Restructuring Object-Oriented Designs Using a Metric-Driven Approach.

Ghassan Salim Alkadi

Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation

https://digitalcommons.lsu.edu/gradschool_disstheses/6970

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
RESTRUCTURING OBJECT-ORIENTED DESIGNS USING A METRIC-DRIVEN APPROACH

A Dissertation

Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Doctor of Philosophy in

The Department of Computer Science

by

Ghassan Salim Alkadi
B.S., Southeastern Louisiana University, 1985
M.S., Louisiana State University, 1991
Summer 1999
ACKNOWLEDGEMENTS

I would like to sincerely thank my advisor Dr. Doris L. Carver for being so helpful in guiding me through the entire ordeal that I went through in finalizing this research. My sincere thanks are extended to my friend Elias Khalaf for supporting me technically and spiritually. Last but not least, I would like to extend my sincere appreciation for my twin brother Ihssan Alkadi for being a great friend and brother in good and bad times.
# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS** ................................................................................................................................. ii

**LIST OF TABLES** ........................................................................................................................................... v

**LIST OF FIGURES** ....................................................................................................................................... vi

**ABSTRACT** .................................................................................................................................................... ix

**CHAPTER 1. INTRODUCTION** ................................................................. 1
  1.1 Object-Oriented Methodologies ................................................................................. 7
  1.2 Factors Affecting the Design Complexity ............................................................. 10

**CHAPTER 2. RELATED WORK** ..................................................................................... 13

**CHAPTER 3. DESIGN ASSISTANT PARAMETERS** .................................................. 21
  3.1 Basic Metric Set ............................................................................................. 21
    3.1.1 Weighted Methods per Class .................................................................. 21
    3.1.2 Depth of Inheritance Tree ..................................................................... 25
    3.1.3 Number of Children .......................................................................... 29
    3.1.4 Coupling Between Objects ................................................................. 30
    3.1.5 Response For a Class ......................................................................... 36
    3.1.6 Lack of Cohesion in Methods ............................................................. 38
  3.2 Computation of Design Assistant Parameters .................................................... 47
    3.2.1 Redesign Using DIT Metric ................................................................. 47
    3.2.2 Redesign Using WMC Metric .................................................................. 52
    3.2.3 Redesign Using NOC Metric .................................................................. 64
    3.2.4 Redesign Using CBO Metric .................................................................. 74
    3.2.5 Redesign Using RFC Metric ................................................................... 80
    3.2.6 Redesign Using LCOM Metric ............................................................... 87
  3.3 Summary ........................................................................................................... 92

**CHAPTER 4. DESIGN EVALUATION ASSISTANT** .................................................. 95
  4.1 Textual Representation of UML Notations .......................................................... 96
  4.2 Description of DEA ....................................................................................... 98
    4.2.1 Evaluating the DIT Metric on A UML Design ........................................ 99
    4.2.2 Evaluating the WMC Metric on A UML Design .................................... 101
    4.2.3 Evaluating the NOC Metric on A UML Design ..................................... 103
    4.2.4 Evaluating the CBO Metric on A UML Design .................................... 104
    4.2.5 Evaluating the RFC Metric on A UML Design .................................... 106
    4.2.6 Evaluating the LCOM Metric on A UML Design .................................. 108

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 5. SUMMARY AND CONCLUSIONS ................................................... 113
5.1 Conclusions and Contributions............................................................. 114
5.2 Future Work....................................................................................... 120

REFERENCES ............................................................................................. 121

VITA ............................................................................................................. 127
LIST OF TABLES

1. Reusability Statistics .................................................................25
2. Average Depth of Inheritance ....................................................28
3. Summary of Library Characteristics .........................................29
4. Metrics Application .................................................................94
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1 (a, b, c, d)</td>
<td>Types of Inheritance Hierarchy</td>
<td>27</td>
</tr>
<tr>
<td>Figure 2 (a)</td>
<td>Fan-in/fan-out example</td>
<td>32</td>
</tr>
<tr>
<td>Figure 2 (b)</td>
<td>RFC counts</td>
<td>33</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Subclassing by type</td>
<td>48</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Subclassing by type after using code reuse</td>
<td>48</td>
</tr>
<tr>
<td>Figure 5</td>
<td>DIT Algorithm</td>
<td>50</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Hierarchy of Depth 1</td>
<td>51</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Hierarchy of Depth 1 and with more than 1 child</td>
<td>51</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Hierarchy of Depth 2 after applying DIT algorithm</td>
<td>51</td>
</tr>
<tr>
<td>Figure 9</td>
<td>WMC Algorithm</td>
<td>57</td>
</tr>
<tr>
<td>Figure 10</td>
<td>WMC Table</td>
<td>58</td>
</tr>
<tr>
<td>Figure 11</td>
<td>WMC Tables</td>
<td>61</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Class Hierarchy</td>
<td>65</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Redesigned Class Hierarchy</td>
<td>66</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Example of NOC</td>
<td>67</td>
</tr>
<tr>
<td>Figure 15</td>
<td>ICBT hierarchy Tree</td>
<td>69</td>
</tr>
<tr>
<td>Figure 16</td>
<td>ICBT A case where NOC = 15 and DIT = 4</td>
<td>70</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Tree with DIT &lt; 2</td>
<td>71</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Tree with DIT &gt; 6</td>
<td>73</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Tree with DIT = 7</td>
<td>74</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Tree with DIT = 6</td>
<td>74</td>
</tr>
<tr>
<td>Figure 21</td>
<td>CBO Algorithm .................................................................................77</td>
<td></td>
</tr>
<tr>
<td>Figure 22</td>
<td>Class Hierarchy Collaboration ............................................................78</td>
<td></td>
</tr>
<tr>
<td>Figure 23</td>
<td>Generic Collaboration Matrix ..............................................................79</td>
<td></td>
</tr>
<tr>
<td>Figure 24</td>
<td>Collaboration Matrix .........................................................................79</td>
<td></td>
</tr>
<tr>
<td>Figure 25</td>
<td>RFC Algorithm ....................................................................................82</td>
<td></td>
</tr>
<tr>
<td>Figure 26</td>
<td>Method Invocation Chart .......................................................................85</td>
<td></td>
</tr>
<tr>
<td>Figure 27</td>
<td>MIG(s) for classes A, B, and C ..............................................................85</td>
<td></td>
</tr>
<tr>
<td>Figure 28</td>
<td>Methods Invocation Chart (revised) .......................................................86</td>
<td></td>
</tr>
<tr>
<td>Figure 29</td>
<td>MIG(s) for class B .................................................................................86</td>
<td></td>
</tr>
<tr>
<td>Figure 30</td>
<td>RRFC Redesign Algorithm ....................................................................86</td>
<td></td>
</tr>
<tr>
<td>Figure 31</td>
<td>LCOM Algorithm ..................................................................................87</td>
<td></td>
</tr>
<tr>
<td>Figure 32</td>
<td>General Cohesion Matrix ......................................................................89</td>
<td></td>
</tr>
<tr>
<td>Figure 33</td>
<td>LCOM = 0 .............................................................................................89</td>
<td></td>
</tr>
<tr>
<td>Figure 34</td>
<td>Cohesion Matrix ...................................................................................90</td>
<td></td>
</tr>
<tr>
<td>Figure 35</td>
<td>After adding more instance variables ...................................................91</td>
<td></td>
</tr>
<tr>
<td>Figure 36</td>
<td>Based on data in Figure (35) .................................................................91</td>
<td></td>
</tr>
<tr>
<td>Figure 37</td>
<td>LCOM Redesign Algorithm ...................................................................93</td>
<td></td>
</tr>
<tr>
<td>Figure 38</td>
<td>DEA Overview .....................................................................................98</td>
<td></td>
</tr>
<tr>
<td>Figure 39</td>
<td>DEA Main Menu ..................................................................................99</td>
<td></td>
</tr>
<tr>
<td>Figure 40</td>
<td>DIT Algorithm output ..........................................................................100</td>
<td></td>
</tr>
<tr>
<td>Figure 41</td>
<td>WMC Algorithm output .......................................................................102</td>
<td></td>
</tr>
<tr>
<td>Figure 42</td>
<td>NOC &amp; DIT Algorithm .............................................................................104</td>
<td></td>
</tr>
</tbody>
</table>
ABSTRACT

The benefits of object-oriented software are now widely recognized. However, methodologies that are used to develop object-oriented software are still in their infancy. There is a lack of methods to assess the quality of the various components that are derived during the development process. The design of a system is a crucial component derived during the system development process. Little attention has been given to assessing object-oriented designs to determine the goodness of the designs. There are metrics that can provide guidance for assessing the quality of the design. The objective of this research is to develop a system to evaluate object-oriented designs and to provide guidance for the restructuring of the design based on the results of the evaluation process. We identify a basic set of metrics that reflects the benefits of the object-oriented paradigm such as inheritance, encapsulation, and method interactions. Specifically, we include metrics that measure depth of inheritance, methods usage, cardinality of subclasses, coupling, class responses, and cohesion. We define techniques to evaluate the metric values on existing object-oriented designs. We then define techniques to utilize the metric values to help restructure designs so that they conform to predetermined design criteria. These methods and techniques are implemented as a part of a Design Evaluation Assistant that automates much of the evaluation and restructuring process.
CHAPTER I. INTRODUCTION

In recent years, software development has become increasingly complex due in part to the tremendous strides achieved in hardware technology. Many software developers create software without considering the software maintenance needed at later stages. Consequently, the program's maintenance process will be costly. Software frequently is not designed with an eye on the future. In order to prevent this situation, there must be a smooth transition from the requirements stage to the implementation stage. The transition must pass through the design stage. Flaws in the design stage will eventually cause problems in the product of the design. Clearly, there is no magic bullet that can solve all the design problems.

Software should be designed with an eye for maintenance. The design evaluation step is an integral part of achieving a high quality design. [IE89] defined quality as:

"The totality of features and characteristics of a product or service that bears on its ability to satisfy given needs."

Design evaluation is a recurring step that should be performed and checked multiple times before committing to the design implementation. [SH92] Shepperd in a review of the history of software engineering metrics focuses on a "range of quality factors, typically maintainability and reliability of the resultant software system." [PF91] on the other hand, equates quality to reliability, availability, and maintainability [HS96]. In object-oriented design, objects and actions are put together when they have a common purpose. There are four key elements for a good object-oriented design [BO91]:

1
• Abstraction: denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer.

• Encapsulation: the process of binding all of the details of an object that do not contribute to its essential characteristics.

• Hierarchy: a ranking or ordering of abstractions.

• Modularity: the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Shaw [SH84] defines abstraction as "a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user while suppressing details that are, at least for the moment, immaterial or diversionary." Abstraction is essential to the object-oriented programming process. It increases the user's readability and ease of understanding the objects. The more effort that can be gathered into abstract classes, the less effort the subclass will require [LK94]. Metrics are needed to assess inheritance and reuse in order to take into account the greater number of abstraction levels inherent in object-oriented systems. Also, they help to address cost estimation and product quality across all life-cycle stages.

Encapsulation is a primary feature of data abstraction. Encapsulation reduces complexity. [DWH97] defines encapsulation as hiding a module implementation in a separate block with a formally specified interface. The interface is a connecting link at a shared boundary that permits independent systems to meet and act on or communicate
with each other. Reliability is increased by encapsulation since other program units cannot change representations directly, either intentionally or accidentally. Consequently, the integrity of such encapsulated objects is increased [SE89].

Inheritance is a relationship among classes wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) classes [BO91]. Single inheritance occurs when a subclass inherits behavior of some superclass. A subclass may change the behavior or structure of some superclass. Multiple inheritance occurs when a subclass inherits from multiple superclasses. Inheritance reduces redundancy in the code and thereby increases its efficiency.

Modularity decreases the degree of complexity of a software system. Each module is defined, refined, and compiled separately. A module is a program unit that can include types, objects, and subprograms that may be called by other units. Object Oriented software systems that implement abstraction, encapsulation, hierarchy, and modularity are a base for a good software design. There are other elements that also contribute to a good design such as typing. Typing falls under two categories dynamic, and static. The kind of typing used in a design depends on the language used. For example, C++ and Object Pascal both support dynamic typing. In dynamic typing, all variables and expressions types are not known until runtime. Whereas, in static typing all variables and expressions types are known during the course of compilation time. Languages can also be strongly typed or weakly typed. In a strongly typed language each variable must have a type. In a weakly typed language, variables do not have types until execution time which may cause problems when running the program.
Again, it is important to note here that modularity features differ from one object-oriented language to another.

Many object-oriented designs are not developed using a metric-driven methodology. System designs that were created for imperative languages conform to different sets of software metrics [WE88] which are not totally applicable to object-oriented designs. [HS96] explained that the suites of object-oriented metrics could easily form the basis for the construction of such evaluative metrics in much the same way that the “essential complexity” metric of McCabe was used as an extension of his standard cyclomatic complexity metric to discriminate between programs written with a structured approach as opposed to a non-structured approach, evidenced by “spaghetti code.”

The six software metrics in [CK94] are proposed to help developers reduce the cost, increase the quality, and decrease the amount of time spent on maintenance. The six metrics are: Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object Classes (CBO), Response for a Class (RFC), and Lack of Cohesion in Methods (LCOM).

1. Weighted Methods Per Class (WMC).

WMC is the number of methods. Objects with a large number of methods tend to be more application specific, which limits the possibility of reuse. The larger the number of methods in a class, the greater the effect on the children because of the inheritance property.
2. Depth of Inheritance Tree (DIT).

DIT is the number of ancestor classes that can affect a class. The deeper a class is in the hierarchy, the higher the degree of methods inheritance, making it more complex to predict its behavior.

3. Number of Children (NOC).

NOC is the number of subclasses that inherit the methods of a parent class. Depth is preferred to breadth in the hierarchy; thus the number of children measure is large for weaker designs. If a class has a large number of children, more testing of the methods is required.

4. Coupling Between Object Classes (CBO).

The CBO for a class is a count of the number of other classes to which it is coupled. It counts class to class connectivity other than by inheritance. CBO is a measure of fan-out, which relates to the notion that an object is coupled to another if two objects act upon each other. Thus, the higher the class coupling, the more rigorous the testing needs to be. Moreover, the larger the number of couples, the higher the amount of changes in other parts of the design, making maintenance of the design more difficult.

5. Response for a Class (RFC).

RFC is a set of methods that can be executed in response to a message received by an object of that class. It measures both external and internal communication. It
specifically includes methods called from outside the object and also measures the communication between objects.


Cohesion measures the inter-relatedness between portions of a program. The degree of similarity for two methods \( M_1 \) and \( M_2 \) in class \( C \) is given by:

\[
\sigma() = \{I_1\} \cap \{I_2\}
\]

where \( \{I_1\} \) and \( \{I_2\} \) are the sets of instance variables used by \( M_1 \) and \( M_2 \).

The LCOM is a count of the number of method pairs whose similarity is zero (i.e., \( \sigma() \) is a null set) minus the count of method pairs whose similarity is not zero. The larger the number of similar methods, the more cohesive the class. A high value of LCOM suggests that classes should be split into two or more classes. If none of the methods in a class utilize instance variables, they have no similarity and consequently the value of LCOM is equal to zero in that class. LCOM is tied to the instance variables and methods of a class; therefore, it is a measure of the attributes of an object class.

The six metrics are explained in detail in Chapter 3. In addition to the metrics in [CK94], [SC93] introduced three additional metrics to supplement the metrics in [CK94]. The metrics are: Weighted Attributes per Class (WAC), Number of Tramps (NOT) which is defined as the total number of extraneous parameters in the methods, and Violations of the law of Demeter (VOD). VOD is described by [LI88] as a component which refers to recursive or concatenated message sends. The Law of
Demeter reduces the coupling between classes and makes it easier to change a class interface [WBJ90]. The Law of Demeter increases information hiding by ensuring that one class cannot depend on the implementation of another [WBJ90]. These metrics, along with metrics proposed by other researchers, were evaluated in [HS96] and [DC97].

In this research we perform design evaluation by measuring the class structure of object-oriented designs using the software metric criteria proposed in [CK94], and we define a technique to restructure the design based on metric values.

1.1 Object-Oriented Methodologies

Many object-oriented methodologists have emerged in recent years. [YO89] divided the methodologists into two categories: revolutionaries and synthesists. Revolutionaries consider object orientation as a radical change that renders conventional ways of thinking about design obsolete. Synthesists, on the other hand, weigh object orientation as an accumulation of sound software engineering principles that users can graft onto their existing methodologies with relative ease [FK92]. Loy [LO90] compares object-oriented methodologies.

In the Shlaer and Mellor methodology [SM88], an Information Model consists of an organization and a graphical notation to describe and define the vocabulary and conceptualize the problem domain. Instances in the problem are identified as objects, their characteristics are abstracted as attributes, and the associations between the instances are abstracted as relationships. The Information Model, which provides software developers with a better understanding of the problem, is used to integrate the
diverse views of the problems. The methodology is applied to real-time control systems, decision support systems, and knowledge-based systems. It provides a structured means of identifying objects within a system by analyzing abstract data types. It is best applied to information systems or re-engineering situations in which data objects are already identified [MHRK93]. Shlaer and Mellor methodology supports the three key elements of abstraction, inheritance and encapsulation.

Existing object-oriented design methodologies do not generally require adherence to design metrics. On the other hand, complexity metrics have been used to help measure the quality of structured designs. Examples include McCabe's cyclomatic number [MC76], Halstead's programming effort [HA77], and the size metric in [BZ88]. McCabe defines the cyclomatic complexity measure based on the control flow in a procedure/function. It is based on the complexity of the directed graph, where a directed graph is derived based on the control flow of a procedure/function. For structured programs, the cyclomatic complexity is the count of Boolean conditions in control constructs. Halstead defines software science metrics based on the lexical tokens in a program. The four counts are: number of unique operators, number of unique operands, total occurrence of operators, and total occurrence of operands. Program complexity is based on the size of a program needed to describe an algorithm where the number of bits needed to describe the algorithm defines the size of the program. Several important dimensions that relate to the detailed definition of classes and inheritance, class and object relationship, encapsulated operations, and message connections are not addressed by conventional methodologies [FK92]. The complexity
metrics made for structured methods have provided support for structured designs, however, these metrics are not directly applicable to object-oriented notions such as classes, inheritance, encapsulation and message passing [WH92]. Therefore, the set of software metrics in [CK94] has been chosen for this research to investigate key object-oriented concepts. The metrics discussed in this work were selected for their usefulness and effectiveness in evaluating object-oriented designs.

[BO89], [WPM89], and [WWW90] have proposed OOD methodologies. [FK92] presented the methodologies in an order based on their similarities to conventional methodologies. [WPM89] stated that the goal of Object Oriented Structured Design (OOSD) should accommodate any software design including the conventional approaches and the object oriented ones. [BO89] described many techniques and tools to assist designers which include informal lists, formal diagrams, and templates. [WWW90] methodology is responsibility driven since the attention during design is on contracts between clients and server objects. [FK92] pointed out the differences among the proposed OOD methodologies, which are:

1. Data design
2. Level of detail in describing the process of OOD
3. Level of detail provided by diagram notations.

[FK92] determined that none of the proposed methodologies achieved the status of widely recognized standards on the order of the conventional methodologies of [YC79] or [DM78]. They stated that OOD has not yet fully matured. In order for object oriented software to fulfill its promised potential, measures of metrics have to be
incorporated such as the ones already existing in the conventional software [CK91]. The six software metrics proposed by [CK94] along with new elements of software design should enable developers to reduce the cost of the software, increase the quality, and help decrease the amount of time spent on the maintenance.

1.2 Factors Affecting the Design Complexity

A great deal of attention has been given to finding an appropriate way to measure the complexity of a software design. The number of methods and the complexity of each method show how much effort and time is spent on developing and maintaining the object. Large numbers of methods within objects limit the possibility of method reuse since these methods are application specific [FK92]. Reusability increases software quality throughout the development of the system. It also decreases the cost and time of the system's development. Reusing the same objects enhances reliability since programmers are now familiar with their performance. The system has to be easily modified so it will be cost and time effective since changes to requirements or error corrections may result in a design change. If the customer asks for a change, then it is important to know how easy it is for the programmer to understand the code before modifying it. Modular structure should be simple so that it can be easily understood and modified without knowledge of other modules. The length of modules should be reasonable, so the code length will not intimidate the programmer when it is time to modify them. Shortening the code should not affect the readability of the code. Usage estimate is a metric that tells us how many times the module has been called or
used by other modules, determining its importance to the class of objects. If a module is rarely used, it should possibly be incorporated within another.

Coupling is a measure of how much modules depend on each other [PF91]. The purpose of object-oriented programming is to eliminate dependability, i.e., if a module is to be modified, then only that module is compiled without affecting other modules. Also, the independence of modules makes it easier to isolate those modules that contain the errors. Cohesion is a measure used to determine the connectivity among the elements in an object. The more connected a module the more cohesive the module. All the elements of a module should be directed toward performing the same function. High cohesion and low coupling are desirable when designing a system. Depth of inheritance is a measure of how the superclasses affect the subclasses. The deeper the class is, the more complex it becomes because of its ability to inherit more methods from superclasses. The availability of such a measure helps designers detect the depth of the class in the hierarchy, which allows them to design the class with the intent of reusing the inherited methods from the superclasses. Objects communicate with each other via message passing. Each message invokes a particular method that causes the object to behave in a particular manner. Methods can be viewed as definitions of responses to possible messages [BA87]. Methods within an object can invoke methods from other objects. If large numbers of methods are invoked in response to a message, it will increase the complexity of the object since testing and debugging the object becomes difficult. The previous measure is important because it assists in appropriate allocation of testing time needed [FK92].
The metrics used in this research have been evaluated against a widely accepted list of software metric evaluation criteria by [CK94]. They cited [HA77] as well as [MC76] complexity measures. [KK88] deals with the development of a computer-aided tool that provides intelligent assistance in the design stage of the software life cycle. They defined complexity measures: module complexity, procedure complexity, and module complexity. In [KK88], they conclude that more cohesion leads to less coupling. They developed a set of test cases that resulted in better maintainable designs. The automated software design assistant developed by [KK88] did not address the applicability of its use on object-oriented designs. In Chapter 3, we provide a description of how these metrics, along with the newly proposed elements of software design, are going to be implemented in the Design Evaluation Assistant (DEA) developed in this research.

In Chapter 2, we review the related work in object-oriented designs that utilize metrics. In Chapter 3, we show a detailed description of each metric used. In Chapter 4, we introduce the DEA tool and the sample outputs generated based on the algorithms and examples used. Chapter 5 provides the summary and conclusions of this research.
CHAPTER 2. RELATED WORK

Numerous research efforts involve object-oriented metrics. [BO86] examined the process of object-oriented development and the influences from advances in abstraction mechanisms. He stated that object-oriented development is fundamentally different from the traditional functional one. [BO86] also stressed the fact that object-oriented development is amenable to automated support and should consider building tools for it. Tools for object-oriented designs help to improve maintainability and understandability of systems complexity. [KE86] indicates that software complexity measures have not realized their potential for the reduction and management of software cost. Furthermore, [KE86] attributes the failure to the lack of a unified approach to the development and the use of these measures. Complexity measures from [HA77] and [MC76] also cited in [KE86].

[CK91] [CK94] introduced a metrics suite for object-oriented designs. [CK91] [CK94] formally evaluated the metrics against a widely accepted list of software metric evaluation criteria. They claimed that such measures applied in a software system could be used to aid management in:

- estimating the cost and schedule of future projects,
- evaluating the productivity impacts of new tools and techniques,
- establishing productivity trends over time,
- improving software quality,
- forecasting future staffing needs, and
- anticipating and reducing future maintenance requirements.

13
The suite proposed by [CK91] is not comprehensive. Some object-oriented design properties are not covered. Also, this suite has been subjected to only limited empirical observation. [CS95] criticized [CK91] [CK94] research for the lack of empirical considerations. They used an example to show the ambiguities associated with the notion of the Number of Methods per class (WMC).

[BS93] presented a conceptual extension of the object-oriented programming paradigm to support the development of high quality software. They introduced a new term, Quality Object-Oriented Language “QOOL”, which enables the inheritance of software metrics. They defined three categories of software quality metrics: goal-syntax, and execution-based metrics. Goal-based metrics assess how well software achieves its functional requirements. Syntax-based metrics examine source code and describe how well it is implemented. Goal-based metrics measure the performance and the fault tolerance of the system. They stated “Quality assurance is often viewed as a crushing burden upon the developer’s back. Yet, if one cannot measure and confirm the quality of a software product, how can its benefits and “qualities” be honestly touted?” They came to the conclusion that “quality metrics should assess how well software satisfies its design goals, how well it is written, and how well it performs, respectively.” They added that in order to evaluate QOOL’s contribution effectively to software engineering, they have to increase the work at the conceptual, empirical, and implementation levels. Also, they asked whether we need to start at the design level rather than the programming level implementing quality metrics?
[CA94] described a general model of cognitive complexity metrics at the programming level. The approach included reference to the ways in which a software engineer uses chunking and tracing to understand the code. Modeling the various programmers' tasks as landscapes can demonstrate the effects of chunking and tracing difficulty on complexity graphically. This landscape visualization illustrates the basic approach of the cognitive complexity model introduced by [CA94]. Many programmers perform a variety of tasks on a wide range of modules. However, in order to have conclusive results, programmers must test the model proposed by [CA94].

[FK92] compares object-oriented design methodologies. They capture essential similarities and differences between the methodologies. They examine the notions and notations advocated by each methodology to decide those that were variants on the same basic idea. [FK92] states that the most important differences between object-oriented and conventional analysis methodologies ultimately stem from the object-oriented requirement of encapsulated operations. In addition, while conventional and object-oriented methodologies both provide tools that define a hierarchy of modules, employ a completely different method of decomposition, and the very definition of the term module is different. [FK92] points out that three areas need more development work. The first area requires that more rigorous mechanism is needed for decomposing very large systems. The second one suggests that the tools for modeling multiple objects are cumbersome. The third area involves focusing on the area of reuse.

[MHRK93] suggests that the Shlaer-Mellor method was a successful one. The Shlaer-Mellor object-oriented method provides a structured means of identifying
objects within a system by analyzing abstract data types. They developed a project for the McDonnell Douglas's Missile System (MGS) Division in which they examined the feasibility of adapting object-oriented analysis to engineer the requirements of a mission-planning system.

[CL93] presents new metrics for object-oriented design. The metrics measure the complexity of a class in an object-oriented design. They conduct an experiment to build the metric system to derive a regression model of the metrics based on the experimental data. The metrics that [CL93] proposes is for the Booch object-oriented design method. In addition, a subjective judgement by an expert is incorporated in the regression model to ensure that the metric system is pragmatic and flexible for the software industry. [CL93] defines: (1) operation complexity, (2) operation argument complexity, (3) attribute complexity (4) operation coupling, (5) class coupling, (6) cohesion, (7) class hierarchy, and (8) reuse metric. This metric system, however, is a preliminary proposal that requires further research based on: (1) theoretical research on individual metrics, (2) experimental work on the contribution of the individual metrics software quality, and (3) pragmatic research in applying the metrics on very large systems in the real world.

[LI95] examined the relationship between software metrics collected from design documents and system maintainability in the object-oriented paradigm. [LI95] divided software metrics into two categories: (1) software product metrics that measure software products, such as source code or design documents. (2) software process metrics that measure the software development process, such as the number of working
hours charged to the development activities in the design and coding phases. The conclusion of their study suggested that they could measure software designs quantitatively using software metrics in the object-oriented paradigm. In addition, it showed that software metrics collected from the design can predict maintenance effort in the two systems that they studied. However, these conclusions were drawn from the study of only two commercial object oriented systems.

[DE96] presented an analytical and empirical evaluation of software reuse metrics. They proposed five metrics in their literature that have been analytically and empirically assessed regarding their capabilities to predict productivity and quality in object oriented systems. Following the lead of [WE88] in the field of complexity measures, they developed axioms that should apply to any measure of reuse benefit. None of the metrics satisfied all the properties.

[WE88] proposed a set of properties of syntactic software complexity measures to serve as a basis for the evaluation of such measures. Using these criteria, she evaluated and compared four well-known complexity measures. [WE88] provided the foundation for comparing and evaluating software complexity measures in a formal way. The set of properties that [WE88] introduced was tested only on conventional structured designs. They have made few attempts, however, to apply those properties on object oriented designs. The work in [WE88] has encouraged a more rigorous look at complexity measures and ultimately would lead to the definition of good meaningful measures.
[LK94] identified a set of meaningful metrics for measuring project progress and quality. The metrics apply specifically to object-oriented software projects. Lorenz & Kidd used metrics that were based on measurements and advice derived from several of actual projects that have successfully used object technology to deliver products. In [LK94] metrics and their use in projects were discussed. For each metric used, they explained its meaning, showed project results and affecting factors, related metrics, thresholds, and suggested actions.

[CU97] summarized how Microsoft uses various techniques and melds them into an overall approach that balances flexibility and structure in software product development. They labeled Microsoft's style of a product as the 'synch-and stabilize' approach. The approach meant continually synchronizing as individuals and team members during the project development rather than at the end of the project. [CU97] stated that the synch-and-stabilize approach is especially suited to fast-paced markets with complex system products, short lifecycles. It is also suited to competition based around evolving product features and actual technical standards.

[BBM96] collected data about faults in object-oriented classes. They verified how much fault-proneness is influenced by cohesion and coupling. They concluded that five of the [CK94] metrics are useful to predict class fault-proneness during the high- and low-level design phases of the life-cycle. They also concluded that object-oriented metrics in [CK94] are better predictors than the best set of 'traditional' code metrics. The results are provided motivation for further investigation and refinement of [CK94] metrics.
[BK98] formalized the concept of design cohesion that was based on a graph model of procedure interface, the input/output independence graph (IODG). They derived a design-level cohesion (DLC) measure, which uses an association-based approach. Also, a design-level functional cohesion (DFC) measure was derived using the slice-based approach. Their findings conclude the following:

- Cohesion can be objectively defined and measured in terms of design-level entities.
- Design-level cohesion measures correspond closely with code-level cohesion measures.
- The design-level measures can be used to help locate poorly-designed modules.
- IODG model provides a flexible tool for a quantitative and qualitative characterization of a software design.

[DM93] proposed the concept of cohesion as “a class is coherent if the methods work together to carry out a single, identifiable purpose.” They identified the set of method calls that is recursively extended to the set of n-calls within a specified visibility boundary [HS96]. Also, they defined the co-response of two methods with visibility boundary B as the common set of methods called from the two methods. Also, [DM93] suggested that the larger the co-response set compared to the total number of distinct methods from the two methods, the greater is the degree of cohesion [HS96]. They derived the cohesion distance formula (3.10) shown in Chapter 3. The coherence distances obtained from this formula are then represented in a distance matrix and used to perform cluster analysis that permits the authors to draw a dendogram.

In summary, there is a lack of object-oriented design evaluation techniques. We utilize the use of the metrics proposed by [CK94] to study design evaluation. Based on
these metrics, we define algorithms to evaluate designs. In Chapter 3 we modify the metrics to encompass more design problem possibilities.

Limitations of the related work include:

1. Lack of experimentation
2. Limited evaluation techniques
3. Lack of object-oriented design methodologies that utilize the metrics to help identify design problems.

The goal of this research is to use object-oriented design metrics to help evaluate the quality of software designs. The evaluation of objects determines if an object needs to be redesigned, modified, or deleted.
CHAPTER 3. DESIGN ASSISTANT PARAMETERS

3.1 Basic Metric Set

We define the six metrics introduced by [CK94] as the parameters of the design assistant. We also include definitions of the same metrics used by other researchers. The six metrics are:

- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling Between Objects (CBO)
- Response For a Class (RFC)
- Lack of Cohesion in Methods (LCOM)

3.1.1 Weighted Methods per Class (WMC)

In [CK94], WMC is defined as:

\[ WMC = \sum_{i=1}^{n} c_i \]  \hspace{1cm} (3.1)

where \( C_1, \ldots, C_n \) is the complexity of methods. The \( c_i \) is the static complexity of each method \( M_1, \ldots, M_n \) that is defined in a class. In addition, "complexity is deliberately not defined more specifically here in order to allow for the most general application of this metric." In other words, WMC = the number of methods if \( c_i \) is 1 and WMC does not include the number of attributes [HS96]. The exclusion of attributes as well as the types of methods "public/private" drew comment in [CS95] [HPVP95a] [G95a]. We adopt the principles that include all method types "public/private." We define WMC as:
\[ WMC = N_0 + N_p + N_i \quad \ldots \quad (3.2) \]

where

- \( N_0 \) is the number of overridden methods
- \( N_p \) is the number of pure "newly added" methods
- \( N_i \) is the number of inherited methods

Numerous overrides indicate subclassing for the convenience of reusing some code and/or instance variables where the new subclass is not purely a specialized type of its superclasses. [LK94] recommends less than 4 overridden methods per instantiated class. Pure methods should decrease down through the layers of the hierarchy. [LK94] recommends less than 4 added methods per instantiated class. Reuse of inherited methods is highly recommended since it utilizes one of the most favorable characteristics of object-oriented techniques.

WMC as defined by [CK94] does not include the number of attributes. It may therefore underestimate the class size [LK94]. Also, WMC introduced by [CK94] does not distinguish attributes from other methods so that the metric applies to Smalltalk but not to C++ [LK94]. [HS91] introduced a formula in which the number of attributes and methods, suitably weighted, are summed to give a class size value \( s_i \):

\[ S_i = (AW_A + MW_M)_i \quad \ldots \quad (3.3) \]

where \( S_i \) is the class size, \( A \) is the number of attributes, \( M \) is the number of methods and \( W_A \) and \( W_M \) are weights for attributes and methods, which take an average value rather than actual value of method complexity. [HS91] introduced the following formula where \( S \) is the summation of all class-level values across the system. It is for \( N \) object classes, each of size \( s_i \):
\[ S = \sum_{i=1}^{N} s_i = \sum_{i=1}^{N} (A W_a + M W_m)_{i} \ldots \ldots (3.4) \]

We introduce the following formula for class size \( S \) to include all types of methods in an object class, where \( N \) is the number of object classes and \( s_i \) is the size of an object class:

\[ S = \sum_{i=1}^{N} s_i = \sum_{i=1}^{N} (A W_a + M W_{mI} + M W_{mH} + M W_{mpr} + M W_{mpB})_{i} \ldots \ldots (3.5) \]

where \( M W_{mI} \) is the weight for local methods

\( M W_{mH} \) is the weight for inherited methods

\( M W_{mpr} \) is the weight for private methods

\( M W_{mpB} \) is the weight for public methods.

The formula is object-oriented language independent, since it applies to different types of programming languages. This feature is important since object-oriented languages deal with methods construction differently. For example, in C++ both public and private methods can be applied to the formula. Other object-oriented languages do not have the capability of writing both public and private methods. Private methods are the only way to write methods in most object-oriented languages. Formula 3.5 also includes the weight for the attributes used in objects. The total number of attributes is counted using the declarations in the class. It represents the number of “data stores” in the class. [LK94] differentiates between the number of instance variables and the number of class variables, placing more emphasis on the number of instance variables because the number of instance variables is a measure of its size. In the research described in this dissertation, we use the number of instance variables because it is a more accurate measure for the number of attributes used in a class.
Reuse and specialization are very important issues that should be considered when dealing with methods metrics. Yap and Henderson-Sellers [YH93] introduced two measures to evaluate the level of reuse within hierarchies. The reuse ratio, $U$, is shown as:

$$U = \frac{\text{number of superclasses}}{\text{total number of classes}} \quad (3.6)$$

If $U$ is near 1, then it suggests a linear hierarchy, whereas, if $U$ is close to zero, then it indicates shallow depth and a large number of leaf classes. A value near to 1 is characteristic of a linear hierarchy and a value near zero indicates a shallow depth and a large number of leaf classes, where, for $n$ subclasses, $U = 1/n \rightarrow 0$ as $n$ increases. For a high use of multiple inheritance [YH93]

$$U = \frac{(n - 1)}{n} \rightarrow 1. \quad (3.7)$$

[YH93] also introduced the specialization ratio, $S$, as follows:

$$S = \frac{\text{number of subclasses}}{\text{number of superclasses}} \quad (3.8)$$

If $S$ is large, then there is a high degree of reuse by subclassing. If both $U$ and $S$ are close to one, then the design is poor. Table 1 summarizes the findings in [YH93]. It shows values of both $U$ and $S$ for ten different class libraries [HS96]. Table 1 depicts the number of subclasses, the number of superclasses, the reuse ratio and the specialization ratio. The most significant reuse ratio was for the Eiffel/S library. The most significant specialization ratio was for the NIH C++. 

24

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Table 1 Reusability Statistics (Yap and Hendersen-Sellers, 1993)

<table>
<thead>
<tr>
<th>Library</th>
<th>No. of Subclasses</th>
<th>No. of Superclasses</th>
<th>Reuse Ratio</th>
<th>Specialization Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor 3.0</td>
<td>118</td>
<td>45</td>
<td>.38</td>
<td>2.62</td>
</tr>
<tr>
<td>Borland 2.0</td>
<td>11</td>
<td>27</td>
<td>.37</td>
<td>2.45</td>
</tr>
<tr>
<td>Borland 3.0</td>
<td>67</td>
<td>131</td>
<td>.40</td>
<td>1.96</td>
</tr>
<tr>
<td>Booch components</td>
<td>4</td>
<td>3</td>
<td>.12</td>
<td>.75</td>
</tr>
<tr>
<td>C++/Views</td>
<td>30</td>
<td>71</td>
<td>.40</td>
<td>2.37</td>
</tr>
<tr>
<td>Eiffel/S²</td>
<td>74</td>
<td>74</td>
<td>.99</td>
<td>1</td>
</tr>
<tr>
<td>NIH C++</td>
<td>16</td>
<td>63</td>
<td>.24</td>
<td>3.94</td>
</tr>
<tr>
<td>Smalltalk/V-Windows</td>
<td>51</td>
<td>172</td>
<td>.30</td>
<td>3.37</td>
</tr>
<tr>
<td>Smalltalk/V for PM</td>
<td>40</td>
<td>138</td>
<td>.29</td>
<td>3.45</td>
</tr>
<tr>
<td>Zinc interface</td>
<td>13</td>
<td>47</td>
<td>.24</td>
<td>3.61</td>
</tr>
</tbody>
</table>

3.1.2 Depth of Inheritance Tree (DIT)

The inheritance hierarchy has a root and leaves. The depth of inheritance of a leaf is always greater than that of the root [CK94]. The DIT(C) is the distance from class C to the root. If multiple inheritance exists, then the DIT is the longest path for the distance. It is a system-level metric that indicates how many levels of inheritance have to be investigated for evaluating the whole class hierarchy.

The deeper the class, the greater the number of methods to inherit, thus making it difficult to maintain. Increased difficulty in maintenance is likely because of the introduction of more public and protected methods. In addition, the introduction of more public and private methods increases the chances of extensions and overrides which in return increases the difficulty of testing [LK94]. [BBM96] introduced a hypothesis ‘H-DIT’ for the DIT metric. They suggested that well-designed object-oriented systems are those structured as forests of classes, rather than as one very large lattice. [BBM96] also suggested that a class located deeper in a class lattice is more fault-prone because the class inherits a large number of definitions from its ancestors.
Moreover, deep hierarchies imply problems of conceptual integrity, i.e.; it becomes unclear which class to specialize from in order to include a subclass in the inheritance hierarchy [DB96].

There is greater potential reuse of inherited methods if the depth is > 2 since reuse further specializes the superclass type of object [LK94]. However, [LK94] indicates that any depth > 5 is enough since more levels indicate the possibility of not subclassing by specialization (is-a) but rather implementation subclassing. Implementation subclassing is the usage of portions of the behavior of data that is not the same type of object as the superclasses [LK94]. Hence, this undesirable type of subclassing results in fragile relationships that break as changes are made [LK94]. For a class, the distance from the root for a class in the hierarchy is called its nesting level. We encounter potential difficulty if a class is deeply nested in the inheritance hierarchy. The deeply nested hierarchy is likely to incur increased complexity and extensive testing of methods. [HS96] computes the average inheritance depth as follows:

\[
\text{Average Inheritance Depth} = \frac{\sum \text{depth of each class}}{\text{number of classes}}. \quad (3.9)
\]

Figure 1 shows the different types of inheritance. There is a need to measure inheritance structures. [LK94] called it the “nesting level.” [TS92] called it class-to-root depth whereas [CK91] called it DIT “Depth of Inheritance Tree.” [HS96] suggests a rough guideline of 6 or 7 DIT\text{max}, where DIT\text{max} is the maximum number of levels in a class hierarchy. It is recommended that DIT\text{max} be either 6 or at most 7 [YH93]. To support their heuristic, they tested data for 10 libraries and summarized their findings in
a table (Table 2) that shows average depth of inheritance for 10 classes. Figure 1d shows a hierarchical fragment where the depth for subclass 1 is 1 since it only inherits

(1a) Linear and deep hierarchy

(1b) Multiple inheritance

(1c) Wide, shallow hierarchy

(1d) Fragment of inheritance Hierarchy

Figure 1 Types of Inheritance Hierarchy
from one parent. The depth for class 2 is \((2+1)/2 = 1.5\) because it inherits from two parents, of which one (superclass 3) inherits from another (superclass 1) and the other (superclass 2) does not inherit from any other superclass [HS96]. The overall average depth of inheritance for the hierarchy in Figure (1d) is the sum of each depth given as: 0 (Superclass 1) + 0 (Superclass 2) + 1 (Superclass 3) + 1 (Subclass 1) + 1.5 (Subclass 2) = 3.5, and the average depth = \(3.5/7 = 0.7\). In Table 3, we find the summary of library characteristics for the hierarchy of some of the library classes used. [YH93] evaluates the structure of 10 class libraries. They use an example to illustrate the data found in Table 2. They use the class STRING which is an equivalent class to VARIABLE_STRING in some libraries. The class STRING is extensively used in most applications, however, its placement within the hierarchy was very different. For example, Actor 3.0 inherits from ByteCollection and through four levels to Object. In Borland C++ a and C++/views, however, it is a second-tier. They concluded that the position of STRING within the hierarchies seems to be consistent only where the libraries share the same developer as in the case of Borland C++ versions 2.0 and 3.0.

<table>
<thead>
<tr>
<th>Library</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Actor 3.0</td>
<td>1 29</td>
</tr>
<tr>
<td>Borland 2.0</td>
<td>4  8</td>
</tr>
<tr>
<td>Borland 3.0</td>
<td>35 53</td>
</tr>
<tr>
<td>Booch components</td>
<td>30  3</td>
</tr>
<tr>
<td>C++/Views</td>
<td>4  15</td>
</tr>
<tr>
<td>Eiffel/5^2</td>
<td>1  1</td>
</tr>
<tr>
<td>NIH C++</td>
<td>3 16</td>
</tr>
<tr>
<td>Smalltalk/V^1</td>
<td>1  39</td>
</tr>
<tr>
<td>Windows</td>
<td>1  35</td>
</tr>
<tr>
<td>Smalltalk/V for PM</td>
<td>7 12</td>
</tr>
</tbody>
</table>
This example shows the depth calculated for the NONE class. As NONE class inherits from everything, its depth is the sum of all other classes' depths \((275.5)/74\) which is the total number of classes + 1 (as it descends from each of those classes)

\[ \text{Depth} = (275.5/74)+1 = 4.72. \]

<table>
<thead>
<tr>
<th>Library</th>
<th>Size (No. of Classes)</th>
<th>Total Number of Methods</th>
<th>Average Number of Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor 3.0</td>
<td>119</td>
<td>1,579</td>
<td>13.27</td>
</tr>
<tr>
<td>Borland 2.0</td>
<td>30</td>
<td>228</td>
<td>7.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>244</td>
<td>8.13</td>
</tr>
<tr>
<td>Borland</td>
<td>166</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Templates</td>
<td>1,014</td>
<td>6.11</td>
<td></td>
</tr>
<tr>
<td>Without Templates</td>
<td>1,087</td>
<td>6.55</td>
<td></td>
</tr>
<tr>
<td>Derivation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Templates</td>
<td>1,098</td>
<td>6.61</td>
<td></td>
</tr>
<tr>
<td>Without Templates</td>
<td>1,183</td>
<td>7.15</td>
<td></td>
</tr>
<tr>
<td>Booch Components</td>
<td>32</td>
<td>340</td>
<td>12.89</td>
</tr>
<tr>
<td>C++/Views</td>
<td>75</td>
<td>967</td>
<td>12.89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,035</td>
<td>13.80</td>
</tr>
<tr>
<td>Eiffel/S</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unrestricted exports</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NIH C++</td>
<td>66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lower limit</td>
<td>1,683</td>
<td>25.50</td>
<td></td>
</tr>
<tr>
<td>Upper limit</td>
<td>1,694</td>
<td>25.67</td>
<td></td>
</tr>
<tr>
<td>Derivation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lower limit</td>
<td>1,823</td>
<td>27.62</td>
<td></td>
</tr>
<tr>
<td>Upper limit</td>
<td>1,838</td>
<td>27.85</td>
<td></td>
</tr>
<tr>
<td>With restricted</td>
<td>572</td>
<td>7.63</td>
<td></td>
</tr>
<tr>
<td>Smalltalk/V- Windows</td>
<td>173</td>
<td>1,825</td>
<td>10.55</td>
</tr>
<tr>
<td>Smalltalk/V- for PM</td>
<td>139</td>
<td>1,518</td>
<td>10.92</td>
</tr>
<tr>
<td>Smalltalk/Objectworks</td>
<td>850</td>
<td>15,989</td>
<td>18.81</td>
</tr>
<tr>
<td>Zinc Interface</td>
<td>54</td>
<td>1,825</td>
<td>10.55</td>
</tr>
<tr>
<td>Instantiation</td>
<td>549</td>
<td>10.17</td>
<td></td>
</tr>
<tr>
<td>Derivation</td>
<td>637</td>
<td>11.80</td>
<td></td>
</tr>
</tbody>
</table>

### 3.1.3 Number of Children (NOC)

NOC is the number of subclasses that inherit methods from a superclass "parent class." [CK94] proposes that depth is preferable to breadth. In general, if the NOC metric has small values, inheritance is not being utilized since there is less reuse of the code in the methods. Small NOC values indicate shallow inheritance hierarchies. In
general, if the NOC metric has large values, the design will not be optimal. A larger number of children indicates improper abstraction of the parent class. It also indicates that more testing and maintenance of the methods are required [HS96]. The NOC metric can be used to prioritize quality assurance efforts. The larger the number of children in a class, the more vulnerable the analysis of its correctness, since more testing is required the deeper we move away from the root [DC97]. Large nesting numbers indicate a potential design problem because designers may have been overly zealous in finding and creating objects. Hence, large nesting numbers result in subclasses that are not specialization of all the superclasses [LK94].

3.1.4 Coupling Between Objects (CBO)

CBO relates to the notion that an object is coupled to another object if one of them acts on the other [CK94]. In other words, if methods of one object use methods or instance variables of another, then coupling occur. The message connections between classes are certainly forms of coupling. Coupling should be minimized and classes should be independent, in effect promoting reuse. [HS96] evaluated coupling in terms of fan-in and fan-out during analysis and design stages. Fan-in and fan-out refer to the number of other collaborating classes irrespective of the number of references made statically or dynamically. In other words, the fan-in and fan-out values are either zero or one for a pair of classes. [LA98] defines coupling as “a dependency between elements (usually types, class, and subsystems), typically resulting from collaboration between the elements to provide a service.” Different ways to measure the amount of coupling between classes [LK94] are:
• number of other classes collaborated with
• amount of collaboration with other classes

Coupling is defined as [BN93]:

“When one object depends implicitly on another, they are tightly coupled. Object instances are tightly coupled with their classes. When one object depends directly on the visibility of another, they are closely related. When one object references another only indirectly through the other’s public interface they are loosely coupled.”

By definition, subclasses are tightly coupled to their superclasses. In [SC93], experiments show that the Wirfs-Brock methodology produces lower values of coupling and higher values of cohesion, as opposed to a data-driven object-oriented methodology that was used by [CO91]. In [STM91], a “fan-down” metric is defined as the number of subclasses that redefine a class. During the design stage, the number of association and aggregation relationships and the argument lists can be counted. Thus, if two classes are coupled in analysis, it is very likely that in later stages of design and implementation they will expand to several connections that would not be shown earlier [HS96]. Moreover, we can evaluate fan-in and fan-out of classes. Fan-in and fan-out refer to the number of other collaborating classes irrespective of the number of references made statically and dynamically; that is for a pair of classes, the fan-in/fan-out value is either zero or one. Figure 2a shows that the analysis fan-out is 2 for A, 0 for B, and 0 for C, whereas the respective fan-in values are 0, 1, and 1. A low fan-out is desirable since a high fan-out is characteristic of the large number of classes needed by the particular class in question [HS96]. We discuss fan-out “CBO” more elaborately in Section 2.4.
[DC97] states that we cannot have zero coupling, because it would suggest that instances could not communicate. However, low coupling is needed to maximize modularity and minimize the dependence on other classes.

![Fan-in/fan-out example](image)

Figure 2a Fan-in/fan-out example

RFC represents the number of message paths but does not discriminate between two messages sent to the same method from different parts of the class in the design [HS96]. Non-inheritance coupling is counted by using the measure CBO “fan-out” because the declaration of an object of a remote ADT creates a potential collaboration. The following heuristics are used:

1. Add a maximum of 1 to the fan-out count disregarding how many messages flow between the two collaborating object classes [HS96].

2. Whenever two object classes collaborate, for each unique service accessed, one is added to the NRM count and thus the RFC count since RFC = NLM + NRM as indicated in [HS96].

3. If a particular service is accessed from different parts of the “client object class,” 0 is added to MPC count [HS96].
Figure (2b) RFC counts

Figure 2b illustrates three ways to count non-inheritance coupling. Class A collaborates with class B. Two messages flow between A and B which gives a value of 1 to fan-out. MPC’s value is two since A uses two different services of objects of class B. RFC’s value is 2 since one is added to the NRM count and hence to the RFC count (RFC = NLM + NRM.) Class D accesses one of B’s services from two different places. Class B does not access any services, which explains why all the values for fan-out, MPC, and RFC are zeros, which gives a different value for MPC and RFC. In Section 2 of this chapter, we show a different approach to count the non-inheritance coupling between these classes. Within class C, although one message makes an internal call, the value of RFC is increased but not added to the coupling value of MPC or fan-out “RFC = 2 \rightarrow NRM = 1 \& NLM = 1.” Recall high fan-outs represent class coupling that in return means excessive complex dependence, whereas high fan-ins suggest good object design and high level of reuse [HS96].

[GS90] defines coupling between objects as the manner and degree of interdependence between them. Coupling between two objects is precisely classical coupling in the absence of inheritance. If a data member of an object is defined to be
public, then it results in common coupling. If an object is declared to be a friend of another object then it again results in common coupling [SC96]. [BE93] defined a friend as an object that is not hierarchically related to another object and has direct access to the underlying implementation to the other object. Outside external coupling and coupling from the side are also known as a friend object [SC96]. [SC96] defines two type of coupling: classical and inheritance. [SC96a] encourages the use of inheritance coupling and discourages the use of classical coupling. He suggests that the level of classical coupling should be as low as possible to promote maintainability and reusability. In contrast, the level of inheritance coupling should be as high as possible to maximize the extent to which we can extract elements of commonality. In addition, inheritance coupling is constrained by the domain being modeled. He supports his claim by providing an example. Card and Glass [CG90] suggested that a low fan-out is desirable since a high fan-out is characteristic of the large number of classes needed by the particular class in question. Haynes and Menzies [HM94] suggested that there might be a linear relationship between class coupling and its size [HS96]. They used Smalltalk systems to show that:

\[ \text{Coupling} = (0.03 \pm 0.0006) \times \text{SLOC} + 5.5 \pm 1.7 \quad (3.10) \]

where SLOC is the use of source lines of code as a measure of software size.

They suggested that the coupling equation might be used as the bases for a “a precise size estimation schema.” However, they noted that to accomplish the precise size estimation schema, the following prerequisites must be followed [HS96]:

1. Independent validation is required in a carefully controlled scientific manner.
2. Further study is required of classes with low coupling values.
3. Languages other than Smalltalk should be used.

4. The source of experimental variations needs to be carefully scrutinized.

CBO counts class/class connectivity other than by inheritance [HS96]. In systems where one class has high CBO “fan-out” count and all other classes have a CBO count of zero, a structured design is the result rather than on object-oriented design [KM93]. Henderson-Sellers suggested that CBO “design fan-out” is easily calculable from a design diagram. He added that at the detailed design level CBO is inadequate. The connection between two object classes permits several services of the server object class to be used. The collaborators (the sum of all services used) introduce a value for the metric NRM “Number of remote methods.” If two classes are coupled in analysis, it is very probable that in late design and implementation, coupling will expand to several connections as well as the addition of message passing, which does not show earlier [HS96]. Li and Henry [LH93] found that the [CK94] definition of coupling as ambiguous [HS96]. They introduced a new definition known as data abstraction coupling (DAC).

\[
\text{DAC} = \text{number of ADTs defined in a class.}
\]

[HS96] suggested that it is clear that both definitions exclude inheritance. Tight coupling of inheritance provides a potential complexity that is not foreseen in the development of traditional metrics [HS96]. Thus, subclasses can essentially access their superclasses’ internal data and methods. When excessive hierarchy depths occur, complexity results, which compounds when methods are overridden in descendant subclasses. High fan-out “class coupling” indicates excessive complex dependence [HS96]. High fan-in represents good object-oriented design and a high level of reuse is
one of the most important qualities of object-oriented designs. [LK94] proposed the number of parameters per method as the strength and quality of the coupling metric. They suggested that a good object-oriented design should pass few objects as parameters (or arguments) to messages. They proposed 0.7 as an appropriate upper limit. Excessive use of parameters and unused parameters are a source of confusion for object-oriented designers [HS96].

In Section 2 of this chapter, we define a CBO metric that measures the fan-out ratio (FOR) to evaluate a design. The FOR metric results are based on a collaboration matrix that shows the coupling involved between objects.

3.1.5 Response For a Class (RFC)

The use of the remote methods leads us to the fifth metric, RFC. RFC is the number of local methods and the number of remote methods. Response for a Class (RFC) is a set of methods that can be potentially executed in response to a message received by an object of that class [CK94]. It is a measure of the potential communication between the class and other classes. If one message invokes a large number of methods, maintenance and testing of the class become more complicated due to the greater level of understanding that is required from the designer. The following definitions are found in [CK94]:

\[
RFC = |RS|,
\]

where, \( RS \) is the response set of the class, given by

\[
RS = M_i \cup \bigcup_{j=1}^{n} \{R_{ij}\},
\]

where, \( M_i \) = set of all methods in the class and

\[
R_i = \{R_{ij}\} = \text{all methods called by } M_i.
\]
In [BBM96], RFC is defined as the number of functions directly invoked by member functions or operators of a class using C++. They also concur with [CK94] that classes with larger response sets implement more complex functionalities and are more fault-prone. [HS96] suggested that RFC simply addresses the notion of "how many methods are accessible from within the class in question." It, however, does not address the frequency of use from different parts of the class. MPC (message passing coupling), on the other hand, addresses the external methods [LH93]. If a message invokes numerous methods as a response, the class becomes more complicated and more testing and debugging are required. Also, the level of understanding becomes more complex for the tester. [LK94] noted that there are many different ways to measure the coupling. They discussed coupling in terms of: number of collaborating classes (CBO or fan-out), and the amount of collaboration (MPC or RFC.) [LK94] did not offer any result or any quantification [HS96].

An equivalent definition of the definition by [CK94] is found in [HS96]:

\[ RFC = NLM + NRM \]

where,

- \( NLM \) is the number of local methods and,
- \( NRM \) is the number of remote methods

As an example, assume:

A::f1() calls B::f1()
A::f2() calls B::f1()
A::f3() calls A::f4(), C::f1()
A::f4() no calls made

then

37
\[ RS = \{A::f1, A::f2, A::f3, A::f4\} \]
\[ U \{B::f1\} \]
\[ U \{B::f2\} \]
\[ U \{A::f4, C::f1\} \]
\[ = \{A::f1, A::f2, A::f3, A::f4, B::f1, C::f1\} \]

and

\[ RFC = 6 \]

CBO and RFC differ as follows [HS96]:

- CBO counts class/class connectivity other than by inheritance. It is a measure of fan out “number of collaborators.”
- RFC measures both internal and external communications.

We define a refined RFC (RRFC) metric in Section 2 of this chapter. Moreover, we provide examples that show the RRFC is more accurate than the RFC metric for certain cases.

### 3.1.6 Lack of Cohesion in Methods (LCOM)

The Lack of Cohesion in Methods (LCOM) metric is an essential metric in the design process. Cohesion is one of the most highly recommended characteristics of object-oriented methodology. (LCOM) is defined as the count of the number of method pairs whose similarity is 0 minus the count of method pairs whose similarity is not zero [CK94]. The definition used by [CK94] is as follows:

Consider Class \( C_1 \) with \( n \) methods \( M_1, M_2, \ldots, M_n \).

Let \( \{I_j\} = \) set of instance variables used by method \( M_i \).

There are \( n \) such sets \( \{I_1, \ldots, I_n\} \).

Let \( P = \{(I_i, I_j) | I_i \cap I_j = 0\} \) and \( Q = \{(I_i, I_j) | I_i \cap I_j \neq 0\} \).
If all $n$ sets $\{I_1\}, \ldots, \{I_n\}$ are 0 then let $P = 0$.

$$\text{LCOM} = |P| - |\emptyset|, \text{ if } |P| > |\emptyset|$$

$$= 0 \text{ otherwise.}$$

The degree of similarity for two methods $M_1$ and $M_2$ in class $C_1$ is given by:

$$\sigma(M_1, M_2) = \{I_1\} \cap \{I_2\}$$

where $\{I_1\}$ and $\{I_2\}$ are the sets of instance variables used by $M_1$ and $M_2$.

The LCOM used by [CK94] is derived from Bunge's [BU77] definition of "similarity" between two objects as the intersection of the sets of their properties. The higher the number of similar methods, the more cohesive the class. Additionally, LCOM is tied to the instance variables and methods of a class. Low cohesion increases complexity since the lack of cohesion implies that classes should possibly be split into two or more subclasses. Classes with poor cohesion should be divided into multiple classes [CK94] and [LK94]. Low usage of global variables is desirable since it indicates good object-oriented design. High usage of global references makes knowledge of objects available to all the objects in the system, encouraging unnecessary coupling [LK94]. In [DC97], \textit{LCOM} is defined as pairs of member functions that can be checked to determine whether or not they share instance variables. Two member functions that share at least one variable are a \textit{good} pair. A \textit{bad} pair is when two member functions do not share a variable [DC97].

In [BK98], module cohesion is defined by using Association-Based Cohesion Measures, which is based on Stevens, Myers, and Constantine's cohesion definition (SMC) [SMC74]. SMC includes coincidental, logical, temporal, procedural, communicational, sequential, and functional cohesion. LCOM can be applied to the
design module. It is derived from a design-level view of a module using an input/output
dependence graph (IODG). IODG is adapted from the variable dependence graph in
[LA93]. It is based on the data and control dependence relationships between
input/output components of a module. Input components are in-parameters and
referenced global variables. Output components are out-parameters, modified global
variables, and “function return” values. In [BK98], the Design-Level Cohesion (DLC)
Measure that uses six relations between a pair of output components based on the IODG
representation is adopted. It uses the strongest relation for each pair of outputs. The
output pair with the weakest cohesion determines the cohesion of the module.

Slice-Based Cohesion measures are used in [BK98]. A program slice is the
portion of the program that might affect the value of a particular identifier at the
specified point in the program. Slices are used to represent the functional components
of a module. Three functional cohesion measures, Weak Functional Cohesion
(WFC), Strong Functional Cohesion (SFC), and Adhesiveness (A) are introduced in
[BO94]. WFC is the ratio of glue tokens to the total number of tokens in a procedure.
SFC is the ratio of superglue tokens to the total number of data tokens in a procedure.
They define adhesiveness as the ratio of the amount of adhesiveness to the total possible
adhesiveness. **Glue tokens** are data tokens common to more than one data slice. The
data slice of a variable is the sequence of data tokens that has a dependence relationship
with the variable. The superglue tokens are tokens common to every data slice of a
module. The adhesiveness of a data token is the number of data slices to which the data
token is common.

40

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
For each design, we identify all input and output variables. We can also identify all the variables and whether they are bound statically or dynamically which will give us an indication of the data dependency. We can borrow, for example, the cohesion metric implementation for [LA93] where different types of dependence are defined to map the LCOM to the relations discussed below. We map the relations in the DLC to two pairs of output components that is based on the IODG representation. In [BK98], the data and control dependence are informally defined using the notation of [LA93] as follows:

1. A variable $y$ has a data dependence on another variable $x$ ($x \rightarrow^{d} y$) if $x$ reaches $y$ through a path consisting of a definition-use and a use-definition.
2. A variable $y$ has a control dependence on another variable $x$ if the value of $x$ determines whether or not the statement containing $y$ will be performed.
3. A variable $y$ is dependent on another variable $x$ ($x \rightarrow y$) when there is a path from $x$ to $y$ through a sequence of data or control dependence “dependence path.”
4. A variable $y$ has condition-control dependence on another variable $x$ ($x \rightarrow^{cc} y$) if $y$ has a control dependence on $x$, and $x$ is used in the predicate of a decision “if-then-else.”
5. A variable $y$ has iteration-control dependence on another variable $x$ ($x \rightarrow^{ic} y$) if $y$ has a control dependence on $x$, and $x$ is used in the predicate of a n iteration structure.
6. A variable $y$ has c-control dependence on another variable ($x \rightarrow^{c} y$) if the dependence path between $x$ and $y$ contains a condition-control dependence but not iteration-control dependence.
7. A variable \( y \) has **i-control dependence** on another variable \( x \) \((x \rightarrow ^i y)\) if the dependence path between \( x \) and \( y \) contains an iteration-control dependence.

The relations are:

1. **Coincidental relation** \((R_1)\):
   \[
   R_1(o_1, o_2) = o_1 \neq o_2 \land \neg (o_1, o_2) \land \neg (o_2, o_1) \land \exists x [(x \rightarrow o_1) \land [(x \rightarrow o_2)]
   \]
   where \( o_1 \) and \( o_2 \) are two outputs of a module that have neither dependence relationship nor dependence on a common input.

2. **Conditional relation** \((R_2)\):
   \[
   R_2(o_1, o_2) = o_1 \neq o_2 \land \exists x [(x \rightarrow c o_1) \land [(x \rightarrow c o_2)]
   \]
   where \( o_1 \) and \( o_2 \) are two outputs c-control dependent on a common input.

A variable \( y \) has a c-control dependence on another variable \( x \) \((x \rightarrow ^c y)\) if the dependence path between variables \( x \) and \( y \) contains a condition-control dependence and not iteration-control dependence.

3. **Iterative relation** \((R_3)\):
   \[
   R_3(o_1, o_2) = o_1 \neq o_2 \land \exists x [(x \rightarrow ^i o_1) \land [(x \rightarrow ^i o_2)]
   \]
   where \( o_1 \) and \( o_2 \) are two outputs i-control dependent on a common input.

A variable \( y \) has an i-control dependence on another variable \( x \) \((x \rightarrow ^i y)\) if the dependence path between variables \( x \) and \( y \) contains an iteration-control dependence.

4. **Communicational relation** \((R_4)\):
   \[
   R_4(o_1, o_2) = o_1 \neq o_2 \land \neg (o_1, o_2) \land \exists x [((x \rightarrow ^d o_1) \land (x \rightarrow ^d o_2) \lor ((x \rightarrow ^p o_1) \land (x \rightarrow ^p o_2))]
   \]
   where \( p,q \in \{d,c,I\} \), and \( p \neq q \). Two outputs are dependent on common input.
5. Sequential relation \( (R_s) \):

\[
R_s(o_1, o_2) = o_1 \neq o_2 \land ((o_1, o_2) \lor (o_2, o_1))
\]

where one output is dependent on the other output.

6. Functional relation \( (R_f) \):

\[
R_f(o_1, o_2) = (o_1 = o_2)
\]

where there is only one output in a module.

Message connections as well as instance variables are forms of cohesion within a class. We should minimize the use of global instance variables to avoid encouraging unnecessary coupling [LK94]. In [LK94] a threshold of .07 per method is suggested. [HS96] suggested to change a system measure of cohesion introduced by [F91] that was not meant for object-oriented systems. [F91] introduced the measure for functional languages. [HS96], however, replaced functional by “abstract” for object-oriented systems.

\[
\text{Cohesion Ratio} = \frac{\text{number of modules having (abstract) cohesion}}{\text{total number of modules}}
\]

[HS96] suggested that while a large value of LCOM indicates poor cohesion, a zero value does not necessarily mean good cohesion. [HS95b] proposed that a better LCOM measure should have values on a percentage range. In their words, “Perfect cohesion is considered to be when all methods access all attributes.” He defined a set of methods \( \{M_i\} \) \( (i = 1, \ldots, m) \) that access a set of attributes \( \{A_j\} \) \( (J = 1, \ldots, a) \). He defined the number of methods accessed by each method as \( \alpha(M_i) \) and the number of methods accessing each datum as \( \mu(A_j) \). The metric proposed is:
The LCOM metric introduced by Henderson-Sellers suggests that the requirements for LCOM include the ability to give values across the full range and not for any specific value to have a higher probability of attainment than any other. [HS96] criticizes [CK94] suggesting that LCOM = 0 does not necessarily mean good cohesion. Also there is guideline on the interpretation of any LCOM value. Is an LCOM value of 8 indicates a low, medium, or abysmal cohesion? They suggest that their LCOM metric gives values that can be uniquely interpreted in terms of cohesion. Dumota and Mingins [DM93] proposed the concept of cohesion as “a class is coherent if the methods work together to carry out a single, identifiable purpose.” They identified the set of method calls that is recursively extended to the set of \( n \)-calls within a specified visibility boundary [HS96]. Also, they defined the co-response of two methods with visibility boundary \( B \) as the common set of methods called from the two methods [HS96]. They defined the co-response as:

\[
\sigma^+_{B}(m,n) = \text{calls}_{B}(m) \cap \text{calls}_{B}(n). \quad (3.12)
\]

Dumota and Mingins [DM93] suggested that the larger the co-response set compared to the total number of distinct methods from the two methods, the greater is the degree of cohesion [HS96]. They then derived the cohesion distance, which is shown in formula (3.13) as:

\[
LCOM = \left( \frac{\sum_{j=1}^{g} \mu(A_j) - m}{1 - m} \right) . \quad (3.11)
\]
where 0 ≤ \( d_b(m,n) \) ≤ 1. The coherence distances obtained from this formula are then represented in a distance matrix and used to perform cluster analysis that permits the authors to draw a dendogram. For total dissimilar methods, the side branches occur at the top of the dendogram, and for totally similar methods they are at the base.

In [BK98] reverse engineering technology is used to generate an input/output dependence graph (IODG) from program code. They used a design-level cohesion (DLC) measure using an association-based approach similar to Stevens [SMC74]. Based on the result obtained by their research, the design-level measures can be obtained before code is written. Hence, it can be used to predict code-level cohesion values. The IODG model provides a visual representation that complements the quantitative information provided by the measures. The visual representation in effect helps the engineer to view an IODG diagram to determine if and how a candidate module should be restructured.

The LCOM introduced by [CK94] has limitations for some cases. They noted that "the LCOM metric for a class where \( |P| = |Q| \) will be zero. This does not imply maximal cohesiveness, since within the set of classes with LCOM = 0, some may be more cohesive than others." \( P \) is the number of pairs that have no similarity and \( Q \) is the number of pairs that have some similarity. While a high value of LCOM implies low similarity and low cohesion, a value of LCOM = 0 does not imply the reverse. If \( |P| ≤ |Q| \), LCOM = 0, and this can occur even for cases of obvious dissimilarity [HS96]. [HS96] extended the example supplied by [CK94] as follows:
\[ I_1 = \{a, b, c, d, e\} \]
\[ I_2 = \{a, b, e\} \]
\[ I_3 = \{x, y, z\} \]
\[ I_3 = \{x, y, z, d\} \]

Consider a class supporting the first three sets.

\[ |P| = 2 \]
\[ |Q| = 1 \]

\[ \Rightarrow \text{LCOM} = 1 \]

Consider a class supporting all four sets.

\[ |P| = 3 \]
\[ |Q| = 3 \]

\[ \Rightarrow \text{LCOM} = 0 \]

When the four sets were considered, LCOM = 0 implying a good cohesive structure, yet intuitively, \( M_1 \) and \( M_2 \) are a pair of cohesive methods and so are \( M_3 \) and \( M_4 \). However, the designer would suspect that two classes should be formed and not one [HS96]. [HS96] supplied another example to reinforce their point. A class with four methods accessing variables according to:

\[ I_1 = \{a, b, c\} \]
\[ I_2 = \{c, d, e\} \]
\[ I_3 = \{e, f, g\} \]
\[ I_4 = \{a, g, h\} \]

\[ \Rightarrow \text{LCOM} = 0 \]

Even though that LCOM = 0, this example shows lack of cohesion.

We define a formula that computes the cohesion ratio and develop a cohesion matrix that shows the cohesiveness of methods within a class object in Section 3.2. We then show examples to compare the LCOM introduced by [CK94] with the new LCOM metric and show results of the comparisons.
3.2 Computation of Design Assistant Parameters

The objective of this research is to define methods to improve existing designs while maintaining the same functionality. The process is:

- For every object in the class hierarchy not including the root, compute the following metrics:
  - DIT
  - WMC
  - NOC
  - CBO
  - RFC
  - LCOM

- For each metric, evaluate the existing designs using algorithms designed in this research.

- Based on the result of the algorithms, recommend actions.

- Redesign the class.

3.2.1 Redesign Using DIT Metric

We now define the methods to use the DIT metric as a part of the redesign process. A class hierarchy in a structure tree has a base called the root. A low number of levels in a hierarchy suggests difficulties in finding the abstractions and specializations to optimize reuse through inheritance. On the other hand, a large number of levels suggests no subclassing by specialization (is-a) [LK94].

An example of subclassing by type is shown in Figure 3. There are some car models produced by GM corporation that have similar specifications and looks but different names, for instance the Chevy Tahoe and the GMC Yukon. The Tahoe is a subclass under Chevy trucks whereas the Yukon is a subclass under GMC trucks. GMC
trucks are at the same level of the Chevy object and one level above the Yukon. If we merge the Yukon truck with the Chevy Tahoe, we reduce testing and reuse code more efficiently. The new tree is depicted in Figure 4. [BBM96] introduced a Hypothesis-DIT ‘H-DIT’ for the DIT metric. They suggested that well-designed object-oriented systems are those structured as forests of classes, rather than as one very large lattice.

Figure 3 Subclassing by type

Figure 4 Subclassing by type after using code reuse

Replicated with permission of the copyright owner. Further reproduction prohibited without permission.
We define a Depth of Inheritance Tree (DIT) algorithm to determine whether we need to extend the number of levels in a class hierarchy. We then introduce examples to show how the algorithm works. The algorithm is given in Figure 5. In the DIT algorithm, we use thresholds where the minimum level of a tree is 2 and the maximum is 6 levels. These thresholds are based on the recommendations of [HS96], [LK94] and [BBM96].

The DIT algorithm determines if we need to extend the number of levels in a hierarchy by checking the degree of similar methods “inherited” and instance variables used among the objects in one level. After we find the objects that are most similar in the use of inherited methods; we then rank them by using the ranking factor, $P_i$. After we sort the percentages, we then make the second highest ranked object obtained from the sorting procedure the child of the highest ranked object. This process results in adding one level to the hierarchy. We show an example of a hierarchy tree that is depicted in Figure 6 where the number of levels is greater than 6. Let us assume that the number of levels is 7. So, the two levels to be affected by this algorithm are only the sixth and seventh levels. Classes $C_2$ and $C_3$ are merged with class $C_1$ without affecting any of the levels preceding level 6 in the class hierarchy.

We show another example where the number of levels is less than 2, and more than one object exists in the child’s level. Figure 7 shows the class hierarchy for the example. After we apply the DIT algorithm, we make object class $C_4$ the child of $C_2$ based on the percentage of inherited methods in each class. The class hierarchy changes in depth, which utilizes the inheritance property and recommends a depth of at least two levels.
DIT Algorithm

Let $T_{\text{min}}$ be the minimum number of levels in a class hierarchy, where $T_{\text{min}} \geq 2$.
Let $T_{\text{max}}$ be the maximum number of levels in a class hierarchy, where $T_{\text{max}} \leq 6$.
Let $P_{0j}$ be the percentage of inherited methods in object $o_j$.

\[
\begin{align*}
\text{Max} \left[ \forall \text{ objects in a class hierarchy} \right] &= \text{depth of the tree} = n \\
\end{align*}
\]

Algorithm_DIT(n);

Begin

If $(n < T_{\text{min}})$ and number of subclasses $< 2$ Then
   Delete_object() /* Check if it should exist at all in the hierarchy*/
Else
   If $(n < T_{\text{min}})$ Then
      Rank objects at level $n$ by similarity, using percentages of inherited methods $P_{IMj}$, as a ranking factor;
      Rank_objects(n);
      Make the second highest ranked object the child of the highest ranked object (Thus adding one level to the hierarchy tree);
   Else
      If $(n > T_{\text{max}})$ Then
         Merge all objects at level $n$ with parent at level $n-1$;
         Call Algorithm_DIT(n-1);
      Else
         No action required;
      Endif;
   Endif;
Return;

Rank_Objects(n);

Begin
   $A = \text{array} [1..N_n]$;
   for each object $o_j$, $j=1..N_n$, where $N_n$ = total number of objects at level $n$
      $A[j] = P_{0j}$; /* Percentage of inherited methods in object $o_j$ */
   Sort($A$); /* $A[1]$ = highest ranking */
Return $A$;
End Rank_Objects;

End Algorithm_DIT;

Figure 5 DIT Algorithm
Figure 6 Hierarchy of Depth 1

Figure 7 Hierarchy of Depth 1 and with more than one child

Figure 8 Hierarchy of Depth 2 after applying DIT algorithm
Also, it should be noted that the change made is minimal and did not affect the original design. The new class hierarchy is shown in Figure 8. The hierarchy in Figure 8 shows that the inheritance characteristic is more efficiently utilized than the hierarchy in Figure 7. Moreover, rather than deleting the hierarchy in Figure 7 since it had an inheritance depth of 1, we extended the hierarchy by 1 level and therefore, justified its existence. Since half of the methods existing in class object \( C_4 \) are inherited methods, this situation will not affect the structure or the behavior of this because \( C_4 \) is still inheriting those methods from the root. The overridden and the pure methods in \( C_4 \) will remain the same without the need for changing their structure. The DIT algorithm also works in the same manner for a hierarchy that has depth larger than 6 levels.

3.2.2 Redesign Using WMC Metric

When an object requests a service from another object via message passing, we have the following scenario [LT97]:

- If a method for the service exists, the object will execute its own method.
- If a method for the service does not exist, the object will delegate the execution of the service requested to an object that has a delegation relationship with it.

There are three types of methods in an object: overridden, inherited, and pure. Overridden methods are methods inherited from a superclass by a subclass. It is known that numerous overrides indicate a design problem. The design problem stems from the fact that specialization becomes nonexistent [LK94]. We measure the severity of overridden methods by considering the percent of total methods that are overridden, that is:
\[
\text{% of Overridden Methods (P_o) = \frac{\text{Number of overridden methods in an object (N_o)}}{\text{Number of methods in an object}}}\quad . \quad (3.14)
\]

[LK94] suggested that number of overridden methods should be three or less as a threshold. However, the number they proposed does not take into account the size of a system. Some systems might need to override more than three methods per an object class. So, we propose an initial threshold of \( P_o = 40\% \). It might be argued that this is still a high percentage for a threshold. The reason is that we are allowing for the number of methods overridden by an object class. Designers tend to override some methods without regard to the goal set for the original method. In other words, some methods are overridden that should not have been overridden.

In addition to overridden methods, WMC includes the methods inherited by a subclass. A subclass inherits both the behavior and the instance variables from the superclass. We consider the percent of total methods that are inherited as:

\[
\text{% of Inherited Methods (P_i) = \frac{\text{Number of inherited methods in an object (N_i)}}{\text{number of total methods in an object}}}\quad . \quad (3.15)
\]

A high \( N_i \) value is a sign of a good design. We set the value at 70\%. We set this value high in order to promote the inheritance and specialization concepts of object-oriented methodologies. If we get a low percentage after applying this formula, we should investigate if these methods can fit better under another class or another object [LK94].

The third component of the WMC is the methods newly added by a subclass. If an object has no local methods, then its existence is questionable [LK94]. We expect each object to have at least one method to justify its existence; otherwise, such object
should cease to exist. The number of newly added methods in the hierarchy should
decrease as we move down the levels. Again we consider the percent of total methods
that are pure as shown in:

% of Pure Methods \( (P_p) \) = \( \frac{\text{Number of pure methods in an object}(N_p)}{\text{Number of total methods in an object}} \) . (3.16)

We set a percentage of 20\% for \( N_p \) as a general threshold value. Some systems
might need to have a larger value. If it is found that if a high number of new methods
are introduced in the lower levels of a deeply nested hierarchy, some methods should be
moved to the higher levels allowing more objects to use the logic [LK94].

Formulas 3.14, 3.15 and 3.16 apply to one object class in the class hierarchy.
The formulas for a class hierarchy are shown below:

\( \text{AOM} \) is the average number of overridden methods,

\( \text{AIM} \) is the average number of inherited methods and,

\( \text{AOP} \) is the average number of pure methods.

\( \text{AIM}, \text{AOM}, \text{and AOP} \) are formally defined in (3.14.1), (3.15.1), and (3.1.16)
respectively.

\[
\text{AOM} = \frac{\sum_{i=1}^{n} \text{Number of overridden methods in object}_i}{\sum_{i=1}^{n} \text{Number of total methods in object}_i} . \quad (3.14.1)
\]

where \( n \) is the number of objects in a class hierarchy.
Number of inherited methods in object 
AIM = \[ \sum_{i=1}^{n} \frac{\text{Number of inherited methods in object } i}{\text{Number of total methods in object } i} \] \hspace{1cm} . (3.15.1)

Number of total methods in object 
where \( n \) is the number of total objects in a hierarchy.

Number of pure methods in a object 
AOP = \[ \sum_{i=1}^{n} \frac{\text{Number of pure methods in a object } i}{\text{Number of total methods in a object } i} \] \hspace{1cm} . (3.16.1)

Number of total methods in a object 
where \( n \) is the total number of objects in a class hierarchy.

The process to evaluate the existing design is:

- Count the total number of methods in all the objects in the class hierarchy.
- For each method, classify each method as: Overridden, Inherited, or Pure "New, Local"
- Apply formula (3.5) to each method based on the method type.
- Evaluate the design by applying the metric algorithm in Figure 9.
- Compare the percentages with the threshold values.
- Redesign the class.

Algorithm WMC, given in Figure 9, supplies the application of the WMC metric to evaluate a design. All testing of methods in each object is done recursively, one level at a time "breadth wise," starting from the terminal nodes "leaves" of the class hierarchy "tree" and working up to the root.

Let \( N_o \) be the number of overridden methods in an object.

Let \( N_i \) be the number of inherited methods in an object.
Let $N_p$ be the number of pure methods in an object.

Let $N_m$ be the total number of methods in an object.

$$P_o = \frac{N_o}{N_m} \leq 1$$

$$P_i = \frac{N_i}{N_m} \leq 1$$

$$P_p = \frac{N_p}{N_m} \leq 1$$

where $P_o + P_i + P_p = 1$

For discussion purposes, we use a table that contains four quadrants for each class. Figure 10 shows the contents of the four quadrants. $C_n$ indicates the class number in the hierarchy. The overridden methods quadrant names all the methods that are inherited and then overridden by the subclass. The inherited methods quadrant names all the methods that are inherited from all its superclass ancestors. The pure methods quadrant names all the methods that are newly introduced in $C_n$. The total number of methods used in an object class $C_n$ is depicted in the fourth quadrant, which is the sum of all methods in the other three quadrants. For example, if $A_{c_4}$ appears in the overridden methods quadrant, that means the current object class has overridden method $A$ that is inherited from object class 4. If method $N$ appears in the pure methods quadrant, then $N$ is the only newly introduced method in $c_4$. If $B$ appears in the inherited methods quadrant, then class $c_4$ inherited only method $B$ from a superclass. Consequently, 4 appears in the total number of methods quadrant. Recall that $P_o$, $P_i$, and $P_p$ are the percentages of the overridden, inherited and pure methods. We apply the WMC algorithm to the hierarchy tree example depicted in Figure 11.
**WMC Algorithm**

Let $T_0$ be the threshold for total of overridden methods in an object, where $0 < T_0 < 1$. Let $T_1$ be the threshold for total of inherited methods in an object, where $0 < T_1 < 1$. Let $T_p$ be the threshold for total of pure methods in an object, where $0 < T_p < 1$.  

$n = \# \text{ of levels in a class hierarchy:}$

\[ \text{for each level } i = n \text{ to } 1 \text{ do} \]

\[ \text{for each object } o_i \text{ at level } I \]

Compute:

\[ P_o \leftarrow \frac{N_o}{N_m} \]

\[ P_i \leftarrow \frac{N_i}{N_m} \]

\[ P_p \leftarrow \frac{N_p}{N_m} \]

Call Test-Object($O_i$);

\[ \text{end for;} \]

\[ \text{end for;} \]

Test_Object($o$);

Begin

If $P_o > T_o$
Then

Move_Object_Up($O_i$);

If $P_p > T_p$
Then

Merge_With_Parent($O_i$);

If $P_i > T_i$
Then

Delete_Object($O_i$);

End;

End;

Move_Object_Up($O$);

Begin

Move object up to the parent hierarchy level;
Make previously overridden methods pure. (in addition to the object's newly introduced methods.)
Remove the methods inherited from the parent object.
Invoke the methods that were inherited from the parent object via message passing.

End;

Figure 9 WMC Algorithm
WMC Algorithm (continued)

Merge_World_Parent(O);
Begin
  Merge child object with parent.
  Add the pure methods in the child object to the pure methods of the parent
  object.
  Compare \( P_o \) with \( T_o \).
  If \( P_o > T_o \) Then
    Call Test_Object(O); “This is done recursively.”
  End;
Delete_Object(O);
Begin
  Transfer any introduced methods to the parent object.
  Delete child object.
  The \# of methods returned \( M_r \) can be calculated as follows:
  \[ M_r = (1 - P_o - P_p) \times N_m \]
End;
End Algorithm_WMC;

Figure 9 (WMC Algorithm continued)

<table>
<thead>
<tr>
<th>( C_n )</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERRIDEN METHODS</td>
<td>INHERITED METHODS</td>
</tr>
<tr>
<td>PURE METHODS</td>
<td>TOTAL NUMBER OF METHODS</td>
</tr>
</tbody>
</table>

Figure 10 WMC Table
We apply the WMC algorithm to the hierarchy tree example depicted in Figure 11. In this example, we start from the terminal nodes of the hierarchy and compare each of the methods percentages against the threshold values.

\[ N = 3, \quad T_o = 0.4 \quad T_i = 0.7 \quad T_p = 0.2 \]

**C_6**: Delete Object(C_6)

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{Ac}_4 & \text{Bc}_1 & \text{CDE} & \text{FGH} \\
\hline
\text{N} & \text{Total} & 9 \\
\hline
\end{array}
\]

\[ P_o = .11 \quad P_i = .78 \quad P_p = .11 \]

Compute the percentages of overridden, inherited, and pure methods. \( P_i \) was greater than \( T_i \), which resulted in the deletion of object class \( C_6 \) and the moving of its methods to \( C_4 \).

**C_3**: Delete Object(C_3)

**C_1**: no change is made since object class \( C_3 \) had no overridden or pure methods.

**C_4**: Delete Object(C_4)

\( P_i \) is greater than \( T_i \), which results in the deletion of object class \( C_4 \) and the moving of its methods to \( C_1 \).

\[
\begin{array}{|c|c|c|}
\hline
\text{Ac}_1 & \text{Bc}_1 & \text{CDEF} \\
\hline
\text{GHN} & \text{Total} & 10 \\
\hline
\end{array}
\]

\[ P_o = .2 \quad P_i = .4 \quad P_p = .4 \]

**C_5**: Merge With Parent(C_5)
$P_p$ is greater than $T_p$, which results in merging object class $C_5$ with $C_2$. Add the pure methods of the $C_5$ to $C_2$.

<table>
<thead>
<tr>
<th>$C_2$</th>
<th>$A$</th>
<th>$B$</th>
<th>$D$</th>
<th>$E$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K, L, M, I$</td>
<td>Total</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_{C_5}$</td>
<td>$P_o = 0.1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$P_f = 0.45$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$P_p = 0.45$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$C_5$: $\text{Merge\_With\_Parent}(C_i)$

$C_2$: $\text{Merge\_With\_Parent}(C_2)$

Both classes $C_1$ and $C_2$ are merged with $C_0$ leaving only one node, namely the root node, in the hierarchy.

The results obtained from the class hierarchy in Figure 11 show that hierarchy has collapsed leaving only the class root node. It is a worst case scenario for the algorithm. A redesign of the whole class hierarchy in this case is needed. The designer should reevaluate all of the pure as well as the overridden methods in the class objects.

The following example shows another application of the algorithm. It is based on the data shown in Figure 12. In this example, we start from the terminal nodes of the hierarchy and test each of the method percentages against the threshold values.

$N = 3$, $T_o = 0.4$  $T_i = 0.7$  $T_p = 0.2$

$C_9$: No action required.

$C_5$: $\text{Move\_Object\_Up}(C_5)$
Figure 11 WMC tables
Move $C_5$ one level up and change its name to $C_n$ since $P_o > T_o$. Compare $C_n$ again at its new level when the current level's testing is exhausted. Object class $C_n$ will contain the previously overridden methods as pure ones as well as its pure methods.

$C_6$: No action required.

$C_7$: No action required.

$C_8$: No action required.

Check $C_n$ "$C_5$" once again. Note that the number of pure methods introduced is larger than the threshold, namely $P_p > T_p$. Merge child object $C_n$ with its parent $C_0$. Add the pure methods in the child object to the methods in the parent object.

$C_n$: Merge_With_Parent($C_n$)

The root class has four more pure methods that are inherited by all other subclasses.

$C_1$: No action required.
$C_2$: No action required.

$C_3$: No action required.

$C_4$: $\text{Delete}_\text{Object}(C_4)$

$P_i$ is greater than $T_i$, which results in the deletion of object class $C_4$ and the moving of its methods to $C_0$.

The application of the algorithm results in the creation of the new class hierarchy tree, which is depicted in Figure 13.

In this section, we provided two examples of applying the WMC metric algorithm to an existing design. One example was a worst case scenario in which the class hierarchy collapsed and another example where some modifications were applied to the hierarchy resulting in a new design. The new changes where related to the thresholds. The redesigned hierarchy in Figure 13 shows fewer object classes in the hierarchy. Fewer children implies less maintenance and testing. For instance, if we look at the most right terminal node of the tree hierarchy (GMC Yukon), we find that object class $C_9$ entity has been not been changed. Its parent $C_8$ has not been changed either. Since object class $C_4$ has been deleted, we moved $C_9$ and $C_8$ one level up without affecting the hierarchy tree structure. Note that even though the depth of the
hierarchy has been reduced by one level, the object classes states and methods were not affected and the design retained its original intended goal. The same applies to the most left node $C_5$ of the hierarchy tree where the hierarchy tree structure remained unaffected by its merging with its parent object class $C_1$.

3.2.3 Redesign Using NOC Metric

A large number of subclasses within a class hierarchy indicates difficulty in modifying and testing, thus introducing more complexity into the class design. Children with common methods that are not overridden or changed should be grouped together. We should combine these classes to a higher level in the hierarchy, reducing the amount of testing and modification. Coupling, cohesion and size are relevant for the NOC metric. There are also many other factors affecting the class and its subclasses such as the number of public methods, the number of private methods, and the number of instance variables. Inheritance is also a main characteristic that is related to NOC. The question becomes “should we distribute the NOC for the whole inheritance tree?”

The specialization and reuse ratio introduced by [HS96] leads to the Average NOC for the whole inheritance tree. An example is given in Figure 14:

The specialization ratio, $S$, measures the extent to which a superclass has captured the abstraction since a large value of $S$ indicates a high degree of reuse by subclassing [HS96]. The reuse ratio, $U$, indicates the extent to which the implementers of the class library have been able to inherit from their own classes to create new classes [HS96]. In the example in Figure 14, there are 4 subclasses and 2 superclasses.

$$S = \frac{4}{2} = 2$$
$$U = \frac{2}{5} = 0.4$$
Figure 12 Class Hierarchy
Figure 13 Redesigned Class Hierarchy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
NOC (1) = 3
NOC (2) = NOC (4) = NOC (5) = 0
NOC (3) = 1
Average NOC = 4/5, where $NOC_i = \text{number of children for each class}$

Figure 14 Example of NOC

These values indicate a broad, shallow structure and a shallow hierarchy. Since there is general agreement that depth is better than breadth, we suggest few classes per hierarchy level. We work with the number of superclasses and subclasses until we get a desired ratio for both reuse and specialization.

To map the NOC metric to a design, we derive upper and lower bounds for NOC. To derive the bounds, we introduced a best case scenario of classes hierarchies which satisfies the following properties:

- $S = 1$
- $U \equiv 1$
• $2 \leq \text{DIT} \leq 6$

• Breadth at higher levels is recommended

An Inverted Complete Binary Tree (ICBT) with all the leaf nodes inheriting from the root node satisfies the previous properties. Such a structure is depicted in Figure 15 with $2 < \text{DIT} < 7$. It has a DIT of $\log_2 n$, where $n = \text{NOC} + 1$. For example, for $\text{NOC} = 15$ ($n = 16$, including the root node) we get the hierarchy depicted in Figure 16, where

$$U = \frac{15}{16} = 0.9375 \text{ } \{\text{a perfect Reuse ratio for this value of } n\}$$

$$S = \frac{15}{15} = 1 \text{ } \{\text{a perfect Specialization ratio}\}$$

Using the minimum and maximum thresholds of the DIT Algorithm, we derive the following lower and upper bounds for $\text{NOC}$ of ICBT based on the fact that the lower bound of DIT is equal to 2 and the upper bound is equal to 6:

$$2^{\text{DIT}_{\text{min}}} - 1 \leq \text{NOC} \leq 2^{\text{DIT}_{\text{max}}} - 1$$

For $\text{DIT}_{\text{min}} = 2$ \& $\text{DIT}_{\text{max}} = 6$,

$$3 \leq \text{NOC} \leq 63.$$ 

Correspondingly, we derive lower and upper bounds for $U$ based on the perfect reuse $U$ ratio. We substitute the values for $U$ to obtain $\frac{3}{4}$ for the lower bound and 0.98 for the upper bound.

$$\frac{3}{4} \leq U \leq \frac{63}{64}$$

$$0.75 \leq U \leq 0.98$$

These values serve as guidelines for the designers for the NOC metric. Enforcement of these values is already embedded in Algorithm_DIT. Figure 16 shows an ICBT where $\text{NOC} = 16$ (including the root node.)

68
Figure 15 ICBT hierarchy Tree. A Generic for all cases from DIT > 2 and DIT < 7
Figure 16 ICBT. A case where NOC = 15 and DIT = 4
Figure 17 illustrates the DIT algorithm after applying it to a hierarchy with \( n < 2 \). The DIT and NOC metric are dependent on one another. Logically, the deeper the levels of DIT, the greater the number of children that are found in the hierarchy. [HS96] suggested that inheritance could be addressed initially by the DIT and NOC metrics. In addition, [KO93] suggested that the number of distinct inheritance hierarchies reflect the number of broad domain foci within the systems. [DV94] also introduced the notion of conceptual entropy, which suggests the direct relationship of the NOC and DIT metrics. Conceptual entropy suggests that deeper hierarchies along with high number of children are less likely to be a true specialization.

![Figure 17 Tree with DIT < 2](image)

We apply the DIT algorithm, where \( n < T_{\text{min}} \), to get the following:

\[
\begin{align*}
    n &= 1 \\
    A &= [p_{c_1}, p_{c_2}, p_{c_3}] \\
        &= [.60, .50, .67] \\
    \text{Sort } A; \\
    A &= [.67, .60, .50]
\end{align*}
\]
\[ [P_{c3}, P_{c1}, P_{c2}] \]

\[ \therefore C_i \text{ will become the child of } C_3. \]

We then apply the DIT algorithm again, where \( n > T_{\text{max}} \) to get the results shown in Figure 18 which depicts a hierarchy tree with \( n > 6 \) in the algorithm. Figure 19 shows that at DIT level 8 we merged object class \( C_0 \) from level 8 with its immediate superclass “parent” \( C_m \) from level 7. At DIT level 7 we merged object classes \( C_0 \) \( C_m \) with \( C_i \) from level 7. Finally, we merged object classes \( C_0 \) \( C_m \) \( C_i \) with \( C_0 \) in level 6 as well as merging \( C_n \) and \( C_k \). Figure 20 depicts the final picture after applying the algorithm.

In this Section we showed two cases, one case where the depth of the hierarchy tree was less than 2, and another case greater than 6. We computed the Percentage of Inherited methods for each object class. Then we sorted the computed values in descending order. Object class \( C_i \) became the child of object class \( C_3 \) since their \( P_i \) values are closest to one another, which indicates that they both use similar methods. In the second case, DIT was 8, which is greater than 6 (threshold.) At level \( n = 8 \), we merged object class \( C_0 \) with its parent object class \( C_m \). At level \( n = 7 \), we merged object class \( C_0C_m \) with object class \( C_k \). Finally at level \( n=6 \), we merged object class \( C_0C_m \) and \( C_i \) with their parent object class \( C_{.} \). Also, we merged \( C_n \) and \( C_k \) at level \( n=6 \). After applying the DIT algorithm as well as the ICBT hierarchy tree, the number of children and the number of hierarchy levels decreased. The decrease in the levels and the children should positively impact testing and maintenance.
Figure 18 Tree with DIT > 6
3.2.4 Redesign Using CBO Metric

We now define a method to use the CBO metric to evaluate a design. Algorithm CBO, given in Figure 21, defines the method.

We define the Fan-out Ratio (FOR) for object $O$ as $\text{FOR}_O$ as:

$$\text{FOR}_O = \frac{\text{Fan-Out}_O}{\text{NOC} - l_O}$$
We subtract $l_0$ because a subclass will not collaborate with its ancestors in the same class hierarchy. Similarly, we compute Fan-in Ratio (FIR) as:

$$FIR_o = \frac{\text{Fan-Ino}}{\text{NOC} - l_0}$$

The class hierarchy in Figure 22 depicts a Fan-in/Fan-out example. Based on the hierarchy depicted in Figure 22, we devise a matrix shown in Figure 24, which shows the collaboration between the object classes. Figure 23 shows how the matrix collaboration works. If class A collaborates with class B, then a value of 1 is set at the intersection cell of columns A and B. If no collaboration exists between class A and B, then a value of zero is set in the intersection cell. The L column signifies the DIT level in which the class object exists. $\text{FOR}_o$ contains the fan-out ratio, computed by:

$$\frac{\text{Sum Class Row}}{\text{No. of Children} - \text{object class level}}$$

The same fan-out count for any two object classes at different depth levels in a hierarchy does not indicate that coupling is equally advantageous/disadvantageous. In fact, the object at the deeper level in the hierarchy will have a higher $\text{FOR}_o$ value to reflect the fact that objects at deeper level should have a lower fan-out. The reason for a higher $\text{FOR}_o$ value is that at deeper levels more specialization and reuse are made than in the shallower levels.

From the example in Figure 22, where object H has the same fan-out count as object B, object H has a more severe case of coupling than B due to its depth. Consequently, we set up the same threshold fan-out value for all the levels in the hierarchy. As discussed in the CBO section, high fan-out indicates a design problem,
whereas high fan-in indicates a good design. We set a threshold of 20% as FORo. We set a value of 1 for any two collaborating object classes in a hierarchy, which is depicted in the collaboration matrix in Figure 24. We show an example of a class hierarchy in Figure 22. We then derived the matrix in Figure 24, which contains FORo and FIRo values. Objects A, B, C, D, E, G, H, and I should be reevaluated by the designer to see if such collaborations between these objects is justified in the class hierarchy. If there is no justification, then the previous objects should be considered for redesigning.

In summary, the collaboration matrix helps the designer find which object classes are coupled with one another. The fan-out ratio FORo, as well as the depth of the level in the hierarchy tree, indicates to the designer if whether a redesign of an object is necessary. The more object classes an object class collaborates with, the higher the need to redesign that object class. The example in Figure 22, object classes such as A and I have to be considered for redesigning. Object class A has three other object classes (B, C, and G) collaborating with it. This type of collaboration indicates that some methods in object class A cannot be executed unless that some methods in B, C, and G are executed, or vice versa. Consequently, coupling is the result, a result that is undesirable in object-oriented designs. The designer, in return, will have to redesign object class A in order not to have the other methods in object classes, namely B, C, and G, be dependent on its methods. A creation of a new object class can be another feasible solution whereby its methods will not be dependent on those in object class A, B, C, and G.
CBO Algorithm

3 Object O, let \( l_o \) be the level of that object in the hierarchy, where
\[
l_o \leq DIT, \text{ Fan-Out}_o \text{ is the fan-out of an object, and Fan-In}_o \text{ is the fan-in of an object.}
\]

Based on the given design, create a matrix that shows the collaboration of objects
Loop until all collaborations between objects are accounted for
  If a collaboration between two objects A & B exists then
    Set the intersection matrix cell of A & B to 1
  Else
    Set the intersection matrix cell of A & B to 0
End_Loop1;

Loop until all cells values have been read
  for every object O
    Compute
      \[ \text{FOR}_O = \text{Fan-Out}_O / (\text{NOC} - l_O) \]
    \[ \text{"NOC is the number of children"} \]
End_Loop2;

End Algorithm_CBO;

Figure 21 CBO Algorithm
Figure 22 Class Hierarchy Collaboration
### Figure 23 Generic Collaboration Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>L</th>
<th>FOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3/8</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2/7</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2/7</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2/7</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2/7</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2/6</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1/6</td>
</tr>
<tr>
<td>L</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIR</td>
<td>3/8</td>
<td>4/8</td>
<td>2/8</td>
<td>0</td>
<td>1/7</td>
<td>2/7</td>
<td>2/7</td>
<td>1/6</td>
<td>1/6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 24 Collaboration Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>L</th>
<th>FOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2/8</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2/7</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2/7</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2/7</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2/7</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2/6</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1/6</td>
</tr>
<tr>
<td>L</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIR</td>
<td>3/8</td>
<td>4/8</td>
<td>2/8</td>
<td>0</td>
<td>1/7</td>
<td>2/7</td>
<td>2/7</td>
<td>1/6</td>
<td>1/6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
3.2.5 Redesign Using RFC Metric

Algorithm RFC in Figure 25 supplies the application of the RFC metric to evaluate a design. We propose a Method Invocation Graph (MIG) to traverse all possible paths upon methods invocation in a class hierarchy. The MIG traces non-inheritance coupling in each object class in the hierarchy. MIG shows the path(s) that an invoked method might take. A message invokes a method, which in turn might invoke other methods locally or remotely. MIG traverses the path(s) that helps calculate a refined RFC metric described below. In the RFC presented by [CK94] and analyzed by [HS96], all methods (NLM) are counted in RFC regardless of whether they invoke other methods or they are not involved in coupling, in addition to all remotely invoked methods (NRM). RFC does not count any methods beyond the first level. In this research, MIGs exhaust all method invocation for each method in every class, repeatedly until one of the following methods is reached:

- A method that has already been invoked along the path of the current method chain.
- A method that invokes no further methods.

We construct one MIG for each method in a class that invokes other methods. The nodes of the graph are of the form C::f, where f is a method in class C. If C1::f1 invokes C2::f2, and C2::f2 invokes C3::f3, then we connect the three nodes as follows:

```
C1::f1 ----> C2::f2 ----> C3::f3
```
If, in turn \( C_3::f_k \) invokes \( C_2::f_m \), then we connect them as follows, only because \( C_2::f_m \) already appeared in the current chain:

\[
\begin{array}{c}
C_1::f_k \\
\rightarrow_
C_2::f_m \\
\rightarrow_
C_3::f_k
\end{array}
\]

Otherwise, a new node is created for \( C_2::f_m \). For the method invocation chart in Figure 26, we construct the corresponding MIGs for methods \( f1(), f2(), \) and \( f3() \) in class \( A \), \( f3() \) in class \( B \), and \( f1() \) and \( f2() \) in class \( C \). We get a total of six MIGs, which is the total number of methods invoking other methods. A MIG graph gives the designer a clearer picture about which classes involve high RFC counts. It gives the designer an overall view of how the whole hierarchy is affected by the RFC count.

We now count the RFC value defined by [HS96] for all the classes in Figure 26.

We show the MIGs for classes \( A \), \( B \), and \( C \) in Figure 27.

Class \( A \) has four methods: \( f1(), f2(), f3(), \) and \( f4() \). We now compute:

\[
\begin{align*}
RS &= \{A::f1, A::f2, A::f3, A::f4\} \\
&\cup \{B::f1, B::f2, C::f3\} \\
&\cup \{B::f1\} \\
&\cup \{A::f4, B::f3, C::f1, C::f2\} \\
&= \{A::f1, A::f2, A::f3, A::f4, B::f1, B::f2, B::f3, C::f1, C::f2, C::f3\}
\end{align*}
\]

\[
\text{RFC} = 10.
\]

Class \( B \) has three methods: \( f1(), f2(), \) and \( f3() \). We now compute:

\[
\begin{align*}
RS &= \{B::f1, B::f2, B::f3\} \\
&\cup \{C::f1\} \\
&= \{B::f1, B::f2, B::f3, C::f1\}
\end{align*}
\]

\[
\text{RFC} = 4.
\]
RFC Algorithm

RFC Algorithm comprises two parts:

1) MIGs
2) RRFC

MIGs:

∃ Class C, let f_i be the method of that class in the hierarchy.

We construct a graph (MIG) for each method in a class that invokes other methods.

The node of the graph have the form C::f_i

For every method in a class C_1::f_i that invokes C_2::f_m and C_2::f_m that invokes C_3::f_k,

we create a graph that have three nodes for each method and 2 edges connecting the three nodes.

Based on the MIGs created, we define RRFC

Let MIG_n be all method invocation graphs for a class.

Let M_m be the total of all methods involved in coupling.

RRFC:

We now compute RRFC for each class by:

\[ RRFC(C) = \sum_{i=1}^{n} MIG_i + \sum_{j=1}^{m} M_j \]

We compute RFC of [CK94] and compare results.

End Algorithm_RFC;

Figure 25 RFC Algorithm
Class C has three methods: fl(), f2(), and f3(). We now compute:
\[
RS = \{C::f1, C::f2, C::f3\} \\
   \cup \{B::f2, B::f3\} \\
   = \{B::f2, B::f3, C::f1, C::f2, C::f3\}
\]
\[
RFC = 5
\]

The redefined RFC (RRFC) metric for any class is the sum of lengths of all method chains of all MIGs for that class, plus the number of local methods involved in coupling. For example, the first MIG of class A has a count of 4, the second 1 and the third 8. Therefore, the RRFC of A is 13 + 4 (number of methods involved in coupling.) Similarly, the RRFC for B is 4 and for C is 5.

To show the value of the RRFC count in measuring coupling, we add one more class, D, to the example depicted in Figure 28. Only class B’s MIGs are affected as shown in Figure 29. We then calculate the new RFC and RRFC.

Class B has three methods: fl(), f2(), and f3(). We now compute:
\[
RS = \{B::f1, B::f2, B::f3\} \\
   \cup \{C::f1\} \\
   \cup \{D::f1\} \\
   = \{B::f1, B::f2, B::f3, C::f1, D::f1\}
\]
\[
RFC = 5 \\
RRFC = 5
\]

Class D has two methods: fl(), and f2(). We now compute:
\[
RS = \{D::f1, D::f2\} \cup \{\Phi\} \\
   = \{D::f1, D::f2\}
\]
\[
RFC = 2 \\
RRFC = 1
\]

Let us now assume that class D has 10 methods instead of the two displayed in Figure 28. We then get the following RFC and RRFC:

Class D has ten methods: fl(), f2(), ..., f(10). We now compute:
\[
RS = \{D::f1, D::f2, ..., D::f10\} \cup \{\Phi\} \\
   = \{D::f1, D::f2, D::f10\}
\]
\[
RFC = 10 \\
RRFC = 1
\]
RRFC responds better to changes in coupling than does the RFC. For the case of D having 10 methods, RRFC remained constant whereas RFC responded dramatically to this change, even though the change had no bearing on coupling! Contrast that with class A, which has high coupling (especially fan-out). RFC in this case could provide incorrect guidance regarding the need to redesign class D. Our RRFC count still showed that class A needs to be redesigned and no action is necessary for class D. Before introducing class D in our example, the weight of class A’s RFC was $10/19 = 52.6\%$ of the total RFC count for all classes. On the other hand, the weight of class A’s RRFC was $17/26 = 65.4\%$ of the total RRFC count for all classes. After introducing class D, where D has 10 methods, the weight of class A’s RFC dropped dramatically to $10/30 = 33\%$, whereas RRFC merely dropped to $17/28 = 60.7\%$. The results of the example obtained after introducing class D is another indication of the improved response of RRFC to coupling over the RFC metric.

We have shown that RRFC responds better to changes in coupling than that of the RFC. We supplied an example in Figure 28. In this example, it was shown that even though we increased the number of methods in class D, RRFC remained constant. On the other hand, RFC responded dramatically to this change, even though that the change had nothing to do with coupling. RFC also showed that the designer need to redesign class D depicted in Figure 28, whereas RRFC did not. The example showed that the designer should not redesign that object class. In other words, RRFC is more accurate measure for our use than the RFC metric.

We define an algorithm in Figure 30 to show when and how to redesign the methods involved in non-inheritance coupling in a class.
Figure 26 Method Invocation Chart

**Class A:**

![Diagram of Class A method invocations]

**Class B:**

![Diagram of Class B method invocations]

**Class C:**

![Diagram of Class C method invocations]

Figure 27 MIG(s) for classes A, B, and C
Class B:

Figure 29 MIG(s) for class B

RRFC Redesign Algorithm

∃ classes C_{1..n}
∃ methods M_{1..i} in class C_m, ∃ methods M_{1..j} in class C_n.

Let class C_m have the method(s) that invoke other method(s) in class C_n.
Let class C_n have its method(s) invoked by other method(s) from class C_m.

For each class in the hierarchy

Compute RRFC & the weight for each class.

Weight = RRFC for each class / (RRFC of class C_m + RRFC of class C_n)

If the weight of a class is > 50%

Then only certain methods M_{1..j} in class C_n should be merged with the methods in class C_m. // The designer must decide which methods should be moved without affecting the inheritance property.//

End;

Figure 30 RRFC Redesign Algorithm
3.2.6 Redesign Using LCOM Metric

We develop a **cohesion matrix** that shows the cohesiveness of methods within a class object. If there is cohesion between methods $m_1$ and $m_2$, we set a value of 1 in the intersection cell; otherwise, a value of 0 is set. We then introduce a formula that computes the cohesion based on the values in the matrix. Figure 31 depicts the definition of the LCOM algorithm.

**LCOM Algorithm**

*LCOM Algorithm comprises two parts:*

1) **Cohesion Matrix**
2) **Cohesion Ratio**

*There exist methods $M_1$ and $M_2$, let $I_m$ be the set of instance variables used in these methods.*

**Cohesion matrix:**

*For every instance variable in each method*

*If a variable is used in more than 1 method*

*Then the intersection cell between these methods is set to 1*

*Else*

*The intersection cell is set to 0;*

**Cohesion Ratio:**

*We now compute the cohesion ratio for each method by:*

Cohesion Ratio = \[
\frac{\text{No. of instance variables in methods involved in cohesion}}{\text{Total no. of methods in a class}}.
\]

*We compute LCOM of [CK94] and compare results of both LCOMs.*

*End Algorithm LCOM;*

Figure 31 LCOM Algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The formula is:

\[
\text{Cohesion Ratio} = \frac{\text{No. of instance variables in methods involved in cohesion}}{\text{Total number of methods in a class}}.
\]  \hspace{1cm} \text{(3.17)}

In formula (3.17), we used the definition supplied by [CK94] for the number of instance variables and shown in Section 3.1. The larger the number of similar methods, the more cohesive the class. We add the contents of the cells from the *cohesion matrix* and then divide by the number of total methods. In our measure LCOM metric, we never get an LCOM of zero that gives the elusion that it is a maximal cohesiveness. Figure 32 depicts the *cohesion matrix* that shows the cohesion of methods in an object class. The *cohesion matrix* shows the number of method pairs whose similarity is not zero (using similar instance variables). We show an example in Figure 33 and 34 of a case where LCOM is zero if we use the [CK94] measure. However, if we use the *cohesion matrix* we obtain different results.

We now compute the LCOM defined in [CK94]:

\[
\begin{align*}
I_1 \cap I_2 &= \{b, c\} & I_1 \cap I_3 &= \emptyset & I_1 \cap I_4 &= \emptyset \\
I_2 \cap I_3 &= \{c, d, e\} & I_2 \cap I_4 &= \emptyset & I_3 \cap I_4 &= \{g\}
\end{align*}
\]

LCOM = number of null intersections - number of nonempty intersections

\[
= 3 - 3 \\
= 0
\]

There is cohesion between three sets of instance variables in the methods tested. However, the LCOM count given by [CK94] results in a value of zero that is an indication of no degree of similarity. They stated that:
“The larger the number of similar methods, the more cohesive the class, which is consistent with traditional notions of cohesion that measure the inter-relatedness between portions of a program. If none of the methods of a class display any instance behavior, i.e., do not use any instance variable, they have no similarity and the LCOM for the class will be zero.”

Whereas if we use the Cohesion Ratio to show that similarity exists between the methods, we get the following:

\[
\text{Cohesion Ratio} = \frac{12}{4} = 3
\]

<table>
<thead>
<tr>
<th></th>
<th>(m_1)</th>
<th>(m_2)</th>
<th>(m_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(m_2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(m_3)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

General Cohesion Matrix

Figure 32 General Cohesion Matrix

\[
\begin{align*}
\text{I}_1 &= \{a, b, c\} \\
\text{I}_2 &= \{b, c, d, e\} \\
\text{I}_3 &= \{c, d, e, f, g\}
\end{align*}
\]

Figure 33 LCOM = 0

89

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<table>
<thead>
<tr>
<th>Cohesion Matrix</th>
<th>m₁</th>
<th>m₂</th>
<th>m₃</th>
<th>m₄</th>
<th>Cohesion</th>
</tr>
</thead>
<tbody>
<tr>
<td>m₁</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>m₂</td>
<td>2</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>m₃</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>m₄</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Cohesion Count</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 34 Cohesion Matrix

This result shows that the object class tested has a cohesion ratio of 3. We propose that a value of 1 for the cohesion ratio be used as the cutoff between high and low cohesion. The reason is that a value of 1 means that, on average, every variable was involved in cohesion, although not every variable has to be involved to get the value of 1. Hence, no redesign process should be applied. From [CK94], the LCOM measure object class redesign should be considered. The closer the Cohesion Ratio is to zero, the higher the probability of an object class redesign process. We then have to collapse the object class into other object classes where similar methods must be put together to ensure a higher degree of cohesion in the class. The Cohesion Ratio will also be applied individually to all object classes in the class hierarchy. We then can add all the ratios obtained from the Cohesion Ratio and compute the average. The designer tests the lowest ratios obtained from the previous calculation. If the instance variables in the methods tested display a low degree of similarity, then the designer will have to disperse these methods by creating new object classes where the dispersed methods will exist. Hence, the designer will ensure higher degree of cohesion in the newly created object classes, since all methods in these classes will be similar in their functionality.
We give another example in Figure 35 and compare the results obtained with the example shown earlier in Figure 33. In Figure 35, we added more instance variables that are involved in cohesive methods. Figure 36 shows the cohesion matrix based on the data in Figure 35.

![Diagram showing cohesion matrix]

Figure 35 After adding more instance variables

<table>
<thead>
<tr>
<th>COHESION MATRIX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>$m_1$</td>
</tr>
<tr>
<td>$m_1$</td>
</tr>
<tr>
<td>$m_2$</td>
</tr>
<tr>
<td>$m_3$</td>
</tr>
<tr>
<td>$m_4$</td>
</tr>
</tbody>
</table>

Cohesion Count 2 4 4 4 14

Figure 36 Based on data in Figure 35

We now compute the LCOM as per [CK94] definition:

$I_1 \cap I_2 = \{b\}$  
$I_1 \cap I_3 = \emptyset$  
$I_1 \cap I_4 = \{a\}$  
$I_2 \cap I_3 = \{d, e\}$  
$I_2 \cap I_4 = \{e\}$  
$I_3 \cap I_4 = \{e, f\}$
LCOM = number of null intersections - number of nonempty intersections

= 1 - 5
= (-4) \rightarrow 0

We now evaluate the cohesion ratio to get $14/4 = 3.5$. The Cohesion ratio and LCOM of [CK94] counts differ for the same number of instance variables used. However, if we increase the number of instance variables in methods that are involved in cohesion, the Cohesion ratio increases whereas LCOM for [CK94] remains the same. Figure 37 defines an algorithm that shows the redesign process using the LCOM metric.

In summary, we have introduced a cohesion ratio and a cohesion matrix that shows methods that are cohesive in an object class. Based on the data in the matrix, we then compute the LCOM. In Figure 34 we computed LCOM defined in [CK94] and then we computed LCOM based on our ratio. We used another example in Figure 33 to show that redesigning the object is not recommended, whereas if we use [CK94] LCOM then a redesign is recommended.

3.3 Summary

Table 4 shows a summary based on the algorithms described in this chapter. For each metric, Table 4 depicts the thresholds, which are general values that we chose based on the literature recommendations. The table also shows the process we apply to evaluate the existing design.
**LCOM Redesign Algorithm**

∃ classes \(C_{1..n}\)
∃ methods \(M_{1..l}\) in classes \(C_{1..n}\)
∃ instance variables \(V_{1..i}\) in methods \(M_{1..l}\)

We perform the steps in the LCOM algorithm defined in Figure (31).

Based on the results obtained from the Cohesion Ratio, do the following:

For each method \(M_t\) in each class \(C_n\) in the hierarchy

Begin

If Cohesion Ratio > 1 then
    No action is required
Else
    \(C_i\) is to be collapsed into other classes where similar methods must be put together;

\[\text{Sum} = \text{Cohesion ratio for each class } C_i + \text{Sum};\]

End;

Compute Cohesion Ratio Average = \(\text{Sum} \div \text{number of classes } (C_n);\)

Compare the average with each of the classes average;

If Cohesion Ratio < Average then “Low degree of similarity” then
    Disperse the methods by creating new objects where the only similar methods exist;

Figure 37
<table>
<thead>
<tr>
<th><strong>Table 4 Metrics Application</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>THRESHOLD</strong></td>
</tr>
<tr>
<td><strong>DIT</strong></td>
</tr>
<tr>
<td><strong>WMC</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>NOC</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>CBO</strong></td>
</tr>
<tr>
<td><strong>RFC</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>LCOM</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 4. DESIGN EVALUATION ASSISTANT (DEA)

The Design Evaluation Assistant is a software assistant that reads the textual representation of a design, which complies with the Unified Modeling Language (UML). DEA consists of seven programs and generates reports based on the results obtained from the programs. The main program provides the different menu options to the user. The other six programs are the metric algorithms discussed in Chapter 3. The result is a toolkit developed in C++ that implements the following main requirements:

1. Input an object-oriented skeleton design written in conformance with the Unified Modeling Language (UML).

2. Store parts of the input obtained in step 1 in a data file
   - Class
   - Objects within a Class
   - Methods “functions”
   - Instance Variables

3. Process the information obtained in step 1 and then apply the six metrics developed in [CK94] to evaluate the designs.

4. Obtain the results from step 3 and generate reports.

5. If needed, propose a redesign of the original design and produce a modified design that maintains the original design goals.

The Unified Modeling Language (UML) is the unification of Booch & OMT, OOSE methods, and a number of other methodologists [UML96]. James Rumbaugh developed OMT and Ivar Jacobson created the OOSE. Booch, Rumbaugh, and
Jacobson joined their work to produce the (UML). One problem the methodologists addressed in the Unified Model Language is the modeling of distributed systems.

Goals of the unification effort were:

- Model systems using object-oriented concepts.
- Enable end users, domain experts, and analysts to use the (UML) model.
- Address the issues of scale inheritance in complex systems.
- Provide a language usable by both humans and machines.

4.1 Textual Representation of Unified Modeling Language (UML) Notations

We use the UML notations and convert the graphical representations to textual representations [UML97]. For each notation used in UML in the left-hand side, we show the equivalent textual representation on the right-hand side:

<table>
<thead>
<tr>
<th>UML Notation</th>
<th>Shape</th>
<th>Meaning</th>
<th>Textual Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(Note)</td>
<td></td>
<td>Comment or Constraint</td>
<td>Regular text</td>
</tr>
<tr>
<td>f(Constraint)</td>
<td></td>
<td>Attributes or Associations</td>
<td>Text written between {}</td>
</tr>
<tr>
<td>f(Name)</td>
<td></td>
<td>Identify a model element</td>
<td>Regular text</td>
</tr>
<tr>
<td>f(Label)</td>
<td></td>
<td>String attached to a graphical symbol</td>
<td>Regular text</td>
</tr>
<tr>
<td>f(Property String)</td>
<td></td>
<td>Keyword = Name of a property String denoting its Value</td>
<td>Regular text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>f(Class_Name)</td>
<td></td>
<td>Name compartment</td>
<td>First letter in caps</td>
</tr>
<tr>
<td>f(Attributes)</td>
<td></td>
<td>List compartment</td>
<td>List of strings</td>
</tr>
<tr>
<td>UML Notation</td>
<td>Shape</td>
<td>Meaning</td>
<td>Textual Representation</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------</td>
<td>----------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>f(Class_Type) →</td>
<td>Class type</td>
<td>Text between guillemets &lt;&lt; &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>f(Class_Template) →</td>
<td>Descriptor</td>
<td>Regular text</td>
<td></td>
</tr>
<tr>
<td>f(Object) →</td>
<td>Instance of a Class</td>
<td>Regular Text</td>
<td></td>
</tr>
<tr>
<td>f(Utility) →</td>
<td>Global variables &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f(Metaclass) →</td>
<td>Class instances</td>
<td>&lt;&lt;Metaclass&gt;&gt; of class</td>
<td></td>
</tr>
<tr>
<td>f(Attributes) →</td>
<td>Show attributes in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Classes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Visibility name: type-expression = initial-value {property-string} where visibility is one of:

+ public visibility
  # protected visibility
- private visibility

where name is an identifier string;
where type-expression is a type of an attribute;
where initial-value is the initial value of a newly created object.
where property-string indicates values that apply to the element.

show operations in classes

Visibility name (parameter-list): return type-expression {property-string}

where visibility is one of:

+ public visibility
  # protected visibility
- private visibility

where name is an identifier string;
where return-type-expression is a type of value returned by the operation;
where name: type-expression = default value.
where name is the name of a formal parameter.
where type-expression is the specification of an implementation type.
where default-value is an optional value expression of the parameter.
where property-string indicates values that apply to the element.
<table>
<thead>
<tr>
<th>UML Notation</th>
<th>Shape</th>
<th>Meaning</th>
<th>Textual Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(\text{Multiplicity}) \rightarrow )</td>
<td>( \Rightarrow )</td>
<td>A subset of non-text string comprising a negative open sequence of integer intervals</td>
<td></td>
</tr>
<tr>
<td>( f(\text{Association_Class}) \rightarrow )</td>
<td>Single model element</td>
<td>Association path &amp; association class have a single name.</td>
<td></td>
</tr>
<tr>
<td>( f(\text{Generalization}) \rightarrow )</td>
<td>Taxonomic</td>
<td>Regular Text relationship between a more general element and a more specific element. discriminator: powertype where discriminator is the name of subtype; where powertype is the name of a type whose instances are subtypes of another type.</td>
<td></td>
</tr>
</tbody>
</table>

### 4.2 Description of DEA

Figure 38 depicts the process in which the DEA operates.

![DEA Overview Diagram](image)

Figure 38 DEA Overview
Figure 39 shows the menu options available in the DEA tool.

![DEA Main Menu](image)

**Figure 39 DEA Main Menu**

### 4.2.1 Evaluating the DIT Metric on A UML Design

The data from Section 3.1 is in Figure 7 and Figure 8. It has been converted to the UML textual representation as described in Section 4.1. The data has the following structure:

```c
struct record_DIT
{
    char class_name[20]; // Total number of methods
    float num_of_methods;
    float num_of_inhrtd; // Total number of inherited methods
    int num_subclasses; // Number of subclasses
    int level_num; // Hierarchy level of object class
    float percent_of_inhrtd; // Percentage of inherited methods
};
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The algorithm is:

Loop;
Read the following values:
Num_of_methods, num_of_inhrtd, num_of_subclasses, level_num;
Implement DIT_Algorithm:
Compute the percentages of inherited methods;
Sort the percentages in ascending order.
End_loop;

The output generated by the execution of this program is in Figure 40.

<table>
<thead>
<tr>
<th>Class</th>
<th>Total Methods</th>
<th>Inherited subclass Level</th>
<th>% of Inherited</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>100</td>
<td>55</td>
<td>0.55</td>
</tr>
<tr>
<td>C4</td>
<td>100</td>
<td>50</td>
<td>0.5</td>
</tr>
<tr>
<td>C3</td>
<td>100</td>
<td>40</td>
<td>0.4</td>
</tr>
</tbody>
</table>

**C4 becomes child of C2.**

The output in Figure 40 shows that a modification in the design is recommended. The object C4 should become the object child of C2. We have a total of hundred methods in each class. The percentage of inherited methods for class C2 is .55,
.5 for class C4 and .4 for class C3. We sorted the percentages of inherited methods in ascending order. Since the percentage of the inherited methods in C4 is the second highest, then C4 becomes the child of C4.

4.2.2 Evaluating the WMC Metric on A UML Design

The data for this metric is in Figure 11. The data is textually interpreted and has the following structure:

```c
struct record_wmc
{
    char class_name[20]; // Total number of methods
    float num_of_methods; // Total number of inherited methods
    float num_of_inhrtd;  // Total number of pure methods
    int num_of_pure;      // Total number of overridden methods
    int num_of_ovrrdn;    
    char name_of_parent[10]; // Parent Class
    int level_num;        // Hierarchy level of object class
    float percent_of_inhrtd; // Percentage of inherited methods
    float percent_of_ovrrdn; // Percentage of overridden methods
    float percent_of_pure;  // Percentage of pure methods
};
```

The following algorithm implements the WMC_Algorithm. All testing of methods in each object is done recursively, one level at a time "breadth wise," starting from the terminal nodes "leaves" of the class hierarchy "tree", and working up to the root. The algorithm is:

Loop;
Read the following values:
Implement the WMC_Algorithm:
Compute the percentages for each method type;
Check for each method percentage and type against the threshold value used for each method type;
If percent_of_inhrtd > Threshold of inherited methods
Then
    Delete object;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
If percent_of_ovrrdn > Threshold of overridden methods
Then
   Move object up one level;
If percent_of_pure > Threshold of pure methods
Then
   Merge object with parent;
End_Loop;

The output generated by the execution of this program is:

<table>
<thead>
<tr>
<th>Class</th>
<th>Total Class methods</th>
<th>Inhert'd methods</th>
<th>Pure methods</th>
<th>overriding methods</th>
<th>% inhered methods</th>
<th>% Pure methods</th>
<th>% overriding methods</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6</td>
<td>9</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0.89</td>
<td>0</td>
<td>0.11</td>
<td>delete object C6</td>
</tr>
<tr>
<td>C5</td>
<td>10</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0.5</td>
<td>0.2</td>
<td>0.3</td>
<td>merge with parent C2</td>
</tr>
<tr>
<td>C4</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0.75</td>
<td>0.25</td>
<td>0</td>
<td>delete object</td>
</tr>
<tr>
<td>C4</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>delete object</td>
</tr>
<tr>
<td>C2</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0.71</td>
<td>0.14</td>
<td>0.14</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.5</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
</tbody>
</table>

Figure 41 WMC Algorithm output

From the output in Figure 41, we recommend that the designer redesign the original design. The class hierarchy collapsed, and the designer should reevaluate all of the pure as well as the overridden methods in the class objects. We compared the percentages of inherited, overridden, and pure methods with the thresholds that were defined in Chapter 3. Based on each comparison, an action is recommended. In classes
C6, C4 and C3 the percentage of inherited methods was larger than the threshold value, which resulted in the deletion of the class. The percentage of pure methods was larger than the threshold in class C5, and therefore the class was merged with its parent C2. Classes C2 and C1 required no action.

4.2.3 Evaluating the NOC Metric on A UML Design

The data for this example is from Figure 17. The data has the following structure:

```c
struct record_noc
{
    char class_name[20];   // Class name
    float num_of_methods;  // Total number of methods
    float num_of_inhrtd;   // Total number of inherited methods
    int num_of_pure;       // Total number of pure methods
    int num_of_ovrdn;      // Total number of overridden methods
    int num_of_vars;       // Total number of instance variables
    int level_num;         // Hierarchy level of object class
    float percent_of_inhrtd; // Percentage of inherited methods
};
```

The algorithm for the NOC metric is as follows:

```c
Begin
Initialize all variables;
Loop;
    Read the following values:
        Class_name, num_of_methods, num_of_inhrtd, num_of_subclasses, level_num;
        num_of_pure, num_of_ovrdn;
        Implement DIT_Algorithm:
        Compute the percentages of inherited methods;
        Sort the percentages in ascending order;
    End_loop;
    Output the results;
        Write the results to NOC.rpt;
End;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Since DIT and NOC metric are dependent on one another, we used the DIT algorithm for both metrics. In this example the number of inherited methods is two while the total number of methods is 6. We compute the percentage of inherited methods and get .33. We apply the same computation to each class and then sort the percentages. We make the class with second highest percentage the child of the class with the highest percentage. Thus, the number of levels increases by one, satisfying the threshold discussed earlier in the redesign using NOC section.

4.2.4 Evaluating the CBO Metric on A UML Design

The data for this section is shown in Figure 24. The data has the following structure:
struct record_CBO
{
    char object_name[4]; // Object Name
    float num_of_children; // Number of children in the hierarchy
    float co_array[10][10]; // Matrix
    int  level_num; // Level in the hierarchy
};

The algorithm is:

Initialize Matrix;
Begin
Loop2;
    Loop1;
    Read the following values:
    Matrix values;
    Sum the values per raw;
    End_Loop1;
    Implement CBO_Algorithm:
    Compute the fan out ratio;
    Compare the fan out ratio to the threshold;
    Based on the result, make the recommendation;
    If co_array[loop1][loop2] > fan_out_ratio
        Recommend redesigning the object
    Else
        The object has low fan_out ratio and no action is required.
    End_loop2;
End;

Based on the data supplied in the collaboration matrix in Figure 24 the following recommendations were made as shown in Figure 41. Classes A, B, C, D, F, G, and H have a high fan-out ratio, which necessitates the redesign of these objects. The threshold used for this program is 20%. Classes E and I have low fan-out ratio that should not require a redesign. It is recommended that the designer redesign the hierarchy based on the results that show high coupling.
Figure 43 CBO Algorithm output

4.2.5 Evaluating the RFC Metric on A UML Design

The data used is given in Figure 28. The data has the following structure:

```c
struct record_rfc {
    char class_name1[10];  // Class name1
    char calling_func[10];  // Calling method in Class 1
    char class_name2[10];  // Class name2
    char called_func[10];  // Called method in Class 2
};
```

The following is the algorithm that we used in the RFC metric:

```c
/* Note: These objects needs to be redesigned due to high coupling. */
```
Begin Algorithm;
Initialize the counts;

Loop; // Input design data
    Read the following values:
    Class_name1,
    calling_func,
    class_name2,
    called_func;
    Compute RFC used by [CK94];
    Implement the RRFC_Algorithm:
    Compute RRFC supplied by us;
    // Based on the data structure that we used, we know the called and the calling function. That made it easy for us to compute the RFC and RRFC by using the sets. //
    We saved the original “old” RRFC count before we added more methods;
    We saved the original “old” RFC count before we added more methods;
    Compare RFC with RRFC and generate recommendations;
    We compared the “old” RFC with the “new” RFC count;
    We compared the “old” RRFC with the “new” RRFC count;
    If “old” RFC $\leftarrow$ new “RFC” count
        Then write the following note:
        Note: It was shown that even when we increase the number of methods in class D RRFC remains constant.
    Else
        No action is required;
End_loop;
End_Algorithm;

The sample output is shown in Figure 44. The RFC and RRFC counts obtained are based on the data depicted in Figure 28. After we have increased the number of methods in Class D from 2 methods to 10, the RFC has increased while the RRFC remained constant.
4.2.6 Evaluating the LCOM Metric on A UML Design

We implement the cohesion ratio and use the data in the matrix depicted in Figure 34. The following data structure is used:

```c
struct record_lcom
{
    float num_of_methods; // Number of methods
    float co_array[10][10]; // Cohesion Matrix
};
```

Following is the algorithm used in the LCOM program:

Begin LCOM_Algorithm
Loop2;
  Loop1;
    Read the following values:
    Number_of_methods;
    
Figure 44 RRFC Algorithm output
Implement the LCOM Algorithm:
For every row in our matrix:
Begin
    Add all the 1s per row;
    Sum Number of methods in a count;
End;
Compute Cohesion Ratio;
End_loop1;
End_loop2;
Compare Cohesion Ratio with threshold value and generate recommendations;
If $LCOM < 1$ then
    Write ('This class hierarchy need to be redesigned due to low level of cohesion among its methods ')
Else
    Write ('This class hierarchy need not be redesigned due to high level of cohesion among its methods ');
End;
End LCOM Algorithm;

Figure 45 LCOM Algorithm output
Figure 45 shows the sample output generated after executing the LCOM program. From the example we used in cohesion matrix in Figure 34, the LCOM count is 3.5 which is larger than the threshold assigned. The recommendation suggests that no redesign is necessary.

In summary, we introduced the DEA tool and its specifications. DEA reads designs that comply with the UML notations and evaluate designs. DEA uses the metric algorithms introduced in Chapter 3. The examples use the same data used in Chapter 3. DEA then provided the recommendations based on the design evaluation after applying the metric algorithms. DEA suggested solutions to detected design problems.

The order in which the six metrics algorithms should be run is:

- DIT
- NOC
- WMC
- CBO
- RFC
- LCOM

The DIT algorithm must be the first. It is essential to know if the class hierarchy being evaluated is going to be restructured. In Section 3.2.1, we presented two examples where the depth of the hierarchy changed. Recall that the thresholds we used were 2 < DIT < 7. If the depth of a hierarchy is less than 2 or greater than 6, then the hierarchy should be restructured. Once the hierarchy is restructured, the other metrics values of the NOC, WMC, RFC, CBO, and LCOM will correspondingly change. The
restructuring that would take effect necessitates running those metric algorithms once again, which will yield different results. That is why we must run the DIT metric algorithm at the beginning.

The NOC metric is highly dependent on the DIT metric. We have presented this dependency in Section 3.2.3. After we check the hierarchy against the ICBT tree, we implement the DIT algorithm to evaluate the design. It is possible that the hierarchy structure will change based on the recommendations done by the DIT algorithm. Moreover, in each object child in the hierarchy there will exist inherited methods and instance variables that could be overridden. These overridden methods will affect the outcome of the results obtained from running the WMC algorithm. Also, in the WMC algorithm we account for the pure and the inherited methods. Consequently, if a child's level has been changed, whether away from or toward the root, then the results of the WMC algorithm will be inaccurate if the algorithm is run before the NOC algorithm.

The WMC algorithm is to run after the DIT algorithm for obvious reasons. The DIT algorithm changes the layout of the class hierarchy, which in return will affect the inherited, overridden and pure methods. For example, if a method that is involved in non-inheritance coupling is moved to a different level, that movement will drastically change the WMC value. The change in the WMC count will give the designer the wrong recommendation. The same applies to running the WMC algorithm before the NOC algorithm.

The CBO collaboration matrix helps the designer find which object classes are coupled with one another. The fan-out ratio FORo, as well as the depth of the level in the hierarchy tree, indicates to the designer whether a redesign of an object is necessary.
Since the depth of the level in the hierarchy is crucial to running the CBO metric algorithm, we have to run the previous algorithms ahead of the CBO algorithm. In addition, an object class whose depth level has changed, whether by merging, or moving it up one level, will result in erroneous values if the CBO algorithm is run before the DIT, NOC, and WMC algorithms.

The RRFC algorithm depends on the MIG graphs. The MIG graph traces non-inheritance coupling in each object class in the hierarchy. MIG shows the path(s) that an invoked method might take. A message invokes a method, which in turn might invoke other methods locally or remotely. The RFC for locally or remotely invoked methods, which exist within classes having a depth level change, will yield inaccurate results. The inaccurate results are obtained because the non-inheritance coupling between the classes will differ based on the changes that were made in the hierarchy. Also, if the number of classes (NOC) or the number of methods (WMC) change, the RRFC algorithm will not obtain the correct results, because we might have more or fewer children and methods depending on the action taken. This is why we should run the RRFC algorithm after the DIT, NOC, WMC, and CBO respectively.

Finally, the LCOM algorithm should be run last. Once the designer has accurate knowledge of the class hierarchy, he/she then can work on the cohesiveness of methods. The LCOM algorithm does not require changes to the depth level. Also, it is not related to the process of merging classes, deleting classes or moving them in the hierarchy. The LCOM algorithm deals only with the methods and their degree of similarities in the classes and that is why it should be run when the design is stable.
CHAPTER 5. SUMMARY AND CONCLUSIONS

The objective of this research was to use object-oriented design metrics to detect design and to help correct design problems before the coding stage begins. We defined six metric algorithms to evaluate object-oriented design hierarchy classes. The algorithms discovered design problems, and provided the user alternative solutions. Providing the alternative solution without changing the intended goal of the design helps reduce time spent on maintenance and testing of the design.

In this research, we divided the work into four major sections which correspond to Chapters 1 through 4 as follows: 1) background, 2) related work, 3) introduction of design assistant parameters as well as computations of design assistant parameters, and 4) the Design Evaluation Assistant (DEA). We discussed the key elements for good object-oriented designs. We included definitions and discussions of the main concepts of applying design metrics to object-oriented designs. In addition, Chapter 1 contained a brief discussion of the six design metrics that are used in this research.

In Chapter 2, we presented the related background information for this research. It included work to introduce object-oriented metrics that can be applied in the design stages. In addition, Chapter 2 discussed some of the limitations of the related work. Moreover, the material referenced in Chapter 2 furnishes a motivation for this research.

In Chapter 3, Section 3.1 was mainly dedicated to the definition and the comprehensive discussion of the design assistant parameters used in this research. We defined the following six metrics: Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object Classes (CBO), Response for a Class (RFC), and Lack of Cohesion in Methods (LCOM). We
defined metric algorithms and applied the algorithms on class hierarchy designs in Section 3.2. We then compared the results, after applying the metrics on the designs with the results of applying the six metrics introduced by [CK94]. In Section 3.3, we summarized the findings in a summary table.

Finally, Chapter 4 included a description of the DEA assistant and the UML. In Section 4.1 we described the conversion of the UML notations to textual representation. In Section 4.2, we showed the use of DEA assistant to evaluate examples of object-oriented design systems. This section showed the results of implementing the DEA on the designs. For each example, we showed the data structure that we used for the designs as well as the results obtained after applying the algorithms on the designs. In Sections 4.1 through 3, we show the conclusions, the contributions of this research and the different research directions for future work, respectively.

5.1 Conclusions and Contributions

The main conclusion of this research is that metrics can be successfully used to improve object-oriented designs. The metric algorithms that we defined evaluated object-oriented designs and discovered design problems. The algorithms that we defined in this research have the following features, which are beneficial to the later stages of the life cycle.

1. The DIT metric algorithm determines whether we need to extend the number of levels in a hierarchy by checking the degree of similar methods "inherited" and instance variables used among the objects in one level. After we find the objects that are most similar in the use of inherited methods, we then rank them by using the ranking factor, $P_t$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
After we sort the percentages, we then make the second highest ranked object obtained from the sorting procedure the child of the highest ranked object. This process will save the designer time and effort creating hierarchies that are either too deep or to shallow.

(2) The WMC metric algorithm application to an object-oriented design shows if the class hierarchy collapses or if some modifications need to be applied to the hierarchy resulting in a new design. The new changes where related to the thresholds that we defined in Chapter 3. The redesigned hierarchy shows fewer object classes in the hierarchy. A low number of children implies less maintenance and testing. It was also shown that even though the depth of the hierarchy has been reduced by one level, the object classes states and methods were not affected and the design retained its original intended goal.

(3) The NOC metric algorithm, which depended on the DIT algorithm, and the ICBT hierarchy tree showed some interesting results. Once we decreased the hierarchy levels and the children, testing and maintenance should be reduced. Also, the algorithm showed that maintaining breadth at higher levels is a desirable feature. Therefore, the inheritance feature, which is one of the most important features of object-oriented programming, is preserved.

(4) The CBO algorithm helps the designer find which object classes are coupled with one another. It includes the fan-out ratio FORo, and the collaboration matrix. The collaboration matrix shows which methods in
a class are dependent on other methods in other object classes. Consequently, undesirable coupling is the result. Based on the results obtained from the collaboration matrix and the FOR ratio, the designer has an overall picture of the methods dependency. Consequently, he can either move dependent methods to the same object class, or he can merge the coupled methods to reduce coupling.

(5) The RFC algorithm has two parts, Method Invocation Graph (MIG) and a refined RFC count. MIG traverses all possible paths upon methods invocation in a class hierarchy. MIG traces non-inheritance coupling in each object class in the hierarchy. MIG shows the path(s) that an invoked method might take. A message invokes a method, which in turn might invoke other methods locally or remotely. MIG traverses the path(s) that helps calculate a refined RFC metric. We have shown that RRFC responds better to changes in coupling than that of the RFC. We have supplied an example, which showed that even though we increased the number of methods in a class, RRFC remained constant. On the other hand, RFC responded dramatically to this change, even though the change was not related to coupling. In other words, RRFC is more accurate measure for our use than the RFC metric.

(6) In the LCOM algorithm, we developed a cohesion matrix that shows the cohesiveness of methods within a class object and a cohesion ratio that computes cohesion of instance variables in methods involved in cohesion in a class. In the LCOM metric, we never get an LCOM of zero that
would give the illusion that it is a maximal cohesiveness. Based on the data that we used in the matrix, we computed LCOM defined in [CK94] and then computed LCOM based on our ratio. We used another example to show that redesigning the object is not recommended, whereas if we use [CK94] LCOM then a redesign is recommended.

The conclusions listed above support our motivation to use the algorithm metrics to evaluate object-oriented designs. The evaluation process reduces design problems. This approach has some limitations. First, the DEA assistant lacks full automation of the system in use. Second, the threshold that we used for the depth of the hierarchy level does not necessarily work for all types of designs. Some designs necessitate the need for having more than six levels per class hierarchy, which imposes a limitation on the DEA assistant.

Other conclusions from this research concern the validity of using the six metric algorithms defined earlier. Each metric we used has substantial bearing on the object-oriented paradigm. In other words, the application of the metrics to an object-oriented design aids the designer in producing a sound design.

There are numerous advantages from using the metrics algorithms. The designer can see the difference in the hierarchy structure instantly right after he/she applies the DIT algorithm. The deeper the class, the greater the number of methods to inherit, making the class difficult to maintain. Increased difficulty in maintenance is likely because of the introduction of more public and protected methods. In addition, the introduction of more public and private methods increases the chances of extensions and overrides which in return increases the difficulty of testing. There is greater
potential reuse of inherited methods if the depth is \( > 2 \) since reuse further specializes the superclass type of object. However, we indicated that any depth \( > 6 \) is enough since more levels indicate the possibility of not subclassing by specialization (is-a) but rather implementation subclassing. Keeping this in mind, we defined the DIT algorithm that determines whether we need to extend the number of levels in a class hierarchy or reduce it. The examples that we showed illustrated the importance of using the DIT metric. The class hierarchy changed in depth, which utilized the inheritance property with minimum design changes.

The WMC algorithm also offers advantages. Based on threshold values, we computed the percentages of inherited, overridden, and pure methods. If the percentage of the inherited methods is higher than the threshold, then the designer is faced with the question “Should this class exist on its own, or should it cease to exist?” If the percentage of the overridden methods is higher than the threshold, then the designer is faced with the question “Should this class be moved up to the parent’s level?” If the percentage of the pure methods is higher than the threshold, then the designer is faced with the question “Should this class be merged with its parent?” The WMC algorithm addresses these important issues and examples are shown to clarify these issues.

The NOC algorithm addresses the close and direct relationship of the NOC and DIT metrics. We recommend the use of the ICBT tree, which stresses the use of at most six levels of a hierarchy. Also, the ICBT shows that the breadth at higher levels of a hierarchy is highly recommended. With these two recommendations in mind, we implemented the DIT algorithm and showed an example where a redesign of the class hierarchy was suggested by DEA assistance. The result was an increase in the
hierarchy by one level that better utilized the inheritance property. Moreover, the reuse and specialization ratios were addressed in the definition of the algorithm used.

The CBO algorithm addresses one of the most important properties that object-oriented methodologies highly discourage. Controlling coupling among methods is very important in the design stage. The CBO algorithm defined a collaboration matrix that depicts the dependency of a method in a class on other methods in other classes. Based on the results obtained from collaborations, the fan-out ratio is then computed. Actions to be taken are recommended.

In the RFC algorithm, we defined the MIG approach. It is easy for the designer to observe the non-inheritance coupling using the MIG graphs. The graph shows each method that is involved in coupling. Unlike union sets as suggested by the RFC algorithm of [CK94], MIG graphs produced a clearer picture for the designer.

The LCOM algorithm utilized the use of a cohesion matrix and a cohesion ratio that is based on the results obtained from the matrix. The designer tests the lowest ratios obtained from the previous results. If the instance variables in the methods tested display low degree of similarity, then the designer will have to disperse these methods by creating new object classes where the dispersed methods will exist. Creating a new object at this stage is crucial. If low cohesion amongst methods is detected at later stages of the software life cycle, a costly maintenance and testing may result. However, if low cohesion is detected at the design stage, then the designer can help ensure a higher degree of cohesion in the newly created object classes, since all methods in these classes will be similar in their functionality.
5.2 Future Work

Possible future research directions are:

(1) Enhance the algorithms and the DEA assistant.

(2) Add more design metrics to the ones already used in this research.

(3) Apply the algorithms to multiple inheritance hierarchies.

The current status of the DEA assistant works only with the six metrics we used in this research. The DEA assistant could be expanded. We could either include already existing metrics or develop other design metrics that will make DEA a more robust evaluating design system. Moreover, DEA should be expanded to evaluate multiple inheritance hierarchies. Multiple inheritance hierarchies are difficult to evaluate. If a hierarchy collapses in a multiple inheritance hierarchy, it will be difficult to evaluate the design due to the lack of the data used for the collapsed hierarchy.
REFERENCES


[BO91] Booch, G., Object Oriented Design with Applications, (Benjamin/Cummings Publishing Co., Inc. 1991.)


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


VITA

Ghassan Salim Alkadi was born in Kuwait in February 1963. He is a Lebanese citizen who has been residing in the United States since August 1989. He is an instructor in the computer science department of Southeastern Louisiana University. He received his bachelor of science degree in computer science from Southeastern Louisiana University, May 1985. He worked for UNISYS division in Kuwait as a software engineer for four years. In fall of 1991 he received his master's degree in Systems Science from Louisiana State University. He is currently pursuing his doctorate in computer science from Louisiana State University. He is an active member of ACM and IEEE. His areas of interests include software engineering, operating systems, computer architecture, and programming languages. In August 1999, the degree of Doctor of Philosophy will be conferred on Ghassan Salim Alkadi.
Candidate: Ghassan Salim Alkadi

Major Field: Computer Science

Title of Dissertation: Restructuring Object-Oriented Designs Using a Metric-Driven Approach

Approved:

[Signatures]

Dean of the Graduate School

EXAMINING COMMITTEE:

[Signatures]

Date of Examination: May 13, 1999