a directed graph. The nodes of the graph are code

¹Software Productivity Research Inc

fragments with exactly one entry and one exit point. The edges are decision points.

2.4.4 Cohesion Metrics

The notion of cohesion and coupling is closely related to complexity. Cohesion is a measure of the "singleness of purpose" of either a module or a data structure. Ideally, all the elements of a system should have a single, well-defined purpose. Cohesion is difficult to measure objectively. Tests in which cohesion was measured subjectively established a correlation between modules with low cohesion and high fault rates. [Card 86]

2.4.5 Data Metrics

Since procedural languages distinguish between code and data, metrics have been developed that measure the amount of data, the complexity of its organization and how it is used. These metrics are useful in cases where the data being manipulated contributes significantly to the complexity of the software system.

The simplest data metric measures either statically or dynamically, the amount of data processed in number of bytes or bytes/second. If the size of the data is not important, one can simply count the number of variables in the modules.

More complex data measures consider how a variable is used. The number of live variables [Dunsmore 79] has also been proposed as a measure of the complexity of a program. A variable is live from its first reference to its last reference. The number

of live variables is computed for each line of code in the system. However, the difficulty associated with the development of a line of code may not be related to the number of variables that the programmer had to consider.

Another metric is the measure of the span of a variable [Elshoff, 76], the number of statements between two successive references to the same variable. A variable that is referenced n times will have $n-1$ spans. Unlike the live variable measure, variable spans measure how a variable is used in the system.

Data coupling measures how data is shared between modules. The fan-in of a module is defined as the number of modules that pass data to the module either directly or indirectly. The fan-out of a module is defined as the number of modules to which data is passed either directly or indirectly.

2.4.6 OOP Metrics

Very little work has been published on software metrics for OOP languages. One of the first papers to contain metrics extracted from an OOP project appeared in OOPSLA'89 [Barry, 89]. This paper presented the results of prototyping a real-time embedded system in Smalltalk. Several metrics were gathered including the number of lines of code and comments, the number messages sent and the number of methods per class and inherited methods. Barry compared three Smalltalk systems: the base V/286 image from Digitalk, the DREO/V image² and the AMEP system. Assuming that the

²The DREO/V image was essentially the base image from Digitalk plus a few utilities for creating documents and managing code.

DREO/V image was used as a starting point; the productivity of the developers was 1.6 classes per person month with each class having, on average, 20.13 methods. Using the more conventional lines of code measure, the productivity was 109.5 lines of code per person-month. The paper suggests that inheritance is used most effectively when objects with a natural hierarchical decomposition are modeled and that inheritance was not as effective for the parts of the system for which an obvious decomposition was not available.³

A more recent paper concerning metrics and OOP appeared in OOPSLA'91 [Chidamber, 91] The author used a theoretical approach to software metrics research to justify the usefulness of the measures. Only static code measures were considered and there were no empirical data to justify the conclusions. The author defines the cohesiveness of a class in terms of groups of methods that access the same instance variables, and suggests that if there is more than one group of methods, the class is not cohesive and should be separated multiple classes. This definition of cohesion ignores the more important issues of the semantics of the class. It is trivial to devise a class that would be highly cohesive by Chidamber's definition, but it would be neither reusable nor a good example of object oriented programming.

³For example, the hard real time code did not use inheritance as much as some other sections.

2.5 Summary

Software metrics continue to receive significant attention by both academics and practitioners. The use of metrics to improve the software development process is motivated by the difficulty to develop large computer systems. Software metrics research requires an understanding of both measurement theory and the problems faced by software developers. The software metrics discussed in this chapter attempt to measure at least one of the following factors: size, quality or complexity. These metrics, together with a model of the software development process, have been used to understand and predict the quality and cost of developing and maintaining large computer systems.

3 OOP Metrics

3.1 Introduction

Object oriented languages differ from more traditional languages such as Fortran and Pascal in three main ways, 1) the use of new language features such as encapsulation, inheritance, polymorphism and late binding, 2) features such as an integrated development environment and incremental compilation, and 3) the approach used to develop systems in an object oriented language. The latter is perhaps the most important advantage of an object oriented language. In this chapter we describe our candidate OOP metrics.

Research on software metrics for OOP is a relatively new. Since very little has been published, the scope of this thesis is quite broad. By examining a number of potential areas for applying software metrics for OOP, it is hoped that a set of metrics will be defined that will be of immediate benefit to practitioners and that the results will stimulate other researchers to continue the work.

The metrics that were defined can be broadly categorized as either *static* or *dynamic*. Static measures ignore the influence of time. One example of a static measure is the number of messages that an object understands. Dynamic measures incorporate the effects of time and can be further categorized as either *runtime* or *evolutionary*. Runtime measures are gathered from the executing program. For example, the classes

of instance variables in an object over the life of the running system. Evolutionary measures are used to quantify the changes in the program over time. For example, the number of methods modified between releases of a system.

The static metrics tool interrogates objects in the image to extract the metrics. An example of a static measure is the number of instance variables in each class. A generic method parsing facility was used for measures that require the source of a method to be parsed in order to derive useful information.

There has been limited work published on metrics gathered from the running system. The runtime tool can be used to gather these metrics. The ability to instrument the running system is a powerful mechanism that facilitates the gathering of measures that would not be possible from the static code. The runtime component operates in two ways, by interrupting the normal execution at a user-specified time interval or by interrupting between each message send. In either case, the runtime stack can be interrogated to extract useful measures.

Evolutionary metrics describe the changes in an application over time. These may give insight into how systems are developed in an object oriented language. Applications in Smalltalk are represented in the image by classes, methods and global variables. Extracting changes from these objects requires the ability to materialize temporary copies of these objects from two applications. Once in the image, they can be compared and the differences recorded.

3.2 Definition of OOP Metrics

The starting point for our metrics for OOP began with the existing metrics for procedural languages and where possible, equivalent similar metrics were defined for object oriented languages. However, it soon became evident that none of the existing metrics measured the significant differences between object oriented languages and procedural languages, such as the use of inheritance to reduce code, the effects of dynamic binding and the iterative nature of developing systems. This lead to the development of metrics that are specific to systems developed in an object oriented language. The set of metrics presented in the next section have undergone several iterations and are the result of constructive advice from several individuals.

3.3 Collecting the Metrics

All projects examined in this thesis used ENVY/Developer, a software development tool from Object Technology International Inc. ENVY/Developer is a multi-user configuration management and version control system for both Digitalk's and ParcPlace's version of Smalltalk. In addition to Smalltalk's organizational constructs of Classes and Methods; ENVY/Developer adds Applications, which contain one or more class editions. A class edition contains none or several methods and are of two types:

class definitions⁴ and class extensions⁵. Applications are a significant addition to the Smalltalk programming environment, because they are a higher level reusable component than individual classes. In addition to containing classes, applications may contain other applications.

ENVY/Developer maintains a complete record of the methods, classes and applications in a shared repository. Classes and methods will be either public or private. This attribute can be used to hide the implementation details of the system and to specify the public interfaces to objects in the system. Software was developed to automatically gather metrics from the repository and create an output file that can be analyzed by standard statistical packages.

A tool for gathering the static, runtime and evolutionary data was implemented to analyze code developed in Digitalk Smalltalk V/286, V/Windows, V/PM as well as Objectworks\Smalltalk. The metric gathering software consists of three tools. The first tool implements a framework for gathering static metrics from the supported Smalltalk images. User configurable options are available for determining which objects are measured; e.g., excluding the base image classes to ensure that only the classes added by an application are measured. The second tool provides the facilities required for gathering metrics from the running program. As it executes, the raw data is gathered. Once the test is completed, the data can be processed and written to disk. The normal

⁴The class definition specifies the superclass, instance variables, class variables and pool dictionaries of the class.

⁵Class extensions only contain methods. The representation of the class is specified by the class definition.

execution of the system is interrupted either between each message send or by causing a timer interrupt to occur at regular intervals. Interrupting between each message send is similar to single stepping through the source in a Smalltalk debugger. When the normal execution is suspended, the run-time state is examined to extract the metrics. User configurable options are also available to specify the classes considered. The third tool extracts changes in collections of Smalltalk classes and methods. This tool relies on the history maintained by ENVY/Developer but it would also be possible to extract this information from a previous Smalltalk image or change log.

3.4 Static Metrics

Static metrics have been defined for methods, classes, class editions and applications. The sections defines the static metrics gathered from these four components.

3.4.1 Method Metrics

Methods define the behaviour of all the objects in a system. Methods can be represented by their source or compiled representation. Both forms are used to define static metrics for methods.

The size of a program has been used in models of programmer productivity⁶. If accurate estimates of a program's size and programmer productivity are available, better

⁶For example, lines-of-code per programmer-month.

estimates of the development effort can be made. The size of a method can be defined in several ways: the number of byte codes, the source length, the lines of code and the lines of comments. The *number of byte codes* generated for the method is independent of both the formatting conventions used by individual programmers and the programmer's choice of variable names. The *source length* includes comments and blank lines of the method's source code. This measure is dependent on the programmer's formatting conventions, the use of comments, as well as the choice of variable names. The number of *lines of code* excludes the message selector, the initial method comment (if there is one) and any blank lines embedded in the source. The number of *lines of comments* excludes the message selector, the body of the method including embedded comments and any blank lines in the comment.

Complexity metrics measure the intricacy of a method. The *number of messages sent* counts the number of unique messages that are sent by a method. Special selectors which are inlined by the compiler are not detected. Neither are selectors that are performed based on the value of a variable which cannot be determined statically. The *number of arguments* accepted by the method may be a measure of its complexity because the objects to which a method can send a message include arguments accepted as parameters to the methods. The *number of temporary variables* is also a measure of the complexity of a method and counts the number of temporary variables allocated by the method when it is executed. The *number of editions* of the method in the repository is yet another measure. ENVY/Developer maintains a complete history of the changes made to a method. Some changes may be trivial, such as adding or removing debugging information, modifying the comment or changing the format of the source;

other changes are more substantial, such as fixing errors or modifying the behaviour of the method. Attaching semantics to this measure is the responsibility of the person doing the measurement and since the method may have been modified for trivial reasons, it may not necessarily indicate that it was difficult to develop.

The next five static method metrics are based on metrics defined for procedural languages. The *number of unique operands, unique operators, operands and operators* form the basis of Halstead's Software Science Model. The last metric in this group is the *Cyclomatic Complexity* of the method, the original intent of which was to count the number of independent paths through the code. For languages such as Pascal, establishing the cyclomatic complexity simply involved decomposing the program into the smallest number of nodes with one entry point and one exit point. The code within the node will be guaranteed to be executed sequentially in the program. Initially, the cyclomatic complexity was directed at automatic test-case generation, but was later used to measure how difficult the code was to understand. The premise being that the more convoluted the logic, the more difficult it will be to understand. Unfortunately, computing the cyclomatic complexity of Smalltalk source using a static analysis would degenerate to every node containing a single message send. Therefore, it is not possible to determine from the source, the code that will be executed as a result of sending one message. The definition used to measure Cyclomatic Complexity for Smalltalk is based on the standard definition for Cyclomatic Complexity that increments the complexity based on the presence of well known keywords such as: "ifTrue:", "whileTrue:" and "do:"

The last static method metric is a flag that identifies methods as either *public* or *private* and can be used to filter methods based on their public/private status. By convention, public methods are available to anyone and private methods should only be sent by instances of the class that provides the method or one of its subclasses, but there is no requirement that the programmer correctly categorize the methods. The analysis of methods based on their public/private status may uncover significant differences between public and private methods.

3.4.2 Class Metrics

The state and behaviour of every object in a Smalltalk system is defined by a class. Classes are arranged in a tree structure that define superclass and subclass relationships. The implementation of a class may add to the state and modify the behaviour inherited from its superclass. Since both state and behaviour can be inherited, productivity improvements can be achieved through an efficient decomposition of objects that exploit code that has already been written and tested. The inclusion of classes and inheritance combined with polymorphism and late binding are significant new features for object oriented languages such as Smalltalk, that are not addressed by traditional metrics. The definitions of static class metrics developed in this thesis follow.

The *number of immediate subclasses of a class* measures the complexity of the inheritance tree and how the system has used inheritance to decompose the problem. The *number of descendants of a class* is related to the number of immediate subclasses and includes all the classes that share the definition supplied by the class. The premise

of these metrics is the belief that there are significant differences between classes that have many subclasses and those that have few. For example, abstract classes such as *Collection* often contain code that can be reused by subclasses; provided they implement the required protocol⁷. The *height of a class* counts the number of superclasses between the class and a class with no superclass⁸. Since objects inherit state and behaviour from their superclass, this metric can be used to identify factors that are affected by a class's position in the hierarchy.

The next group of metrics measure the complexity of the state introduced by the class. The *number of instance variables* counts the number of variables that are part of each instance and the *number of class variables* counts the number of class variables that are shared by instances of the class.

Reuse is often claimed to be one of the major benefits of OOP. The reuse metric measures the degree of reuse by examining the methods defined in the class and all of the methods that the class inherits. The *number of inherited methods* counts the number of messages implemented by all of the superclasses of the class. The *normalized number of inherited methods* removes methods defined in the root of the inheritance tree because methods defined at the root typically implement system support methods required by all objects and should therefore, not be counted. The *number of reused methods* counts the number of distinct messages that instances of the class will

⁷For example, *Collection* implements the iterator method: *select*:, *collect*:, *reject*: and *inject*:*into*: which can be reused by any of its subclasses that implement (or inherit) a definition for the *do*: method.

⁸Usually *Object*

respond to, that are not implemented by the class. The *number of refined methods* counts the number of distinct messages for which there is a new definition provided by the class. In this case, the class is either replacing the inherited method or adding new behaviour to an existing method. The *number of new methods* is a measure of the number of messages for which this class provides the first implementation in the inheritance chain. In this case, the class is defining a new method that has not yet been defined in any of its superclasses.

The methods defined in a class are measured by three metrics, the *number of methods implemented* and the *number of public and private methods*. Measuring the number of public and private methods in the class may provide useful information about the relationship between the class's public interface and the private messages that support it. The significance of this metric depends entirely on how the developer has chosen to categorize methods; therefore, the code will need to be examined before drawing conclusions.

The *average space in bytes occupied by instances* of the class is a static count of the memory space occupied by instances of the class. The number of instances metric computes the *number of instances of the class* that are in the image. These metrics can be used to assess static memory usage of objects in the system.

3.4.3 Application Metrics

Applications contain one or more classes and are higher level reusable components than classes. An application contain none or several other applications and specify the

applications that must be present in the image before it can be used. The static application metrics capture these essential features.

The first group of metrics measure the complexity of the application based on how it defines new classes and extends existing ones. The *number of defined classes* counts the number of class definitions provided by the application. The class definition specifies the representation of the class. The number of class definitions introduced by the application is a measure of the complexity added to the system. The *number of extended classes* counts the number of classes to which the application has added methods. A class extension cannot replace existing methods, nor can it modify the definition of the class. The number of class extensions introduced by the application measures the number of classes that were extended to provide the application behaviour. The *number of undefined classes* counts the number of undefined classes in the application. An undefined class is a placeholder for the name of a class that has yet to be implemented. Typically, undefined classes are found during the development of the system, and should not be found in a finished product. The *number of classes* is the sum of the defined, extended and undefined classes. The *number of public classes* counts the number of public classes and the *number of private classes* counts the number of private ones.

The second group of static application metrics measures the complexity of the application in terms of how it relates to other applications in the image. The *number of prerequisite applications* counts the number of applications that are direct prerequisites of the application. The *number of all prerequisite applications* counts all the

applications that are prerequisites of the application. The direct prerequisites also includes all of the inherited prerequisites and is similar to the all superclasses metric defined for classes. The *number of dependent applications* counts all of the applications that have this application as a prerequisite, either directly or indirectly. The *number of subapplications* counts the number of subapplications and is similar to the number of subclasses of a class metric. The *number of all subapplications* metric is similar to the all subclasses metric defined for classes. The *number of visible classes* metric counts the number of classes that can be referenced directly by the classes defined or extended in the application. For a class to be visible, it must be defined in an application or subapplication in the list of all prerequisite applications.

Finally, the *number of users* counts the group members in the application. A static measure of the evolution of an application can be measured by counting the *number of editions* in the library. As mentioned before, care must be taken when semantics are attached to the number of editions of an object in the library because the changes may be unrelated to the difficulty of developing the application. The *lines of comments* and the number of *lines of description* for an application are the final static application metrics gathered in this thesis.

3.4.4 Class Edition Metrics

Since more than one application can add methods to a class, it is reasonable to measure how applications add methods to classes in the system. The following metrics measure class editions.

The first group of static class edition metrics measures the complexity of a class edition based on the methods. The *number of methods* counts the number of methods in the class edition. The *number of public methods* counts the number of public methods and the *number of private methods* counts the number of private methods in the class edition. The results from these metrics may provide useful information about the relationship between the class's public interface and the private messages that support it.

The number of *lines of comments* and *lines of description* define another group of static class edition metrics.

A static measure of the evolution of a class edition can be measured by counting *the number of editions* in the library. While some changes may be trivial, such as adding or removing debugging information, or modifying the comment; other changes will be more substantial such as fixing errors or modifying the behaviour of the class edition. Care must be taken when semantics are attached to the number of editions of a class edition.

3.5 Runtime Metrics

These metrics are gathered from the running system and provide information that is not available from the static objects. This is especially important for languages such as Smalltalk, which support dynamic binding. Since variables in Smalltalk are not assigned types, it is not possible to always determine the class of an object referenced in a method. Runtime metrics can record this information.

Runtime metrics can also improve the quality of a system's test suite. One of the requirements of an adequate test suite is that it exercises all paths in the program. Runtime metrics can improve the quality of a test suite by determining the set of methods that have been executed.

All computation in a pure OOP language is accomplished by sending messages to objects. In order to understand how a system works one needs to understand the individual components and how they interact with the other objects in the system. This may be achieved by an accurate measure of the communication complexity by runtime metrics.

The simplest runtime metric, the *Method Coverage Metric* marks methods each time they are executed. After running the test, all of the methods that were executed at least once will be marked. This metric can be used to identify methods that have not been tested. The *Method Execution Metric* counts the number of times the method was executed and can be used to determine the distribution of method execution.

The relationship between methods on the runtime stack can be use to determine collaborating objects. The *Method Collaborator Metric* records the relationship between compiled methods either on each message send or at a user specified interval. The *Class Stability Metric* records the classes of each argument and local variable in the executing method. If variables can be identified as always containing objects of the same class, it may be possible for compilers to exploit this information to improve the runtime performance of the system.

3.6 Evolutionary Metrics

All large software systems evolve over time. Evolutionary Metrics examine changes in a system over time and may produce valuable insights into the evolution of software systems written in an object oriented language. One of the benefits of OOP languages is that they support rapid prototyping. The cost of prototyping functionality is much lower in an object oriented language than a procedural language, because of the combined benefits of the development environment and new language features. If this is true, it should be possible to track the evolution of a system. Given two versions of a system, the metrics that are gathered are: the number of added, reused and removed classes. A class is considered the same if it has the same name. For classes that were reused, changes in the class hierarchy are recorded. The number of added, removed, identical, equal, bigger and smaller methods are counted. A method is considered the same if it is in the same class and has the same selector.

3.7 Summary

This chapter has defined the metrics developed in this thesis. Three categories of metrics were described: Static, Runtime and Evolutionary. In each category, the definitions for the metrics and their motivation was presented. As discussed in Chapter 2, there is very little published material on what should be measured from an object oriented program. The metrics presented here are an attempt at defining a collection of useful metrics. In the next chapter we will investigate the results of applying these metrics to a wide variety of software systems.

4 Applying the Metrics

4.1 Introduction

In previous chapters, the motivation for gathering software metrics and the particular requirements of metrics for OOP languages were discussed. Using these requirements, a collection of candidate metrics were identified. This chapter describes the results from gathering these metrics from a number of projects. This chapter contains four major sections. The first section contains a brief description of each project. The second presents the results of applying the static metrics. The third and fourth sections contain the results from the runtime and evolution metrics respectively.

4.2 The Projects

The projects analyzed in this thesis include research prototypes and commercially available products and range in size from a several hundred methods to over ten thousand. To give a more accurate representation of each project, the data for the projects has been normalized to remove the classes and methods supplied by the base image.⁹ A brief description of each project follows:

⁹Otherwise, the effects of the base image would bias the results. This is especially important for small projects.

Project 1 - Simulation System

This is a simulation system written in Smalltalk and contains a complex user interface and a simulation engine. It also defines a script language for describing simulations and compiles the script into executable Smalltalk code. Seventy five person-months were required to complete this project. The number of developers varied from three to five.

Project 2 - A Team Programming Tool for Objectworks\Smalltalk

This is a team programming tool for Objectworks\Smalltalk. This product includes all of the base image code developed by ParcPlace except their development browsers, which were replaced. This project is a port of the code from Project 3. All of the classes and methods in the base image were measured for this project, which required 16 person-months. The project team varied in size from one to three developers.

Project 3 - Team Programming Tool for Digitalk V/286

This is a team programming tool for Digitalk/V286. This product includes all of the base image code developed by Digitalk except their development browsers. All of the classes and methods in the base image were measured for this project. Four person years of effort were required to deliver this project. The project team ranged in size from one to four developers.

Project 4 - Embedded Smalltalk Application

The metrics gathered from this project are from the Smalltalk portion of a real-time embedded application. The static metric data presented in this chapter was gathered

from the development image.

Project 5 - Computational Geometry Toolkit

This is a computational geometry toolkit and contains code written by several graduate and undergraduate students. A comprehensive set of data types were implemented along with a user interface for exploring computational geometry algorithms. The available geometric objects includes hierarchies of: Arcs, Circles, Lines, Polygons, Planes and Vertices, as well as hierarchies of Lists, Trees, Graphs and Nodes.

Project 6 - User Interface Builder

This product is a commercially available user interface builder written in Smalltalk. It contains layout tools and automatically generates Smalltalk code to create the windows specified by the user.

Project 7 - Code Formatter

This product is a replacement user interface for Project 3. In addition to the features supplied by the standard user interface, this product can automatically format Smalltalk source according to user configureable options. This project was completed by one developer in three months.

Project 8 - Multiprocessing Extensions

This product implements a variation of the Actor [Hewitt 77] multiprocessing model for Smalltalk by extending its process model to include Tasks and Actors. Tasks

implement a synchronous message passing facility and Actors are the active objects that send and receive messages. Using the support of a real-time operating system, Smalltalk Actors can communicate with all Actors running in a multiprocessor environment. This project was completed by one developer in three months.

Project 9 - Runtime Application Loader

This product implements a mechanism for quickly loading and unloading applications from a running Smalltalk system. Once the objects have been saved on disk, they can be loaded on demand. This product is useful when the memory requirements of a system are important. Materializing the objects stored on disk does not require compiling source code, which is a significant performance benefit. One developer completed this project in four months.

Project 10 - Expert System Framework

This is a beta release of an expert system framework written in Smalltalk. The framework supplies a base set of components necessary for building expert systems. A set of fundamental building blocks are provided for the expert system developer. These components include the fundamentals for facts, rules, inferencing engines and user tools. Expert systems will extend these primitive components to provide the required behaviour. In addition to a functional expert system kernel, an example provided in the product demonstrates how to extend the framework. This project was completed in six months by one developer.

Project 11 - Mailer

This is an internal release of a mail system written in Smalltalk. This product is modeled on the DiskBrowser from Digitalk and has evolved over a number of years. All of the standard mail features are supported. Six person weeks of effort were spent on this project.

Project 12 - Packager

This is a product for creating stand-alone Smalltalk applications. The packager is run in the development image and extracts objects required by the runtime system. New image files containing only the objects required by the runtime system are created by the packager. The packager enables the user to specify rules to control the inclusion or exclusion of objects from the runtime image. The collection of rules that were created can be stored on disk. One person year of effort was spent on this project.

Project 13 - Parser

This product is collection of reusable components for building lexical analyzers and LALR parsers in Smalltalk. The parsing tables generated by LEX or YACC are extracted and stored in Smalltalk. Three person months were required to complete this project.

4.3 Static Metrics

This section presents the results obtained by measuring the methods, classes, class editions and applications in each of the 13 projects. Four areas of interest are examined for each category. First, the values computed by the metrics and their average values are presented. This information is useful for determining typical values and for comparing results between projects. Next the data was analyzed to determine if there was a normal distribution of the values, since many statistical analysis procedures assume a normal distribution. The third section identifies statistically related metrics which may be eliminated to reduce the cost of collecting the data. The final section identifies metrics that did not provide useful information and the ones that did.

Insufficient data was available to perform a rigorous statistical analysis. The statistics presented in this section are based on the data that was available. Further research is required to empirically verify these results.

4.3.1 Method Metrics

Table 1 shows the results of applying the static method metrics to each of the 13 projects and includes for each metric, its mean value, standard deviation and the results of testing the data for a normal distribution. The columns from left to right in the tables are the number of methods in the project, and for these methods, the means of the static method metrics. These metrics are: the number of bytes in the compiled method, the string length, lines of code, lines of comment, the number of messages

sent, the number of temporary variables declared, the number of editions of the method in the library, the number of unique operators, the number of unique operands and the cyclomatic complexity of the method.

Project	Number	Bytes	Length	Comnt	Code	Msgs	Args	Temps	Editns	Opnds	Opatr	Cyclo
1	11972	36.14	287.07	1.97	4.91	3.96	.49	.54	1.02	8.47	3.96	.49
2	11033	20.62	294.39	1.77	5.23	4.16	.82	1.40	1.01	11.95	4.16	.81
3	5807	48.26	385.35	2.30	6.80	5.44	.68	.86	1.88	11.43	5.44	.87
4	9522	36.14	268.68	1.09	4.05	3.90	.63	.51	1.42	11.25	3.90	.43
5	4475	54.41	402.31	2.17	6.64	5.28	.74	1.14	1.00	13.78	5.28	1.26
6	1834	19.97	183.60	.39	3.91	4.25	.42	.94	1.00	10.32	4.25	.72
7	1891	58.40	426.78	1.76	8.29	6.52	.43	1.09	2.54	12.45	6.52	1.12
8	367	32.68	212.21	.80	4.46	3.88	.67	.44	2.04	9.18	3.88	.50
9	298	52.81	541.92	3.21	8.80	4.85	1.14	1.42	3.22	14.78	4.85	.93
10	974	26.36	209.71	1.59	3.75	2.83	.46	.46	1.62	6.29	2.83	.40
11	401	53.24	369.01	1.60	7.64	5.81	.65	1.10	1.84	12.48	5.81	1.11
12	450	74.77	458.34	.90	9.02	7.53	.73	1.45	2.2	14.12	7.53	1.71
13	251	36.72	339.33	1.61	8.44	3.36	.39	.49	3.41	12.29	3.36	.64
Mean	3790.3	42.34	336.82	1.62	6.30	4.75	.63	.91	1.86	11.44	4.75	.84
St.Dev.	4389.7	16.19	106.72	.73	2.00	1.32	.20	.38	.81	2.39	1.32	.38
Normal	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table I Summary of Static Method Metrics

Based on the collected data, there is a normal distribution for all metrics except the number of methods in each product. A correlation was run to determine the linear relationships between the metrics. The results are presented in Table 10 which includes both the Pearson product-moment correlation coefficients, and the coefficient's significant level.¹⁰ Evidently, there is a significant correlation between the number of byte codes, the size of the method, the lines of code in a method, the number of messages the method sends, the number of unique operators and operands and its cyclomatic complexity. Since there is a high degree of correlation between many of these metrics, it is not necessary to gather all of them. The correlation between token

¹⁰The significance level is the probability of obtaining the stated correlation value from a sample of the same size when the true population correlation is zero.

counts and lines of code is consistent with similar studies that compared the metrics in procedural languages. The number of arguments and the number of temporary variables are related and so are the number of lines of code and the number of lines of comment.

Some of the metrics did not generate useful information. Since the complete development history was not available for most projects; the number of method editions significantly underestimates the true number. More research is required to determine if this metric is useful in cases where the complete development history is available. Due to its high correlation to size measures such as the number of byte codes, it is doubtful that the Cyclomatic Complexity metric as defined for OOP is a useful complexity metric. A high correlation between Cyclomatic Complexity and lines of code has been previously reported in studies from procedural languages. It is likely that the complexity of a Smalltalk program is not in the source code for a method and that it is more affected by the communication complexity between objects.

4.3.2 Class Metrics

Table 2 shows the results of applying the static class metrics to each of the 13 projects. Table 11 contains the results from the correlation analysis. The columns from left to right in the tables are the number of classes in the project, and for these classes, the means of the static class metrics. These metrics are: the number of subclasses, the number of all subclasses, the height of the class, the number of class variables, the number of instance variables, the number of methods per instance variable, the number of inherited methods, the normalized number of inherited methods, the number of

reused methods, the number of refined methods, the number of new methods, the number of methods defined by the class, the number of public methods, the number of private methods, the number of instances of the class and the average size of the instances. When insufficient data was available to compute a value '.' is used.

Project	Num	Subels	AllSub	Height	CVar	IVar	M/ivar	Inhrt	N-Inh	Reuse	Refd	New	Meths	Priv	Pub	Insts	AvgSz
1	773	.77	2.56	3.80	.33	1.57	2.50	219.78	112.78	216.32	3.46	9.47	13.09	4.78	8.15	.45	53.85
2	481	1.01	7.53	3.36	1.19	2.08	5.04	168.99	50.99	164.22	4.77	13.81	18.58	2.05	16.53	178.00	277.53
3	234	1.05	2.61	2.49	.76	2.17	5.01	117.32	40.32	113.64	3.68	16.23	22.04	9.26	10.65	179.84	57.85
4	512	.70	2.10	3.50	5.55	6.57	3.85	141.43	53.43	138.60	2.83	14.52	19.19	2.10	15.25	6.27	44.46
5	198	.81	1.94	3.31	.10	1.97	4.43	141.13	63.13	137.05	4.07	14.82	18.90	.75	12.14	0	.
6	124	.71	2.09	4.09	.14	2.23	1.17	198.43	74.43	196.04	2.39	9.74	12.14	0	12.13	0	.
8	92	.67	1.34	2.71	.57	2.61	3.19	120.81	29.81	118.44	2.36	15.41	20.68	12.61	5.16	0.00	77.14
8	21	.52	.52	2.80	.52	1.09	2.63	131.95	54.95	129.80	2.14	9.95	15.80	3.80	8.28	0	.
9	13	.38	.38	1.76	1.46	4.15	3.30	81.30	4.30	80.46	.84	15.61	22.76	10.07	6.38	0	.
10	116	.68	1.11	2.42	0.00	1.27	2.69	93.31	14.31	91.31	1.99	5.69	9.59	2.29	5.39	0	.
11	12	.25	.25	2.25	.16	3.08	21.11	94	15	92.75	1.25	27.08	31.33	10.91	17.41	0.00	16
12	29	.37	.55	2.41	.27	1.82	4.08	111.65	20.65	109.96	1.68	11.48	18.20	4.58	8.58	.24	12
13	22	.72	2.04	3.27	.95	3.09	1.90	101.68	24.68	101.27	.40	9.27	10.68	7.5	2.18	0	.
Mean	202.07	.66	1.92	2.93	.92	2.59	4.68	132.44	42.98	129.93	2.45	13.31	17.92	5.43	10.32	28.06	41.44
St.Dev.	240.40	.23	1.88	.67	1.46	1.45	5.06	41.67	29.85	40.90	1.28	5.23	5.87	4.17	5.18	66.97	75.97
Normal	No	Yes	No	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No

Table II Summary of Static Class Metrics

Unlike the previous set of metrics, there were several class metrics which did not display a normal distribution. This may be the result of the small sample size. Three groups of related metrics were identified. The first group measures the structural complexity of the classes, including the number of subclasses and the number of all subclasses. Since both metrics are intuitively related, it is not surprising that they are also statistically. Project 2 had the highest value for the all subclasses metric. The Objectworks\Smalltalk version of the team programming tool contains many examples of deep inheritance hierarchies, which are reflected in the value of the metric. In Project 11 there is very little use of subclassing which results in low values for the subclassing metrics. A significant correlation was observed in the metrics that count the number of methods in each class. Each metric, although statistically related, measures different aspects of how the behaviour of the classes is implemented. Project 11 appears again as an extreme case based on the average number of methods per class. Classes in this product implement many more new methods on average than in any other product. In addition, few methods are inherited or refined. The results of these metrics combined with the limited use of subclasses and the number of methods per instance variable suggests that there are classes in Project 11 which are very large.

The amount of reuse in Project 1 is significantly different from Project 11. On average, classes reuse over 100 methods from their superclasses and add fewer than ten new ones. The structural complexity metrics also suggest that classes have been factored to maximize reuse.

Unfortunately, the categorization of methods as either public or private was inconsistently applied in the measured projects. Therefore, the applicability of results obtained from a statistical analysis of these metrics is suspect. Since there were very few instances of the classes being measured; metrics that counted their number and size returned zero most of the time. The only exceptions are Project 2 and Project 3 which counted base image classes.

4.3.3 Class Edition Metrics

Table 3 summarizes the results of applying the static class edition metrics to each of the 13 projects. See Table 12 for the results of the correlation analysis. The columns from left to right in the tables are the number of class editions in the project, the means of the static class edition metrics. These metrics are the number of methods, the number of public methods, the number of private methods, the lines of comment, the lines of description and the number of editions of the class edition in the library.

Other than highlighting the lack of comments and descriptions in some projects, the measures of comments and descriptions did not provide useful information. For the reasons mentioned previously, the editions metric does not accurately reflect the number of class editions created. The public and private metrics also suffer from the inconsistency in which classes were categorized between projects. Based on the available data, the usefulness of these metrics is questionable.

Project	Num	Meth	Priv	Pub	Comnt	Desc	Eds
1	1532	6.65	2.42	4.22	1.06	0.00	1.11
2	533	16.76	1.85	14.90	6.73	0	1.17
3	307	15.16	7.06	8.10	1.02	0	4.84
4	620	14.48	1.75	12.73	10.22	0.00	4.04
5	206	18.23	.73	17.50	0	0	1.21
6	178	9.11	0	9.11	0	0	1.11
7	140	11.88	8.32	3.55	8.18	0	5.97
8	35	7.65	2.31	5.34	5.34	0	4.54
9	23	10.47	5.73	4.73	.52	.17	11.91
10	202	4.45	1.31	3.13	5.18	0	2.50
11	18	19.05	7.27	11.77	0	0	7.33
12	43	9.41	3.09	6.32	0	0	7.55
13	27	8.11	6.11	2	19.14	0	4.48
Mean	297	11.64	3.68	7.95	4.41	0.00	4.44
St.Dev	418.37	4.67	2.80	4.91	5.68	0.00	3.20
Normal	No	Yes	Yes	Yes	No	No	Yes

Table III Summary of Static Class Edition Metrics

4.3.4 Application Metrics

The last group of static metrics were extracted from the applications in each of the 13 projects. A summary of the results can be found in Tables 4 and 13. The only metrics that tested positive for a normal distribution were measures of the prerequisite structures.

Since the number of classes in an application is the sum of the number of defined and extended classes, the observed correlation between these metrics is understandable. Little information was gained by counting the number of private classes and the number of group members, since private classes and group ownership were not used in the measured projects. In several projects, the ownership of classes was mapped to a single user in the released database which removes the class ownership information. Metrics that counted the lines of comments and descriptions were also hampered by the lack of data. As mentioned previously, the number of editions metric underestimates the true number of editions because the complete development history of many projects was not available. These metrics did not provide enough data to compare results between projects, but may be useful when more data is available.

The number defined and extended is a good measure of the complexity of an application. Comparing Projects 1 and 2, it is evident that applications in Project 1 define significantly more classes than Project 2. This is offset, somewhat, by increased number of class extensions and prerequisite applications in Project 1. There may be a trade-off between large applications and class extensions or prerequisites.

Project	Num	Def	Ext	Undef	Class	Pub	Priv	Membs	Prereq	AllPrq	Depdnt	Subs	AllSub	AllPnt	Visibl	Commt	Descrp	Eds
1	98	7.88	7.74	0	15.63	7.88	0	1.38	1.83	13.29	1.62	.62	.81	.81	473.63	.11	0	1.46
2	22	21.86	2.36	0	24.22	20.18	1.68	1	1.16	1.66	1.33	.72	1.09	1.09	210.90	0	0	2.09
3	31	7.54	2.35	0	9.90	6.38	1.16	1	1.22	2.66	1.44	.70	1	1	122.25	0	0	13.25
4	95	5.37	1.14	0	6.52	5.37	0	1.98	1.78	12.65	1.41	.46	.57	.65	276.16	.66	0.00	10.62
5	1	198	8	0	206	198	0	1	3	9	0	0	0	0	396	0	0	2
6	1	124	54	0	178	124	0	1	2	7	0	0	0	0	574	0	0	10
7	12	7.66	4	0	11.66	7.66	0	1.58	1.66	5.66	1.83	.5	.58	.58	163.16	1.08	0	23.5
8	3	7	4.66	0	11.66	7	0	3	1	7.5	.5	.33	.33	.33	210.66	0	0	23
9	3	4.33	3.33	0	7.66	4.33	0	1.66	1.5	3.5	.5	.33	.33	.33	100.66	0	0	61.66
10	8	14.5	10.75	0	25.25	14.5	0	1	1.87	5.87	1.25	0	0	0	179.75	0	0	5.87
11	4	3	1.5	0	4.5	3	0	1.25	2.5	4.5	.5	.5	.5	.5	16.5	0	0	18
12	4	7.25	3.5	0	10.75	7.25	0	1.25	1.5	4.5	.5	.5	.5	.5	187.5	18.75	0	16.25
13	3	7.33	1.66	0	9	7.33	0	1	1.5	5.5	.5	.33	.33	.33	137.66	4.66	0	5.66
Mean	21.92	31.97	8.07	0	40.05	31.76	.21	1.39	1.73	6.40	.87	.38	.46	.47	245.94	1.94	0	14.87
St.Dev	34.27	59.41	14.10	0	67.95	59.48	.54	.57	.54	7.51	.62	.25	.35	.35	145.68	5.21	0	15.97
Normal	NO	NO	NO	NO	NO	NO	NO	NO	YES	YES	YES	YES	YES	YES	NO	NO	NO	NO

Table IV Summary of Static Application Metrics

4.3.5 Taking a Closer Look

The high level view of the metrics presented in the previous four sections highlights the relationships between metrics, their average values and outliers¹¹. In this section we will take a more detailed look at individual projects.

One of the benefits of OOP is its ability to reuse code. There are several types of reuse: Inherited methods can be reused, reusable components can be used and frameworks can be reused. But since they are difficult to design, it is unlikely that a project will be composed entirely of such components.¹² Table 5 contains metrics from the applications in Project 1. The major function of each application has been categorized as either application specific or user interface specific. Applications that are candidates for reuse in other products are also identified.¹³ Forty percent of the application developed for the product are candidates for reuse. Further research is required to determine if this value is typical of successful projects written in Smalltalk.

¹¹Outliers are results that differ greatly from others in the same sample.

¹²The definition of a "reusable component" is one or more classes that can be reused with minimal modifications in projects which are not necessarily in the same application domain.

¹³Reusable applications have a "R" beside them.

#	Def	Ext	Undef	Class	Pub	Priv	Membs	Prereq	AllPrq	Depdnt	Subapp	AllSub	AllPnt	Visibl	Comnt	Descrp	Eds
98	7.88	7.74	0	15.63	7.88	0	1.38	1.83	13.29	1.62	.62	.81	.81	473.63	.11	0	1.46
Image	3	0	0	3	3	0	2	1	1	1	0	0	0	85	0	0	2
UI R	1	0	0	1	1	0	1	1	1	1	0	0	0	83	0	0	1
APP	1	0	0	1	1	0	1	1	1	1	0	0	0	96	0	0	1
APP R	2	.5	0	3	2.5	0	1	2	3	3	.5	.5	.5	90	0	0	1
APP R	1	0	0	2	2	0	1	1	2	1	0	0	0	90	0	0	1
APP R	2	14.5	0	15.5	1	0	1.5	2	4	7	.5	.5	.5	161	0	0	1.5
APP R	1	3	0	4	1	0	1	1	4	1	0	0	0	97	0	0	1
APP R	2	2.5	0	12.5	10	0	1	2	5	2	.5	.5	.5	137	5.5	0	1
UI R	1	1	0	3	2	0	1	1	5	1	0	0	0	163	0	0	1
APP	1	0	0	2	2	0	1	1	5	1	0	0	0	163	0	0	1
APP	14	2.28	0	11.5	9.21	0	1.35	1	5	5	.92	1.78	1.78	290	0	0	1.35
APP	1	17	0	37	20	0	2	1	5	2	0	0	0	181	0	0	2
APP R	2	3	0	4	1	0	1	1	5	1	.5	.5	.5	163	0	0	1
APP R	1	0	0	2	2	0	1	1	5	1	0	0	0	163	0	0	1
APP R	1	0	0	5	5	0	1	1	5	1	0	0	0	166	0	0	1
APP	1	47	0	52	5	0	2	1	6	1	0	0	0	295	0	0	1
APP	1	3	0	11	8	0	2	3	10	1	0	0	0	327	0	0	2
APP	8	15.25	0	26.37	15.25	0	1.25	4	11	2	.87	.87	.87	465	0	0	1.25
APP R	1	11.12	0	20	19	0	2	1	6	2	0	0	0	182	0	0	2
UI R	1	1	0	4	4	0	1	1	7	1	0	0	0	186	0	0	1
UI R	1	4	0	8	4	0	1	1	7	1	0	0	0	186	0	0	1
APP	9	16.66	0	24.44	7.77	0	1.55	1	12	1	.88	.88	.88	535	0	0	1.55
APP	1	3	0	45	3	0	2	1	12	1	0	0	0	468	0	0	2
UI R	1	49	0	54	49	0	2	2	9	3	0	0	0	237	0	0	2
APP R	9	4.55	0	10.77	6.22	0	1.44	3	13	1	.88	1.22	1.22	430	0	0	1.66
APP R	1	8	0	35	27	0	2	1	10	1	0	0	0	264	0	0	4
UI R	1	1	0	20	19	0	1	3	12	1	0	0	0	405	0	0	1
UI	1	9.5	0	27	17.5	0	1	1	11	1	.5	.5	.5	299	0	0	1
UI	1	13	0	14	13	0	2	5	21	1	0	0	0	489	0	0	2
UI	1	0	0	3	3	0	2	2	23	1	0	0	0	494	0	0	2
UI	14	10.28	0	17.21	6.92	0	1.35	4	30	4	.92	1.21	1.21	814	0	0	1.57
UI	2	1.5	0	3	1.5	0	1	1	31	2	.5	.5	.5	817	0	0	1
APP	2	7	0	10	3	0	1	1	31	2	.5	.5	.5	820	0	0	1
UI	4	11.5	0	13.5	2	0	1.5	2	32	2	.75	.75	.75	841	0	0	1.5
UI	1	0	0	15	15	0	2	2	32	2	0	0	0	885	0	0	2
APP	3	15.66	0	16.66	1	0	1.66	4	37	0	.66	.66	.66	924	0	0	1.66
UI	1	0	0	4	4	0	2	2	34	1	0	0	0	892	0	0	2
APP	1	0	0	4	4	0	1	5	40	0	0	0	0	932	0	0	3

Table V Project 1 Application Metrics

Upon closer examination, it is evident that most objects are simple. Therefore, the average values reported in the summary tables are biased by the few objects that are very complex. For example, most applications in Table 5 define five or fewer classes. Therefore, the mean value of almost eight classes per application is biased by the few that define many classes.

This skewed distribution is evident in many metrics. For example, the vast majority of methods are one line long and most messages have one implementor (See Figures 1, 2 and 3).

Figure 1 Lines of Code for Methods in Project 4

Figure 2 Lines of Comment for Methods in Project 4

Figure 3 Number of implementors of messages in Project 4

Figures four through nine contain histograms for selected class metrics from Project 4. Figure 7 which contains a histogram of class height metrics is the only figure that suggests a normal distribution of the data. All other figures have similar skewed distributions as was observed in static method metrics.

The number of instances of a class is a static snapshot of the system when the metrics were collected. Most classes had no instances. Classes with the most instances were associated with CompiledMethods and Classes.

Figure 4 Number of Subclasses for Classes in Project 4

Figure 5 Number of all Subclasses for Classes in Project 4

Figure 7 Height of Classes in Project 4

Figure 8 Number of instance variables for Classes in Project 4

Mean: 19.2

Median: 13

Mode: 0

Figure 9 Number of methods for Classes in Project 4

Mean: 6.5

Median: 5

Mode: 2

Figure 10 Number of classes for Applications in Project 4

4.3.6 Programmer Productivity

In addition to promoting reuse, it has been suggested that programmer productivity is significantly improved when an object oriented language is used. However, it is not clear how to measure productivity in an object oriented language. Since inheritance and reuse are fundamental requirements of good object oriented programs, it is counter intuitive to reward productivity based on lines of code per person-month.¹⁴ Only new or substantially modified code was included in the productivity calculations which gives a more accurate measure of the amount of code developed in a month but significantly underestimates the "Bang for the Buck" [DeMarco, 82] of reusing classes in the base image or classes developed in other projects.

Based on the available data the average number of methods per person-month was: 148. There is a significant difference between the productivity observed from projects that are production quality vs those which are ports or prototypes. The observed productivity from completed systems was 107 methods per person-month. This number more than doubled to 216 methods per person month when only prototypes and ports were measured. These numbers support the claim that it is possible to produce a prototype relatively quickly, but it takes considerably longer to deliver the finished product. Translating these productivity numbers into the number of methods or classes per person-month gives 10 classes per person-month on average. For prototype and production systems the number of classes per person-month is 15 and 7 respectively.

¹⁴A better model would be to reward programmers when their components are reused in other projects. This has been proposed by several proponents of OOP.

Based on 107 methods per person-month and an average method size of 8 lines of comment and code yields 856 delivered source lines of code per person-month. The average productivity from procedural languages has been reported at 300 delivered source lines per month [Walston and Felix, 77]. Productivity numbers are influenced significantly by the effects of individual capabilities. Since most of the projects measured in this thesis were developed by small teams, this effect is likely to be significant.

4.3.7 Cost Models

The desire to accurately predict both the cost and the time required to develop a system is important to both software developers and their managers. The data gathered thus far suggests that developing such a model is possible. When aggregate measures of the number of methods per class and the size of methods are collected a normal distribution was detected. The observed standard deviation suggests that many values are close to the mean. A cost model for developing systems in Smalltalk may achieve significant accuracy based on the simple assumption that classes have an average of twenty methods with an average of six lines of code each. It is likely that estimating the number of classes will be more accurate than estimating the number of methods or lines of code because OOP design methodologies like CRC identifies them.

Programmer productivity, on the other hand, varied significantly more than the code

metrics. Productivity numbers will need to be experimentally determined for individual organizations to improved the accuracy. The variance in these numbers are affected significantly by the experience of individual programmers, the quality requirements of the product¹⁵ and the difficulty of the application being developed. Nevertheless, once these numbers have been determined, acceptable predictions of time and cost can be computed.

4.4 Applying Runtime Metrics

The runtime metrics gather information that is not available statically. The results from the class stability metric are interesting, since most of the arguments and instance variables always pointed to objects of the same class. This suggests that runtime optimization could exploit this information to improve performance. Class Stability was observed in all the test cases. In one case, there were 571 unique variables, of these 545 or 95 percent always pointed to an object of the same class. The remaining five percent pointed to objects from an average of 4.1 classes. Most of the time, these objects were either part of the same hierarchy, or the method was implemented by one of the Collection classes¹⁶.

By examining the communication pattern between objects, we find the details of the system's runtime behaviour. The communication complexity is a significant factor in the complexity of an object oriented system. There are four types of possible collabor-

¹⁵i.e. Is the product a conceptual prototype, or is it a commercial product.

¹⁶Collections make significant use of polymorphism supported by Smalltalk's dynamic binding.

ation: sending a message to yourself, sending a message implemented in a superclass, sending a message implemented in a subclass, and sending a message to an unrelated object. The information generated by the runtime metrics exposes the interaction between objects. This information is essential for understanding how groups of objects cooperate to perform complex tasks. Filters can be used to limit the scope to a specific set of objects which will remove unwanted interactions being reported.

In project 7, the first type of collaboration accounted for five percent. A further 12 percent of the collaboration was between a class and one of its parent classes; and 10 percent was between a class and one of its subclasses. The remaining 73 percent were the result of collaboration between objects that were not related by inheritance.

4.5 Evolutionary Metrics

Extracting measures from a project as it evolves from prototype to product may give insights into how software systems are developed in an object oriented language. The evolutionary metrics were applied to several projects for which the source code was available. For example, comparing the changes between the first alpha release and the most recent beta release of Project 10, the total number of classes grew by one to 121 and the total number of methods increased by three to 974. There were 13 new classes

added to the system and 12 were removed. A total of 818 methods were reused, 156 were added and 153 were removed. Of the reused methods 117 or 15% of them were identical. Since 85% of the methods changed between versions, stability has not yet been achieved in the project. For the methods that changed, most of them 588 (71%) increased in size. Of the 818 methods that were reused, 624 of them had more than one text difference. Of these, 524 of them were limited to one change. One hundred methods had two changes, nine methods had three changes and four methods had four changes each.

A late beta of Project 3 and its initial product release was also compared. During the ten months between these versions a considerable amount of code was modified. Most of the changes were to the browser classes and more methods were added than removed. This is an indication of added user interface features. No classes were removed from the user interface and several classes were added to support dialogue boxes.

The changes in the browsers for Project 2 and Project 3 is the final example of the evolution of an application. The browser code in Project 2 originated from the code that was measured in Project 3. Between projects the average method size decreased by 30 percent while the number of lines of comments has increased. The average height of classes has also increased due to the inclusion of abstract classes. Abstract classes and a reorganization of the hierarchy have also increased the amount of reused code. There has also been a better factoring of public and private methods which will make the code easier to understand. The total size of the user interface components in

Project 2 decreased in spite of the fact that the features in Project 2 are a superset of the features in Project 3. This is an example of an often stated benefit of OOP: That a consolidation phase will reduce code size due to better factoring of behaviour and the inclusion of abstract classes.

4.6 Summary

These results have an immediate practical benefit for developers and managers who are trying to understand and improve their software development process. Based on the available data there are a few key metrics that should be gathered from methods, classes and applications. The size of methods should be measured. Since there is a significant correlation between them, one or two size metrics should be sufficient. If the data is available, the number of editions of each method should also be counted. Finally, the lines of comments in each method should be gathered. The organization of classes should be measured, this requires computing the height of the inheritance tree and the number of subclasses. The number of new, reused, refined and normalized inherited methods should also be counted. If the data is available, counting the number of public and private methods should be gathered. Finally, applications should be measured. The number of classes defined and extended by the application should be measured along with its prerequisite applications and subapplications. The results of gathering these metrics should be compared with the values presented in this chapter and outliers should be investigated. The runtime metrics are useful tools for

understanding the runtime behaviour of a system and can also be used to improve test case coverage. The evolution metrics will extract the changes in a software system. This information is useful in determining areas in a large project that have changed significantly.

5 Conclusions and Future Work

5.1 Conclusions

Software metrics are a fundamental component of the software maturity framework as defined by the Software Engineering Institute¹⁷. They will become even more important as software developers are compelled to achieve a minimum process maturity to compete for software development contracts. If object oriented programming languages gain wider acceptance, metrics will play an important role.

OOP languages and development environments affect not only the code that is written but also the way systems are designed. The metrics presented in this thesis are an initial attempt at defining a suite of metrics for object oriented languages. This thesis identified three categories of metrics: Static, Runtime and Evolutionary. For each category, metrics were developed to capture the essence of software development in an object oriented language. Software was developed to automatically gather the metrics and was applied to several projects.

Several significant results have been derived from the static metric data. Notably, results concerning the size and structure of object oriented systems. For example, inheritance is a significant benefit of object oriented languages because it enables code

¹⁷The Software Engineering Institute is part of the Software Technology for Adaptable, Reliable Systems research program by the DoD. Its mandate is to encourage the development of improved software processes.

to be reused. The ratio between added and reused code ranges between a factor of two and ten. The amount of reused code appears to increase with the size of the project which reinforces the benefits of OOP. In addition to reusing inherited methods, a significant amount of higher level reusable components were observed. In one project approximately forty percent of them were reusable. It is these large grained reusable components that will significantly improve the productivity of software developers.

Measures of the number of methods per class and the average size of methods suggest that cost models that are based on the number of classes and programmer productivity can be used to predict the development time and cost of future products.

Most objects in a Smalltalk application are simple. For example, most methods are one line long. This skewed distribution was observed in methods, classes and applications and suggests a pervasive style for Smalltalk applications.

Examining the runtime behaviour of a Smalltalk application showed that most variables point to objects of a single class. The cases in which there were multiple classes was dominated by Smalltalk's Collection classes and by hierarchies of related objects. For example, variables in code that is inherited pointed to instances of the classes that inherited it.

The evolution metrics were useful in performing a post mortem analysis of the changes that took place in a project. The data supports the model of object oriented software development proposed by McKenna [McKenna 92] in which development progresses

in alternating stages of adding features followed by a period of consolidation.

Based on the available data, metrics that provided useful information were identified. A statistical analysis was also performed to identify statistically related metrics. This information will enable others to immediately apply the proposed metrics to their work.

5.2 Future Research Topics

There are several topics not covered in this thesis that may provide interesting results. Measuring software defects in projects developed in an object oriented language is one of the more interesting areas. This area was not considered because of the lack of available data. It would be interesting to determine the cause of the defects and to see if dynamic binding is a significant factor and to determine the correlation between defects and the metrics developed in this thesis for example. If a significant correlation was observed, a predictive measure of program quality could result.

Insufficient data was available to determine the effectiveness of some of the proposed metrics. More data is required to empirically validate the proposed metrics.

It would also be interesting to compare the results from object oriented languages with results gathered from procedural languages. This will permit comparisons of

programmer productivity and quality metrics.

The proposed metrics and the observed results are likely to apply to other object oriented programming languages like C++, Eiffel and Objective C. Further research is required to determine if this is the case. There are many areas that could be investigated. For example, do the proposed metrics generate useful information for languages other than Smalltalk? While all object oriented programming languages share a fundamental set of language features, there are significant differences between pure object oriented languages like Smalltalk and hybrid languages like C++. The proposed metrics that measure attributes that the languages have in common are likely to be useful. It would also be useful to determine if there is a pervasive "style" for each language. For example, do systems developed in a language that supports multiple inheritance use it to factor the system more effectively than is possible with single inheritance and are there significant differences in class and method sizes between systems written in Smalltalk and C++. The effects of multiple inheritance on the complexity, quality and maintainability of a complex system are still open questions. In addition to comparing the effects of language features, it may be useful to examine the effects of the development environment on productivity and quality? Care should be taken to control the effects of the tremendous variation in capabilities of individual programmers.

5.3 The Last Word

In conclusion, software metrics have become an important area of study in software engineering research. Object oriented languages and development environments are significantly changing the way in which applications are developed. Metrics for object oriented systems is a new area in which there is considerable room for future research. The results obtained by applying the metrics to several substantial projects has shown the usefulness of the proposed metrics. Hopefully the research presented in this thesis has provided insights into the requirements for metrics for object oriented languages and a foundation for work to follow.

6 References

Albrecht 79

Albrecht, A.J. Measuring application development productivity. *Proceedings of the Joint SHARE/GUIDE Symposium* (October 1979) pg 83-92.

Boehm 81

Boehm, B.W. *Software Engineering Economics*. Englewood Cliffs, NJ; Prentice-Hall. 1981.

Boehm and Papaccio 90

Boehm, B.W. and Papaccio P.N. *"Understanding and Controlling Software Costs"*. Software State-of-the-Art: Selected Papers, New York, New York; Dorset House Publishing, 1990.

Card 86

Card, D.N. Church, V.E. and Agresti, W.W. "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, 12 no. 2 (February 1986), 264-271.

Chidamber 91

Chidamber, S.R. Towards a Metrics Suite for Object Oriented Design, *OOPSLA'91 Proceedings*, 1991 pg 197-211.

Conte, Dunsmore, Shen, 86

Conte, S.D. Dunsmore, H.E. Shen, V.Y. *Software Engineering Metrics and Models*, Menlo Parc, California; The Benjamin/Dummings Publishing Company, Inc. 1986.

DeMarco 82

Demarco, T. *Controlling Software Projects - Measurement Management & Estimation*. New Jersey; Yourdon Press. 1982.

DeNatale, Irby, Lalonde, Leathers and Phillips 90

Rick DeNatale, Charles Irby, John LaLonde, Burton Leathers, Reed Phillips. "Panel - OOP in the Real World". *OOPSLA-ECOOP'90 Proceedings*, 1990. pp 299-302.

Dunsmore 76

Dunsmore, H.E. Gannon, J.D. Data Referencing: an empirical investigation. *IEEE Computer* 12, 12 (December 1979) pg 50-59.

Deming 82

Deming, W.E. *Out of the Crisis*. Cambridge, MA; MIT Center for Advanced Engineering Study, 1982.

Elshoff 76

Elshoff, J.L. An analysis of some commercial PL/I programs. *IEEE Transactions on Software Engineering* SE2-2 (June 1976) pg 113-120.

Goldberg and Robson 83

Goldberg, A and Robson, D. *Smalltalk-80 - The Language and its Implementation*. Massachusetts; Addison-Wesley Publishing Company. 1983.

Halstead 77

Halstead, M.H. *Elements of Software Science*. New York; Elsevier North-Holland, 1977.

Humphrey 91

Humphrey, W. S. *Managing the Software Process*, Massachusetts; Addison-Wesley Publishing Company. 1991.

McCabe 76

McCabe, T.J. A Complexity Measure. *IEEE Transactions on Software Engineering* SE- 2,4 (December 1976), pg 437-439.

McKenna 92

McKenna, Jeff. A proposal for change management in Smalltalk. *The Smalltalk Report*, Feb 92.

Rubey and Hartwick 68

Rubey R.J. and Hartwick R. D. "Quantitative measures of program quality". *Proceedings of the ACM National Conference*, 1968. pp 671-677.

Stroustrup 86

Stroustrup, B. *The C++ Programming Language*. Massachusetts; Addison-Wesley Publishing Company. 1986.

Walston and Felix, 77

Walston, C.E. and Felix, C.P. A method of programming measurement and estimation. *IBM System Journal* 16,1 (1977) pg 54-73.

Wirfs-Brock, Wilkerson, Wiener 90

Wirfs-Brock, R. Wilkerson, B. Wiener, L. *Designing Object-Oriented Software*, Englewood Cliffs, New Jersey; Prentice Hall, 1990.

Appendix A: Definitions

Bytes	The size of a method's byte code array.
Length	The string size of a method. Includes message selector, comment and code.
Comnt	The number of lines of comment in a method. Does not include embedded comments.
Code	The number of non-blank lines of code in the method.
Msgs	The number of messages sent by the method.
Args	The number of arguments passed to the method.
Temps	The number of temporary variables declared by the method.
Editns	The number of method editions contained in the library.
Opnds	The number of operands in the method. An operand counts the number of variables (instance, temporary, class, global etc) and constants in the method. The number of unique operands is computed as well.
Opatr	The number of operators in the method. All tokens that are not operands are counted. Operators include i.e messages sent in the method. The number of unique operators is computed as well.
Cyclo	The Cyclomatic Complexity of the method. The definition of Cyclomatic Complexity for Smalltalk is based on "well known" messages like: #ifTrue:, #ifTrue:ifFalse:, #do:, #select: etc.

Table VI Method Metrics

Subcls	The number of subclasses of the class.
AllSub	The number of descendents of the class. Recursively include all of the classes that have the class as a parent.
Height	The number of classes between the class and a class with no superclass (usually Object). This metric measures the number of classes that can provide inherited state and behavior.
CVar	The number of class variables in the class.
IVar	The number of instance variables defined by the class.
M/Ivar	Methods per instance variable.
Inhrt	The number of messages inherited by the class.
N-Inh	The number of messages inherited by the class after removing messages defined by the root (Object).
Reuse	The number of inherited messages that are not redefined in the class.
Refd	The number of inherited messages that are redefined in the class.
New	The number of messages that are not defined in one of the class's superclasses.
Methods	The number of methods defined in the class.
Priv	The number of private methods defined in the class.
Public	The number of public methods defined in the class.
Insts	The number of instances of the class in the image.
AvgSz	The average size of the instances

Table VII Class Metrics

Meth	The number of methods in the class edition.
Priv	The number of private methods in the class edition.
Pub	The number of public methods in the class edition.
Commnt	The number of lines of comment in the class edition.
Desc	The number of lines of description in the class edition.
Eds	The number of editions of the class edition in the library.

Table VIII Class Edition Metrics

Def	The number of classes defined by the application.
Ext	The number of classes extended by the application.
Undef	The number of undefined classes in the application.
Class	The number of classes in the application. The number of classes includes both defined and extended classes.
Pub	The number of public classes in the application.
Priv	The number of private classes in the application.
Membs	The number of group members in the application.
Prereq	The number of applications that are prerequisites of the application. Inherited prerequisites are not included in this metric.
AllPrq	The total number of applications that are either direct or inherited prerequisites of the application.
Depdnt	The number of applications that have this application as a prerequisite (direct or inherited)
Subs	The number of subapplications in the application.
AllPnt	The number of parent applications.
Visibl	The number of classes that are visbile. A visible class is defined by the application or by an application that is a prerequisite.
Comnt	The number of lines of comment.
Descrp	The number of lines of description.
Eds	The number of editions of the application in the library. application.

Table IX Application Metrics

Appendix B: Correlation Tables

	Number	Bytes	Length	Comm	Code	Msgs	Args	Temps	Editns	Opnds	Opatr	Cyclo
S=	1.0000	-0.3445	-0.2292	0.1204	-0.4393	-0.2315	0.0189	-0.0967	-0.6257	0.196	-0.231	-0.3185
	0.0000	0.2490	0.4512	0.6952	0.1330	0.4466	0.9511	0.7532	0.0221	0.5201	0.4466	0.2888
Bytes		1.0000	0.8321	0.2796	0.8064	0.8647	0.3069	0.5009	0.4073	0.6680	0.8647	0.8266
S=	0.2490	0.0000	0.0004	0.3549	0.0009	0.0001	0.3078	0.0812	0.1672	0.0126	0.0001	0.0005
Length			1.0000	0.6674	0.8989	0.6880	0.5897	0.6832	0.5510	0.8282	0.6880	0.7156
S=	0.4512	0.0004	0.0000	0.0127	0.0000	0.0093	0.0339	0.0100	0.0510	0.0005	0.0093	0.0060
Comm				1.0000	0.4456	0.0340	0.5595	0.3095	0.3153	0.3469	0.0540	0.0993
S=	0.6952	0.3549	0.0127	0.0000	0.1270	0.8608	0.1468	0.3033	0.2939	0.2455	0.8608	0.7469
Code					1.0000	0.6797	0.3176	0.722	0.7173	0.7942	0.6797	0.7323
S=	0.1330	0.0009	0.0000	0.1270	0.0000	0.0106	0.2902	0.0411	0.0058	0.0012	0.0106	0.0044
Msgs						1.0000	0.2487	0.7028	0.1492	0.6664	1.0000	0.9183
S=	0.2315	0.8647	0.6880	0.0540	0.6797	1.0000	0.4125	0.0074	0.6265	0.0129	0.0000	0.0000
Args							1.0000	0.6240	0.1529	0.5690	0.2487	0.3246
S=	0.0189	0.3069	0.5897	0.5595	0.3176	0.2487	0.4125	0.0000	0.0226	0.6179	0.0424	0.4125
Temps								1.0000	0.0560	0.7603	0.7028	0.8028
S=	-0.0967	0.5009	0.6832	0.3095	0.5720	0.7028	0.6240	1.0000	0.8556	0.0026	0.0074	0.0010
Editns									1.0000	0.3989	0.1492	0.1548
S=	0.6257	0.4073	0.5510	0.3153	0.7173	0.1492	0.1529	0.0560	0.0000	0.1769	0.6265	0.6134
Opnds										1.0000	0.6664	0.7557
S=	-0.1964	0.6680	0.8282	0.3469	0.7942	0.6664	0.5690	0.7603	0.3989	1.0000	0.0129	0.0028
Opatr											1.0000	0.9183
S=	0.4466	0.0001	0.0093	0.8608	0.0106	0.0000	0.4125	0.0074	0.6265	0.0129	0.0000	0.0000
Cyclo												1.0000
S=	-0.3185	0.8266	0.7156	0.0993	0.7323	0.9183	0.3246	0.8028	0.1548	0.7557	0.9183	1.0000
	0.2888	0.0005	0.0060	0.7469	0.0044	0.0000	0.2791	0.0010	0.6134	0.0028	0.0000	0.0000

Table X Correlations Between Static Method Metrics

Number	NumberSubcls	AllSub	Height	CVar	IVar	M/IVar	Inhrt	N-Inh	Reuse	Reid	New	Metbs	Priv	Pub	Insts	AvgSz
S=	1.0000	0.5491	0.5910	0.3732	0.1346	-0.1823	0.7321	0.7458	0.7247	0.6793	-0.1515	-0.1778	-0.344	0.2932	0.2978	0.5026
	0.0000	0.0520	0.0334	0.0325	0.2091	0.6611	0.5510	0.0644	0.0034	0.0051	0.0106	0.6211	0.5611	0.2495	0.3308	0.0800
Subcls	0.5491	1.0000	0.7547	0.5199	0.0907	-0.1476	-0.4420	0.4655	0.4705	0.4507	0.7516	0.3158	0.3607	-0.321	0.1276	0.6911
S=	0.0520	0.0000	0.0029	0.0686	0.7681	0.6303	0.1304	0.1089	0.1047	0.1222	0.0030	0.2932	0.2259	0.2836	0.6778	0.0089
AllSub	0.5910	0.7547	1.0000	0.4838	0.1381	-0.0643	-0.1694	0.5002	0.3624	0.4872	0.7154	-0.108	-0.1529	-0.345	0.3464	0.7425
S=	0.0334	0.0029	0.0000	0.0939	0.6527	0.8346	0.5800	0.0817	0.2236	0.0912	0.0060	0.7232	0.6179	0.2477	0.2463	0.0036
Height	0.5933	0.5199	0.4838	1.0000	0.1625	-0.0026	-0.3671	0.8637	0.8294	0.8656	0.4594	-0.3745	-0.5203	-0.632	0.2462	-0.0018
S=	0.0325	0.0686	0.0939	0.0000	0.5957	0.9932	0.2172	0.0001	0.0005	0.0001	0.1142	0.2074	0.0683	0.0204	0.4175	0.9953
CVar	0.3732	0.0907	0.1381	0.1625	1.0000	0.8852	-0.1049	-0.0024	0.0177	-0.0043	0.0609	0.0932	0.1119	-0.116	0.2030	0.0398
S=	0.2091	0.7681	0.6527	0.5957	0.0000	0.0001	0.7329	0.9938	0.9542	0.9888	0.8433	0.7619	0.7158	0.7056	0.5059	0.8971
IVar	0.1346	-0.1476	-0.0643	-0.0026	0.8852	1.0000	0.1205	-0.1934	-0.1973	0.1905	-0.2079	0.3559	0.3310	0.1471	0.1891	-0.1223
	0.6611	0.6303	0.8346	0.9932	0.0001	0.0000	0.6948	0.5266	0.5181	0.5330	0.4954	0.2327	0.2693	0.6313	0.5361	0.6906
M/IVar	-0.1823	-0.4420	-0.1694	-0.3671	-0.1049	0.1205	1.0000	0.3117	-0.3106	-0.3131	-0.1421	0.8641	0.7879	0.4001	0.5137	0.0283
S=	0.5510	0.1304	0.5800	0.2172	0.7329	0.6948	0.0000	0.2998	0.3017	0.2975	0.6431	0.0001	0.0014	0.1755	0.0725	0.9268
Inhrt	0.7321	0.4655	0.5002	0.8637	-0.0024	-0.1934	-0.3117	1.0000	0.9458	0.9996	0.6139	-0.2989	-0.3652	-0.536	0.2830	0.1157
S=	0.0044	0.1089	0.0817	0.0001	0.9938	0.5266	0.2998	0.0000	0.0000	0.0000	0.0256	0.3211	0.2198	0.0589	0.3487	0.7066
N-Inh	0.7458	0.4705	0.3624	0.8294	0.0177	-0.1973	-0.3106	0.9458	1.0000	0.9445	0.6117	-0.2933	0.3566	-0.506	0.2639	0.0432
S=	0.0034	0.1047	0.2236	0.0005	0.9542	0.5181	0.3017	0.0000	0.0000	0.0000	0.0263	0.3306	0.2316	0.0775	0.3835	0.8884
Reuse	0.7247	0.4507	0.4872	0.8656	-0.0043	-0.1905	-0.3131	0.9996	0.9445	1.0000	0.5941	-0.3037	-0.3717	-0.532	0.2702	0.0985
S=	0.0051	0.1222	0.0912	0.0001	0.9888	0.5330	0.2975	0.0000	0.0000	0.0000	0.0323	0.3131	0.2110	0.0610	0.3718	0.7486
Reid	0.6793	0.7516	0.7154	0.4594	0.0609	-0.2079	-0.1421	0.6139	0.6117	0.5941	1.0000	-0.0270	-0.0097	-0.444	0.5789	0.6160
S=	0.0106	0.0030	0.0060	0.1142	0.8433	0.4954	0.6431	0.0256	0.0263	0.0323	0.0000	0.9300	0.9749	0.1283	0.0382	0.0250
New	-0.1515	-0.3158	-0.1089	-0.3745	0.0932	0.3559	0.8641	-0.2989	-0.2933	-0.3037	-0.0270	1.0000	0.9641	0.5491	0.5592	0.1470
S=	0.6211	0.2932	0.7232	0.2074	0.7619	0.2327	0.0001	0.3211	0.3306	0.3131	0.9300	0.0000	0.0000	0.0519	0.0469	0.6317
Metbs	-0.1778	-0.3607	-0.1529	-0.5203	0.1119	0.3310	0.7879	-0.3652	0.3566	-0.3717	-0.0097	0.9641	1.0000	0.5750	0.5065	0.1829
S=	0.5611	0.2259	0.6179	0.0683	0.7158	0.2693	0.0014	0.2198	0.2316	0.2110	0.9749	0.0000	0.0000	0.0398	0.0773	0.5496
Priv	-0.3442	-0.3218	-0.3454	-0.6323	-0.1161	0.1471	0.4001	-0.5362	0.5063	0.5324	0.4442	0.5491	0.5750	1.0000	-0.3624	0.0186
S=	0.2495	0.2836	0.2477	0.0204	0.7056	0.6313	0.1755	0.0589	0.0775	0.0610	0.1283	0.0519	0.0398	0.0000	0.2236	0.9518

Table XI Correlations Between Static Class Metrics

	Number	Meth	Priv	Pub	Comnt	Desc	Eds
Number	1.0000	-0.1207	-0.2896	0.0503	-0.0802	-0.1969	-0.5000
S=	0.0000	0.6943	0.3371	0.8704	0.7943	0.5190	0.0819
Meth	-0.1207	1.0000	0.2140	0.8295	-0.1986	-0.0755	0.0377
S=	0.6943	0.0000	0.4826	0.0005	0.5154	0.8062	0.9026
Priv	-0.2896	0.2140	1.0000	-0.3679	0.1854	0.2183	0.6434
S=	0.3371	0.4826	0.0000	0.2161	0.5441	0.4736	0.0177
Pub	0.0503	0.8295	-0.3679	1.0000	-0.2948	-0.1970	-0.3318
S=	0.8704	0.0005	0.2161	0.0000	0.3280	0.5188	0.2679
Comnt	-0.0802	-0.1986	0.1854	-0.2948	1.0000	-0.2059	-0.0993
S=	0.7943	0.5154	0.5441	0.3280	0.0000	0.4997	0.7467
Desc	-0.1969	-0.0755	0.2183	-0.1970	-0.2059	1.0000	0.6991
S=	0.5190	0.8062	0.4736	0.5188	0.4997	0.0000	0.0078
Eds	-0.5000	0.0377	0.6434	-0.3318	-0.0993	0.6991	1.0000
S=	0.0819	0.9026	0.0177	0.2679	0.7467	0.0078	0.0000

Table XII Correlations Between Static Class Edition Metrics

	NumberDef	Ext	Undef	Class	Pub	Priv	Membs	Prereq	AllPrq	Depdnt	Subapp	AllSub	AllPnt	Visibl	Comnt	Descrp	Eds	
Number	1.0000	-0.2655	-0.1836	0.0000	-0.2702	-0.2657	0.0476	0.1493	-0.0624	0.7003	0.6152	0.4386	0.4774	0.5135	0.3022	-0.1765	0.0000	-0.2864
S=	0.0000	0.3805	0.5481	1.0000	0.3718	0.3802	0.8771	0.6262	0.8394	0.0077	0.0252	0.1337	0.0989	0.0727	0.3156	0.5639	1.0000	0.3427
Def	-0.2655	1.0000	0.5306	0.0000	0.9844	0.9999	-0.1169	-0.3181	0.6743	0.1985	-0.5788	-0.6445	-0.5373	-0.541	0.6422	-0.1682	0.0000	-0.3009
S=	0.3805	0.0000	0.0621	1.0000	0.0000	0.0000	0.7035	0.2895	0.0115	0.5156	0.0382	0.0174	0.0583	0.0559	0.0179	0.5827	1.0000	0.3177
Ext	-0.1836	0.5306	1.0000	0.0000	0.6714	0.5316	-0.1765	-0.2256	0.2139	0.1155	-0.4095	-0.5602	-0.4795	-0.485	0.7359	-0.1414	0.0000	-0.1418
S=	0.5481	0.0621	0.0000	1.0000	0.0120	0.0615	0.5639	0.4586	0.4827	0.7071	0.1647	0.046	0.0973	0.0929	0.0041	0.6448	1.0000	0.6440
Undef	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
S=	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Class	-0.2702	0.9844	0.6714	0.0000	1.0000	0.9845	-0.1389	-0.3249	0.6340	0.1975	-0.5910	-0.6797	-0.5692	-0.574	0.7142	-0.1764	0.0000	-0.2925
S=	0.3718	0.0000	0.0120	1.0000	0.0000	0.0000	0.6509	0.2787	0.0200	0.5176	0.0334	0.0106	0.0423	0.0401	0.0061	0.5642	1.0000	0.3320
Pub	-0.2657	0.9999	0.5316	0.0000	0.9845	1.0000	-0.1259	-0.3150	0.6776	0.2032	-0.5814	-0.6490	-0.5432	-0.547	0.6435	-0.1665	0.0000	-0.2985
S=	0.3802	0.0000	0.0615	1.0000	0.0000	0.0000	0.6817	0.2944	0.0109	0.5055	0.0371	0.0164	0.0550	0.0528	0.0176	0.5865	1.0000	0.3217
Priv	0.0476	-0.1169	-0.1765	0.0000	-0.1389	-0.1259	1.0000	-0.2957	-0.4358	-0.5366	0.3491	0.5651	0.7185	0.7057	0.212	-0.1623	0.0000	-0.2241
S=	0.8771	0.7035	0.5639	1.0000	0.6509	0.6817	0.0000	0.3266	0.1366	0.0586	0.2423	0.0441	0.0057	0.0070	0.4850	0.5962	1.0000	0.4615
Membs	0.1493	-0.3181	-0.2256	0.0000	-0.3249	-0.3150	-0.2957	1.0000	-0.3804	0.3123	0.0404	0.0837	-0.0157	0.0032	-0.131	-0.1082	0.0000	0.3975
S=	0.6262	0.2895	0.4586	1.0000	0.2787	0.2944	0.3266	0.0000	0.1998	0.2989	0.8957	0.7856	0.9592	0.9916	0.6674	0.7249	1.0000	0.1785
Prereq	-0.0624	0.6743	0.2139	0.0000	0.6340	0.6776	-0.4358	-0.3804	1.0000	0.3504	-0.3825	-0.5134	-0.5121	-0.506	0.4372	-0.1600	0.0000	-0.2427
S=	0.8394	0.0115	0.4827	1.0000	0.0200	0.0109	0.1366	0.1998	0.0000	0.2404	0.1970	0.0727	0.0736	0.0772	0.1352	0.6015	1.0000	0.4242
AllPrq	0.7003	0.1985	0.1155	0.0000	0.1975	0.2032	-0.5366	0.3123	0.3504	1.0000	0.1117	-0.1949	-0.2009	-0.166	0.6200	-0.1632	0.0000	-0.3223
S=	0.0077	0.5156	0.7071	1.0000	0.5176	0.5055	0.0586	0.2989	0.2404	0.0000	0.7162	0.5233	0.5104	0.5867	0.0238	0.5942	1.0000	0.2828
Depdnt	0.6152	-0.5788	-0.4095	0.0000	-0.5910	-0.5814	0.3491	0.0404	-0.3825	0.1117	1.0000	0.6146	0.6659	0.6769	-0.240	-0.1880	0.0000	-0.1461
S=	0.0252	0.0382	0.1647	1.0000	0.0334	0.0371	0.2423	0.8957	0.1970	0.7162	0.0000	0.0254	0.0130	0.0110	0.4290	0.5385	1.0000	0.6339
Subapp	0.4386	-0.6445	-0.5602	0.0000	-0.6797	-0.6490	0.5651	0.0837	-0.5134	-0.1949	0.6146	1.0000	0.9690	0.9674	-0.369	0.1347	0.0000	0.0212
S=	0.1337	0.0174	0.0464	1.0000	0.0106	0.0164	0.0441	0.7856	0.0727	0.5233	0.0254	0.0000	0.0000	0.0000	0.2140	0.6607	1.0000	0.9450
AllSub	0.4774	-0.5373	-0.4795	0.0000	-0.5692	-0.5432	0.7185	-0.0157	-0.5121	-0.2009	0.6659	0.9690	1.0000	0.9981	-0.277	0.0120	0.0000	-0.0953
S=	0.0989	0.1335	0.0973	1.0000	0.0423	0.0550	0.0057	0.9592	0.0736	0.5104	0.0130	0.0000	0.0000	0.0000	0.3578	0.9689	1.0000	0.7568
AllPnt	0.5135	-0.5445	-0.4851	0.0000	-0.5742	-0.5475	0.7057	0.0032	-0.5067	-0.1665	0.6769	0.9674	0.9981	1.0000	-0.272	0.0073	0.0000	-0.0995
S=	0.0727	0.0559	0.0929	1.0000	0.0401	0.0528	0.0070	0.9916	0.0772	0.5867	0.0110	0.0000	0.0000	0.0000	0.3685	0.9809	1.0000	0.7463
Visibl	0.3022	0.6422	0.7359	0.0000	0.7142	0.6435	-0.2129	-0.1319	0.4372	0.6200	-0.2403	-0.3695	-0.2779	-0.272	1.0000	-0.1805	0.0000	-0.4513
S=	0.3156	0.0179	0.0041	1.0000	0.0061	0.0176	0.4850	0.6674	0.1352	0.0238	0.4290	0.2140	0.3578	0.3685	0.0000	0.5550	1.0000	0.1216
Comnt	-0.1765	-0.1682	-0.1414	0.0000	-0.1764	-0.1665	-0.1623	-0.1082	-0.1600	-0.1632	-0.1880	0.1347	0.0120	0.0073	-0.180	1.0000	0.0000	-0.0121
S=	0.5639	0.5827	0.6448	1.0000	0.5642	0.5865	0.5962	0.7249	0.6015	0.5942	0.5385	0.6607	0.9689	0.9809	0.5550	0.0000	1.0000	0.9687
Descrp	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
S=	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Eds	-0.2864	-0.3009	-0.1418	0.0000	-0.2925	-0.2985	-0.2241	0.3975	0.2427	0.3223	-0.1461	0.0212	0.0953	0.099	-0.451	0.0121	0.0000	1.0000

Table XIII Correlations Between Static Application Metrics

END

2/3/1 2/92

FIN