Object-Oriented Software Metrics

by

Xiaowei Liu

A thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfilment of the Requirements

for the degree of

MASTER OF SCIENCE

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba, Canada

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-41734-4

Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

OBJECT-ORIENTED SOFTWARE METRICS

BY

XIAOWEI LIU

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

Quality assurance is one of the important non-functional software requirements which many software products fail to satisfy. Current software market is driven mostly by urgency and competition. This poses a serious problem to software quality assurance, customer satisfaction and reliability of the software products. One of the methods to ensure software quality is a metrics-based approach. Software metrics have been used to quantitatively evaluate software products.

Software metrics proposed and used for procedural paradigm have been found inadequate for object-oriented software products, mainly because of the distinguishing features of the object-oriented paradigm such as inheritance and polymorphism. In this thesis, we propose a new set of metrics for object-oriented software; this set is targeted towards evaluating the efforts required for testing object-oriented programs. The new metrics have been analytically evaluated using widely known properties for software metrics. They have also been critically compared with other metrics published in the literature. Experimental evaluation of these metrics have also been presented in the thesis.

Acknowledgements

I wish to express my sincere thanks and gratitude to my supervisor, Dr. Kasilingam Periyasamy, for his guidance and advise throughout the course of this research work. I also wish to thank Dr. V.S.Alagar, Concordia University, for his invaluable discussions about the research.

I wish to thank my thesis examining committee members Dr. D.Scuse and Dr. Padmanabhan (Department of Applied Mathematics), who gave me suggestions on improving this work, and Dr. H.Cameron for chairing my thesis defense.

I also wish to thank my parents, my sister, and my friends who gave my support throughout the years of my study in the University of Manitoba.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past couple of decades, the speed of computer hardware development has far exceeded software productivity development [40]. As computers are used in all sorts of everyday activities, the demand for sophisticated and flexible software also increases. Currently, software market is driven by urgent market needs which drive software developers to produce software without delay in delivery. Such urgency poses a lot of problems in producing quality software. In addition, software maintenance becomes extremely difficult. In most cases, the delivered product is not reliable. Hence, quality assurance, customer satisfaction and reliable products are immediate needs of current software industries.

Software quality can be assured in several ways. Quality control can be enforced at each stage of the software development process. Alternatively, one can evaluate the quality of a software product after it is implemented. The former case generally uses mathematically-based approaches such as formal specifications, while the latter case

uses more quantified approaches such as testing and metrics-based approaches. This thesis is a contribution to the latter case of quality assurance. Some researchers call the quantitative approach as software measurement.

Measurement is used in everyday life. For example, we calculate distances of different routes on a trip; such measurement helps in predicting the reachability of certain destinations on time. As another example, the price of a commodity in a grocery store denotes teh value of the commodity. For our purposes in this thesis, the definition of measurement given in [21] is used, which is stated as follows:

> *Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them using a set of clearly defined rules.*

In some cases, measurement of certain entities become almost impossible because it is hard to come up with clearly defined rules to describe certain entities. For example, the measurement to evaluate human intelligence or to describe the maturity of certain discipline is hard to achieve. Software measurement poses a similar problem in the sense that the rules for describing software behavior are not well established. Part of this problem is due to lack of foundations in defining the measurement. As a result, software measurement defined by one group is not appreciated or not even approved by another group. Nevertheless, software measurement gives an excellent opportunity for software developers to evaluate their own products, to convince themselves of the outcome of the software development process and to estimate or predict the efforts required for a future product. For example, a software manager may wish to estimate the cost and delivery time required for a future product based on a

quantitative evaluation of a recently completed product. A customer of a software product may expect a quantitative measurement on the size and some characteristics of the product.

In order to evaluate a measurement, a set of expected or estimated results must be made available before evaluation. By this way, the actual measured values can be compared with the estimated values to ensure satisfaction. Such a measurement combined with expected results is called "metrics" [22].

Research on software metrics is an ongoing process for several decades. They were primarily used for cost estimation in the 70's, but quickly changed towards productivity measurement, evaluating reliability models and very recently to capability maturity measurement [40]. Software metrics is used by all people involved in the development processes including customers and managers on one hand, and project leaders, designers and programmers on the other hand. Depending on who uses the metrics, a different set of definitions is used for describing the same software product. As an example, a software manager may be interested in lines of code[1], while a project manager may be interested in estimating the number of hours required to develop the same number of lines to calculate the productivity of programmers.

Software metrics are helpful in several activities of the software development life cycle, and therefore contribute to the overall objective of software quality. The use of software metrics has been fully recognized by the software engineering community and has been included in several standards, such as *ANSI/IEEE p-1061/D21* and *ISO/IEC 9000* [2]. There are also successful industrial examples which applied software metrics to their software products since 1980s; these showed some important benefits. The

[1] *Lines of code* (LOC) is one of the earliest measurement techniques used in software industries.

enterprises which applied metrics include Siemens companies in Europe and in the United States, Data Logic Limited in the United Kingdom, and ETNOTEAM Spa in Italy [40]. Interim reports from these companies show that successful application of software metrics has reduced development cost by 25% per year and further has shortened development schedules by 10% per year.

Yu and Lamb [45] made a survey of software metrics ranged from 1978 to 1991. For example, Chapin's Q metric concentrates on the role of input and output data of modules in a structured design. McCabe and Butler applied the cyclomatic complexity metric to structured design and suggested that portions of a procedure's flow graph that do not involve calls to other procedures do not contribute to design complexity. Henry and Kafura's information flow metrics indicate that the prime factor determining structural complexity is the connectivity of a procedure to its environment. The more complex procedures in a system are those through which large amounts of information flow.

Weyuker proposed a set of properties to serve as a basis for the evaluation of metrics [44], which was one of the first attempts to assess metrics. These properties are used by Chidamber and Kemerer from MIT, and Kim *et al.* from Osada University, Japan. Both groups and Abreu *et al.* from Portugal, have introduced a set of metrics for object-oriented software.

Despite the fact that metrics have been researched and used in industries for years, there is no standard or generalized metrics definition which is applicable to every software product. Each software project somehow tailors the information and so research continues to play a major role in software metrics area. Recently, research on object-oriented software metrics has received more attention, due to the popularity of

the object-oriented approach in software development. Henderson-Sellers listed the key differences between object-oriented and traditional (procedural) software development processes [29]. Accordingly, object-oriented software

1. provides more interactive/recursive life cycle, which imposes the need for a framework for methodological steps and activities;

2. supports reuse;

3. provides a better system structuring such as subsystem, classes and methods;

4. supports peer-to-peer message passing rather than hierarchical decomposition of control flow; and

5. greatly advocates the use of inheritance structures.

In addition to the above differences, "cohesion" and "coupling" also pose different problems in the object-oriented approach.

Cohesion has long been recognized as a highly desirable property in software components, because it measures separation of responsibilities, independence of components, and control of complexity. Typically, cohesion refers to the collaboration among the constituents of a software product. It reflects how well complexity has been controlled in software design and the integrity of functionality in a module; it has a significant effect on a component's reusability, understandability, effectiveness, and adaptability [6]. In a procedural language, a function can be a component, and the statements in the function can be the parts. In object-oriented paradigm, a class can be a component, and the attributes and methods become its parts. The cohesion implicit in

an *object* is very high since all operations to modify or inspect attributes of the object will be provided by that object. The inheritance property in the object-oriented approach increases the dependency of subclasses on the superclass(es), and hence reduces cohesion.

Coupling is a measure of interdependence of components. High coupling implies strong interconnections between program units, while loose coupling implies independence. Object-oriented designs reflect the real world as independent objects, which leads to loose coupling, if designed well. Once again, inheritance causes tight coupling among classes. Because inheritance and polymorphism are heavily used in object-oriented paradigm, a study of metrics for object-oriented software becomes an important research project.

In this thesis, a new set of object-oriented software metrics is introduced. This set of metrics is targeted towards estimating the efforts required to test object-oriented programs. The metrics do not indicate actual number of test cases required for a given program; rather, it gives a subjective but quantitative estimate of the efforts required for testing. The results are justified based on analytical approaches, rather than compared. Part of the problem for this decision is due to the lack of tools available at the time of conducting this research. Accordingly, the work presented in this thesis is analytically compared with related work by other researchers in the same area.

The rest of the thesis is organized as follows: Chapter 2 provides a quick overview of research on metrics for traditional (procedural) software and research done on object-oriented software metrics. The new metrics for testing efforts are introduced in Chapter 3. For a quick review, the metrics in Chapter 3 are also summarized in

tables at the end of the thesis. Chapter 4 describes the analytical comparison of the new metrics with others published in the literature. In Chapter 5, an empirical analysis of the new metrics is given, based on a simple program. The thesis concludes in Chapter 6 with comments on future work in this direction.

Chapter 2

Previous Research in Software Metrics

In this thesis, traditional software approach refers to the software that use functional decomposition and data flow development methods. These methods commence by considering the system's behavior and/or data separated [41].

2.1 Overview of Traditional Software Metrics

Software metrics is not a new phenomenon. The original work of applying quantitative methods to software development was established in the 1970s , which include trends such as code complexity measures, software project cost estimation, software quality assurance and software development process [40]. These four primary technology trends have evolved into the metrics used today.

In addition to Yu and Lamb's survey (mentioned in Chapter 1), many other scholars have also contributed to this area. For example, a variety of approaches have been proposed for evolving software metrics. One of them is the Goal/Question/Metrics (GQM) paradigm developed by Basili and Rombach [8].

GQM approach indicates a basic fact of measurement: *measurement activities must have clear objectives* [20]. That is, before any measurement can be undertaken, one should know exactly which entities are the subjects of interest, and focus only on those attributes of the chosen entities that are significant.

GQM is intended as a rigorous method for evolving metric models. It represents systematic approach for setting project goals and defining them in an operational and tractable way. Goals are to be refined into a set of quantifiable questions that specify metrics. GQM can also be explained as follows: every software measurement activity should commence with the identification of a goal, which should then be decomposed into several questions, and finally decomposed into metrics.

Fenton has done research in the area of metrics validation [20]. He stated that *software measurement, like measurement in an other discipline, must adhere to the science of measurement if it is to gain widespread acceptance and validity.* He used scientific measurement to highlight both weaknesses and strengths of software metrics.

Data collection is as important as the definitions of software metrics, since *software measurement is only as good as the data that is collected and analyzed* [21]. Basili and Weiss introduced a methodology [7] for collecting valid data for evaluating software. They claimed that in order to ensure accuracy of the data, validation should be performed concurrently with software development and data collection.

2.2 Overview of Object-Oriented Software Metrics

The popularity of the object-oriented approach to software development has created an intensive interest and more challenging problems to metrics development and metrics evaluation. The object-oriented approach centers around modeling the real world in terms of objects; in contrast, the traditional procedural approaches emphasize a function-oriented view which separates data and procedures [16]. In object-oriented approach, each kind of data and related operations are collected into a single system entity. Therefore, even though the traditional software metrics are modified to assess object-oriented software system, they are inadequate to cover all the new and unique aspects of object-orientation. Thus, new metrics which reflect the characteristics of object-oriented paradigm must be defined.

2.2.1 Characteristics of Object-Oriented Design Method

The term "object-oriented" means that software is organized as a collection of discrete objects that encapsulate both data structure and behavior [42]. Generally, the characteristics required by an object-oriented approach include four aspects: identity, classification, polymorphism, and inheritance.

Identity means that data is quantified into discrete, distinguishable entities called *objects*. Each object has its own inherent identity; that is, two objects are distinct as long as their identities are different, even if all their attribute values (such as name) are identical.

Classification means that objects with the same data structure and behavior are grouped into one *class.* Each class describes a possibly infinite set of individual objects. Each object is an instance of its class.

Polymorphism means the same operation may be implemented differently on different classes. The specific implementation of an operation by a certain class is called a *method.* An object-oriented operator that is polymorphic, may have more than one method implementing it.

Inheritance is the mechanism that allows sharing of attributes and operations among classes based on a hierarchical relationship. Using inheritance, a class can be refined into subclasses. Each subclass inherits all the properties of its superclass and may add its own unique properties. The properties of the superclass need not be repeated in each subclass. The ability to distill common properties of several classes into a common superclass and to inherit the properties from the superclass can greatly reduce repetition within designs and programs. Inheritance also leads to code reuse in an efficient way.

2.2.2 Previous Research on Object-Oriented Software Metrics

Research on object-oriented software metrics has been done in several aspects, such as metrics evolving model, defining new metrics, validating OO metrics, etc.

Gowda and Winslow [23] proposed an approach for deriving object-oriented metrics. This approach centered on dividing a software development procedure into several

phases, and defining different categories of metrics based on the emphasizing points of each phase. All the metrics are to be categorized in terms of subsystems, collaborations at subsystem levels, class levels, message flow at class/object levels, and etc. For example, in the object modeling stage in Object Modeling Technique (OMT), class metrics and subsystem metrics about classes, subclasses, attributes and methods will be suitable. Whereas in dynamic modeling stage, system metrics which measures transitions, data flows, etc, will be useful. The previously introduced GQM model can also be used in evolving object-oriented metrics.

Yet, major work in this area is centering around defining new object-oriented software metrics and analyzing these metrics. Chidamber and Kemerer defined a metrics suite for object-oriented design [15]. These metrics were later refined the definitions and analytically evaluated against Weyuker's measurement principles [16]. This metric suite is an important contribution and is one of the widely-used metric suites. Basili et al[9] assessed these metrics by collecting data on the development of eight medium-sized information management systems based on identical requirements. Based on empirical and quantitative analysis, the advantages and drawbacks of these object-oriented metrics are discussed. His conclusion is that except for the metric *Lack of Cohesion in Methods* (LOCM), other metrics in this suite appear to be useful in predicting class fault-proneness during the early phases of the life cycle.

Churcher and Shepperd [17] indicated that it was premature to apply Chidamber and Kemerer's metrics, since there remained uncertainty about the precise definitions of many of the quantities to be observed and their impact upon subsequent indirect metrics. For example, the metric *Weighted Methods per Class* (WMC) counts only methods with distinct names, while methods with same name but different parameters

and return type, such as multiple-defined constructors, might be ignored. The set of inherited methods poses the same question. Therefore, it is very important to precisely specify the mapping from a language-independent set of metrics to specific programming languages and sets of observations. The usefulness of any set of metrics will be fully achieved only when their application to specific languages is clearly specified, as mentioned by Basili *et al.* [9].

Graham [24] indicated that it is not clear whether the metric *Number of Children* (NOC) allows for dynamic classification schemes. It is obvious that NOC will change at runtime. He suggested the maximum, minimum, mean, and mode values for NOC should be collected in such circumstances. Graham also found that there were some inconsistencies in the metric LCOM. This metric measures the non-overlapping of sets of instance variables used by the methods of a class. It is the percentage of methods that do not access an attribute, averaged over all attributes. Graham showed that LCOM appears to increase with cohesion where it is expected to decrease. Therefore, he gave a new definition for LCOM that overcomes the above problem. The new definition provides a metric that decreases as cohesion increases and gives values that discriminate classes of intuitively different cohesion.

Gowda and Winslow [23] analyzed Chidamber's metrics and gave the conclusion that those metrics basically address the class structures, their attributes and methods, but do not assess all system design and resulting, final system, which is often needed by managers.

Abreu *et al.* proposed the MOOD (Metrics for Object Oriented Design) set of metrics [1, 2]. The goal of developing this set of metrics centered on how to improve the object-oriented design process to achieve better maintainability. These metrics allow

the use of the mechanisms of the object-oriented paradigm to be evaluated. Each of these metrics quantifies a distinct feature: encapsulation, inheritance, polymorphism and message-passing.

Kim *et al.*'s metrics [31, 32] are proposed for computing the complexity of an object-oriented program. These metrics are used to examine program complexity from three dimensional viewpoints: syntax dimensional, inheritance dimension, and interaction dimension.

We will further discuss the above three sets of metrics in Chapter 4. Other metrics sets include Ebert and Morschel's metrics for quality analysis and improvement of object-oriented software implemented using SmallTalk [18]. Hitz and Montazeri [30] proposed three metrics. Their job focused on selecting appropriate attributes for object-oriented software measurement. Martin [37] described a set of metrics that can be used to measure the quality of an object-oriented design in terms of interdependence between the subsystems of that design. Milankovic-Atkinson and Georgiadou's metrics [38] can be used to assess the design rules to improve the quality of object-oriented software, primarily to enable better reuse.

There also have been a number of papers addressing the issue of validating software metrics. Weyuker [44] discussed complexity metrics and proposed nine properties which she believed any syntactic complexity measure should fulfill. The list is a widely known formal analytical approach. Therefore, Chidamber and Kemerer [16] and Kim *et al.* [32] had referred to Weyuker's properties to evaluate their object-oriented metrics.

However, Weyuker's properties are not without criticism. For example, Cherniavsky

and Smith [14] suggested these properties should be used carefully since the properties may only give necessary, but not sufficient conditions for good complexity metrics. Fenton [21] suggested that Weyuker's properties attempt to characterize incompatible views of complexity. This discussion shows that new measures are being justified according to disputed criteria. Based on this thought, Kitchenham, Pfleeger and Fenton [33] proposed a validation framework which, they believe, can overcome the above problems. Their framework includes a structural model which describes the objects involved in measurement and their relationships, various models and protocols used to define the measurement elements, and the theoretical and empirical validation issues which assure whether or not the measurements are appropriate. Both Weyuker's properties and Kitchenham's framework are suitable not only for object-oriented paradigm, but for functional pradigm as well.

Kim *et al.* proposed a framework for analyzing object-oriented metrics [31]. The framework evaluates the scope or the capability of complexity metrics for object-oriented programs [31, 32]. Bansiya and Davis [5] have developed an automated tool that supports object-oriented metrics, called QMOOD++. Unfortunately, this tool is strongly related to a certain set of metrics, developed by Etzkorn, Bansiya and Davis [19].

Chapter 3

Liu's Object-Oriented Software Metrics

This chapter describes a new set of metrics; this set of metrics is targeted towards evaluating the testing efforts required for object-oriented programs. The metrics indicate a quantitative measurement of testing efforts required, and do not indicate the actual number of test cases. Further work in this direction will refine these metrics to evaluate the actual number of these cases.

For the sake of future usage, we name the new metrics set proposed in this chapter as "Liu's metrics set".

This set of Liu's metrics is divided into three categories:

1. Metrics based on inheritance hierarchy;
2. Metrics based on polymorphism;

3. Metrics based on interactions among objects.

According to Harrold and McGregor [28], these categories differentiate object-oriented programs from procedural programs; at the same time, these categories make testing methods for procedural programs become inadequate for object-oriented programs. The definitions for the metrics introduced in this chapter are based on the concepts of object-orientation and thus can be applied to programs written in any object-oriented language. To facilitate evaluation of these metrics, a language specific metric corresponding to each general metric is also given in this chapter. C++ has been chosen for deriving these language-specific metrics.

3.1 Metrics Based on Inheritance Hierarchy

Inheritance is a mechanism which implements generalization/specialization relationship among classes in an object-oriented program. The specialization relationship describes how a subclass can be made as a specialization of a superclass by adding more features to, or by redefining some of, the inherited features of the superclass. Since subclass objects can be substituted for superclass objects due to polymorphism, testing a superclass requires testing of all its subclasses as well. The inheritance mechanism implements the specialization relationship in three ways:

- by adding more features[1] to the inherited features
- by modifying or redefining some of the inherited features

[1]The term "feature" refers to an attribute or a method; it is borrowed from the object-oriented language Eiffel.

- by deleting or restricting some of the inherited features

Though the third category is not common in usage, it is included here for completion. Following these discussions, the new metrics are defined as follows:

1. IHC: metrics based on changing features;

2. IHA: metrics based on adding new features;

3. IHD: metrics based on deleting features.

For convenience, we use the notation "⟨*classname*⟩*_tree*" to indicate the tree/subtree rooted at "⟨*classname*⟩" in the inheritance hierarchy. For example, in Figure 3.1, the box in dotted lines refers to "*A_tree*".

We further attach a level number to each node in the inheritance hierarchy. These level numbers are used in the metrics definitions. The root node of the hierarchy has the number *0*, with its immediate descendents having level number *1*; subsequent levels will have their numbers uniformly increased.

3.1.1 Metrics Based on Changing Features (IHC)

The term "changing feature" refers to an attribute or a method in the superclass whose name is retained in the subclass, but its type (in case of attribute) or definition (in case of methods) changed in the subclass. Generally, object-oriented programming languages do not permit changing a type of an attribute in the subclass. Therefore,

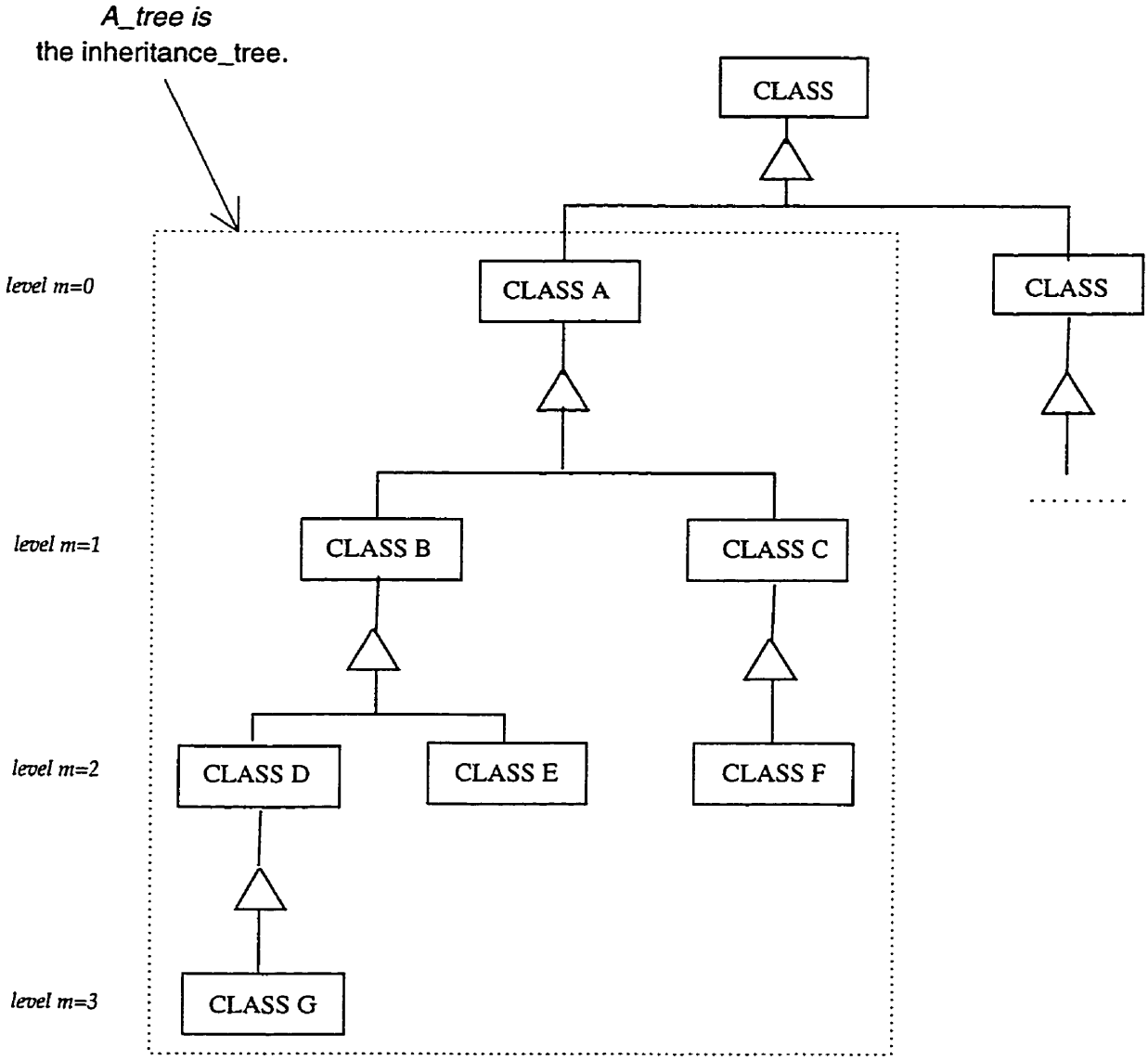

Figure 3.1: An Inheritance Hierarchy

we consider only inherited methods in this section. The methods with same name but different signature[2] will be considered as being changed.

The metric IHC represents the efforts required to test a superclass due to redefinition of its methods in the subclasses. As the inheritance hierarchy becomes higher and higher, it becomes necessary to test all subclasses for a given superclass. Consequently, the redefinitions of a given method in a superclass increase the testing efforts.

Based on these discussions, the IHC metric is defined as follows:

$$\begin{aligned}&\text{IHC}(\langle classname\rangle\_tree \text{ at level } m)\\ &\quad = \text{IHC}(all\_subclasses \text{ at level } (m+1)) +\\ &\quad\quad \textstyle\sum_{i=1}^{n} \text{IHC}(\langle class_i\rangle\_tree \text{ at level } (m+1))\end{aligned} \tag{e-1}$$

where n refers to the number of immediate subclasses of $\langle classname\rangle$, and

$$\begin{aligned}&\text{IHC}(all\_subclass \text{ at level } (m+1))\\ &= \frac{\begin{array}{c}\text{number of changed methods}\\ \text{in subclasses at level } (m+1)\end{array}}{\begin{array}{c}\text{number of all the methods}\\ \text{in subclasses at level } (m+1)\end{array}} \times (\text{Method Percentage})\end{aligned} \tag{e-2}$$

If there is no subclass for $\langle classname\rangle$, then $\text{IHC}(\langle classname\rangle\_tree) = 0$.

As an example, consider the inheritance hierarchy given in Figure 3.1. Using the above metrics definition, we observe that

$$\text{IHC}(A\_tree) = [\text{IHC}(A\text{'s subclasses at level } 1)] + [\text{IHC}(B\_tree)+\text{IHC}(C\_tree)]$$

[2]Each method has a unique signature, determined by combining its identifier, parameters, and return type [17].

We can also deduce, for example, that IHC(*G_tree*) = 0 and IHC(*E_tree*) = 0, since "*G_tree*" and "*E_tree*" do not have subclasses.

Let us now elaborate the calculation of IHC(*all_subclasses*), since it is the core of the IHC metrics definition. The calculation of IHC(⟨*all_subclasses*⟩) uses the following terms:

MI(*subclass_i*): number of methods in *subclass_i*, which are inherited but not modified.

MC(*subclass_i*): number of methods in *subclass_i*, which are inherited and re-defined in *subclass_i*.

MA(*subclass_i*): number of newly introduced attributes in *subclass_i*.

MD(*subclass_i*): number of all methods in *subclass_i*.

$$MD = MI + MA + MC$$

AD(*subclass_i*): number of all attributes in *subclass_i*.

$$AD = AI + AA$$

n: number of immediate subclasses for a given class named ⟨*classname*⟩.

MP(*all_subclasses*): Method Percentage for all subclasses in ⟨*classname*⟩*_tree*.

$$MP(all\_subclasses) = \left(\frac{\sum_{i=1}^{n} MD(subclass\_i)}{\sum_{i=1}^{n} AD(subclass\_i) + \sum_{i=1}^{n} MD(subclass\_i)} \times 100 \right) \%$$

Now, IHC(*all_subclasses* at level (*m+1*)) is defined as:

IHC(*all_subclasses* at level $(m+1)$)

$$= \frac{\sum_{i=1}^{n} \text{MC}(subclass\_i)}{\sum_{i=1}^{n} \text{MD}(subclass\_i)} \times \text{MP}(\langle all\_subclasses \rangle) \quad \text{(e-3)}$$

The possible values of IHC(*all_subclasses* at level (*m+1*)) can be computed as follows:

IHC(*all_subcalsses* at level (*m+1*))

$$= \frac{\sum_{i=1}^{n} \text{MC}}{\sum_{i=1}^{n} \text{MD}} \times \left(\frac{\sum_{i=1}^{n} \text{MD}}{\sum_{i=1}^{n} \text{AD} + \sum_{i=1}^{n} \text{MD}} \times 100 \right) \%$$

$$= \left(\frac{\sum_{i=1}^{n} \text{MC}}{\sum_{i=1}^{n} \text{AD} + \sum_{i=1}^{n} \text{MD}} \times 100 \right) \% \quad \text{(e-4)}$$

$\sum$MC is the sum of changed methods in the subclasses, while $\sum$MD is the sum of all the methods in the subclasses. The changed methods are parts of total methods, since MD is the sum of MI, MA, and MC. Therefore $\sum_{i=1}^{n}$ MC can be only less than or equal to $\sum_{i=1}^{n}$ MD. Thus, the value of IHC(*all_subcalsses* at level (*m+1*)) will be between 0 and 1. This is a significant result as compared to similar metrics published in the literature.

3.1.2 Metrics Based on Adding Features (IHA)

The "adding features" refers to a set of attributes and/or methods which are newly defined in the subclasses. Each attributes defined in subclasses can be treated as new attribute. For a new method, both its name and signature should be different from

those defined in superclass.

The metric IHA indicates the efforts needed to test a superclass due to the increasing of new attributes and methods in the subclasses. The IHA metric is defined below:

$$\begin{aligned}&\text{IHA}(\langle classname\rangle\_tree \text{ at level } m)\\&\quad = \text{IHA}(all\_subclasses \text{ at level } (m+1)) \ +\\&\quad\ \sum \text{IHA}(\langle subclass\rangle\_tree \text{ at level } (m+1))\end{aligned} \tag{e-5}$$

where n is the number of immediate subclasses of $\langle classname\rangle$, and

$$\begin{aligned}&\text{IHA}(all\_subclasses \text{ at level } (m+1))\\&= \frac{\begin{array}{l}\text{number of added attributes}\\ \text{in subclasses at level } (m+1)\end{array}}{\begin{array}{l}\text{number of all the attributes}\\ \text{in subclasses at level } (m+1)\end{array}} \times (\text{Attribute Percentage})\\&\quad + \frac{\begin{array}{l}\text{number of added methods}\\ \text{in subclasses at level } (m+1)\end{array}}{\begin{array}{l}\text{number of all the methods}\\ \text{in subclasses at level } (m+1)\end{array}} \times (\text{Method Percentage})\end{aligned} \tag{e-6}$$

As in the previous subsection, IHA(*all_subclasses*) is the core of IHA metric definition. We will introduce some more terms before elaborating the calculation of IHA(*all_subclasses*).

AI(*subclass_i*): number of attributes in *subclass_i*, which are inherited from *superclass*.

AA(*subclass_i*): number of newly introduced attributes in *subclass_i*.

AP(*all_subclasses*): Attribute Percentage for all subclasses in $\langle classname\rangle\_tree$. AP(*all_subclasses*)

$$= \left(\frac{\sum_{i=1}^{n} \mathrm{AD}(all\_subclasses)}{\sum_{i=1}^{n} \mathrm{AD}(all\_subclasses) + \sum_{i=1}^{n} \mathrm{MD}(all\_subclasses)} \times 100 \right) \%.$$

Now IHA(*all_subclasses*) can be defined as:

$$\begin{aligned} &\mathrm{IHA}\ (all\_subclass \text{ at level } (m+1)) \\ &= \frac{\sum_{i=1}^{n} \mathrm{AA}(subclass\_i)}{\sum_{i=1}^{n} \mathrm{AD}(subclass\_i)} \times \mathrm{AP} + \frac{\sum_{i=1}^{n} \mathrm{MA}(subclass\_i)}{\sum_{i=1}^{n} \mathrm{MD}(subclass\_i)} \times \mathrm{MP} \end{aligned} \quad \text{(e-7)}$$

The possible values of IHA(*all_subclasse* at level (*m+1*)) will be calculated as follows:

$$\begin{aligned} &\mathrm{IHA}(all\_subclasses \text{ at level } (m+1)) \\ &= \frac{\sum_{i=1}^{n} \mathrm{AA}}{\sum_{i=1}^{n} \mathrm{AD}} \times \left(\frac{\sum_{i=1}^{n} \mathrm{AD}}{\sum_{i=1}^{n} \mathrm{AD} + \sum_{i=1}^{n} \mathrm{MD}} \times 100 \right) \% \\ &\quad + \frac{\sum_{i=1}^{n} \mathrm{MA}}{\sum_{i=1}^{n} \mathrm{MD}} \times \left(\frac{\sum_{i=1}^{n} \mathrm{MD}}{\sum_{i=1}^{n} \mathrm{AD} + \sum_{i=1}^{n} \mathrm{MD}} \times 100 \right) \% \\ &= \left(\frac{\sum_{i=1}^{n} \mathrm{AA} + \sum_{i=1}^{n} \mathrm{MA}}{\sum_{i=1}^{n} \mathrm{AD} + \sum_{i=1}^{n} \mathrm{MD}} \times 100 \right) \% \end{aligned} \quad \text{(e-8)}$$

$\sum$AA is the sum of newly introduced attributes in subclasses. It is part of all the attributes in subclass, which is $\sum$AD (AD=AI+AA). $\sum$MA is the sum of newly introduced methods in subclass. It is part of all the methods in subclass, which is

$\sum$MD (MD=MI+MA+MC). Therefore, $\sum_{i=1}^{k}$ AA+$\sum_{i=1}^{n}$ MA can be only less than or equal to $\sum_{i=1}^{n}$ AD + $\sum_{i=1}^{n}$ MD. Thus, the value of IHA(*all_subclasses* at level (*m+1*)) will be between 0 and 1.

3.1.3 Metrics Based on Deleting Features (IHD)

The "deleted features" refers to the attributes and methods which have been deleted (if permitted by the object-oriented programming language used in implementing the software product); otherwise it refers to those which will not be used in the subclasses. The IHD metric represents the percentage of deleting features of the inheritance tree.

The IHD metric is defined as follows:

$$\begin{aligned}&\text{IHD}(\langle classname\rangle\_tree \text{ at level } m)\\&\quad = \text{IHD}(all\_subclasses \text{ at level } (m+1))\\&\qquad + \sum \text{IHD}(\langle subclass\_i\rangle\_tree \text{ at level}(m+1))\end{aligned} \tag{e-9}$$

The core part of IHD metric is IHD(*all_subclasses*). It can be calculated as follows:

IHD(*all_subclasses* at level (*m+1*))

$$\begin{aligned}&= \frac{\begin{array}{l}\text{number of attributes which are deleted or}\\ \text{not used in subclasses at level } (m+1)\end{array}}{\begin{array}{l}\text{number of attributes in subclasses}\\ \text{at level } (m+1)\end{array}} \times (\text{Attribute Percentage})\\&\quad + \frac{\begin{array}{l}\text{number of methods which are deleted or}\\ \text{not used in subclasses at level } (m+1)\end{array}}{\begin{array}{l}\text{total number of methods in subclasses}\\ \text{at level } (m+1)\end{array}} \times (\text{Method Percentage})\end{aligned} \tag{e-10}$$

C++ does not provide any direct mechanism by which a subclass can delete/restrict any inherited feature. But there are some access rules about the inheritance mecha-

nism, which will cause some attributes or methods of the superclass not be used in subclasses directly. The following are two of them:

1. Private members from the superclass are not accessible by the subclass.
2. Private members inherited from the superclass private section can be accessed in the subclass only by means of the public methods of the superclass.

Based on the above discussion, IHD(*all_subclasses* at level (*m*+1)) can be defined as:

$$\begin{aligned}&\text{IHD}(\textit{all\_subclasses} \text{ at level}(m+1))\\ &= \frac{\sum_{i=1}^{n} \text{ANU}(\textit{subclass\_i})}{\sum_{i=1}^{n} \text{AD}(\textit{subclass\_i})} \times \text{AP} + \frac{\sum_{i=1}^{n} \text{MNU}(\textit{subclass\_i})}{\sum_{i=1}^{n} \text{MD}(\textit{subclass\_i})} \times \text{MP} \qquad \text{(e-11)}\end{aligned}$$

Here are some terms used in this metric:

ANU(*subclss_i*): number of attributes inherited by *subclass_i*, but not used.

MNU(*subclass_i*): number of methods inherited by *subclass_i*, but not used.

We will futher work on (e-11), and get:

$$\begin{aligned}&\text{IHD}(\textit{all\_subclass} \text{ at level } (m+1))\\ &= \frac{\sum_{i=1}^{n} \text{ANU}}{\sum_{i=1}^{n}} \times \left(\frac{\sum_{i=1}^{n} \text{AD}}{\sum_{i=1}^{n} \text{AD} + \sum_{i=1}^{n} \text{MD}} \times 100 \right) \% \\ &+ \frac{\sum_{i=1}^{n} \text{MNU}}{\sum_{i=1}^{n}} \times \left(\frac{\sum_{i=1}^{n} \text{MD}}{\sum_{i=1}^{n} \text{AD} + \sum_{i=1}^{n} \text{MD}} \times 100 \right) \%\end{aligned}$$

$$= \left(\frac{\sum_{i=1}^{n} \text{ANU} + \sum_{i=1}^{n} \text{MNU}}{\sum_{i=1}^{n} \text{AD} + \sum_{i=1}^{n} \text{MD}} \times 100 \right) \% \qquad \text{(e-12)}$$

ANU and MNU are the attributes and methods inherited but never used. They are part of AI and MI, thus they are parts of AD and MD. Therefore, ($\sum$ANU+$\sum$MNU) will be less than or equal to ($\sum$AD+$\sum$MD), resulting the value of IHD(*all_subclasses* at level (*m+1*)) to be always between 0 and 1.

3.1.4 Further Discussions

If the Denominator is Zero

Let's have a look at equation (e-4), (e-8) and (e-12) . All of them have ($\sum_{i=1}^{n}$ AD + $\sum_{i=1}^{n}$ MD) as the denominator. If ($\sum_{i=1}^{n}$ AD + $\sum_{i=1}^{n}$ MD) is zero, which means both $\sum_{i=1}^{n}$ AD and $\sum_{i=1}^{n}$ MD are zero, the metrics will have an infinite value. Is this kind of situation possible?

Case 1: $\sum_{i=1}^{n}$ AD(*subclass_i*) is zero.

This means there are no attributes defined in superclass. Practically, this is abusing the notion of a *class*, which is a group of objects with the same data structure and behavior. If there are no attributes, then there are no objects that could be instantiated from this class.

Case 2: $\sum_{i=1}^{n}$ MD(*subclass_i*) is zero.

This means that there are no methods defined, neither in superclass, nor in subclasses. Theoretically, it is possible. Practically, this is meaningless. First, there is no behavior of this class, which once again abuses the notion of a class. Second, grouping data structure and behavior into a class is one of the basic differences between object-oriented and traditional systems. If there is no method, then a class in C++ is equal to a structure data type in C. The program will then loose the benefits provided by object-oriented concepts.

The Meaning of the Metric Value

When a subclass inherits a superclass, the number of attributes and methods it inherits and those it defines decide the value of IHC, IHA and IHD. There are four possible cases in a subclass.

Case 1: AA $\ggg$[3] AI, and MA $\ggg$ MI+MC,
where AI is the number of all the attributes defined in the superclass;
and (MI+MC) is the number of all the methods in the superclass.

The subclasses define more attributes and methods than they inherit. IHC will be close to 0, while IHA will be close to 1.

The inheritance mechanism is used to group objects with the same data structure and behavior. If the subclasses have only 1%[4] of attributes and methods that are the same, it is meaningless to extract that 1% common attributes and methods into a superclass. This is not a good scenario.

[3]The symbol $\ggg$ means the value on the left is much larger than the value on the right.
[4]We exaggerate the number to explain the situation.

Case 2: AA $\lll$[5] AI, and MA $\lll$ MI+MC.

The subclasses inherit more attributes and methods than they define. IHA will be close to 0, and IHC will be close to 1. Contrary to the first case, in this case, the subclasses keep lots similarity with each other. From implementation view point, a lot of code will be saved by having a superclass. Moreover, this superclass is intended to have good portability or reusability.

Case 3: AA $\lll$ AI, and MA $\ggg$ MI+MC.

Most of the attributes for subclasses in this case are coming from their superclasses, while the subclasses define a lot of new methods. IHC will be close to 0. This means the subclasses keep same data identity, but have more variable behaviors. We feel this case can also reflect a good inheritance mechanism.

Case 4: AA $\ggg$ AI, and MA $\lll$ MI+MC.

The subclasses can perform a lot similar functions with less amount of same data identity. IHC will be close to 1. Consider a group of sibling subclasses, namely "bird", "plane" and "balloon". The commonality is that they all can *fly*, with quite different mechanisms. We would say the superclass of these subclasses does not reflect the abstraction idea in real world. Therefore, this case is not so good.

[5]The symbol $\lll$ means the value on the left is much smaller than the value on the right.

3.2 Metrics Based on Polymorphism

In the context of implementation, polymorphism is defined such that an operation may behave differently in different classes. Metrics of polymorphism will reveal the efforts required to test the superclass caused by polymorphism. At the design stage, the polymorphism can be found out through the object model [42], since methods with same name but different signatures will appear in different classes in an inheritance hierarchy. When the coding is finished, different size of each polymorphic method will reveal the efficiency of run-time method resolution brought by polymorphism mechanism.

3.2.1 Polymorphism Factor Based on Overriding Methods (PFOM)

The metric PFOM can be used in the early stage of software development life cycle. It represents the efforts needed for testing superclass due to methods being overridden in subclasses. PFOM can be defined as:

$$\text{PFOM} = \frac{\begin{array}{l}\text{actual number of overridden methods}\\ \text{in an inheritance hierarchy}\end{array}}{\begin{array}{l}\text{maximum possible number of}\\ \text{override methods}\end{array}} \qquad \text{(e-13)}$$

In C++, the polymorphic methods are declared as *virtual* in the superclass, and overridden in the subclasses. There are other multiple defined scenarios in C++, such as operator overloading and multiple constructors. Since multiple constructors have closer relationship with the host class than with subclass, we will not put multiple constructors in this metric.

PFOM($\langle classname \rangle\_tree$) =

$$\begin{cases} 0, & \text{if there are no multiple defined scenarios} \\ \dfrac{\sum_{i=1}^{m} \text{NIV}(subclass\_i) + \sum_{i=1}^{m} \text{NOLO}(subclass\_i)}{m \times [\text{NIV}(superclass) + \text{NOP}(\langle classname \rangle\_tree)]}, & \text{otherwise} \end{cases} \quad \text{(e-14)}$$

where

NIV(*subclass_i*): number of methods in *subclass_i*, which are the implementations of the same virtual method defined in the *superclass*.

NOLO(*subclass_i*): number of overloaded operators in *subclass_i*.

NOP(*subclass_i*): number of system-defined operators which have been overloaded in this *subclass_i*.

m: number of all the subclasses of the $\langle classname \rangle\_tree$.

The subclasses at different inheritance levels will either concretize a virtual method, or keep the original virtual method. For a virtual method, it can be concretized by any subclasses at the same level in different ways. Once it is concretized by a subclass, the redefined method will be passed on to next level in this $\langle subclass \rangle\_tree$. Otherwise, the original virtual method will be passed on. Therefore, technically every subclass in this inheritance hierarchy has the chance to concretize all the virtual methods defined in the superclass.

In the second equation of the metric PFOM, the denominator indicates that in case of no virtual method defined, the total number of methods defined for the same functions. The numerator indicates the actual number of overridden methods. Therefore,

the finial value of PFOM(⟨*classname*⟩*_tree*) will be between 0 and 1.

3.2.2 Average Changing Rate of Virtual Methods (ACRV)

This metric compares number of statements in the virtual methods defined in *superclass* and those in its overridden versions in subclasses.

C++ contains facilities for run-time method resolution. Method resolution and the ability to override a method in a subclass are only available if the method is declared *virtual* in the superclass. The implementation of run-time method resolution is efficient. At run-time, a virtual method is resolved by retrieving the method structure from the object and selecting a member to find the method address [42].

We can use the metric ACRV to check the efficiency by using run-time method resolution. It is defined as below:

$$\mathrm{ACRV}(\langle classname\rangle\_tree) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{k}\mathrm{NSV}(subclass\_i, virtual\_method\_j)}{m * \sum_{j=1}^{k}\mathrm{NSV}(superclass, virtual\_method\_j)} \quad \text{(e-15)}$$

where

NSV(*class,method*): number of statements in *"method"* defined in *"class"*. It is is zero in the program, we set it to 0.5. The justification for this arrangement is not to leave the denominator be zero. Moreover, a value of 0.5 is reasonably small.

m: number of all the subclasses in the *superclass_tree*.

k: number of virtual-methods defined in the *superclass*.

When a subclasses chooses to implement a virtual method, the number of statements for this method in the subclass will increase. The numerator in (e-15) will greater than or equal to the denominator. The value of ACRV(*superclass_tree*) will be greater than or equal to 0. When ACRV is equal to zero, it means no subclasses implement any of the virtual methods, which is less possible to happen. The more methods are implemented, the greater the value for ACRV.

3.3 Metric Based on Interaction among Objects

The interaction among classes will greatly affect the effort needed to maintain software. Since object interactions are defined by message passing, any changes to a class definition will directly affect the interaction. The metrics of interaction among classes will reveal this kind of effort.

The Object Modeling Technique [42] describes different kinds of relationships between classes, such as association, aggregation, and inheritance. We have developed this catalogue of metrics based on the above relationships.

C++ has provided several techniques to implement the relationships introduced in OMT. These techniques include the use of pointers to objects, usage of reference, etc. The following are some examples.

Implementing association in C++ using embedded pointers:

```
class Customer {
private:
  Automobile *mycar;
```

```
  ... ...
};

class Automobile {
private:
  Customer *owner;
  ... ...
};
```

Implementing multiple associations in C++ using template class:

```
class Ticker {
private:
  Passenger *traveler;
  ... ...
};

class Passenger {
private:
  List(Ticker) tickets;
  ... ...
};
```

Implementing standard conversions from derived objects to base objects in C++ using references:

```
class Derived : public Base { ... }
Base b;
Derived d;
```

```
Base *bp = &d;
Base & br = d;
```

3.3.1 Interaction of Objects from Program Viewpoint (IFPV)

This metric examines the interaction from the viewpoint of the whole program. The result indicates the percentage of classes in the program which have relation with *Class_C*. In this metric, we will check how many superclasses a class have, but won't check the number of subclasses. The reason is that we believe the superclasses have more impact on subclasses than subclasses do on superclasses.

$$\text{IFPV}(Class\_C) = \left(\frac{\text{NCP}(Class\_C) + \text{NTC}(Class\_C) + \text{NSUP}(Class\_C)}{\text{NCIP}} \times 100 \right) \% \quad \text{(e-16)}$$

where

NCP: number of class pointers used in *Class_C*.

NTC: number of template classes used in *Class_C*, other than itself. Every class pointer will be counted only once, even if it might appear several times.

NSUP: number of superclasses of *Class_C*, other than itself. Every class pointer will be counted only once, even if it might appear several times.

NCIP: number of classes in the program.

3.3.2 Interaction of Object from Class Viewpoint (IFCV)

This metric examines the interaction of a method from methods within a class. The result represents the percentage of attributes, methods and parameters in *Class_C* which has relationship with other classes. We didn't include the number of local variables in this metrics. This metrics is intended to evaluated a design, where there is no concept of "local variables".

$$\text{IFCV}(Class\_C) = \left(\frac{\text{NDC}(Class\_C) + \text{NPC}(Class\_C) + \text{NRC}(Class\_C)}{\text{AD}(Class\_C) + \text{MD}(Class\_C) + \text{NPAR}(Class\_C)} \times 100 \right) \% \quad \text{(e-17)}$$

where

NDC: number of attributes/parameters/methods in *Class_C* directly defined as other class type.

NPC: number of attributes/parameters/methods in *Class_C* pointed to by other classes.

NRC: number of attributes/parameters/methods in *Class_C* referenced by other classes.

NPAR: number of parameters of all the methods in *Class_C*.

Chapter 4

Comparison of Metric Sets

In this chapter, we will compare the metrics proposed in this thesis with three other metrics sets that were collected from the literature. While there are numerous publications on software metrics, the three sets of metrics chosen for comparison are considered as benchmarks in this area. Some researchers do criticize some of these metrics; nevertheless, the three sets of metrics address several fundamental issues with regard to object-oriented software metrics, thus giving rise to a basis for comparison. The three sets are:

- Chidamber and Kemerer's metrics suite [16]
- MOOD metrics set [1]
- Kim et al's metrics set [32]

Since it is hard to establish a one-to-one comparison between the metrics, we select only a subset of metrics from the above three sets. A justification for the partial

comparison comes from the fact that these three metrics sets as well as the one described in Chapter 3 of this thesis are all targeted towards different goals. So, we select only those fundamental metric components and their compositions in deriving new metrics.

4.1 Metrics Based on Inheritance Hierarchy

In this section, we compare the metrics based on inheritance hierarchy.

4.1.1 Chidamber and Kemerer's Metrics Suite

Chidamber and Kemerer proposed the following two metrics based on inheritance hierarchy.

Metric Name: Depth of Inheritance Tree (DIT)

Description: This metric indicates the height of the inheritance hierarchy. According to Chidamber and Kemerer, this metric can be used to predict the behavior of an object, the design complexity and the potential for reuse of the classes in the inheritance hierarchy.

Metric Name: Number of Children (NOC)

Description: The metric counts the number of immediate subclasses subordinated to a class in the inheritance hierarchy. It gives an indication of potential reuse

of a class at a given level. Indirectly, the same metric can also help predicting the testing effort required for the same class.

Both DIT and NOC are basic metrics, which can be directly gathered from the inheritance hierarchy of an object-oriented design. They are based on the internal structure (namely attributes and methods) of the design. Chidamber and Kemerer claimed that DIT and NOC can be used to predict the external characteristics[1] of a design such as design complexity and testing efforts. However, this claim is hard to prove.

Many researchers criticized that Chidamber and Kemerer's metrics suite is inadequate to evaluate external characteristics of a design [20, 30]. Binkley and Schach [12] conducted experiments to evaluate several object-oriented metrics including Chidamber and Kemerer's. According to their experiments, DIT and NOC had a success rate of only 28% in predicting design complexity. Binkley and Schach also pointed out that DIT and NOC are at a high level of abstraction and are not adequate for evaluating a design.

4.1.2 MOOD Metrics Set

There are two metrics in MOOD metrics set based on the inheritance hierarchy.

Metric Name: Attribute Inheritance Factor (AIF)

[1]External characteristics are the characteristics such as design complexity and class reusability, which can only be measured with respect to how the product, process, or resource relates to other entities in its environment [20].

Description: This metric is defined as the ratio of inherited attributes to the total attributes in a complete program.

$$\text{AIF} = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)},$$

where

TC = total number of classes in the program

$A_i(C_i)$ = number of attributes inherited by class C_i

$A_a(C_i)$ = number of attributes that can be accessed by class C_i

$A_i(C_i) \subseteq A_a(C_i)$

The authors of the MOOD metrics set claim that AIF is a good measure of inheritance, and represents the degree of reuse of the attributes.

Metric Name: Method Inheritance Factor (MIF)

Description: This metrics is defined as the ratio of the inherited methods to the total methods in a program. Like AIF, MIF is also claimed to be a measure of inheritance and indicates the level of reuse.

$$\text{MIF} = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)},$$

where

$M_i(C_i)$ = number of methods inherited in class C_i

$M_a(C_i)$ = number of methods that can be accessed by class C_i

$M_i(C_i) \subseteq M_a(C_i)$

Both AIF and MIF are strongly related to the inheritance hierarchy. While Chidamber and Kemerer claim that DIT and NOC indicate an abstraction of the inheritance hierarchy based on number of classes, both AIF and MIF by the MOOD metrics indicate the level of reuse in more detail. This is because they are defined on the internal structures of the classes. Compared to DIT and NOC, MOOD metrics' AIF and MIF give better indication of the complexity of a program.

4.1.3 Kim's Metrics Set

There is only one metric from this set that is based on the inheritance hierarchy.

Metric Name: Degree of Reuse (DOR)

Description: Evaluated for each class separately, this metric defines the ability or potential of reuse of a class. It is defined as:

$\mathrm{DOR}(C_i)=\sum_{k=1}^{r(C_i)}[k/(t+tr)]$,

where

$r(C_i)$ = reused number of each class C_i in the program; typically, it refers to the number of subclasses of C_i

t = total number of classes in the program

tr = total sum of all $r(C_i)$ś in the program

Since DOR is, once again, based on the number of classes, rather than the internal structures of the classes, it is at the same level of abstraction as that of DIT and NOC. Hence, the usefulness or applicability of DOR is under the same criticism as that of Chidamber and Kemerer's metrics suite.

4.1.4 Liu's Metrics Set

The metrics IHA, IHC and IHD defined in the previous chapter are based on the inheritance hierarchy. These three metrics are quite comparable to AIF and MIF of MOOD metrics because they are based on the internal structure of a class. Thus, the level of abstraction of Liu's metrics is lower than that of Chidamber and Kemerer's and Kim's metrics sets. While the goal of MOOD metrics set is to evaluate the level of reuse, the goal of Liu's metrics set is to evaluate the testing efforts required to test individual classes. Thus, the focus of the latter is on the individual hierarchies, while the former is targeted towards the entire program. It is quite common that all the classes in a program do not participate in a single inheritance hierarchy or not in any inheritance hierarchy. In this context, we claim that Liu's metrics set is more detailed than MOOD metrics set.

4.2 Metrics Based on Polymorphism

In this section, we will compare the metrics based on polymorphism. None of the metrics in Chidamber and Kemerer's metrics set and in Kim's metrics set examine this object-oriented characteristics. Therefore, in this section, we compare only MOOD and Liu's metrics sets.

4.2.1 MOOD Metrics Set

There is one metric in MOOD that is based on polymorphism.

Metric Name: Polymorphism Factor (PF)

Description: This metric is proposed as a measure of polymorphism in a program. It is defined as the number of methods that redefine inherited methods, divided by the maximum number of possible distinct polymorphic situations in a program.

$$\text{PF} = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]},$$

where

$DC(C_i)$ = number of class inheriting from class C_i

$M_n(C_i)$ = number of new methods in class C_i

$M_o(C_i)$ = number of overriding methods in class C_i

This metric can be viewed as an indirect measure of the relative amount of dynamic binding in a program, which is strongly related to polymorphism and inheritance. The authors of the MOOD metrics set claimed that an appropriate use of polymorphism in object-oriented project designs should decrease the defect density as well as rework [2]. However, higher value of PF tend to reduce the benefits achieved through these metric.

Harrison [26, 27] analyzed the validity of PF and concluded that this metric exhibits an unexpected discontinuity when the denominator is zero. Since the denominator includes the number of subclasses of a superclass ($DC(C_i)$), the value of the denominator for a system without any inheritance will be zero, which will give an infinite value for the metric itself. That would be meaningless in analyzing software.

4.2.2 Liu's Metrics Set

The metrics PFOM and ACRV defined in Liu's metrics set are related to polymorphism.

The metric PFOM is quite comparable to PF of MOOD metrics set. As the goals of MOOD metrics set and Liu's metrics set are different, the focus of metrics PFOM and PF are different, with PFOM focusing on the inheritance hierarchies individually and PF focusing on the entire program.

The metric ACRV of Liu's metric indicates the level of polymophism from within the classes of each individual inheritance hierarchy. It can also give an indication of the efficiency of late binding.

4.3 Metrics Based on Interaction between Objects

In this section, we will make the comparison based on the interaction between objects. The metrics will be chosen from all the four metrics sets.

4.3.1 Chidamber and Kemerer's Metrics Suite

There are two metrics in this set that are related to interaction between objects.

Metric Name: Coupling between Object Classes (CBO)

Description: This metric counts the number of other classes to which the current

class is coupled[2]. CBO can be viewed as an indication of the effort needed for maintenance and testing. Excessive coupling between classes is detrimental to modular design and prevents reuse.

Binkley and Schach [12] suggested CBO provided only crude estimate of the coupling. There is no distinction made between different types of coupling, such as the number of parameters and how they are passed. Nevertheless, they did not deny the usefulness of coupling metric. Binkley and Schach claimed coupling is indeed a good metric for design quality, as long as the details of the coupling are taken into account. Once again, this indicates that Chidamber and Kemerer's metrics suite is at a high level of abstraction and is not adequate for evaluating an object-oriented design.

Metric Name: Response for a Class (RFC)

Description: This metric calculates the occurrences of calls to other classes from a particular class. The authors claimed that this metric is a measure of the potential communication between the class and other classes, thus predicting the complexity of testing and debugging of the class.

RFC = |RS|, where |RS| is the response set for the class

RS = $\{M\} \bigcup_{all\ i} \{R_i\}$

$\{R_i\}$ = the set of methods called by method i

$\{M\}$ = the set of methods in the class

Like CBO, the metric RFC does not distinguish between different types of coupling. For example, parameters passed by value or by reference appear identical under this

[2]Two classes are coupled when methods declared in one class use methods or instance variables defined in the other class [16].

measurement. However, compared to CBO, RFC can give better prediction, because RFC examines every call of all the methods in a class. Binkley and Schach showed that the successful prediction rate of RFC is much higher than that of CBO [12].

4.3.2 MOOD Metrics Set

There is one metric in MOOD metrics set that is based on interaction between objects.

Metric Name: Coupling Factor (COF)

Description: This metric is defined as the actual number of couplings not caused by inheritance, divided by the maximum possible number of couplings in a program.

$$\text{COF} = \frac{\sum_{i=1}^{TC}\left(\sum_{j=1}^{TC} is\_client(C_i, C_j)\right)}{TC^2 - TC - 2 \times \sum_{i=1}^{TC}(C_i)},$$

where

$is\_client(C_i, C_j) = 1$, if client class C_i contains at least one non-inheritance reference to a method or attribute of supplier class C_j;

or $is\_client(C_i, C_j) = 0$, otherwise.

$2 \times \sum_{i=1}^{TC}(C_i)$ = the maximum number of couplings due to inheritance.

The metric COF removes the coupling caused by inheritance, while both CBO and RFC do not differentiate the causes of interactions between classes. Comparing to RFC, the coupling factor gathered by COF refers to the whole system. Therefore,

if we want to get an approximate evaluation of maintenance efforts for the whole system, COF will be the proper choice. When focusing on the design of a single class, RFC will be more useful.

4.3.3 Kim's Metrics Set

Kim et al defined three metrics based on interaction between objects: CBI, UCL, and MPC. In the metric CBI, a syntax complexity metric IMC is used to assist the calculation.

Metric Name: Degree of Coupling of Inheritance (CBI)

Description: This metric examines each class separately. It is defined as the product of total complexity of methods in a class, and number of subclasses of th is class. Kim et al claimed that the value of CBI indicates the degree how far the implementation of the class is dependent on other classes inherited. This metric is defined as:

$\text{CBI}(C_i) = (\sum \text{IMC}(M_j)) \times ih(C_i)$,

where

$\text{IMC}(M_j)$ = the degree of internal methods complexity for each method M_j in class C_i

$ih(C_i)$ = the number of subclasses which of class C_i

This metric is only related to the interactions which are caused purely by inheritance. In order to examine the effect of interactions in a program, the metric has to be used in conjunction with other metrics.

In CBI, a syntax complexity metric IMC is used.

Metric Name: Degree of Internal Method Complexity (IMC)

Description: For each method in a class, or each function in a program, this metric defines the syntax complexity based on all the operators used in that method or that function. It applied the effort equation of Halstead's software science. It is defined as:

$$\text{IMC}(M_i)=\frac{(N_1 + N_2) \times log_2(n_1 + n_2)}{(n_1/2) \times (n_2/N_2)},$$

where

n_1=number of unique operators in the method M_i

n_2=number of unique operands in the method M_i

N_1=total number of appearances of operators in the method M_i

N_2=total number of appearances of operands in the method M_i

In a method or a function, each unique operator can appear in different statements which will have different impact on the program. For example, the operator "[ ]" can be used as index of an array, such as *num[i]*. Or, it can be used in an equation like *[(a+b)*c]/(c+d)*. This metric fails to differentiate the above situations. Moreover, the number of operators and operands do not have direct link with system complexity and maintenance cost. Therefore, the metric CBI, which uses IMC, is at a high level of abstraction and not adequate for evaluating a software design.

Metric Name: Number of classes used in a class, except for its superclasses and subclasses (UCL)

Description: This metric is calculated for each class individually. It examines four kinds of interactions among classes via message passing, and calculates the

occurrence times of each kind of interaction. It is simply defined as the sum of all occurrences.

UCL(C_i)=$u_1 + u_2 + u_3 + u_4$,

where

u_1 = number of classes in S, which are defined as data member of class C_i

u_2 = number of classes in S, which are defined as data member of method M_j in class C_i

u_3 = number of classes in S, which are defined as pointer in C_i

u_4 = number of classes in S, which are defined as pointer in method M_j of class C_i

S = the set of classes which do not have "is-a" relationship with C_i in the program

In contrast to the previous metric CBI, the metric UCL indicates the interactions which are not caused by the classes within same inheritance hierarchy. The definitions for u_1, u_2, u_3, and u_4 are good approaches to describe the implementation of different kinds of interactions happening in a class. However, this metric counts the same class several times, which will make the metric value larger than the actual complexity caused by interaction.

Metric Name: Number of Send Statements of a Class (MPC)

Description: This metric is computed for each class separately. It is defined as:

MPC(C_i)=$\sum[s(m_{ij}) + r(m_{ij})] \times v(m_{ij})$,

where

$s(m_{ij})$ = number of ";" in each member function m_{ij} of $M(C_i)$

$r(m_{ij})$ = number of reserved works in each m_{ij} of $M(C_i)$, such as "if", "switch",

"case", "default", "for", "while" "do-while"

$v(m_{ij})$ = number of m_{ij} used by other classes

$M(C_i)$ = a set of member functions, which are used in class C_i and are called from other classes

If we take the interaction discussed in metric UCL as "fan-in" interaction, then those appeared in metric MPC will be "fan-out" interaction. The difference between UCL and MPC is that the former focuses on each class as a whole, while the latter discusses each statement in detail. Thus, UCL and MPC indicate the complexity of interactions from different viewpoints.

4.3.4 Liu's Metrics Set

The metrics IFPV and IFCV in Liu's metrics set are based on the interaction among objects.

Both IFPV and IFCV use same kind of definition as used in metric CBI. The metrics IFPV and IFCV include more implementation options, such as templates and references, which tend to include more kinds of interaction that can occur between objects. Metrics UCL and MPC use an actual count to present the interaction complexity, while IFPC and IFCV use a relative measure.

Chapter 5

Evaluation of Liu's Metrics Set

The goal of this chapter is two-fold: (1) to analytically evaluate the Liu's metrics set using a set of properties for object-oriented software metrics by Weyuker; and (2) to apply the metrics to some simple programs and empirically evaluate the metrics.

5.1 Analytical Evaluation Using Weyuker's Properties

Weyuker [44] defined nine properties with which a metrics set can be analytically evaluated. Weyuker's list of properties is well-known in the area of software metrics and has been treated as an analytical tool for several years. Some researchers [14, 20] do criticize Weyuker's properties for being more theoretical and formal; nevertheless, these properties have been applied to several metrics sets including the metrics defined by Chidamber and Kemerer, and Kim's metrics set.

In this section, we will list all the nine properties defined by Weyuker and for each one of them, we informally justify how that property is met by Liu's metrics proposed in Chapter 3. The following notations are used to describe the properties:

P, Q and R denote arbitrary, but distinct, program bodies; sometimes, they also denote classes [16].

$|P|$ denotes the complexity of the program P or the value of a metric for P; $|P| \geq 0.0$.

$P;Q$ denotes the combination of two programs P and Q. One can choose any composition operator in place of ";" in combining P and Q.

We have also used Cherniavsky and Smith's explanations [14], wherever appropriate, to explain these properties.

Property 1 : $(\exists P)\ (\exists Q)\ (|P| \neq |Q|)$.
Informally, this property asserts that not every program (or class) has the same metric value; stated otherwise, there are at least two different programs (or classes) which have different complexities.

The IHC metric definition, for example, is based on the number of attributes and methods defined in a class and its subclasses. It is very rare to find two classes having the same subclass hierarchy[1]. Thus, the value of IHC is generally different for two different classes, thus conforming the fact that $\text{IHC}(C_1) \neq \text{IHC}(C_2)$, where $C_1 \neq C_2$.

Property 2 : Let c be a nonnegative number. Then, there are only finitely many programs or classes of complexity c in a given application.

[1] A subclass hierarchy of a class C is that portion of an inheritance hierarchy in which C is the root of the hierarchy.

Property 2 states that there are a finite number of programs (or classes) which possess the same complexity or metric value.

The statement of property 2 as stated above should be interpreted in the following way: Given a program of complexity c, there are only finite number of programs having the same number of complexity c. Weyuker argues that this statement will be generally true for program written using conventional program languages. The justification comes from the fact that conventional languages have finite number of distinct operators and the complexity is measured using expressions involving these distinct operators.

According to Chidamber and Kemerer's analysis, their metric suite satisfies this property; they claim that the complexity of object-oriented programs measured in terms of number of classes in a program, and there are only finite number of classes in each program [16].

We claim that Liu's metrics also satisfy this property. Liu's metrics set contains metrics based on classes and some metrics based on their components such as statements and attributes. Similarly to the claims by Weyuker, Chidamber and Kememer, we also argue that the number of components in a class as well as the number of classes in an application are both finite. Besides the complexity is measured in terms of these finite number of components and classes. Therefore, Liu's metrics satisfy this property.

Property 3 : $(\exists P)\ (\exists Q)\ (|P| = |Q|)$.

Informally, this property indicates that there exist at least two different programs or classes P and Q that are equally complex.

This definition is too vague because it does not precisely say whether P and Q belong to the same program if they refer to classes, or whether they belong to the same application if they refer to programs. Since metric definitions are expressed as numeric formulas, it is possible to find two classes or programs having the same metric value. However, it is not guaranteed that these two programs or classes will belong to the same application. We, therefore, do not consider this property for evaluation of Liu's metrics set.

Property 4 : $(\exists P)\ (\exists Q)\ (P \equiv Q \text{ and } |P| \neq |Q|)$.

This property expresses that two programs (or classes), which provide the same functionality, can have different complexities.

The intuition behind property 4 is that even though two programs (or classes) perform the same function, it is the details of the implementation that determine the program's (or class') complexity. That is, we are measuring the complexity of the program (or class), not the functions being computed by the program (or class). Since all the measures in Liu's metrics set are implementation dependent, they all satisfy this property.

Property 5 : $(\forall P)\ (\forall Q)\ (|P| \leq |P;Q| \text{ and } |Q| \leq |P;Q|)$.

This indicates that the complexity for the combination of two programs (or classes) will never be less than the complexity of either of the component programs (or classes).

Generally speaking, when two classes P and Q are combined, the total number of attributes and methods will increase, so will be the number of inherited, redefined

and new attributes and methods, etc. If P and Q have inheritance relationship, say P is Q's superclass, then the above data will be no less than that of Q's. For the metrics in Liu's metrics set, they are all defined as the ratio of different combination of the above data. Therefore, the values of numerators and denominators in those metrics are uncertain. It is hard to compare the value of $|P|$, $|Q|$ and $|P;Q|$. In other words, it may be possible to find, for example, a pair P and Q such that $|P| = |P;Q|$. Thus, we tend to disagree that this property need not be true for every pair of programs or classes.

Property 6 :

(a) $(\exists P)\ (\exists Q)\ (\exists R)\ (|P| = |Q|$ *and* $|P;R| \neq |Q;R|)$.

(b) $(\exists P)\ (\exists Q)\ (\exists R)\ (|P| = |Q|$ *and* $|R;P| \neq |R;Q|)$.

In essence, property 6 suggests that when two programs or classes P and Q having equal complexity are composed independently with another program or class R, the resulting programs or classes will have different complexities, irrespective of the order of composition.

Consider for example, two classes P and Q having the value 3/5 for the metric PFOM (Polymorphic Factor based on Overriding Methods). According to the definition of PFOM, there are 3 methods in P which are actually overridden and P has 5 methods that could be overridden. The same case applies to the class Q. Now consider a class R which is inherited into P and Q, independently; that is, the composition operator in $R;P$ and $R;Q$ is the inheritance mechanism. In the resulting inheritance hierarchies, the value of PFOM($R;P$) and PFOM($R;Q$) may be possibly different. This is because some of the subclasses of P might override some methods of R while the subclass of Q

may not override any method of R. Therefore, one can easily conclude that $|R;P|$ and $|R;Q|$ may be quite different. Similar arguments can be given for the complexities $|P;R|$ and $|Q;R|$. Thus, we claim that the Liu's metrics set satisfies the property 6.

Property 7: There are program bodies P and Q, such that Q is formed by permuting the order of the statements of P, and $|P| \neq |Q|$.

This property means changing the order of statements may change the complexity of the program (or class).

Weyuker discussed two aspects of this property [44]. On one hand, since the complexity of a program is completely independent of the location of statement, the statement count will not satisfy this property. On the other hand, since the placement of statements may affect the interaction of the program, and hence affect the program's complexity, this property would hold if the program is evaluated using the data flow measure. Chidamber and Kemerer argue that in object-oriented designs, a class is an abstraction of the problem space, and hence the order of statements within the class definition has no impact on eventual execution or use [16]. This will be true only for metrics based on class levels; for example, all metric definitions given by Chidamber and Kemerer are based on classes only. Therefore, changing the statement order within the class definition does not change the metric involving the class. In Liu's metrics set, some of the metrics are based on classes, and hence this property is not satisfied. However, those metrics which are based on statements within a method may suffer a changing metric value, if their order of execution changed. For example, the metric PFOM and ACRV are based on the number of overridden methods, which is still based on the statements inside a method. So PFOM and ACRV may change

when the order of statements within method changed, thus conforming the property 7.

Property 8 : If P is a renaming of Q, then $|P| = |Q|$.

This property states that uniformly changing variable or function names should not affect a program's (or class') complexity.

It is obvious that the measurements of each metrics in Liu's metrics set do not depend on the name of program (or class) and/or the names of the variables or functions within the program or class. Renaming does not change the number of components or interactions within a program (or class). Thus, each metric in Liu's metrics set satisfies this property.

Property 9 : $(\forall P)\ (\forall Q)\ ((|P| + |Q|) \leq |P; Q|)$.

This property states that a combined program (or class) will be more complex that its constituents parts.

We claim that this definition is too vague since it requires a precise description and semantics for the operators "+" and ";". Therefore, we do not consider this property for evaluation.

5.1.1 Summary of the Evaluation Using Weyuker's Properties

Table 5.1 summarizes the evaluation results of Liu's metrics set with regard to Weyuker's properties.

Table 5.1: Analytical Evaluation Using Weyuker's Properties

Metrics	Properties								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
IHC	O	O	?	O	?	O	X	O	?
IHA	O	O	?	O	?	O	X	O	?
IHD	O	O	?	O	?	O	X	O	?
PFOM	O	O	?	O	?	O	?/O	O	?
ACRV	O	O	?	O	?	O	?/O	O	?
IFPV	O	O	?	O	?	O	X	O	?
IFCV	O	O	?	O	?	O	X	O	?

Symbols:

O: The metric satisfies the property.

X: The metric does not satisfies the property.

?: It is uncertain whether the metric will or will not satisfy the property.

As the result of evaluation, all the proposed metrics satisfy properties 1, 2, 4, 6 and 8. The metrics do not satisfy property 7, which is inappropriated for object-oriented metrics [14, 16]. The metrics gave an uncertain answer for properties 3, 5 and 9. This implies Weyuker's properties are suitable to evaluate simple metrics at an abstract level, but not for metrics which need complicated calculations. As it is stated by Cherniavsky and Smith [14], Weyuker's properties "should be used carefully and intelligently in evaluating complexity measures". The uncertainty of properties 3, 5 and 9 also indicates that combination of programs (or classes) does not necessarily

increase the final complexity.

5.2 Analytical Evaluation Using a Framework

Kim et al [31] proposed a framework to analyze the scope of metrics for evaluating the complexity of object-oriented programs. The framework contains three aspects: (1)syntax complexity, (2) inheritance complexity and (3) interaction complexity. Each aspect is expanded into five attributes.

The goal of this framework is to analyze the scope of a metrics set to see if that set covers a majority part of object-oriented paradigm, especially those features which make them different from traditional software metrics. This is different from Weyuker's properties, in the sense that the former evaluates a metrics set independent of others.

The attributes of Kim's framework are defined in a quantitative way, which makes they sound like another set of metrics, such as "number of methods", or "number of classes". It is likely to confuse the metrics and the framework that analyzes metrics. Besides these, we feel that it is necessary to add "polymorphism" into the framework, since polymorphism is another important characteristic of the object-oriented paradigm.

5.2.1 The Framework

We present Kim's approach of the framework in the form of a series of questions. For our evaluation, we choose a subset of Kim's approach and add some questions related

to polymorphism. Following are the questions:

Q1: Do the metrics measure the number of classes in a program?

Q2: Do the metrics measure the number of methods and attributes in a class?

Q3: Do the metrics measure the lines of codes of a program or any fractions of the program?

Q4: Do the metrics measure the degree of cohesion between methods in a class?

Q5: Do the metrics measure the depth of an inheritance tree?

Q6: Do the metrics measure the number of children of a class or its direct ancestors?

Q7: Do the metrics measure the degree of reuse by inheritance?

Q8: Do the metrics measure the number of polymorphic methods in a class?

Q9: Do the metrics measure the degree of applying polymorphism feature in an inheritance hierarchy?

Q10: Do the metrics measure the number of messages send by and received by a class?

Q11: Do the metrics measure the degree of coupling between classes?

5.2.2 Evaluation of the Metrics Using the Framework

Following are the answers for the questions in the framework to Liu's metrics.

Table 5.2: Analytical Evaluation of Metrics Using a Framework

Metrics	Framework										
	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**
Chidamber & Kemerer's Metrics	X	O	X	O	O	O	X	X	X	O	O
Kim et al's Metrics	O	O	X	O	O	O	O	X	X	O	O
MOOD Metrics Suite	O	O	X	X	X	O	O	O	O	O	O
liu's Metrics Set	O	O	O	X	O	O	O	O	O	O	O

Symbols:

O: The metric gives a positive answer to the question.

X: The metric gives a negative answer to the question.

For example, Chidamber and Kemerer's metrics [16] are widely used, but they fail to measure polymorphism as shown in Q8 and Q9; neither does Kim's metrics set[32].

5.3 Empirical Evaluation

In this section, we show the empirical results of applying Liu's metrics to a simple program. Figure 5.1 shows the class diagram for the program and Table 5.3 to 5.6 summarize the results.

Following is the program to which the metrics have been applied.

```
#include <stdio.h>
#include <string.h>

typedef char LongString[256];

class TreeItem {
  public:
   virtual void StringFormat(char* the_string)
           {strcpy(the_string,"??");}
  };

class IntTreeItem : public TreeItem {
  private:
   int data;
```

```
  public:
   IntTreeItem(int newvalue) {data=newvalue;}
   void StringFormat(char* the_string) {sprintf(the_string,"%d",data);}
   };

class TreeNode {
  private:
   TreeItem *data;
   TreeNode *left,*right;
  public:
   TreeNode(TreeItem *root_item) {data=root_item; left=right=NULL;}
   void StringFormat(char*) ;
   void Insert(TreeNode*,char*) ;
  };

class Tree : public TreeItem {
  private:
   TreeNode *top;
  public:
   Tree() {top=NULL;}
   void StringFormat(char*) ;
   void Insert(TreeItem*);
  };

void Tree::StringFormat(char* the_string) {
```

```
  if (top)
    top->StringFormat(the_string);
  else
    sprintf(the_string,"<>");
  }

void Tree::Insert(TreeItem* new_item) {
  TreeNode* new_node = new TreeNode(new_item);
  LongString its_string;
  new_item->StringFormat(its_string);
  if (top)
    top->Insert(new_node,its_string);
  else
    top = new_node;
  }

void TreeNode::StringFormat(char* the_string) {
  LongString temp;

  sprintf(the_string,"<");
  data->StringFormat(temp);
  strcat(the_string,temp);
  strcat(the_string,",");

  if (left) {
```

```
    left->StringFormat(temp);
    strcat(the_string,temp);
    strcat(the_string,",");
    }
  else
    strcat(the_string,"<>,");

  if (right) {
    right->StringFormat(temp);
    strcat(the_string,temp);
    strcat(the_string,">");
    }
  else
    strcat(the_string,"<>>");
  }

void TreeNode::Insert(TreeNode* new_node, char* its_string) {
  LongString temp;
  data->StringFormat(temp);
  if (strcmp(its_string,temp)<0)
    if (left)
        left->Insert(new_node,its_string);
    else
        left = new_node;
  else
```

```
    if (right)
        right->Insert(new_node,its_string);
    else
        right = new_node;
  }

void main() {
  Tree *T1 = new Tree;
  Tree *T2 = new Tree;
  LongString temp;

  T1->Insert(new IntTreeItem(10));
  T1->Insert(new IntTreeItem(5));
  T1->Insert(new IntTreeItem(20));
  T1->Insert(new IntTreeItem(30));

  T1->StringFormat(temp);
  printf("The StringFormat of T1 is:%s\n",temp);

  T2->Insert(T1);
  T2->Insert(new IntTreeItem(99));

  T2->StringFormat(temp);
  printf("The StringFormat of T2 is:%s\n",temp);
```

```
printf("That about does it.\n");
}
```

The class diagram of the program GenTree.cpp is shown in Figure 5.1. This figure uses the symbols of Object-Oriented Modeling Tool [42] and shows the components of each class and the inheritance relationships between each class. Most of the information required by in Liu's metrics set can be collected from the class diagram.

Table 5.3, 5.4, 5.5, 5.6 and 5.7 show the data collected from GenTree.cpp. In order to demonstrate how the data is counted, we include the names of attributes and methods that were counted.

In this program, we have NOP=0, and NCIP=4.

Table 5.4: Metrics Based on Inheritance

Class_Name	HIC	HIA	HID
TreeItem_tree	0.2857	0.7143	0

Table 5.5: Metrics for Virtual Methods

Class_Name	NIV	NSV	NOLO
TreeItem	1	virtual void StringFormat() NSV=1	0
IntTreeItem	1	void StringFormat() NSV=1+0.5	0
TreeItem	1	void StringFormat() NSV=4+0.5	0

Table 5.6: Metrics Based on Polymorphism

Class_Name	PFOM	ACRV
TreeItem_tree	1.0000	3.0000

5.3.1 Discussion

From the program we can see that the superclass *TreeItem* does not have any attributes. This is the case of having no data identity. The only method *TreeItem* has the virtual function StringFormat(), which is redefined in both of the subclasses. We can treat this inheritance as case 1 or case 4 in "Meaning of Metric Value" in section 3.1.4 of Chapter 3,

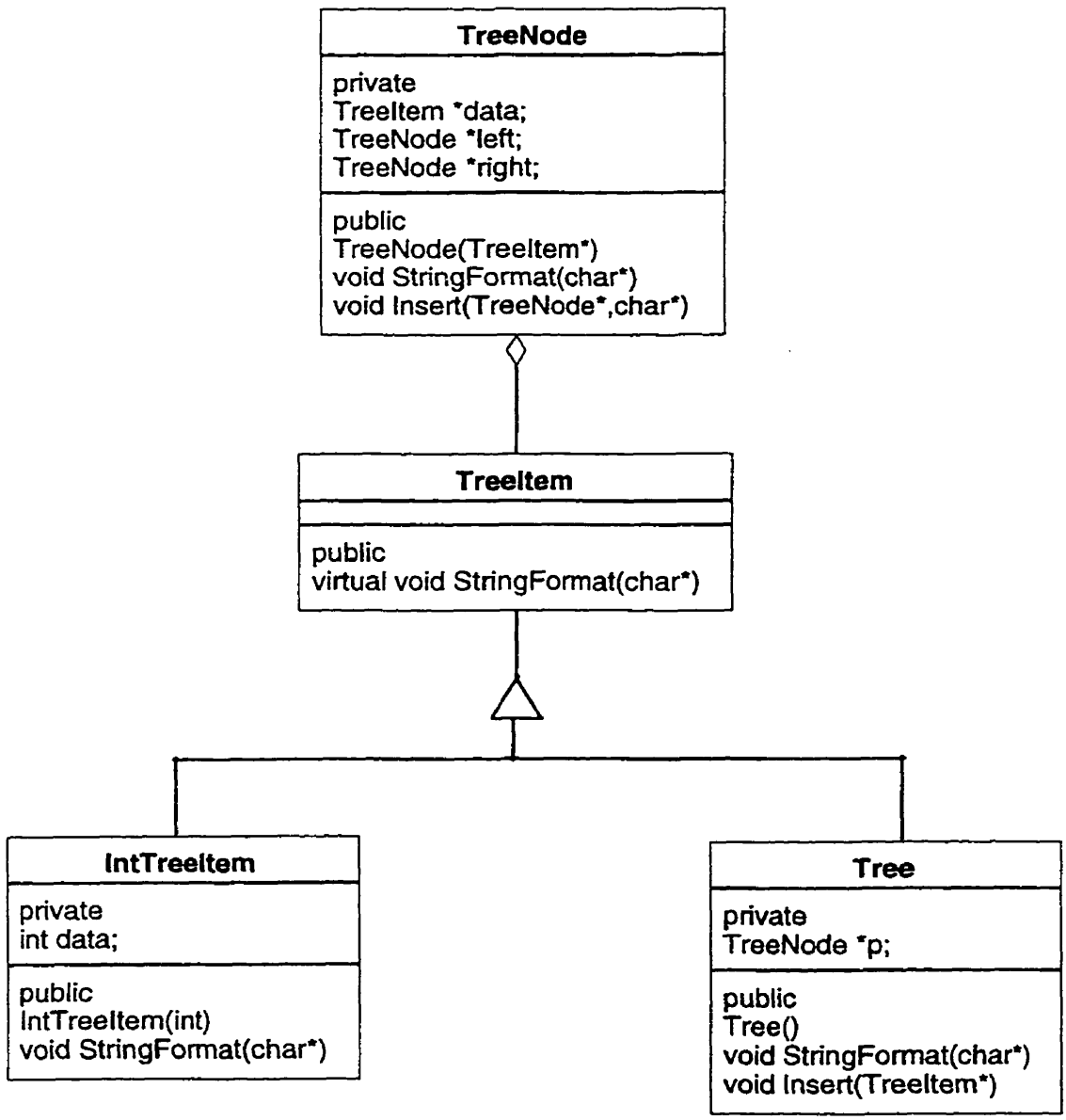

Figure 5.1: Class Diagram for the Program GenTree.cpp

Table 5.3: Metrics for Attributes and Methods

Class_Name	AA	AI	AD	ANU	MC	MA	MI	MD	MNU
TreeItem	0	–	0	–	–	1 virtual void StringFormat()	–	1	–
IntTreeItem	1 int data	0	1	0	1 void StringFormat()	1 *IntTreeItem*()	0	2	0
Tree	1 TreeNode *top	0	1	0	1 void StringFormat()	2 *Tree*() Insert()	0	3	0
TreeNode	3 TreeItem *data TreeNode *left TreeNode *right	–	3	–	–	3 *TreeNode*() StringFormat() Insert()	–	3	–

Table 5.7: Metrics Based on Interactions

Class_Name	NCP	NTC	NSUP	NDC	NPC	NRC	NPAR	MD	AD	IFPV	IFCV
TreeItem	0	0	0	0	0	0	1	1	0	0	0
IntTreeItem	0	0	1 TreeItem	0	0	0	2	2	1	0.2500	0
Tree	2 TreeNode*(1) TreeItem*(1)	0	1 TreeItem	0	2 TreeNode * TreeItem*	0	2	3	1	0.7500	0.2000
TreeNode	1 TreeItem*	0	0	0	2 TreeItem *data TreeItem *root_item	0	4	3	3	0.2500	0.2000

which is not good.

Both PFOM(*TreeItem_tree*) and ACRV(*TreeItem_tree*) are larger than zero. That means this inheritance tree do use polymorphism.

Let's look at the interactions. *TreeItem* does not have any message passing with other classes except for the inheritance, thus IFPV and IFCV for *TreeItem* are 0. Both *IntTreeItem* and *TreeNode* have relationship with only one class, while *Tree* have relationship with all three classes. Therefore, IFPV(*Tree*) is larger than IFPV(*IntTreeItem*) and IFPV(*TreeNode*). IFCV(*IntTreeItem*) is 0, while IFPV(*IntTreeItem*) is not zero. That means the class that interacts with *IntTreeItem* does not appear in the form of pointer of attribute or method, etc; then it can be only caused by *IntTreeItem*'s superclass. IFCV(*Tree*) and IFCV(*TreeNode*) have same value. This implies that both IFCV(*Tree*) and IFCV(*TreeNode*) have same degree of connection with other classes.

5.3.2 Other Empirical Evaluation

During the tenure of this thesis, there were not suitable tools that can extract the data required for Liu's metrics. There were two generate tools that are used to reduce a small portion of our manual work. The first tool was QMOOD++, developed by Bansiya and Davis [5]. The second tool was CCCC developed by Littlefair [36]. QMOOD++ was used to get the information such as "class hierarchy", "number of subclasses" and "number of superclasses". The CCCC tool offered very little help.

We have applied the Liu's metrics to two fairly large programs. They are named as PROJECT and SPAS [3]. The analysis results for these two programs and the metrics values are listed in the Appendixes. Appendix B and D are the metrics data defined and calculated by QMOOD++ and CCCC, some of which are used in Liu's metrics set. Ap-

pendix E and F are the metrics values for Liu's metrics set. Most of the data in these two appendixes are obtained manually.

Chapter 6

Conclusion and Future Work

This thesis is a contribution to the quantitative analysis of object-oriented programs. The thesis has introduced a new set of metrics for object-oriented programs, targeted towards evaluating the efforts required for testing object-oriented programs.

The definitions are evaluated analytically by (1) comparing them with other metrics sets chosen from the literature, and (2) analyzing them with respect to a set of properties outlined by Weyuker. It is to be noted that Weyuker's properties have been used to analytically evaluate several metrics sets that are commonly used and hence they became a benchmark for evaluating software metrics. The new set of metrics introduced in this thesis has also applied to two fairly large programs and the results have been included in the thesis.

Empirical evaluation of a metrics suite requires extensive tool support. During the tenure of this thesis, we were unable to find a suitable tool to extract the basic metric quantities from object-oriented programs. Therefore, it was hard to apply the Liu's metrics set to several programs. The two tools QMOOD++ and CCCC that were used for empirical evaluation in this thesis offered very little help; consequently, much of the work on empirical evaluation

was done manually. This necessitates the development of a tool as an immediate follow-up of this thesis.

As stated in the introduction of this thesis, research in software metrics is an ongoing process. Depending on the need and software trend, the research on metrics will continue to evolve. Since it is the goal of this new set of metrics to evaluated the efforts required for testing, we do anticipate that the work reported in this thesis will evolve with the research on object-oriented program testing.

Appendix A

Summary of Liu's Metrics

This appendix summarizes Liu's metrics set introduced in this thesis. These metrics are listed in Table A.1. The metrics IHC, IHA, IHD, PFOM, and ACRV are based on or related to inheritance hierarchy. IFPV and IFCV are more general.

Table A.1: Metrics for Testing Object-Oriented Software

Name	**Description**
IHC	Metrics of Inheritance Hierarchy Based on Changing Features
IHA	Metrics of Inheritance Hierarchy Based on Adding Features
IHD	Metrics of Inheritance Hierarchy Based on Deleting Features
PFOM	Polymorphism Factor Based on Overriding Methods
ACRV	Average Changing Rate of Virtual Methods
IFPV	Interaction of Objects from Program Viewpoint
IFCV	Interaction of Objects from Class Viewpoint

Table A.2: Basic Metrics

Name	Description
AI(*subclass_i*)	number of attributes in *subclass_i*, which are inherited from *superclass*
AA(*subclass_i*)	number of newly introduced attributes in *subclass_i*
AD(*subclass_i*)	number of all attributes in *subclass_i*
ANU(*subclass_i*)	number of attributes inherited by *subclass_i*, but not used
AP(*all_subclasses*)	Attribute Percentage for all subclasses in ⟨*classname*⟩*_tree.*
MI(*subclass_i*)	number of methods in *subclass_i*, which are inherited but not modified
MC(*subclass_i*)	number of methods in *subclass_i*, which are inherited and redefined in *subclass_i*
MA(*subclass_i*)	number of newly introduced attributes in *subclass_i*
MD(*subclass_i*)	number of all methods in *subclass_i*
MNU(*subclass_i*)	number of methods inherited by *subclass_i*, but not used
MP(*all_subclasses*)	Method Percentage for all subclasses in ⟨*classname*⟩*_tree*
NIV(*subclass_i*)	number of methods in *subclass_i*, which are the implementations of the same virtual method defined in the *superclass*
NOLO(*subclass_i*)	number of overloaded operators in *subclass_i*
NOP(*subclass_i*)	number of system-defined operators which have been overloaded in this *subclass_i*
NCIP	total number of classes defined in a program
NSV(*class, method*)	number of statements in *method* defined in *class*
NCP	number of class pointers used in *Class_C*, other than itself

continued on next page

continued from previous page

Name	**Description**
NTC	number of template classes used in *Class_C*, other than itself
NSUP	number of superclasses of *Class_C*
NDC	number of attributes, parameters and methods in a class directly defined as other class type
NPC	number of attributes, parameters and methods in a class pointed to by other classes
NRC	number of attributes, parameters and methods in a class referenced by other classes
NPAR	number of parameters of all the methods in a class
n:	number of immediate subclasses for a given class named ⟨*classname*⟩
m	number of all the subclasses of the ⟨*classname*⟩*_tree*
k	number of virtual_methods defined in the superclass

Appendix B

Data Extracted from Program "PROJECT"

This appendix includes data extracted from program "PROJECT" using QMOOD++ and CCCC.

```
=================Display Class Hierachy=================

  LockQueue
  DataItem
      KSIDataItem
  Pattern
  Database
  KS
      Gen
      Stat
```

```
  Control

Number of Classes Defined = 9
Number of Base Classes    = 2


=================Class Hierarchy Window=================

=============** LockQueue **==============
 int activeReaders;
 int activeWriters;
 boolean waitingReaders;
 boolean waitingWriters;

public:
 void readlock ( );
 void writelock ( );
 void readunlock ( );
 void writeunlock ( );
 LockQueue ( );
 LockQueue ( );

CLASS EXTERNAL MEASURES (LockQueue)
---------------------------------------
[DOI] Depth of Inheritance          = 0
[NOC] Number of Children            = 0
[NOA] Number of Ancestors           = 0
```

```
CLASS INTERNAL MEASURES (LockQueue)

----------------------------------------

[NOM] Number of Methods                 = 6

[CIS] Class Interface Size              = 6

[NOT] Number Of Trivial Methods         = 0

[NOP] Number Of Polymorphic Methods     = 0

[NOO] Number Of Operator Methods        = 0

[NPT] Number Of Parameter Types         = 0

[NPM] Number Of Parameters Per Method   = 0.00

[NOD] Number Of Attributes              = 4

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 0

[NPA] Number of Public Attributes       = 0

[CSB] Class Size in Bytes               = 16

[CSM] Class Size Metric                 = 10
```

```
CLASS DESIGN PROPERTY MEASURES (LockQueue)

----------------------------------------

[CAM] Cohesion Among Methods of Class   = 1

[DCC] Direct Class Coupling             = 0

[MCC] Maximum Class Coupling            = 0

[DAC] Direct Attribute Based Coupling   = 0

[MAC] Maximum Attribute Based Coupling  = 0

[MPC] Maximum Parameter Based Coupling  = 0

[VOM] Virtuality of Methods             = 0

[CCN] Class Complexity Based on Nodes   = 0
```

```
==============** KSIDataItem **==============
 int ksi;
 boolean activated;
Size: KSIDataItem  32 = 24 + 8

public:
 inline KSIDataItem ( );
```

CLASS EXTERNAL MEASURES (KSIDataItem)

[DOI] Depth of Inheritance	= 1
[NOC] Number of Children	= 0
[NOA] Number of Ancestors	= 1

CLASS INTERNAL MEASURES (KSIDataItem)

[NOM] Number of Methods	= 9
[CIS] Class Interface Size	= 4
[NOT] Number Of Trivial Methods	= 6
[NOP] Number Of Polymorphic Methods	= 1
[NOO] Number Of Operator Methods	= 4
[NPT] Number Of Parameter Types	= -17761
[NPM] Number Of Parameters Per Method	= 0.11
[NOD] Number Of Attributes	= 5
[NAD] Number Of Abstract Data Types	= 1

```
[NRA] Number Of Reference Attributes   = 1
[NPA] Number of Public Attributes      = 0
[CSB] Class Size in Bytes              = 32
[CSM] Class Size Metric                = 14

CLASS DESIGN PROPERTY MEASURES (KSIDataItem)
-------------------------------------------
[CAM] Cohesion Among Methods of Class  = 1.00
[DCC] Direct Class Coupling            = 2
[MCC] Maximum Class Coupling           = 9
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 8
[MPC] Maximum Parameter Based Coupling = 7
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


==============** DataItem **==============
 Database * db;
 int itemID;
 LockQueue rwlock;
Size: DataItem  24 = 0 + 24

public:
 inline int itemID ( );
 DataItem ( );
 virtual DataItem ( );
```

```
protected:
 inline void readlock ( );
 inline void readunlock ( );
 inline void writelock ( );
 inline void writeunlock ( );
 void notifyChange ( DataItem * );
```

CLASS EXTERNAL MEASURES (DataItem)

```
[DOI] Depth of Inheritance              = 0
[NOC] Number of Children                = 0
[NOA] Number of Ancestors               = 0
```

CLASS INTERNAL MEASURES (DataItem)

```
[NOM] Number of Methods                 = 8
[CIS] Class Interface Size              = 3
[NOT] Number Of Trivial Methods         = 5
[NOP] Number Of Polymorphic Methods     = 1
[NOO] Number Of Operator Methods        = 4
[NPT] Number Of Parameter Types         = 1
[NPM] Number Of Parameters Per Method   = 0.13
[NOD] Number Of Attributes              = 3
[NAD] Number Of Abstract Data Types     = 1
[NRA] Number Of Reference Attributes    = 1
[NPA] Number of Public Attributes       = 0
```

```
[CSB] Class Size in Bytes                = 24
[CSM] Class Size Metric                  = 11

CLASS DESIGN PROPERTY MEASURES (DataItem)
-----------------------------------------
[CAM] Cohesion Among Methods of Class  = 1.13
[DCC] Direct Class Coupling            = 2
[MCC] Maximum Class Coupling           = 9
[DAC] Direct Attribute Based Coupling  = 1
[MAC] Maximum Attribute Based Coupling = 8
[MPC] Maximum Parameter Based Coupling = 7
[VOM] Virtuality of Methods            = 1
[CCN] Class Complexity Based on Nodes  = 0


=============** Pattern **==============
public:
 void waitForMatch ( );
 void signalMatch ( );
 boolean isMatched ( );
 void resetMatches ( );
 void clearPattern ( );
 Pattern ( );
 Pattern ( );

CLASS EXTERNAL MEASURES (Pattern)
---------------------------------------
```

```
[DOI] Depth of Inheritance              = 0

[NOC] Number of Children                = 0

[NOA] Number of Ancestors               = 0


CLASS INTERNAL MEASURES (Pattern)

----------------------------------------

[NOM] Number of Methods                 = 7

[CIS] Class Interface Size              = 7

[NOT] Number Of Trivial Methods         = 0

[NOP] Number Of Polymorphic Methods     = 0

[NOO] Number Of Operator Methods        = 0

[NPT] Number Of Parameter Types         = 0

[NPM] Number Of Parameters Per Method   = 0.00

[NOD] Number Of Attributes              = 0

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 0

[NPA] Number of Public Attributes       = 0

[CSB] Class Size in Bytes               = 0

[CSM] Class Size Metric                 = 7


CLASS DESIGN PROPERTY MEASURES (Pattern)

----------------------------------------

[CAM] Cohesion Among Methods of Class   = 1

[DCC] Direct Class Coupling             = 0

[MCC] Maximum Class Coupling            = 0

[DAC] Direct Attribute Based Coupling   = 0

[MAC] Maximum Attribute Based Coupling  = 0
```

```
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


==============** Database **==============
 LockQueue rwlock;
 Control * ctl;
Size: Database  20 = 0 + 20

public:
 void clear ( );
 void add ( DataItem * );
 void remove ( DataItem * );
 Database ( Control * );
 Database ( );

CLASS EXTERNAL MEASURES (Database)
----------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0

CLASS INTERNAL MEASURES (Database)
----------------------------------------
[NOM] Number of Methods                = 5
[CIS] Class Interface Size             = 5
```

```
[NOT] Number Of Trivial Methods         = 0
[NOP] Number Of Polymorphic Methods     = 0
[NOO] Number Of Operator Methods        = 0
[NPT] Number Of Parameter Types         = 3
[NPM] Number Of Parameters Per Method   = 0.60
[NOD] Number Of Attributes              = 2
[NAD] Number Of Abstract Data Types     = 1
[NRA] Number Of Reference Attributes    = 1
[NPA] Number of Public Attributes       = 0
[CSB] Class Size in Bytes               = 20
[CSM] Class Size Metric                 = 7

CLASS DESIGN PROPERTY MEASURES (Database)
-----------------------------------------
[CAM] Cohesion Among Methods of Class   = 0.53
[DCC] Direct Class Coupling             = 2
[MCC] Maximum Class Coupling            = 9
[DAC] Direct Attribute Based Coupling   = 1
[MAC] Maximum Attribute Based Coupling  = 8
[MPC] Maximum Parameter Based Coupling  = 7
[VOM] Virtuality of Methods             = 0
[CCN] Class Complexity Based on Nodes   = 0


==============** Gen **==============

 Pattern instPattern;
```

```
public:
 Pattern * getInstPattern ( );
 void run ( KSI * );
 Gen ( );
 Gen ( );
```

CLASS EXTERNAL MEASURES (Gen)

[DOI] Depth of Inheritance	= 1
[NOC] Number of Children	= 0
[NOA] Number of Ancestors	= 1

CLASS INTERNAL MEASURES (Gen)

[NOM] Number of Methods	= 8
[CIS] Class Interface Size	= 8
[NOT] Number Of Trivial Methods	= 2
[NOP] Number Of Polymorphic Methods	= 3
[NOO] Number Of Operator Methods	= 0
[NPT] Number Of Parameter Types	= -17761
[NPM] Number Of Parameters Per Method	= 0.25
[NOD] Number Of Attributes	= 1
[NAD] Number Of Abstract Data Types	= 1
[NRA] Number Of Reference Attributes	= 0
[NPA] Number of Public Attributes	= 0
[CSB] Class Size in Bytes	= 0
[CSM] Class Size Metric	= 9

```
CLASS DESIGN PROPERTY MEASURES (Gen)

----------------------------------------

[CAM] Cohesion Among Methods of Class  = 0.63

[DCC] Direct Class Coupling            = 0

[MCC] Maximum Class Coupling           = 0

[DAC] Direct Attribute Based Coupling  = 0

[MAC] Maximum Attribute Based Coupling = 0

[MPC] Maximum Parameter Based Coupling = 0

[VOM] Virtuality of Methods            = 0

[CCN] Class Complexity Based on Nodes  = 0
```

```
==========** Stat **==========
 Pattern instPattern;

public:
 Pattern * getInstPattern ( );
 void run ( KSI * );
 Stat ( );
 Stat ( );

private:
 void extractOldStats ( Database *, IntegerDataItem &,
                        IntegerDataItem &, IntegerDataItem & );
 void makeCondPattern ( Pattern * );
 void makeStatPattern ( Pattern &, boolean );
```

```
CLASS EXTERNAL MEASURES (Stat)

----------------------------------------

[DOI] Depth of Inheritance             = 1

[NOC] Number of Children               = 0

[NOA] Number of Ancestors              = 1


CLASS INTERNAL MEASURES (Stat)

----------------------------------------

[NOM] Number of Methods                = 11

[CIS] Class Interface Size             = 8

[NOT] Number Of Trivial Methods        = 2

[NOP] Number Of Polymorphic Methods    = 3

[NOO] Number Of Operator Methods       = 0

[NPT] Number Of Parameter Types        = -17757

[NPM] Number Of Parameters Per Method  = 0.82

[NOD] Number Of Attributes             = 1

[NAD] Number Of Abstract Data Types    = 1

[NRA] Number Of Reference Attributes   = 0

[NPA] Number of Public Attributes      = 0

[CSB] Class Size in Bytes              = 0

[CSM] Class Size Metric                = 12


CLASS DESIGN PROPERTY MEASURES (Stat)

----------------------------------------

[CAM] Cohesion Among Methods of Class  = 0.45

[DCC] Direct Class Coupling            = 2
```

```
[MCC] Maximum Class Coupling            = 6
[DAC] Direct Attribute Based Coupling   = 0
[MAC] Maximum Attribute Based Coupling = 3
[MPC] Maximum Parameter Based Coupling = 6
[VOM] Virtuality of Methods             = 0
[CCN] Class Complexity Based on Nodes   = 0


=============** KS **==============
public:
 virtual Pattern * getInstPattern ( ) = 0;
 virtual void run ( KSI * ) = 0;
 inline KS ( );
 inline virtual KS ( );

CLASS EXTERNAL MEASURES (KS)
----------------------------------------
[DOI] Depth of Inheritance              = 0
[NOC] Number of Children                = 0
[NOA] Number of Ancestors               = 0

CLASS INTERNAL MEASURES (KS)
----------------------------------------
[NOM] Number of Methods                 = 4
[CIS] Class Interface Size              = 4
[NOT] Number Of Trivial Methods         = 2
[NOP] Number Of Polymorphic Methods     = 3
```

```
[NOO] Number Of Operator Methods        = 0

[NPT] Number Of Parameter Types         = 1

[NPM] Number Of Parameters Per Method   = 0.25

[NOD] Number Of Attributes              = 0

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 0

[NPA] Number of Public Attributes       = 0

[CSB] Class Size in Bytes               = 0

[CSM] Class Size Metric                 = 4


CLASS DESIGN PROPERTY MEASURES (KS)

----------------------------------------

[CAM] Cohesion Among Methods of Class   = 1.25

[DCC] Direct Class Coupling             = 0

[MCC] Maximum Class Coupling            = 0

[DAC] Direct Attribute Based Coupling   = 0

[MAC] Maximum Attribute Based Coupling  = 0

[MPC] Maximum Parameter Based Coupling  = 0

[VOM] Virtuality of Methods             = 3

[CCN] Class Complexity Based on Nodes   = 0



==============** Control **==============

 KSPat;

 KS * ks;

 Pattern * pat;

 KSIPat;
```

```
 KSI * ksi;
 Pattern * pat;
 Database * db;
 LockQueue rwlock;

public:
 void registerKS ( KS * );
 void unregisterKS ( KS * );
 void regCondPattern ( KSI *, Pattern * );
 void unregCondPattern ( KSI * );
 void killKSI ( KSI * );
 void killAllKSI ( );
 Database * getDatabase ( );
 Control ( );
 Control ( );

private:
 void destroyInstance ( KSI * );
```

CLASS EXTERNAL MEASURES (Control)

[DOI] Depth of Inheritance = 0

[NOC] Number of Children = 0

[NOA] Number of Ancestors = 0

CLASS INTERNAL MEASURES (Control)

```
[NOM] Number of Methods                 = 10
[CIS] Class Interface Size              = 9
[NOT] Number Of Trivial Methods         = 0
[NOP] Number Of Polymorphic Methods     = 0
[NOO] Number Of Operator Methods        = 0
[NPT] Number Of Parameter Types         = 4
[NPM] Number Of Parameters Per Method   = 0.70
[NOD] Number Of Attributes              = 6
[NAD] Number Of Abstract Data Types     = 1
[NRA] Number Of Reference Attributes    = 5
[NPA] Number of Public Attributes       = 0
[CSB] Class Size in Bytes               = 36
[CSM] Class Size Metric                 = 16

CLASS DESIGN PROPERTY MEASURES (Control)
----------------------------------------
[CAM] Cohesion Among Methods of Class   = 0.42
[DCC] Direct Class Coupling             = 3
[MCC] Maximum Class Coupling            = 5
[DAC] Direct Attribute Based Coupling   = 3
[MAC] Maximum Attribute Based Coupling  = 4
[MPC] Maximum Parameter Based Coupling  = 4
[VOM] Virtuality of Methods             = 0
[CCN] Class Complexity Based on Nodes   = 0


===============Class Relationships Window===============
```

```
LockQueue

DataItem

   DataItem

   Database

DataItem

KSIDataItem

Pattern

Database

   DataItem
```

Appendix C

Metrics for Program "PROJECT"

Table C.1: Metrics for Attributes and Methods

Class_Name	AA	AI	AD	ANU	MC	MA	MI	MD	MNU
DataItem	3	–	3	–	–	8	–	8	–
KSIDataItem	2	3	5	3	0	1	8	9	2
KS	0	–	0	–	–	4	–	4	–
Gen	1	0	1	0	2	2	2	6	2
Stat	1	0	1	0	2	5	2	9	2

Table C.2: Metrics Based on Inheritance

Class_Name	HIC	HIA	HID
DataItem_tree	0	0.2222	0.5919
KS_tree	0.2353	0.5294	0.2353

Table C.3: Metrics for Virtual Methods in *DataItem_tree*

Class_Name	NIV	NSV	NOLO	PFOM	ACRV
DataItem	1	virtual ~DataItem() NSV=0.5	0	0	0
KSIDataItem	0	NSV=0	0	–	–

Table C.4: Metrics Based for Virtual Methods in *KS_tree*

Class_Name	NIV	NSV			NOLO	PFOM	ACRV
KS	3	virtual *getInt-Pattern() NSV=0.5	virtual run() NSV=0.5	virtual ~KS() NSV=0.5	0	0.6667	41
Gen	2	NSV=4	NSV=26	NSV=0.5	0	–	–
Stat	2	NSV=4	NSV=21	NSV=0.5	0	–	–

Table C.5: Metrics Based on Polymorphism

Class_Name	PFOM	ACRV
DataItem_tree	0	0
KS_tree	0.6667	41.00

Table C.6: Metrics Based on Interactions

Class_Name	NCP	NTC	NSUP	NDC	NPC	NRC	NPAR	MD	AD	IFPV	IFCV
LockQueue	0	0	0	0	0	0	0	6	4	0	0
DataItem	1	0	0	1	1	0	1	8	3	0.1111	0.1667
KSIDataItem	1	0	1	1	1	0	1	9	5	0.2222	0.1333
Pattern	0	0	0	0	0	0	0	7	0	0	0
Database	2	0	0	1	4	0	3	5	2	0.2222	0.5000
KS	2	0	0	0	2	0	1	4	0	0.2222	0.4000
Gen	2	0	1	1	2	0	1	6	1	0.3333	0.3750
Stat	3	0	1	1	4	4	8	9	1	0.4444	0.5000
Control	4	0	0	1	13	0	7	10	8	0.4444	0.5600

Appendix D

Data Extracted from Program "SPAS"

This appendix includes data extracted from program "SPAS" using QMOOD++ and CCCC.

```
===============Display Class Hierachy==============


  wxObject
      wxPrintData
      wxEvent
      wxSystemEventClassStruc
      wxSystemEventNameStruc
      wxToolBarTool
  DarrochAnalysis
  LeastSqrAnalysis
  MomentAnalysis
```

```
  PPetersenAnalysis
  SchaeferAnalysis
  DarrochResults
  Environment
  FormatSet
  IndexSet
  LabelSet
  LeastSqrResults
  MomentResults
  PPetersenResults
  Replicator
      Simulation
  ResultSet
  SchaeferResults
  wxRegex
  wxSubString
  wxString

Number of Classes Defined = 26
Number of Base Classes    = 2


===============Class Hierarchy Window===============


================** wxObject **================
 int refCount;
```

```
public:
 wxObject ( );
 virtual wxObject ( );
 inline int refCount ( );
 inline void ref ( int ref );
 inline int refCount ( ) const;
 inline int refCount ( ) const;
```

CLASS EXTERNAL MEASURES (wxObject)

[DOI] Depth of Inheritance = 0

[NOC] Number of Children = 0

[NOA] Number of Ancestors = 0

CLASS INTERNAL MEASURES (wxObject)

[NOM] Number of Methods = 6

[CIS] Class Interface Size = 6

[NOT] Number Of Trivial Methods = 4

[NOP] Number Of Polymorphic Methods = 1

[NOO] Number Of Operator Methods = 3

[NPT] Number Of Parameter Types = 1

[NPM] Number Of Parameters Per Method = 0.17

[NOD] Number Of Attributes = 1

[NAD] Number Of Abstract Data Types = 0

[NRA] Number Of Reference Attributes = 0

[NPA] Number of Public Attributes = 0

```
[CSB] Class Size in Bytes              = 4
[CSM] Class Size Metric                = 7

CLASS DESIGN PROPERTY MEASURES (wxObject)
-----------------------------------------
[CAM] Cohesion Among Methods of Class  = 1.17
[DCC] Direct Class Coupling            = 0
[MCC] Maximum Class Coupling           = 0
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 1
[CCN] Class Complexity Based on Nodes  = 0


================** wxPrintData **================
 int printFromPage;
 int printToPage;
 int printMinPage;
 int printMaxPage;
 int printNoCopies;

public:
 wxPrintData ( );
 wxPrintData ( );
 int GetFromPage ( );
 int GetToPage ( );
```

```
 int GetMinPage ( );
 int GetMaxPage ( );
 int GetNoCopies ( );
 void SetFromPage ( int private 19 );
 void SetToPage ( int private 20 );
 void SetMinPage ( int private 21 );
 void SetMaxPage ( int private 22 );
 void SetNoCopies ( int private 23 );
 void operator = ( const wxPrintData & );

CLASS EXTERNAL MEASURES (wxPrintData)
-----------------------------------------
[DOI] Depth of Inheritance             = 1
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 1

CLASS INTERNAL MEASURES (wxPrintData)
-----------------------------------------
[NOM] Number of Methods                = 19
[CIS] Class Interface Size             = 19
[NOT] Number Of Trivial Methods        = 4
[NOP] Number Of Polymorphic Methods    = 1
[NOO] Number Of Operator Methods       = 4
[NPT] Number Of Parameter Types        = -17760
[NPM] Number Of Parameters Per Method  = 0.37
[NOD] Number Of Attributes             = 6
[NAD] Number Of Abstract Data Types    = 0
```

```
[NRA] Number Of Reference Attributes   = 0
[NPA] Number of Public Attributes      = 5
[CSB] Class Size in Bytes              = 24
[CSM] Class Size Metric                = 25

CLASS DESIGN PROPERTY MEASURES (wxPrintData)
-------------------------------------------
[CAM] Cohesion Among Methods of Class  = 0.37
[DCC] Direct Class Coupling            = 1
[MCC] Maximum Class Coupling           = 1
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 1
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** wxEvent **===============
 char * eventHandle;
 WXTYPE eventType;
 WXTYPE eventClass;
 WXTYPE objectType;
 wxObject * eventObject;
 long timeStamp;

public:
 wxEvent ( );
```

```
 wxEvent ( );
 inline WXTYPE eventType ( );
 inline WXTYPE eventClass ( );
 inline WXTYPE objectType ( );
 inline wxObject * eventObject ( );
 inline virtual long timeStamp ( );
 virtual void SetTimestamp ( const long int ts );
```

CLASS EXTERNAL MEASURES (wxEvent)

[DOI] Depth of Inheritance = 1

[NOC] Number of Children = 0

[NOA] Number of Ancestors = 1

CLASS INTERNAL MEASURES (wxEvent)

[NOM] Number of Methods = 14

[CIS] Class Interface Size = 14

[NOT] Number Of Trivial Methods = 9

[NOP] Number Of Polymorphic Methods = 3

[NOO] Number Of Operator Methods = 3

[NPT] Number Of Parameter Types = -17761

[NPM] Number Of Parameters Per Method = 0.14

[NOD] Number Of Attributes = 7

[NAD] Number Of Abstract Data Types = 0

[NRA] Number Of Reference Attributes = 2

[NPA] Number of Public Attributes = 6

```
[CSB] Class Size in Bytes              = 22
[CSM] Class Size Metric                = 21

CLASS DESIGN PROPERTY MEASURES (wxEvent)
----------------------------------------
[CAM] Cohesion Among Methods of Class  = 0.50
[DCC] Direct Class Coupling            = 1
[MCC] Maximum Class Coupling           = 1
[DAC] Direct Attribute Based Coupling  = 1
[MAC] Maximum Attribute Based Coupling = 1
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 2
[CCN] Class Complexity Based on Nodes  = 0


================** wxSystemEventClassStruc **================
 WXTYPE eventClass;
 wxEventConstructor eventConstructor;
 char * eventDescription;

public:
 inline wxSystemEventClassStruc ( );
 inline eventDescription ( );

CLASS EXTERNAL MEASURES (wxSystemEventClassStruc)
-------------------------------------------
[DOI] Depth of Inheritance             = 1
```

```
[NOC] Number of Children                = 0

[NOA] Number of Ancestors               = 1


CLASS INTERNAL MEASURES (wxSystemEventClassStruc)

-----------------------------------------

[NOM] Number of Methods                 = 8

[CIS] Class Interface Size              = 8

[NOT] Number Of Trivial Methods         = 6

[NOP] Number Of Polymorphic Methods     = 1

[NOO] Number Of Operator Methods        = 4

[NPT] Number Of Parameter Types         = -17761

[NPM] Number Of Parameters Per Method   = 0.13

[NOD] Number Of Attributes              = 4

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 1

[NPA] Number of Public Attributes       = 3

[CSB] Class Size in Bytes               = 10

[CSM] Class Size Metric                 = 12


CLASS DESIGN PROPERTY MEASURES (wxSystemEventClassStruc)

-----------------------------------------

[CAM] Cohesion Among Methods of Class   = 0.88

[DCC] Direct Class Coupling             = 0

[MCC] Maximum Class Coupling            = 0

[DAC] Direct Attribute Based Coupling   = 0

[MAC] Maximum Attribute Based Coupling  = 0

[MPC] Maximum Parameter Based Coupling  = 0
```

```
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================**  wxSystemEventNameStruc **================
 WXTYPE eventClass;
 WXTYPE eventType;
 char * eventName;

public:
 inline wxSystemEventNameStruc ( );
 inline eventName ( );

CLASS EXTERNAL MEASURES (wxSystemEventNameStruc)
-----------------------------------------
[DOI] Depth of Inheritance             = 1
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 1

CLASS INTERNAL MEASURES (wsSystemEventNameStruc)
-----------------------------------------
[NOM] Number of Methods                = 8
[CIS] Class Interface Size             = 8
[NOT] Number Of Trivial Methods        = 6
[NOP] Number Of Polymorphic Methods    = 1
[NOO] Number Of Operator Methods       = 4
[NPT] Number Of Parameter Types        = -17761
```

```
[NPM] Number Of Parameters Per Method  = 0.13
[NOD] Number Of Attributes             = 4
[NAD] Number Of Abstract Data Types    = 0
[NRA] Number Of Reference Attributes   = 1
[NPA] Number of Public Attributes      = 3
[CSB] Class Size in Bytes              = 12
[CSM] Class Size Metric                = 12

CLASS DESIGN PROPERTY MEASURES (wxSystemEventNameStruc)
-------------------------------------------
[CAM] Cohesion Among Methods of Class  = 0.88
[DCC] Direct Class Coupling            = 0
[MCC] Maximum Class Coupling           = 0
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** wxToolBarTool **================
 int toolStyle;
 wxObject * clientData;
 int index;
 float x;
 float y;
 float width;
```

```
 float height;
 char * shortHelpString;
 char * longHelpString;

public:
 wxToolBarTool ( );
 inline void h ( float w, float h );
 inline float width ( );
 inline float height ( );
```

```
CLASS EXTERNAL MEASURES (wxToolBarTool)
-----------------------------------------
[DOI] Depth of Inheritance             = 1
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 1
```

```
CLASS INTERNAL MEASURES (wxToolBarTool)
-----------------------------------------
[NOM] Number of Methods                = 10
[CIS] Class Interface Size             = 10
[NOT] Number Of Trivial Methods        = 7
[NOP] Number Of Polymorphic Methods    = 1
[NOO] Number Of Operator Methods       = 4
[NPT] Number Of Parameter Types        = -17760
[NPM] Number Of Parameters Per Method  = 0.30
[NOD] Number Of Attributes             = 10
[NAD] Number Of Abstract Data Types    = 0
```

```
[NRA] Number Of Reference Attributes   = 3

[NPA] Number of Public Attributes      = 9

[CSB] Class Size in Bytes              = 40

[CSM] Class Size Metric                = 20


CLASS DESIGN PROPERTY MEASURES (wxToolBarTool)

----------------------------------------

[CAM] Cohesion Among Methods of Class  = 0.70

[DCC] Direct Class Coupling            = 1

[MCC] Maximum Class Coupling           = 1

[DAC] Direct Attribute Based Coupling  = 1

[MAC] Maximum Attribute Based Coupling = 1

[MPC] Maximum Parameter Based Coupling = 0

[VOM] Virtuality of Methods            = 0

[CCN] Class Complexity Based on Nodes  = 0



================** DarrochAnalysis **================

public:

 DarrochAnalysis ( );

 void failure ( );


CLASS EXTERNAL MEASURES (DarrochAnalysis)

----------------------------------------

[DOI] Depth of Inheritance             = 0

[NOC] Number of Children               = 0
```

```
[NOA] Number of Ancestors                = 0


CLASS INTERNAL MEASURES (DarrochAnalysis)

-----------------------------------------

[NOM] Number of Methods                  = 2

[CIS] Class Interface Size               = 2

[NOT] Number Of Trivial Methods          = 0

[NOP] Number Of Polymorphic Methods      = 0

[NOO] Number Of Operator Methods         = 0

[NPT] Number Of Parameter Types          = 0

[NPM] Number Of Parameters Per Method    = 0.00

[NOD] Number Of Attributes               = 0

[NAD] Number Of Abstract Data Types      = 0

[NRA] Number Of Reference Attributes     = 0

[NPA] Number of Public Attributes        = 0

[CSB] Class Size in Bytes                = 0

[CSM] Class Size Metric                  = 2


CLASS DESIGN PROPERTY MEASURES (DarrochAnalysis)

------------------------------------------

[CAM] Cohesion Among Methods of Class    = 1

[DCC] Direct Class Coupling              = 0

[MCC] Maximum Class Coupling             = 0

[DAC] Direct Attribute Based Coupling    = 0

[MAC] Maximum Attribute Based Coupling   = 0

[MPC] Maximum Parameter Based Coupling   = 0

[VOM] Virtuality of Methods              = 0
```

```
[CCN] Class Complexity Based on Nodes  = 0


================** LeastSqrAnalysis **================
public:
 LeastSqrAnalysis ( );
 void failure ( );

CLASS EXTERNAL MEASURES (LeastSqrAnalysis)
------------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0

CLASS INTERNAL MEASURES (LeastSqrAnalysis)
------------------------------------------
[NOM] Number of Methods                = 2
[CIS] Class Interface Size             = 2
[NOT] Number Of Trivial Methods        = 0
[NOP] Number Of Polymorphic Methods    = 0
[NOO] Number Of Operator Methods       = 0
[NPT] Number Of Parameter Types        = 0
[NPM] Number Of Parameters Per Method  = 0.00
[NOD] Number Of Attributes             = 0
[NAD] Number Of Abstract Data Types    = 0
[NRA] Number Of Reference Attributes   = 0
[NPA] Number of Public Attributes      = 0
```

```
[CSB] Class Size in Bytes              = 0
[CSM] Class Size Metric                = 2

CLASS DESIGN PROPERTY MEASURES (LeastSqrAnalysis)
-------------------------------------------
[CAM] Cohesion Among Methods of Class  = 1
[DCC] Direct Class Coupling            = 0
[MCC] Maximum Class Coupling           = 0
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** MomentAnalysis **================
public:
 MomentAnalysis ( );
 void failure ( );

CLASS EXTERNAL MEASURES (MomentAnalysis)
-----------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0

CLASS INTERNAL MEASURES (MomentAnalysis)
```

[NOM] Number of Methods = 2

[CIS] Class Interface Size = 2

[NOT] Number Of Trivial Methods = 0

[NOP] Number Of Polymorphic Methods = 0

[NOO] Number Of Operator Methods = 0

[NPT] Number Of Parameter Types = 0

[NPM] Number Of Parameters Per Method = 0.00

[NOD] Number Of Attributes = 0

[NAD] Number Of Abstract Data Types = 0

[NRA] Number Of Reference Attributes = 0

[NPA] Number of Public Attributes = 0

[CSB] Class Size in Bytes = 0

[CSM] Class Size Metric = 2

CLASS DESIGN PROPERTY MEASURES (MomentAnalysis)

[CAM] Cohesion Among Methods of Class = 1

[DCC] Direct Class Coupling = 0

[MCC] Maximum Class Coupling = 0

[DAC] Direct Attribute Based Coupling = 0

[MAC] Maximum Attribute Based Coupling = 0

[MPC] Maximum Parameter Based Coupling = 0

[VOM] Virtuality of Methods = 0

[CCN] Class Complexity Based on Nodes = 0

```
================** PPetersenAnalysis **================

public:

 PPetersenAnalysis ( );

 void failure ( );


CLASS EXTERNAL MEASURES (PPetersenAnalysis)

-----------------------------------------

[DOI] Depth of Inheritance              = 0

[NOC] Number of Children                = 0

[NOA] Number of Ancestors               = 0


CLASS INTERNAL MEASURES (PPetersenAnalysis)

-----------------------------------------

[NOM] Number of Methods                 = 2

[CIS] Class Interface Size              = 2

[NOT] Number Of Trivial Methods         = 0

[NOP] Number Of Polymorphic Methods     = 0

[NOO] Number Of Operator Methods        = 0

[NPT] Number Of Parameter Types         = 0

[NPM] Number Of Parameters Per Method   = 0.00

[NOD] Number Of Attributes              = 0

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 0

[NPA] Number of Public Attributes       = 0

[CSB] Class Size in Bytes               = 0

[CSM] Class Size Metric                 = 2
```

```
CLASS DESIGN PROPERTY MEASURES (PPetersenAnalysis)
-------------------------------------------
[CAM] Cohesion Among Methods of Class  = 1
[DCC] Direct Class Coupling            = 0
[MCC] Maximum Class Coupling           = 0
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** SchaeferAnalysis **================
public:
 SchaeferAnalysis ( );
 void failure ( );

CLASS EXTERNAL MEASURES (SchaeferAnalysis)
-------------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0

CLASS INTERNAL MEASURES (SchaeferAnalysis)
-------------------------------------------
[NOM] Number of Methods                = 2
[CIS] Class Interface Size             = 2
```

```
[NOT] Number Of Trivial Methods         = 0

[NOP] Number Of Polymorphic Methods     = 0

[NOO] Number Of Operator Methods        = 0

[NPT] Number Of Parameter Types         = 0

[NPM] Number Of Parameters Per Method   = 0.00

[NOD] Number Of Attributes              = 0

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 0

[NPA] Number of Public Attributes       = 0

[CSB] Class Size in Bytes               = 0

[CSM] Class Size Metric                 = 2


CLASS DESIGN PROPERTY MEASURES (SchaeferAnalysis)

-----------------------------------------

[CAM] Cohesion Among Methods of Class   = 1

[DCC] Direct Class Coupling             = 0

[MCC] Maximum Class Coupling            = 0

[DAC] Direct Attribute Based Coupling   = 0

[MAC] Maximum Attribute Based Coupling  = 0

[MPC] Maximum Parameter Based Coupling  = 0

[VOM] Virtuality of Methods             = 0

[CCN] Class Complexity Based on Nodes   = 0



================** DarrochResults **===============

 int fail_flag;
```

```
public:
 DarrochResults ( int s, int t );
 DarrochResults ( );

CLASS EXTERNAL MEASURES (DarrochResults)
-----------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0

CLASS INTERNAL MEASURES (DarrochResults)
-----------------------------------------
[NOM] Number of Methods                = 2
[CIS] Class Interface Size             = 2
[NOT] Number Of Trivial Methods        = 0
[NOP] Number Of Polymorphic Methods    = 0
[NOO] Number Of Operator Methods       = 0
[NPT] Number Of Parameter Types        = 2
[NPM] Number Of Parameters Per Method  = 1.00
[NOD] Number Of Attributes             = 1
[NAD] Number Of Abstract Data Types    = 0
[NRA] Number Of Reference Attributes   = 0
[NPA] Number of Public Attributes      = 1
[CSB] Class Size in Bytes              = 4
[CSM] Class Size Metric                = 3

CLASS DESIGN PROPERTY MEASURES (DarrochResults)
```

```
-----------------------------------------

[CAM] Cohesion Among Methods of Class  = 0.75

[DCC] Direct Class Coupling            = 0

[MCC] Maximum Class Coupling           = 0

[DAC] Direct Attribute Based Coupling  = 0

[MAC] Maximum Attribute Based Coupling = 0

[MPC] Maximum Parameter Based Coupling = 0

[VOM] Virtuality of Methods            = 0

[CCN] Class Complexity Based on Nodes  = 0



================** Environment **================

public:


CLASS EXTERNAL MEASURES (Environment)

-----------------------------------------

[DOI] Depth of Inheritance             = 0

[NOC] Number of Children               = 0

[NOA] Number of Ancestors              = 0


CLASS INTERNAL MEASURES (Environment)

-----------------------------------------

[NOM] Number of Methods                = 0

[CIS] Class Interface Size             = 0

[NOT] Number Of Trivial Methods        = 0

[NOP] Number Of Polymorphic Methods    = 0

[NOO] Number Of Operator Methods       = 0
```

```
[NPT] Number Of Parameter Types        = 0
[NOD] Number Of Attributes             = 0
[NAD] Number Of Abstract Data Types    = 0
[NRA] Number Of Reference Attributes   = 0
[NPA] Number of Public Attributes      = 0
[CSB] Class Size in Bytes              = 0
[CSM] Class Size Metric                = 0

CLASS DESIGN PROPERTY MEASURES (Environment)
-------------------------------------------
[CAM] Cohesion Among Methods of Class  = 1.00
[DCC] Direct Class Coupling            = 0
[MCC] Maximum Class Coupling           = 0
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** FormatSet **================
 int datawidth;
 int dataprec;
 int resultswidth;
 int resultsprec;
 int resultsprep;
 int displaycovar;
```

```
public:
 FormatSet ( const int dw, const int dp, const int rw, const int rp,
             const int dc, const int rpp );

CLASS EXTERNAL MEASURES (FormatSet)
-------------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0

CLASS INTERNAL MEASURES (FormatSet)
-------------------------------------------
[NOM] Number of Methods                = 1
[CIS] Class Interface Size             = 1
[NOT] Number Of Trivial Methods        = 0
[NOP] Number Of Polymorphic Methods    = 0
[NOO] Number Of Operator Methods       = 0
[NPT] Number Of Parameter Types        = 2
[NPM] Number Of Parameters Per Method  = 6.00
[NOD] Number Of Attributes             = 6
[NAD] Number Of Abstract Data Types    = 0
[NRA] Number Of Reference Attributes   = 0
[NPA] Number of Public Attributes      = 6
[CSB] Class Size in Bytes              = 24
[CSM] Class Size Metric                = 7
```

```
CLASS DESIGN PROPERTY MEASURES (FormatSet)
-----------------------------------------
[CAM] Cohesion Among Methods of Class  = 1.00
[DCC] Direct Class Coupling            = 0
[MCC] Maximum Class Coupling           = 0
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** IndexSet **================
 int * set;
 int size;

public:
 IndexSet ( int private 28 );
 IndexSet ( );
 int * operator [] ( int elem );
 IndexSet & operator = ( const IndexSet & );

CLASS EXTERNAL MEASURES (IndexSet)
-----------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0
```

```
CLASS INTERNAL MEASURES (IndexSet)

-----------------------------------------

[NOM] Number of Methods                = 4

[CIS] Class Interface Size             = 4

[NOT] Number Of Trivial Methods        = 0

[NOP] Number Of Polymorphic Methods    = 0

[NOO] Number Of Operator Methods       = 2

[NPT] Number Of Parameter Types        = 3

[NPM] Number Of Parameters Per Method  = 0.75

[NOD] Number Of Attributes             = 2

[NAD] Number Of Abstract Data Types    = 0

[NRA] Number Of Reference Attributes   = 1

[NPA] Number of Public Attributes      = 0

[CSB] Class Size in Bytes              = 8

[CSM] Class Size Metric                = 6


CLASS DESIGN PROPERTY MEASURES (IndexSet)

-----------------------------------------

[CAM] Cohesion Among Methods of Class  = 0.58

[DCC] Direct Class Coupling            = 1

[MCC] Maximum Class Coupling           = 1

[DAC] Direct Attribute Based Coupling  = 0

[MAC] Maximum Attribute Based Coupling = 0

[MPC] Maximum Parameter Based Coupling = 1

[VOM] Virtuality of Methods            = 0

[CCN] Class Complexity Based on Nodes  = 0
```

```
================** LabelSet **================
 int size;
 int second_length;
 int length;
 char * set;
 char * second_set;

public:
 LabelSet ( int nmlabels );
 LabelSet ( );
 char * operator [] ( int elem );
 int number ( );
 int second_number ( );
 LabelSet & operator = ( const LabelSet & );
 void copy_second_set ( );
 void consective_pool_without_group ( int begin, int end );
 void second_consective_pool_without_group ( int begin, int end );
 void non_consective_pool ( int row1, int row2 );
 void second_non_consective_pool ( int row1, int row2 );
 void second_expand ( LabelSet *, int idx *, int element, int save_length );
 void expand_one_row ( LabelSet *, int copy_location, int location,
                       LabelSet * );
 void second_expand_one_row ( int location, LabelSet * );
 void drop ( int element );
 void second_drop ( int element );
```

```
CLASS EXTERNAL MEASURES (LabelSet)

-----------------------------------------

[DOI] Depth of Inheritance              = 0

[NOC] Number of Children                = 0

[NOA] Number of Ancestors               = 0


CLASS INTERNAL MEASURES (LabelSet)

-----------------------------------------

[NOM] Number of Methods                 = 16

[CIS] Class Interface Size              = 16

[NOT] Number Of Trivial Methods         = 0

[NOP] Number Of Polymorphic Methods     = 0

[NOO] Number Of Operator Methods        = 2

[NPT] Number Of Parameter Types         = 4

[NPM] Number Of Parameters Per Method   = 1.44

[NOD] Number Of Attributes              = 5

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 2

[NPA] Number of Public Attributes       = 3

[CSB] Class Size in Bytes               = 20

[CSM] Class Size Metric                 = 21


CLASS DESIGN PROPERTY MEASURES (LabelSet)

-----------------------------------------

[CAM] Cohesion Among Methods of Class   = 0.48

[DCC] Direct Class Coupling             = 1
```

```
[MCC] Maximum Class Coupling           = 1
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 1
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** LeastSqrResults **================
 double estimate;
 double variance;
 double chisqr;
 double gsqr;
 double mixstat;
 double markstat;
 int fail_flag;

public:
 LeastSqrResults ( int s, int t );
 LeastSqrResults ( );

CLASS EXTERNAL MEASURES (LeastSqrResults)
-----------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0
```

```
CLASS INTERNAL MEASURES (LeastSqrResults)

----------------------------------------

[NOM] Number of Methods                 = 2

[CIS] Class Interface Size              = 2

[NOT] Number Of Trivial Methods         = 0

[NOP] Number Of Polymorphic Methods     = 0

[NOO] Number Of Operator Methods        = 0

[NPT] Number Of Parameter Types         = 2

[NPM] Number Of Parameters Per Method   = 1.00

[NOD] Number Of Attributes              = 7

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 0

[NPA] Number of Public Attributes       = 7

[CSB] Class Size in Bytes               = 52

[CSM] Class Size Metric                 = 9


CLASS DESIGN PROPERTY MEASURES (LeastSqrResults)

----------------------------------------

[CAM] Cohesion Among Methods of Class   = 0.75

[DCC] Direct Class Coupling             = 0

[MCC] Maximum Class Coupling            = 0

[DAC] Direct Attribute Based Coupling   = 0

[MAC] Maximum Attribute Based Coupling  = 0

[MPC] Maximum Parameter Based Coupling  = 0

[VOM] Virtuality of Methods             = 0

[CCN] Class Complexity Based on Nodes   = 0
```

```
================** MomentResults **================
 matrixp theta;
 double estimate;
 double variance;
 double mixstat;
 double markstat;
 int fail_flag;

public:
 MomentResults ( int s );
 MomentResults ( );
```

```
CLASS EXTERNAL MEASURES (MomentResults)
----------------------------------------
[DOI] Depth of Inheritance            = 0
[NOC] Number of Children              = 0
[NOA] Number of Ancestors             = 0

CLASS INTERNAL MEASURES (MomentResults)
----------------------------------------
[NOM] Number of Methods               = 2
[CIS] Class Interface Size            = 2
[NOT] Number Of Trivial Methods       = 0
[NOP] Number Of Polymorphic Methods   = 0
[NOO] Number Of Operator Methods      = 0
```

```
[NPT] Number Of Parameter Types        = 1
[NPM] Number Of Parameters Per Method  = 0.50
[NOD] Number Of Attributes             = 6
[NAD] Number Of Abstract Data Types    = 0
[NRA] Number Of Reference Attributes   = 0
[NPA] Number of Public Attributes      = 6
[CSB] Class Size in Bytes              = 40
[CSM] Class Size Metric                = 8

CLASS DESIGN PROPERTY MEASURES (MomentResults)
-----------------------------------------
[CAM] Cohesion Among Methods of Class  = 1.50
[DCC] Direct Class Coupling            = 0
[MCC] Maximum Class Coupling           = 0
[DAC] Direct Attribute Based Coupling  = 0
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** PPetersenResults **================
 double estimate;
 double variance;
 double mixstat;
 double markstat;
```

```
public:
 PPetersenResults ( );


CLASS EXTERNAL MEASURES (PPetersenResults)
------------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0

CLASS INTERNAL MEASURES (PPetersenResults)
------------------------------------------
[NOM] Number of Methods                = 1
[CIS] Class Interface Size             = 1
[NOT] Number Of Trivial Methods        = 0
[NOP] Number Of Polymorphic Methods    = 0
[NOO] Number Of Operator Methods       = 0
[NPT] Number Of Parameter Types        = 0
[NPM] Number Of Parameters Per Method  = 0.00
[NOD] Number Of Attributes             = 4
[NAD] Number Of Abstract Data Types    = 0
[NRA] Number Of Reference Attributes   = 0
[NPA] Number of Public Attributes      = 4
[CSB] Class Size in Bytes              = 32
[CSM] Class Size Metric                = 5

CLASS DESIGN PROPERTY MEASURES (PPetersenResults)
```

```
-----------------------------------------

[CAM] Cohesion Among Methods of Class  = 1

[DCC] Direct Class Coupling            = 0

[MCC] Maximum Class Coupling           = 0

[DAC] Direct Attribute Based Coupling  = 0

[MAC] Maximum Attribute Based Coupling = 0

[MPC] Maximum Parameter Based Coupling = 0

[VOM] Virtuality of Methods            = 0

[CCN] Class Complexity Based on Nodes  = 0



================** Replicator **================

 const  int Accumulator * allstats[5];

 int numstatistics;

 long initseed1;

 long initseed2;


public:

 Replicator ( );

 virtual Replicator ( );

 int Multinomial ( long private 30 *, long private 31 *,

                   long private 32 *, float private 33 *,

                   long private 34 *, long private 35 * );

 int Binomial ( long private 36 *, long private 37 *,

                float private 38 *, long private 39 * );


CLASS EXTERNAL MEASURES (Replicator)
```

```
-----------------------------------------

[DOI] Depth of Inheritance            = 0

[NOC] Number of Children              = 0

[NOA] Number of Ancestors             = 0


CLASS INTERNAL MEASURES (Replicator)

-----------------------------------------

[NOM] Number of Methods               = 4

[CIS] Class Interface Size            = 4

[NOT] Number Of Trivial Methods       = 0

[NOP] Number Of Polymorphic Methods   = 1

[NOO] Number Of Operator Methods      = 0

[NPT] Number Of Parameter Types       = 9

[NPM] Number Of Parameters Per Method = 2.50

[NOD] Number Of Attributes            = 4

[NAD] Number Of Abstract Data Types   = 0

[NRA] Number Of Reference Attributes  = 1

[NPA] Number of Public Attributes     = 2

[CSB] Class Size in Bytes             = 32

[CSM] Class Size Metric               = 8


CLASS DESIGN PROPERTY MEASURES (Replicator)

-----------------------------------------

[CAM] Cohesion Among Methods of Class = 0.39

[DCC] Direct Class Coupling           = 0

[MCC] Maximum Class Coupling          = 0

[DAC] Direct Attribute Based Coupling = 0
```

```
[MAC] Maximum Attribute Based Coupling = 0
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 1
[CCN] Class Complexity Based on Nodes  = 0


================** Simulation **================
 char * currentfile;
 char * filepath;
 int display_selection;
 char * old_iteration;
 char * old_converge_value;
 int dis_cov;
 int old_start_value;

public:
 Simulation ( wxFrame *, char title *, const int x, const int y,
              const int w, const int h, const int type );
 Simulation ( );
 void OnMenuCommand ( int id );
 void OnSize ( int w, int h );
 int loadFile ( );
 int stream_write ( char inter_stream *, char file_name * );
 char * formatData ( );

CLASS EXTERNAL MEASURES (Simulation)
-----------------------------------------
```

```
[DOI] Depth of Inheritance              = 1

[NOC] Number of Children                = 0

[NOA] Number of Ancestors               = 1


CLASS INTERNAL MEASURES (Simulation)

-----------------------------------------

[NOM] Number of Methods                 = 11

[CIS] Class Interface Size              = 11

[NOT] Number Of Trivial Methods         = 0

[NOP] Number Of Polymorphic Methods     = 1

[NOO] Number Of Operator Methods        = 4

[NPT] Number Of Parameter Types         = -17750

[NPM] Number Of Parameters Per Method   = 2.00

[NOD] Number Of Attributes              = 11

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 5

[NPA] Number of Public Attributes       = 7

[CSB] Class Size in Bytes               = 60

[CSM] Class Size Metric                 = 22


CLASS DESIGN PROPERTY MEASURES (Simulation)

-------------------------------------------

[CAM] Cohesion Among Methods of Class   = 0.14

[DCC] Direct Class Coupling             = 0

[MCC] Maximum Class Coupling            = 0

[DAC] Direct Attribute Based Coupling   = 0

[MAC] Maximum Attribute Based Coupling  = 0
```

```
[MPC] Maximum Parameter Based Coupling = 0
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** ResultSet **================
 matrixp betacovar;
 matrixp theta;
 double loglike;
 double estimate;
 double variance;
 double chisqr;
 double gsqr;
 double mixstat;
 double markstat;

public:
 ResultSet ( int s, int t );
 ResultSet ( );

CLASS EXTERNAL MEASURES (ResultSet)
-----------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0

CLASS INTERNAL MEASURES (ResultSet)
```

[NOM] Number of Methods = 2

[CIS] Class Interface Size = 2

[NOT] Number Of Trivial Methods = 0

[NOP] Number Of Polymorphic Methods = 0

[NOO] Number Of Operator Methods = 0

[NPT] Number Of Parameter Types = 2

[NPM] Number Of Parameters Per Method = 1.00

[NOD] Number Of Attributes = 9

[NAD] Number Of Abstract Data Types = 0

[NRA] Number Of Reference Attributes = 0

[NPA] Number of Public Attributes = 9

[CSB] Class Size in Bytes = 64

[CSM] Class Size Metric = 11

CLASS DESIGN PROPERTY MEASURES (ResultSet)

[CAM] Cohesion Among Methods of Class = 0.75

[DCC] Direct Class Coupling = 0

[MCC] Maximum Class Coupling = 0

[DAC] Direct Attribute Based Coupling = 0

[MAC] Maximum Attribute Based Coupling = 0

[MPC] Maximum Parameter Based Coupling = 0

[VOM] Virtuality of Methods = 0

[CCN] Class Complexity Based on Nodes = 0

```
================** SchaeferResults **================
 double estimate;
 double mixstat;
 double markstat;
 matrixp N_mtx;
 int fail_flag;

public:
 SchaeferResults ( int s, int t );
 SchaeferResults ( );
```

```
CLASS EXTERNAL MEASURES (SchaeferResults)
-----------------------------------------
[DOI] Depth of Inheritance             = 0
[NOC] Number of Children               = 0
[NOA] Number of Ancestors              = 0

CLASS INTERNAL MEASURES (SchaeferResults)
-----------------------------------------
[NOM] Number of Methods                = 2
[CIS] Class Interface Size             = 2
[NOT] Number Of Trivial Methods        = 0
[NOP] Number Of Polymorphic Methods    = 0
[NOO] Number Of Operator Methods       = 0
[NPT] Number Of Parameter Types        = 2
[NPM] Number Of Parameters Per Method  = 1.00
```

```
[NOD] Number Of Attributes              = 5
[NAD] Number Of Abstract Data Types     = 0
[NRA] Number Of Reference Attributes    = 0
[NPA] Number of Public Attributes       = 5
[CSB] Class Size in Bytes               = 32
[CSM] Class Size Metric                 = 7

CLASS DESIGN PROPERTY MEASURES (SchaeferResults)
-----------------------------------------
[CAM] Cohesion Among Methods of Class   = 0.75
[DCC] Direct Class Coupling             = 0
[MCC] Maximum Class Coupling            = 0
[DAC] Direct Attribute Based Coupling   = 0
[MAC] Maximum Attribute Based Coupling  = 0
[MPC] Maximum Parameter Based Coupling  = 0
[VOM] Virtuality of Methods             = 0
[CCN] Class Complexity Based on Nodes   = 0
```

```
================**  wxRegex **================
private:
 inline wxRegex ( const wxRegex & );
 inline void operator = ( const wxRegex & );

public:
 wxRegex ( const char t *, const int fast, const int bufsize,
           const char transtable * );
```

```
wxRegex ( );
int Match ( const char s *, int len, const int pos ) const;
int Search ( const char s *, int len, int matchlen &,
             const int startpos ) const;
int match_info ( int start &, int length &, const int nth ) const;
int OK ( ) const;
```

CLASS EXTERNAL MEASURES (wxRegex)

[DOI] Depth of Inheritance = 0

[NOC] Number of Children = 0

[NOA] Number of Ancestors = 0

CLASS INTERNAL MEASURES (wxRegex)

[NOM] Number of Methods = 8

[CIS] Class Interface Size = 6

[NOT] Number Of Trivial Methods = 2

[NOP] Number Of Polymorphic Methods = 0

[NOO] Number Of Operator Methods = 1

[NPT] Number Of Parameter Types = 5

[NPM] Number Of Parameters Per Method = 2.00

[NOD] Number Of Attributes = 0

[NAD] Number Of Abstract Data Types = 0

[NRA] Number Of Reference Attributes = 0

[NPA] Number of Public Attributes = 0

```
[CSB] Class Size in Bytes              = 0

[CSM] Class Size Metric                = 8


CLASS DESIGN PROPERTY MEASURES (wxRegex)

-----------------------------------------

[CAM] Cohesion Among Methods of Class  = 0.42

[DCC] Direct Class Coupling            = 1

[MCC] Maximum Class Coupling           = 1

[DAC] Direct Attribute Based Coupling  = 0

[MAC] Maximum Attribute Based Coupling = 0

[MPC] Maximum Parameter Based Coupling = 1

[VOM] Virtuality of Methods            = 0

[CCN] Class Complexity Based on Nodes  = 0



==============** wxSubString **==============

 wxString;

 wxString & S;

 short pos;

 short len;

Size: wxSubString  8 = 0 + 8


protected:

 inline wxSubString ( wxString &, int p, int l );

 inline wxSubString ( const wxSubString & );


public:
```

```
inline wxSubString ( );
wxSubString & operator = ( const wxString & );
wxSubString & operator = ( const wxSubString & );
wxSubString & operator = ( const char t * );
wxSubString & operator = ( char c );
inline int Contains ( char c ) const;
inline int Contains ( const wxString & ) const;
inline int Contains ( const wxSubString & ) const;
inline int Contains ( const char t * ) const;
inline int Contains ( const wxRegex & ) const;
inline int Matches ( const wxRegex & ) const;
ostream & operator << ( ostream &, const wxSubString & );
inline unsigned int Length ( ) const;
inline int Empty ( ) const;
const char * Chars ( ) const;
int OK ( ) const;
```

```
CLASS EXTERNAL MEASURES (wxSubString)
-----------------------------------------
[DOI] Depth of Inheritance            = 0
[NOC] Number of Children              = 0
[NOA] Number of Ancestors             = 0

CLASS INTERNAL MEASURES (wxSubString)
-----------------------------------------
[NOM] Number of Methods               = 18
[CIS] Class Interface Size            = 16
```

```
[NOT] Number Of Trivial Methods        = 11
[NOP] Number Of Polymorphic Methods    = 0
[NOO] Number Of Operator Methods       = 5
[NPT] Number Of Parameter Types        = 10
[NPM] Number Of Parameters Per Method  = 0.89
[NOD] Number Of Attributes             = 3
[NAD] Number Of Abstract Data Types    = 0
[NRA] Number Of Reference Attributes   = 1
[NPA] Number of Public Attributes      = 0
[CSB] Class Size in Bytes              = 8
[CSM] Class Size Metric                = 21

CLASS DESIGN PROPERTY MEASURES (wxSubString)
-------------------------------------------
[CAM] Cohesion Among Methods of Class  = 0.18
[DCC] Direct Class Coupling            = 3
[MCC] Maximum Class Coupling           = 6
[DAC] Direct Attribute Based Coupling  = 1
[MAC] Maximum Attribute Based Coupling = 1
[MPC] Maximum Parameter Based Coupling = 6
[VOM] Virtuality of Methods            = 0
[CCN] Class Complexity Based on Nodes  = 0


================** wxString **================
 wxSubString;
 enum  CaseCompare;
```

```
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 const ;
 enum  const  int StripType;

protected:
 int Match ( int private 42, int private 43, int private 44,
             const char private 45 *, const int private 46 ) const;
 int _gsub ( const char private 47 *, int private 48,
             const char private 49 *, int private 50 );
 int _gsub ( const wxRegex &, const char private 52 *, int private 53 );
 inline wxSubString _substr ( int private 54, int private 55 );
```

```
public:
 int Search ( int private 56, int private 57, const char private 58 *,
              const int private 59 ) const;
 int Search ( int private 60, int private 61, char private 62 ) const;
 inline wxString ( );
 inline wxString ( const wxString & );
 inline wxString ( const wxSubString & );
 inline wxString ( const char t * );
 inline wxString ( const char t *, int len );
 inline wxString ( char c );
 inline wxString ( );
 wxString & operator = ( const wxString & );
 wxString & operator = ( const char y * );
 wxString & operator = ( char c );
 wxString & operator = ( const wxSubString & );
 wxString Copy ( ) const;
 wxString Replicate ( char c, int n );
 wxString Replicate ( const wxString &, int n );
 inline wxString & operator += ( const wxString & );
 wxString & operator += ( const wxSubString & );
 inline wxString & operator += ( const char t * );
 wxString & operator += ( char c );
 inline wxString & cs ( const char cs * );
 inline wxString & s ( const wxString & );
 inline wxString & rep ( char c, const int rep );
 inline wxString & Prepend ( const wxString & );
 wxString & Prepend ( const wxSubString & );
```

```
wxString & Prepend ( const char t * );
wxString & Prepend ( char c );
inline wxString & this ( char c, const int rep );
inline void Cat ( const wxString &, const wxString &, wxString & );
inline void Cat ( const wxString &, const wxSubString &, wxString & );
inline void Cat ( const wxString &, const char private 70 *,
                  wxString & );
inline void Cat ( const wxString &, char private 73, wxString & );
inline void Cat ( const wxSubString &, const wxString &, wxString & );
inline void Cat ( const wxSubString &, const wxSubString &, wxString & );
inline void Cat ( const wxSubString &, const char private 82 *,
                  wxString & );
inline void Cat ( const wxSubString &, char private 85, wxString & );
inline void Cat ( const char private 87 *, const wxString &,
                  wxString & );
inline void Cat ( const char private 90 *, const wxSubString &,
                  wxString & );
inline void Cat ( const char private 93 *, const char private 94 *,
                  wxString & );
inline void Cat ( const char private 96 *, char private 97,
                  wxString & );
inline void Cat ( const wxString &, const wxString &, const wxString &,
                  wxString & );
inline void Cat ( const wxString &, const wxString &,
                  const wxSubString &, wxString & );
inline void Cat ( const wxString &, const wxString &,
                  const char private 109 *, wxString & );
```

```
inline void Cat ( const wxString &, const wxString &,
                  char private 113, wxString & );
inline void Cat ( const wxString &, const wxSubString &,
                  const wxString &, wxString & );
inline void Cat ( const wxString &, const wxSubString &,
                  const wxSubString &, wxString & );
inline void Cat ( const wxString &, const wxSubString &,
                  const char private 125 *, wxString & );
inline void Cat ( const wxString &, const wxSubString &,
                  char private 129, wxString & );
inline void Cat ( const wxString &, const char private 132 *,
                  const wxString &, wxString & );
inline void Cat ( const wxString &, const char private 136 *,
                  const wxSubString &, wxString & );
inline void Cat ( const wxString &, const char private 140 *,
                  const char private 141 *, wxString & );
inline void Cat ( const wxString &, const char private 144 *,
                  char private 145, wxString & );
inline void Cat ( const char private 147 *, const wxString &,
                  const wxString &, wxString & );
inline void Cat ( const char private 151 *, const wxString &,
                  const wxSubString &, wxString & );
inline void Cat ( const char private 155 *, const wxString &,
                  const char private 157 *, wxString & );
inline void Cat ( const char private 159 *, const wxString &,
                  char private 161, wxString & );
inline void Cat ( const char private 163 *, const wxSubString &,
```

```
                    const wxString &, wxString & );
inline void Cat ( const char private 167 *, const wxSubString &,
                    const wxSubString &, wxString & );
inline void Cat ( const char private 171 *, const wxSubString &,
                    const char private 173 *, wxString & );
inline void Cat ( const char private 175 *, const wxSubString &,
                    char private 177, wxString & );
inline void Cat ( const char private 179 *, const char private 180 *,
                    const wxString &, wxString & );
inline void Cat ( const char private 183 *, const char private 184 *,
                    const wxSubString &, wxString & );
inline void Cat ( const char private 187 *, const char private 188 *,
                    const char private 189 *, wxString & );
inline void Cat ( const char private 191 *, const char private 192 *,
                    char private 193, wxString & );
int CompareTo ( const char cs *, CaseCompare ) const;
int CompareTo ( const wxString &, CaseCompare ) const;
inline int Index ( char c, const int startpos ) const;
inline int Index ( const wxString &, const int startpos ) const;
inline int Index ( const wxString &, int startpos,
                    CaseCompare ) const;
inline int Index ( const wxSubString &, const int startpos ) const;
inline int Index ( const char t *, const int startpos ) const;
inline int Index ( const char t *, int startpos, CaseCompare ) const;
inline int Index ( const wxRegex &, const int startpos ) const;
int Freq ( char c ) const;
int Freq ( const wxString & ) const;
```

```
int Freq ( const wxSubString & ) const;
int Freq ( const char t * ) const;
int First ( char c ) const;
int First ( const char cs * ) const;
int First ( const wxString & ) const;
int Last ( char c ) const;
int Last ( const char cs * ) const;
int Last ( const wxString & ) const;
wxSubString At ( int pos, int len );
wxSubString operator () ( int pos, int len );
wxSubString At ( const wxString &, const int startpos );
wxSubString At ( const wxSubString &, const int startpos );
wxSubString At ( const char t *, const int startpos );
wxSubString At ( char c, const int startpos );
wxSubString At ( const wxRegex &, const int startpos );
wxSubString Before ( int pos );
wxSubString Before ( const wxString &, const int startpos );
wxSubString Before ( const wxSubString &, const int startpos );
wxSubString Before ( const char t *, const int startpos );
wxSubString Before ( char c, const int startpos );
wxSubString Before ( const wxRegex &, const int startpos );
wxSubString Through ( int pos );
wxSubString Through ( const wxString &, const int startpos );
wxSubString Through ( const wxSubString &, const int startpos );
wxSubString Through ( const char t *, const int startpos );
wxSubString Through ( char c, const int startpos );
wxSubString Through ( const wxRegex &, const int startpos );
```

```
wxSubString From ( int pos );
wxSubString From ( const wxString &, const int startpos );
wxSubString From ( const wxSubString &, const int startpos );
wxSubString From ( const char t *, const int startpos );
wxSubString From ( char c, const int startpos );
wxSubString From ( const wxRegex &, const int startpos );
wxSubString After ( int pos );
wxSubString After ( const wxString &, const int startpos );
wxSubString After ( const wxSubString &, const int startpos );
wxSubString After ( const char t *, const int startpos );
wxSubString After ( char c, const int startpos );
wxSubString After ( const wxRegex &, const int startpos );
inline wxString from ( int from, int to ) const;
wxString & Del ( int pos, int len );
wxString & Del ( const wxString &, const int startpos );
wxString & Del ( const wxSubString &, const int startpos );
wxString & Del ( const char t *, const int startpos );
wxString & Del ( char c, const int startpos );
wxString & Del ( const wxRegex &, const int startpos );
inline wxString & this ( ) const;
inline wxString & this ( int pos );
inline wxString & this ( int pos, int len );
wxString & Insert ( int pos, const char private 198 * );
wxString & Insert ( int pos, const wxString & );
inline int GSub ( const wxString &, const wxString & );
inline int GSub ( const wxSubString &, const wxString & );
inline int GSub ( const char pat *, const wxString & );
```

```
inline int GSub ( const char pat *, const char repl * );
inline int GSub ( const wxRegex &, const wxString & );
wxString & Replace ( int pos, int n, const char private 200 * );
wxString & Replace ( int pos, int n, const wxString & );
int Split ( const wxString &, wxString [], int maxn, const wxString & );
int Split ( const wxString &, wxString [], int maxn, const wxRegex & );
wxString Join ( wxString [], int n, const wxString & );
wxString CommonPrefix ( const wxString &, const wxString &,
                        const int startpos );
wxString CommonSuffix ( const wxString &, const wxString &,
                        const int startpos );
wxSubString Strip ( StripType, const char c );
inline wxString Reverse ( const wxString & );
inline wxString Upcase ( const wxString & );
inline wxString Downcase ( const wxString & );
inline wxString Capitalize ( const wxString & );
inline void Reverse ( );
inline void Upcase ( );
inline void Upcase ( );
inline void Downcase ( );
inline void Downcase ( );
inline void Capitalize ( );
inline char & operator [] ( int i );
inline char & pos ( int pos );
inline char Elem ( int i ) const;
inline char Firstchar ( ) const;
inline char Lastchar ( ) const;
```

```
char * private 203 ( ) const;
const char * Chars ( ) const;
char * GetData ( );
void sprintf ( const char fmt *, ... private 204 );
inline ostream & operator << ( ostream &, const wxString & );
ostream & operator << ( ostream &, const wxSubString & );
istream & operator >> ( istream &, wxString & );
int Readline ( istream &, wxString &, const char terminator,
               const int discard_terminator );
inline unsigned int Length ( ) const;
inline int Empty ( ) const;
int IsAscii ( ) const;
int IsWord ( ) const;
int IsNumber ( ) const;
int IsNull ( ) const;
inline int IsNull ( ) const;
inline void Alloc ( int newsize );
inline int Allocation ( ) const;
void Error ( const char msg * ) const;
int OK ( ) const;
```

CLASS EXTERNAL MEASURES (wxString)

[DOI] Depth of Inheritance = 0

[NOC] Number of Children = 0

[NOA] Number of Ancestors = 0

```
CLASS INTERNAL MEASURES (wxString)

-----------------------------------------

[NOM] Number of Methods                 = 177

[CIS] Class Interface Size              = 173

[NOT] Number Of Trivial Methods         = 88

[NOP] Number Of Polymorphic Methods     = 0

[NOO] Number Of Operator Methods        = 28

[NPT] Number Of Parameter Types         = 16

[NPM] Number Of Parameters Per Method   = 1.95

[NOD] Number Of Attributes              = 0

[NAD] Number Of Abstract Data Types     = 0

[NRA] Number Of Reference Attributes    = 0

[NPA] Number of Public Attributes       = 2

[CSB] Class Size in Bytes               = 0

[CSM] Class Size Metric                 = 177


CLASS DESIGN PROPERTY MEASURES (wxString)

-----------------------------------------

[CAM] Cohesion Among Methods of Class   = 0.16

[DCC] Direct Class Coupling             = 3

[MCC] Maximum Class Coupling            = 4

[DAC] Direct Attribute Based Coupling   = 0

[MAC] Maximum Attribute Based Coupling  = 0

[MPC] Maximum Parameter Based Coupling  = 4

[VOM] Virtuality of Methods             = 0

[CCN] Class Complexity Based on Nodes   = 0
```

Appendix E

Metrics on program "SPAS"

Table E.1: Metrics for Attributes and Methods in *wxObject_tree*

Class_Name	AA	AI	AD	ANU	MC	MA	MI	MD	MNU
wxObject	1	–	1	–	–	6	–	6	–
wxPrintData	5	1	6	1	0	13	6	19	6
wxEvent	6	1	7	1	0	8	6	14	6
wxSystemEventClassStruc	3	1	4	1	0	2	6	8	6
wxSystemEventClassStruc	3	1	4	1	0	2	6	8	6
wxToolBarTool	9	1	10	1	0	4	6	10	6

Table E.2: Metrics for Attributes and Methods in *Replicator_tree*

Class_Name	AC	AA	AI	AD	ANU	MC	MA	MI	MD	MNU
Replicator	–	4	–	4	–	–	4	–	4	–
Simulation	0	7	4	11	4	0	7	4	11	4

Table E.3: Metrics Based on Inheritance

Class_Name	HIC	HIA	HID
wxObject_tree	0	0.6111	0.3888
Replicator_tree	0	0.6364	0.3636

Table E.4: Metrics for Virtual Methods in *wxObject_tree*

Class_Name	NIV	NSV	NOLO
wxObject	1	virtual ~wxObject() NSV=0.5	0
wxPrintData	0	NSV=0	0
wxEvent	0	NSV=0	0
wxSystemEventClassStruc	0	NSV=0	0
wxSystemEventNameStruc	0	NSV=0	0
exToolBarTool	0	NSV=0	0

Table E.5: Metrics for Virtual Methods in *Replicator_tree*

Class_Name	NIV	NSV	NOLO
Replicator	1	~Replicator() NSV=0.5	0
Simulation	0	NSV=0	0

Table E.6: Metrics Based on Polymorphism

Class_Name	PFOM	ACRV
wxObject_tree	0	0
Replicator_tree	0	0

Table E.7: Metrics Based on Interactions

Class_Name	NCP	NTC	NSUP	NDC	NPC	NRC	NPAR	MD	AD	IFPV	IFCV
wxObject	0	0	0	0	0	0	1	6	1	0	0
wxPrintData	0	0	1	0	0	1	6	19	6	0.0385	0.0333
wxEvent	1	0	1	0	2	0	1	14	7	0.0769	0.0910
wxSystemEventClassStruc	0	0	1	0	0	0	0	8	4	0.0385	0
wxSystemEventClassStruc	0	0	1	0	0	0	0	8	4	0.0385	0
wxToolBarTool	1	0	1	0	1	0	2	10	10	0.0769	0.0455
DarrochAnalysis	0	0	0	0	0	0	0	0	2	0	0
LeastSqrAnalysis	0	0	0	0	4	4	8	0	2	0	0
MomentAnalysis	0	0	0	0	0	0	0	0	2	0	0
PPetersenAnalysis	0	0	0	0	0	0	0	0	2	0	0
SchaeferAnalysis	0	0	0	0	0	0	0	0	2	0	0
DarrochResults	0	0	0	0	0	0	2	2	1	0	0
Environment	0	0	0	0	0	0	0	0	0	0	0
FormatSet	0	0	0	0	0	0	6	1	6	0	0
IndexSet	0	0	0	0	0	0	3	4	2	0	0
LabelSet	0	0	0	0	0	0	21	16	5	0	0
LeastSqrResults	0	0	0	0	0	0	2	2	7	0	0
MomentResults	0	0	0	0	0	0	1	2	6	0	0
PPetersenResults	0	0	0	0	0	0	0	1	4	0	0
Replicator	1	0	0	0	0	0	10	4	4	0.0385	0.0556
Simulation	1	0	1	0	0	0	12	11	11	0.0769	0.0333
ResultSet	0	0	0	0	0	0	2	2	9	0	0
SchaeferResults	0	0	0	0	0	0	2	2	5	0	0
wxRegex	0	0	0	0	0	0	16	8	0	0	0
wxSubString	0	0	0	0	0	5	16	18	3	0	0.1351
wxString	0	0	0	0	0	43	344	177	19	0	0..769

Bibliography

[1] F.B. Abreu, M. Goulao, and R. Esteves. "Toward the Design Quality Evaluation of Object-Oriented Software Systems", *Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, pp. 44-57, October 1995.*

[2] F.B. Abreu and W. Melo. "Evaluation the Impact of Object-Oriented Design on Software Quality", *Proceedings of the 3rd International Software Metrics Symposium, Berlin, Germany, pp. 90-99, March 1996.*

[3] A.N. Arnason, C.W. Kirby, C.J. Schwarz, and J.R. Irvine. "Computer Analysis of Data from Stratified Mark-Recovery Experiments for Estimation of Salmon Escapements and Other Populations", *Canadian Technical Report of Fisheries and Aquatic Sciences 2106, 1996.*

[4] J. Avotins, "An Object-oriented Method for Evolving and Evaluating Object-oriented Design Metric Models", *Ph.D Thesis, Department of Software Development, Monash University, Australia, 1996.*

[5] J. Bansiya and C. Davis. "Automated Metrics and Object-Oriented Development", *Dr. Dobb's Journal, pp. 42-48, December 1997.*

[6] J. Bansiya, L. Etzkorn, C. Davis, and W. Li. "A Class Cohesion Metric for Object-Oriented Designs", Accepted for publication and to appear in the *Journal of Object-Oriented Programming, 1998.*

[7] V.R. Basili and D.M. Weiss. "A Methodology for Collecting Valid Software Engineering Data", *IEEE Transactions on Software Engineering, Vol.10, No.6, pp. 728-738, November 1984.*

[8] V.R. Basili and H.D. Rombach. "The TAME Project: Towards Improvement-Oriented Software Environment", *IEEE Transactions on Software Engineering, Vol.10, No.6, pp. 758-773, June 1988.*

[9] V.R. Basili, L. Briand, and W.L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering, Vol.22, No.10, pp. 751-761, October 1996.*

[10] R.V. Binder. "State-based Testing", *Object Magazine, Vol.5, No.4, pp. 75-78, July/August 1995.*

[11] R.V. Binder. "Testing for Reuse: Libraries and Frameworks", *Object Magazine, Vol.6, No.6, pp.77-80, August 1996.*

[12] A.B. Binkley and S.R. Schach. "A Comparison of Sixteen Quality Metrics for Object-Oriented Design", *Information Processing Letters 58 (1996), pp. 271-275, 1996.*

[13] G. Booch, *Object Oriented Design with Applications (Second Edition)*, The Benjamin/Cumming Publishing Company, Inc., Redwood City, California, USA, 1991.

[14] J.C. Cherniavsky and C.H. Smith."On Weyuker's Axioms for Software Complexity Measures", *IEEE Transactions on Software Engineering, Vol.17, No.6, pp. 636-638, June 1991.*

[15] S.R. Chidamber and C.F. Kemerer. "Towards a Metrics Suite for Object-Oriented Design", *OOPSLA'91, Phoenix, Aeizona, USA, pp. 197-211, 1991.*

[16] S. R. Chidamber and C. F. Kemerer. "A Metrics Suite for Object-Oriented Design", *IEEE Transactions on Software Engineering, Vol.20, No.6, pp. 476-493, June 1994.*

[17] N.I. Churcher and M.J. Shepperd. "Comments on 'A Metrics Suite for Object Oriented Design' ", *IEEE Transactions on Software Engineering, Vol.21, No.3, pp. 263-265, March 1995.*

[18] C. Ebert and I. Morschel. "Metrics for Quality Analysis and Improvement of Object-Oriented Software", *Information and Software Technology 39 (1997) pp. 497-509, 1997.*

[19] L. Etzkorn, J. Bansiya, and C. Davis. "Design and Code Complexity Metrics for OO Classes", Accepted for publication in *Journal of Object-Oriented Programming, January 1998.*

[20] N. Fenton. "Software Measurement: A Necessary Scientific Basis", *IEEE Transactions on Software Engineering, Vol.20, No.3, pp. 199-206March 1994.*

[21] N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*, International Thomson Computer Press, 1997.

[22] F.P. Ginac. *Software Quality Assurance*, Prentice Hall, Upper Saddle River, New Jersy, USA, 1998.

[23] R.G. Gowda and L.E. Winslow. "An Approach for Deriving Object-Oriented Metrics", *Proceedings of the IEEE 1994 National Aerospace and Electronics Conference NACON, Vol.2, pp. 897-904, 1994.*

[24] I. Graham. "Making Progress in Metrics", *Object Magazine, Vol.6, Iss.8, pp. 68-73, October 1996.*

[25] R. Harrison, L.G. Samaraweera, M.R. Dobie, and P.H. Lewis. "An Evaluation of Code Metrics for Object-Oriented Programs", *Information and Software Technology 38(1996), pp. 443-450, 1996.*

[26] R. Harrison, S. Counsell, and R.Nithi. "An Overview of Object-Oriented Design Metrics", *8th International Workshop on Software Technology and Engineering Practice, 1997.*

[27] R. Harrison, S. Counsell, and R. Nithi. "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", *IEEE Transactions on Software Engineering, Vol.24, No.6, pp. 491-496, June 1998.*

[28] M.J. Harrold, and J.D. McGregor. "Incremental Testing of Object-Oriented Class Structures", *Proceedings of 14th International Conference on Software Engineering, pp. 68-80, May 1992.*

[29] B. Henderson-Sellers. "OO Metrics Programme", *Object Magazine, Vol.5, Iss.6, pp. 72-76,78-79,95, October 1995.*

[30] M. Hitz and B. Montazeri. "Measuring Product Attributes of Object-Oriented Systems", *Proceedings of the 5th European Software Engineering Conference, Barcelona, Spain, pp. 124-136, September 1995.*

[31] E.M. Kim, O.B. Chang, S. Kusumoto, and T. Kikuno. "Analysis of Metrics for Object-Oriented Program Complexity", *Proceedings of COMPSAC'94, pp.201-207, 1994.*

[32] E.M. Kim, S. Kusumoto, T. Kikuno, and O.B. Chang. "Heuristics for Computing Attribute Values of C++ Program Complexity Metrics", *1996 IEEE 20th Annual International Computer Software and Applications Conference, Seoul, Korea, pp. 104-109, August 1996.*

[33] B. Kitchenham, S.L. Pfleeger, and N. Fenton. "Towards a Framework for Software Measurement Validation", *IEEE Transactions on Software Engineering, Vol.21, No.12, pp. 929-944, December 1995.*

[34] D.C. Kung, J.Gao, P. Hsia, Y. Toyoshima and C. Chen. "On Regression Testing of Object-Oriented Programs", *The Journal of Systems and Software, Vol.32, No.1, pp. 21-40, January 1996.*

[35] R. Lafore. *Object-Oriented Programming in Turbo C++*. Waite Group, 1991.

[36] T. Littlefair. "An Investigation into the Role of Software Metrics in Software Quality Improvement", A research project for the degree of Master of Science, Edith Cowan University, Australia. http://www.fste.ac.cowan.edu.au/~tlittlef/project.html

[37] R. Martin, "OO Design Quality Metrics", *Report on Object Analysis and Design 2, November/December 1995.*

[38] M. Milankovic-Atkinson and E. Georgiadou. "Metrics for Reuse of Object-Oriented Software", *Software Quality Management IV. Improving Quality, pp. 363-374, 1997.*

[39] J.D. McGregor. "Managing Metrics in an Iterative Environment", *Object Magazine, Vol.5, Iss.6, pp. 65-71, October 1995.*

[40] K.H. Möller and D.J. Paulish. *Software Metrics.* Chapman & Hall, London UK, 1993.

[41] L.H. Rosenberg and L.E. Hyatt. "Software Quality Metrics for Object-Oriented Environments", *Technical Report, NASA Software Assurance Technology Center, SATC-TR-95-1001, June 1995.*

[42] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[43] R. Whitty. "A Survey of Research and Practice in Object-Oriented Software Metrics", Empirical Software Engineering Research Group, Department of Computing, Borunemouth University, UK, *December 1996.*

[44] E.J. Weyuker. "Evaluating Software Complexity Measures", *IEEE Transactions on Software Engineering, Vol.14, No.9, pp. 1357-1365, September 1988.*

[45] X. Yu and D.A. Lamb. "Metrics Applicable to Software Design", *Annals of Software Engineering 1, pp. 23-41, 1995.*