Reliability and Maintainability Related Software Coupling Metrics in C++ Programs

Chandrashekar Rajaraman
CS Department
The University of Iowa
<rajar@cs.uiowa.edu>

Michael R. Lyu
Information Sciences and Technologies Research Lab.
Bellcore
<lyu@bellcore.com>

Abstract

This paper describes some difficulties that one encounters in the testing and maintenance of C++ programs, which may result in program unreliability. Inheritance and polymorphism are key concepts in object-oriented programming (OOP), and are essential for achieving reusability and extendibility, but they also make programs more difficult to understand. We have tried to show by arguments and by some empirical evidence that widely used complexity metrics like lines of code, cyclomatic complexity, and Software Science's metrics may not be appropriate to measure the complexity of C++ programs and those written in other object-oriented languages, since they do not address concepts like inheritance and encapsulation, apart from having other weaknesses. Some measures using a notion from the world of functional decomposition - coupling, are defined for C++ programs. Two of them - CC and AMC - and equivalent ones for the three widely used complexity metrics (for comparison) are computed for five C++ programs. Our preliminary results show that our coupling measures correlate better with difficulty of testing and maintenance than the three widely used complexity metrics.

I. Introduction

The object-oriented paradigm is revolutionizing software engineering, by providing a new and potentially better way to analyze a problem, design a solution, and implement it. Many goals of software engineering like maintainability, reliability, and reusability, are said to be more easily achieved using this paradigm than with traditional ones based on funtional decomposition. According to Biggerstaff ([Bigge 87]), this paradigm has a good balance between power and generality. In his framework, procedural-based solutions are also depicted having a good balance, but are considered less effective than object-oriented (abbreviated as OO hereafter) solutions. Encapsulation capabilities create self-contained objects that are easily incorporated into a new design, thus promoting reusability [Kerni 84]. Some studies have determined that the object-oriented approach is quantitatively more beneficial than a procedural one in terms of software maintenance ([Henry 90], [Mancl 90]).

Some important questions that must be answered at this juncture are: what makes the object-oriented paradigm different from earlier paradigms, how do these differences help in achieving the goals of software engineering more easily, *and are these goals really being achieved as claimed?* In order to answer the italicized question, the ability to measure is needed, for which *appropriate* measures are required [Denic 81].

Some previous work has recognized the shortcomings of extant metrics and the need for new metrics for OO software. Some empirical suggestions have been made, but little work has been done to define metrics with a sound theoretical foundation [Chida 91]. In this paper, we will define four measures of coupling, primarily for C++ software, though they could be extended to other OO languages:

(1) Class Inheritance-related Coupling (CIC)
(2) Class Non-Inheritance-related Coupling (CNIC)
(3) Class Coupling (CC)
(4) Average Method Coupling (AMC)

The organization of this paper is thus: Section II criticizes the three most widely used metrics as to their aptness for object-oriented as well as traditional software. Section III provides some background about the foundations of software measurement. Section IV defines our coupling measures. Section V deals with our validation approach. Section VI deals with the collection of raw data. Section VII presents the preliminary results of our study. Section VIII contains our conclusions, and future research direction. We will use "member function" and "method" interchangeably.

II. A Criticism of Widely Used Complexity Metrics

There are two types of criticisms that can be applied to current software metrics for object-oriented software. In the first category, we have those that are directed against conventional metrics that are applied to conventional, non-OO software design and development. They are criticized for having no firm theoretical bases ([Vesse 84], [Kearn 86]), and for failing to display "normal predictable behavior" [Weyuk 88]. Weyuker defined a set of nine properties to serve as the basis for the evaluation of syntactic software complexity measures, which she used to evaluate cyclomatic complexity, statement count, Oviedo's data flow complexity ([Ovied 80]), and Halstead's effort measure. Her study found serious drawbacks with all four metrics.

The second type of criticism that can be applied to current software metrics is specific to object-oriented design and development. In the object-oriented approach, data and procedures are not separated as they are in the older, conventional approaches that take a function-oriented view that clearly separates data and procedures. Once we consider the different notions behind these two views, it is not very surprising to find that none of the traditional metrics addresses concepts like inheritance, encapsulation of procedures and data, and passing of messages.

(1) statement count

It is a very intuitive measure of software complexity. From an abstract viewpoint, *the more detail that an entity possesses, the more difficult it is to understand*. That is, the entity is complex. So, a program (entity) that has 100 statements (details) is inherently more complex than one that has 10 statements. However, a drawback is - it is not easy to define what a statement is. Once this definition is made, it is simple to compute the statement count. Its simplicity is the major reason for its wide use, despite its other drawbacks [Weyuk 88]. Statement count views a program's components as possessing inherent complexity regardless of their context in the program, this means that it is insensitive to interactions among the program's various components.

(2) Halstead's Software Science

Halstead introduced software science to measure properties of programs [Halst 77]. Using his notation,

n_1 = number of unique operators
n_2 = number of unique operands
N_1 = total number of operators
N_2 = total number of operands

Then, the program volume V is defined to be

$$V = (N_1+N_2)\log_2(n_1+n_2)$$

The potential volume V* is defined as the minimum possible volume for a given algorithm. Programming effort is then defined to be:

$$E = V^2/V^*$$

The Halstead's effort measure predicted that it would take longer to produce the initial part of the program than the entire program, and its doing so raises serious questions about its use as a syntactic complexity measure [Weyuk 88].

(3) McCabe's Number or Cyclomatic Complexity

McCabe ([Mccab 76]) has defined the complexity of a program to be:

$$v = e - n + 2p$$

where
e = number of edges in a program flow graph
n = number of nodes
p = number of connected components

A drawback with the cyclomatic complexity is that it rates too many programs as equally complex; that is, it is not sensitive enough to capture what might be reasonably considered differences in program complexity [Weyuk 88]. Moreover, it views a program's components as possessing intrinsic complexity, irrespective of their context in the program, thus ignoring the interactions among them.

Nodes are sequential blocks of code, and edges are decisions causing a branch. It is quite obvious that the definition of nodes is not granular enough to account for the complexity of each statement in nodes in a OO or non-OO program's flow graph. For instance, consider two consecutive statements: an object sending a message to another object, and an assignment statement. They would both be "lumped" together in a node, totally ignoring the fact that they differ in their inherent individual complexities. Further, if $p = 1$, then $v = \# + 1$ where # is the number of predicates in the program. One of the points of contention (applicable to both OO and non-OO) in this definition is: How to treat compound predicates?

A simple count of lines, statements or "tokens" in any program, whether OO or non-OO, cannot fully capture its complexity. This is because, in a program, there is a great deal of interaction between modules, and in OO software, you have classes in addition to modules, adding a dimension to this interaction. The above three metrics simply ignore such dependencies, implicitly assuming that each component of a program is a separate entity. On the other hand, our metrics attempt to quantify the interactions among classes assuming that the interdependencies involved contribute to the total complexity of the program units, and ultimately to that of the whole software.

III. Software Measurement Foundations

Most of the software engineering methods proposed in the last twenty-five years provide tools, rules, and heuristics for producing software products that are characterized by *structure* [Fento 90]. This structure is present in the development process as well as in the products themselves. Its presence in the products is identified as modularity, low coupling, high cohesion, encapsulation and others. These are all *internal* attributes. Experts in software engineering agree that the presence of these attributes will ensure the existence of the *external* attributes expected by software users, e.g. reliability, maintainability, and reusability. This is treated almost as an axiom. Despite the important intuitive relationships that exist between the internal structure of software products and their external attributes, there has been little scientific work to establish precise relationships between the internal and external attributes. An important reason for this is that there is a lack of understanding of how to measure important internal software attributes of software products [Fento 90].

Measurement theory provides a relevant basis for deriving measures of software attributes [Baker 90]. It gives us a framework for numerically characterizing intuitive properties or attributes of objects and events. Applying the basic criteria of measurement theory to software measures requires the identification and/or definition of

- attributes of software products and processes. These attributes need to be aspects of software that have both intuitive and well-understood meanings.
- abstractions that capture the attributes.
- important relationships and orderings that exist between the objects being modelled and that are determined by the attributes of the models.
- order-preserving mappings from the models to number systems.

If all of the above criteria are satisfied, then the resultant mapping will be called a software measure. With this background, we will now define some measures of coupling, primarily for C++ software, though they can be extended to other object-oriented languages.

IV. Features of Object-Oriented Programs

Booch has defined object-oriented design as the process of identifying objects and their attributes, identifying operations suffered by and required of each object, and establishing interfaces between objects [Booch 86]. The design of objects involves three steps:

1) definition of objects

2) attributes of objects

3) communication between objects

The design of *methods* involves the definition of procedures which implement the attributes and operations suffered by objects. The design of *classes* is therefore at a higher level of abstraction than the traditional procedural approach which is closer to methods design. The task of class design makes OO design different from procedural design.

The fundamental concepts of OO design as outlined by Booch are shown in Figure 1, and readers are referred to [Booch 86] for a more detailed discussion.

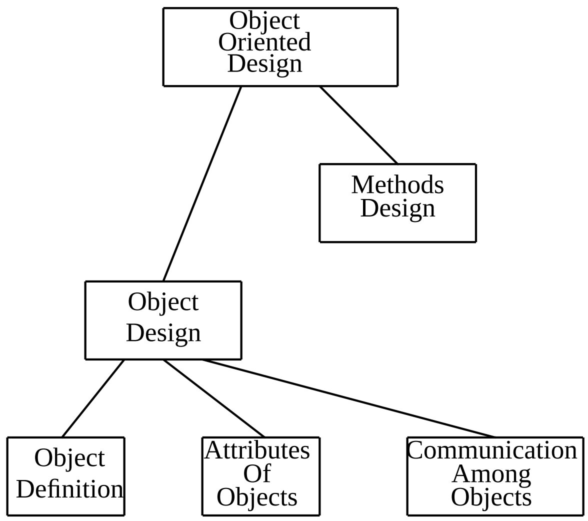

Figure 1: Elements of Object Oriented Design

Software testing and maintenance heavily involve program analysis and understanding. Generally, C++ programs seem to have a large number of small member functions, and hence the system may have a large number of small modules. Maybe this is true for programs written in other OO programming languages as well ([Wilde 90], [Ponde 91]). In fact, it appears to be good OO style to write small methods [Liebe 89]. This has important repercussions for both testing and maintenance. If the code that needs to be understood for a simple task is scattered widely, then good browsers are needed to ease the tester's and maintainer's tasks, and even they may not afford an appreciable advantage if some understanding of context is required. In such cases, a great deal of cross-references alone will not be sufficient, and *relationship links* will need to be identified. This underlines the need for semantics-based tools.

Another problem arises with polymorphism, especially in a dynamically typed environment ([Ponde 92], [Taenz 89], [Wilde 91], [Lejte 91]). It is not possible to determine at compile time the function body that will be executed for a given function call. This will be determined only at run-time. If a function f is invoked on object O, it is the dynamic type of O that determines which f's body is invoked. This can seriously hamper a programmer's understanding of a program merely from a static trace of it. He may have to consider all potential dynamic types of O, all probable types if some application-related knowledge is available, and both of these numbers may be very large. Worse, he may have to perform dynamic analysis, which is cumbersome and can easily be inexhaustive, leading to incorrect conclusions. Therefore, many simple tasks in conventional programs can become major undertakings in object-oriented programs.

A dependency is a relationship between two entities in a system A-->B such that when A is modified, one must be concerned about side-effects in B. Some dependencies in a program are inevitable, and in fact, are desirable, but a large number of them will make the program difficult to test and maintain. The use of inheritance and polymorphism increases the kinds of dependencies considerably. Some of them are: class- class, class-method, class-message, class-variable, and method-variable dependencies, where class, method, message and variable are four kinds of entities. Therefore, a class that is low in a class hierarchy will be more difficult to modify than one higher up since an understanding of a greater number of classes is required. The more a class references variables and uses methods *not defined* in the class, the less self-contained is the class. That is, greater are the class' dependencies, and clearly, greater is the difficulty of testing and maintaining the class.

V. Definitions of Our Coupling Metrics

The software attribute that we consider in this paper is coupling. It has been defined as *a measure of the degree of interdependence between modules* [Press87], and *the degree of interaction between modules* [Myers 78]. Though coupling is a notion from structured design, it is still applicable to object-oriented design – at the levels of modules, classes and objects. In this paper, we are concerned only with coupling between classes.

Coupling for a class has been defined as *a count of the number of non-inheritance related couplings with other classes* [Chida 91]. When methods of one class use methods or instance variables of one that belongs to another class hierarchy, then we have coupling between the classes. A class with strong coupling – high interrelation with other classes – is harder to understand, change, or correct by itself. The greater the number of couplings, the higher the sensitivity to changes in other parts of the design. This makes testing and maintenance more difficult. Coupling also affects testing. The higher the inter-object(class) coupling, the more rigorous the testing needs to be [Chida 91]. A measure of coupling would therefore be useful in identifying parts of a product that are "complex" from the point of view of testing and maintenance.

There is some clash of interests between inheritance and coupling. While it is desirable to have weak coupling between classes, inheritance promotes coupling between superclasses and their subclasses, to take advantage of the commonality among abstractions. There is no question that inheritance is crucial to achieving reusability and extendibility, apart from being a powerful modelling tool of key relationships between concepts in the application domain, but it does increase the dependencies and has adverse effects on code understandability and testability (in the absence of sophisticated semantics-based tools). Therefore, we include inheritance in our definition of coupling:

> "*Coupling is a measure of the association, whether by inheritance or otherwise, between classes in a software product.*"

The model that we use to study coupling is a directed multigraph (Figure 2). It is a graph that may have many arcs between two nodes. Each node corresponds to a class, and each arc corresponds to a variable reference or member function use. For example, in Figure 2, an arc from A to C signifies that class A makes a reference to a variable or uses a member function that has been defined in C.

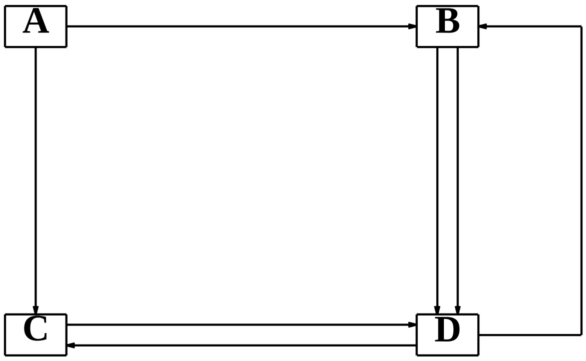

Figure 2: A Directed Multigraph

Based on the above discussions, we define four new coupling metrics as follows:

(1) Class Inheritance-related Coupling (CIC) and (2) Class Non-Inheritance-related Coupling (CNIC)

For a class, there are usually two kinds of clients: objects that invoke operations upon instances of the class, and subclasses that inherit from the class. With inheritance, coupling will occur when a class accesses a variable or uses a member function defined in a proper ancestor class. For a class, we define a count of such accesses and uses as *2lass Inheritance-related Coupling (CIC)*.

A way in which non-inheritance-coupling can occur is by the use of *friends*. A friend is defined as a method typically involving two or more objects of different classes, whose implementation for any one class may reference the private parts of all the corresponding classes that are also friends. Global variables and functions also cause non-inheritance-related coupling. For a class, *Class Non-inheritance-related Coupling (CNIC)* is defined as a count of the accesses to variables and uses of member functions that have been defined neither in the class nor in any of its proper ancestors.

We define *Method Coupling (MC)* as follows,

$$MC = number \ of \ non\text{-}local \ \text{references}$$

For a method, we define a non-local reference as one that references a variable or method *not defined* in the class where the method is defined or redefined. MC is nothing but the sum of inheritance-related and non-inheritance-related couplings at the method level.

$$MC = gv + gf + om + iv$$

where
gv = # global variable references
gf = # global function uses
om = # messages to other classes
iv = # references to instance variables of other classes

(3) Class Coupling (CC)

Consider a class C, with methods M_1, M_2, ... M_n, where $MC_1, MC_2, \cdots MC_n$ are the method couplings (MCs) of the respective methods, then

$$Class \ Coupling \ (CC) \ of \ C = \sum_{i=1}^{n} MC_i$$

where n is the number of methods *belonging* to the class.

CC is equal to the number of outgoing arcs from the node corresponding to the class in the multigraph representation of the program.

(4) Average Method Coupling (AMC):

For a class, this is defined as the ratio of its Class Coupling to its number of member functions.

$$AMC = CC/n$$

where

CC = Class Coupling
n = number of member functions in the class

This measure would provide the average couplings of member functions in a class.

VI. Validation Efforts

Our main concern was prompt and easy access to the software developers for obtaining reliability and maintainability related data. We also wanted to analyze "real" programs instead of programs developed in a simulated environment. We obtained four such software from the Center for Computer-Aided Design, Simulation and Design Optimization of Mechanical Systems

(CCAD) at the University of Iowa. CCAD is actively engaged in the development of software for CAD applications in mechanical engineering. It consists of over a hundred student research assistants, full-time research staff, and faculty of the Department of Mechanical Engineering. In fact, the research being conducted there has been largely responsible for the University of Iowa being selected in a nation-wide competition as the site for the $32 million National Advanced Driving Simulator (NADS) project. Four software projects obtained from CCAD (identified as "ccad_<id>"), together with one from the University of South Carolina (identified as "usc_1"), are briefly described below:

- ccad_1, ccad_11: They are class libraries for dynamics computations. ccad_1 is an earlier version of ccad_11. Both have over 10 classes; the former has over 89K LOC, the latter more than 137K LOC.
- ccad_2, ccad_22: They are also class libraries for dynamics computations. ccad_2 is a very early version of ccad_22; the former has 27K LOC, the latter 167K LOC.
- usc_1: It is a class library for image processing applications, developed at the University of South Carolina. It consists of over 60 classes, and over 15K LOC.

All the classes in each of the five software were ranked by the developers in the order of perceived difficulty of testing and maintenance. We then computed CC and AMC for all the classes. The classes were ranked based on CC and AMC values, and by corresponding values obtained using the three widely used complexity metrics. Rank correlations of these values with perceived-difficulty ranks were then computed.

To extract the data that we needed from our five data points, we used "PC-Metric for C++" marketed by SET Laboratories, "CodeCheck Tool" of Abraxas Software, and developed some code of our own.

A point to note: there is a loss of quantitative information by using ranks. If data like mean time to failure(mttf), and mean time to repair(mttr) were available, then a "better" validation of our measures could have been done.

VII. Preliminary Results

The values that we computed from the five software projects have been presented in Tables 1, 2, and 3. In this section we have the following naming convention:

"loc" - Lines Of Code
"hss" - Halstead's Software Science
"mn" - McCabe's Number

For instance, CC refers to Class Coupling defined earlier, while CC_mn refers to the *equivalent* measure computed using cyclomatic number; likewise for CC_hss, and CC_loc. This naming convention applies to AMC also.

Table 1 shows #classes and #methods for the investigated projects.

	# classes	# methods
ccad_1	12	91
ccad_11	16	97
ccad_2	12	59
ccad_22	40	321
usc_1	70	279

Table 1: Class and Method Data in the Projects

Table 2 lists sample CC and AMC data for a data point: ccad_1. The correlation between the ranks of the classes based on column headers versus the perceived difficulty ranks will be presented in Table 3. For instance, the correlation coefficient between CC_hss ranks and perceived difficulty ranks in Table 2 is the entry (ccad_1, CC_hss) in Table 3. Tables similar to Table 2 have been computed for the other data points also, and they were used to compute the correlation coefficients presented in Table 3.

Table 3 contains the correlation coefficients of CC vs perceived difficulty and AMC vs perceived difficulty for all software projects. In Table 3, columns 1, 3, 5, and 7 contain the correlation values with perceived difficulty of CC, CC_loc, CC_hss, and CC_mn respectively. Columns 2, 4, 6, and 8 contain the correlation values with perceived difficulty of AMC, AMC_loc, AMC_hss, and AMC_mn respectively.

Two observations were made:

(1) CC and AMC correlate with perceived difficulty better than all the other CC and AMC values, and this is true for all the software projects analyzed.

	CC	AMC	CC_loc	AMC_loc	CC_hss	AMC_hss	CC_mn	AMC_mn	perceived difficulty
class 1	21	8.97	2134	508	123	53	12	4.3	1
class 2	10	7.86	4329	1296	163	46	12	4.5	2
class 3	22	4.6	8976	354	216	54	15	4.9	3
class 4	23	9.17	3209	325	287	65	15	4.9	4
class 5	23	8.9	7098	312	257	51	21	5.0	5
class 6	65	5.6	5690	929	234	67	15	5.2	6
class 7	55	10.23	11247	638	256	71	24	6.2	7
class 8	46	10.31	3431	523	293	56	17	5.2	8
class 9	50	12.23	7896	632	198	57	21	5.4	9
class 10	70	13.4	6654	865	365	60	28	6.2	10
class 11	40	17.32	17415	2112	241	63	30	5.3	11
class 12	56	13.34	10908	2206	265	78	30	5.6	12

Table 2: Sample CC and AMC Data for the Ccad_1 Data Point

	CC	AMC	CC_loc	AMC_loc	CC_hss	AMC_hss	CC_mn	AMC_mn
ccad_1	0.76	0.81	0.65	0.64	0.59	0.62	0.74	0.75
ccad_11	0.78	0.82	0.62	0.62	0.61	0.61	0.74	0.76
ccad_2	0.75	0.78	0.65	0.67	0.63	0.62	0.71	0.72
ccad_22	0.78	0.79	0.67	0.67	0.62	0.62	0.72	0.73
usc_1	0.74	0.77	0.65	0.66	0.64	0.63	0.63	0.62

Table 3: Correlation Coefficients for All Software Projects

(2) AMC_mn’s correlation with perceived difficulty is comparable to those of our metrics for all data points except usc_1. Our speculation is that there is a definite difference in the qualities of the software developers; those of usc_1 are more knowledgeable in C++ and object-oriented programming, and hence have exploited its language constructs more fully than those of ccad_1, ccad_11, ccad_2, and ccad_22. The latter developers may have programmed in C++ as they would in a traditional programming language based on functional decomposition.

VIII. Conclusions and Future Research

Some key features of C++ and other OO languages - inheritance and dynamic binding - that are essential to achieving reusability and extendibility, can make the task of program understanding difficult. The main reason for this is the absence of tools that address these special requirements. Currently available cross-referencers and browsers are not enough. In addition to ordinary cross-referencing information, relationship links have to be identified. More sophisticated tools that are semantics-based are necessary. We have tried to quantify the difficulty of OO program analysis and understanding by defining some measures for programs in a representative language - C++, using the well-understood notion of coupling. Two of them - CC and AMC - and equivalent ones for the three widely used complexity metrics (for comparison) were computed for five C++ programs, and their rank correlations with perceived difficulty of testing and maintenance were computed. Our preliminary results show that CC and AMC had the maximum correlation, though the differences were not statistically significant.

We plan to study the well-definedness and consistency of the measures over a larger cross section of C++ software. We also plan to refine the idea of CIC by taking into account the depths in the class hierarchy tree the class making the references and the classes it makes references to, are. This would provide insights regarding the manageable depth for a class hierarchy tree from the testing and maintenance perspective. Such information would certainly affect class design. Since communication between objects is at the heart of object-oriented design, we hope our research direction will lead us to the optimal non-zero value for coupling, which will correspond to efficient communication between objects, and substantially reduce the difficulty of testing and maintenance.

References

[Baker 90]
Baker, A.L., J.M.Bieman, N.E.Fenton, D.A.Gustafson, A.C.Melton, and R.W.Witty, "A Philosophy for Software Measurement," *Journal of Systems and Software*, 12, 277-281 (1990).

[Bigge 87]
Biggerstaff, T., and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, March 1987, pp.41-49.

[Booch 86]
Booch, G., "Object Oriented Development," *IEEE Transactions on Software Engineering*, SE-12, February, 211-221, 1986.

[Chida 91]
Chidamber, Shyam R. and Chris F. Kemerer., "Towards a Metrics Suite for Object-Oriented Design," *OOPSLA* 1991.

[Denic 81]
Denicoff, Marvin and Robert Grafton "Software Metrics: A Research Initiative," In Alan J. Perlis, Frederick Sayward, and Mary Shaw editor, *Software Metrics: An Analysis and Evaluation*. MIT Press, Cambridge, Massachusetts, 1981.

[Fento 90]
Fenton, N. and A.Melton, "Deriving Structurally Based Software Measures," *Journal of Systems and Software*, 12, 177-187, 1990.

[Halst 77]
Halstead, M.H., *Elements of Software Science*, Elsevier, New York, 1977.

[Henry 90]
Henry, Sallie M. and Matt Humphrey, "A Controlled Experiment to Evaluate Maintainability of Object-Oriented Software," *Proceedings of IEEE Conference on Software Maintenance 1990*, pp. 258-265.

[Kearn 86]
Kearney, J.K., et al. "Software Complexity Measurement," *Communications of the ACM*, 29 (11), 1986, pp. 1044-1050.

[Kerni 84]
Kernighan, B.W., "The Unix System and Software Reusability," *IEEE Transactions on Software Engineering*, September 1984, pp.513-518.

[Lejte 91]
Lejter, M., Scott Meyers and Steven P. Reiss: "Support for Maintaining Object-Oriented Programs", *Proceedings of IEEE Conference on Software Maintenance '91*, pp. 171-178.

[Liebe 89]
Lieberherr, Karl J. and Ian M. Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, Volume 6, Number 5, September 1989, pp. 38-48.

[Mancl 90]
Mancl, Dennis and William Havanas, "A Study of the Impact of C++ on Software Maintenance," *Proceedings of IEEE Conference on Software Maintenance 1990*, pp. 63-69.

[Mccab 76]
McCabe, T.J., "A Complexity Measure," *IEEE*

Transactions on Software Engineering, 2(4), (1976).

[Myers 78]
Myers, G.J., *Composite/Structural Design*, Van Nostrand Reinhold, New York, NY, 1978.

[Ovied 80]
Oviedo, E.I., "Control flow, data flow, and program complexity," in *Proceedings of IEEE COMPSAC*, Chicago, IL, Nov.1980, pp.146-152.

[Ponde 92]
Ponder, Carl and Bill Bush, "Polymorphism Considered Harmful," *ACM SIGPLAN Notices*, Volume 27, Number 6, June 1992.

[Taenz 89]
Taenzer, et al. "OO SW Reuse: The Yoyo Problem," *Journal of Object-Oriented Programming*, September/October 1989, pp.30-35.

[Vesse 84]
Vessey, I. and R.Weber, "Research on Structured Programming: An Empiricist's Evaluation, " *IEEE Transactions on Software Engineering*, SE-10(4), 1984, 394- 407.

[Weyuk 88]
Weyuker, E., "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering*, Volume 14, Number 9, September 1988, 1357-1365.

[Wilde 91]
Wilde, N. and Ross Huitt: "Maintenance Support for Object-Oriented Programs," *Proceedings of IEEE Conference on Software Maintenance*