Component Integration Metrics and Their Evaluation

Prapanna Parthasarathy
Western Kentucky University

Follow this and additional works at: http://digitalcommons.wku.edu/theses
Part of the Computer Sciences Commons

Recommended Citation
http://digitalcommons.wku.edu/theses/414

This Thesis is brought to you for free and open access by TopSCHOLAR®. It has been accepted for inclusion in Masters Theses & Specialist Projects by an authorized administrator of TopSCHOLAR®. For more information, please contact topscholar@wku.edu.
Component Integration Metrics & Their Evaluation

A Thesis
Presented to
The Faculty of the Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science

By
Prapanna Tamarapu Parthasarathy
May 2007
INTEGRATED COMPONENT METRICS & THEIR EVALUATION

Date Recommended 4/30/2007

Director of Thesis

Committee member

Committee member

Dean, Graduate Studies and Research  May 8, 2007
ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. V.L. Narasimhan who guided me throughout the thesis, read numerous revisions and helped make some sense out of the confusion. I would also like to thank my committee members, Dr. Shindhlem and Dr. Wang, who offered guidance at various occasions. Thanks to my twin sister, Prasanna and my parents who have been supportive of my higher education. Thanks to my brother Rangu for giving me the inspiration to write this thesis and to Amar for providing me software packages, solutions to various problems, and tireless encouragement. Special thanks to Sailaja and Vani, for lending their laptops and, to Sumana for helping me with several IEEE papers. Last but not the least, I wish to thank everyone who helped me not only on my thesis, but during my stay at WKU with pleasant memories.
# Table of Contents

**Acknowledgements** i

**Abstract** vii

1 **Introduction** 3

   1.1 Overview of software engineering 3

   1.2 What is a component based software engineering? 4

   1.3 Why measure quality? 6

   1.4 Existence of other metrics and their role 7

   1.5 Motivation for this work 9

   1.6 An overview of the thesis 10

   1.7 Outline of the thesis 11

   1.8 Contributions to thesis 12

   1.9 Summary 13

2 **Literature survey** 14

   2.1 The definition of the metrics 14

   2.2 Metadata 24

   2.3 Summary 25

3 **Comparison of Metrics** 26

   3.1 Overview of comparison 26

   3.2 Tabular form of metrics used in comparison 27

   3.3 Analysis of behavior of metrics used in comparison 27
7.2 Conclusion

7.3 Future Research

References
List of Figures

1) Fig 4.1 Software architecture of metric evaluation system  
2) Fig: 4.2 UML sequence diagram of the metric evaluation system  
3) Fig: 5.1 Step-by-step algorithm for CRIT Instability
List of Tables

1) Table of metrics used for comparison 27
2) Relative values for metrics ideal for the quality factor: reusability 28
3) Relative values for metrics ideal for the quality factor: Complexity 29
4) Relative values for metrics ideal for the quality factor: Testability 30
5) Relative values for metrics ideal for the quality factor: Modularity 31
6) Characteristics of the benchmark software programs 41
7) Table 6: Table of metric values for benchmark software 48
Software Engineering (SE) has been described as the discipline devoted to the design, development, and use of computer software, covering not only the technical aspects of building software systems, but also management issues develops highly complex software. The crisis in SE, due to the lack of well-defined formal processes, has led to poorly designed products with high maintenance costs and whose behavior becomes unpredictable. Component Based Software Engineering (CBSE) is currently a preferred approach to system design to overcome the crisis of SE, since it promotes software re-use, facilitates adaptability and faster system development. A component provides a function or a set of related functions, which forms a reusable program building block that can be combined with other components to form an application. A component with qualities such as, reusability, testability, modularity, complexity, proper to communicate and stability reduces maintenance costs. The components thus integrated, should be able to interoperate so that an operational application that results in reduced maintenance costs can be composed with minimal effort. Metrics are used to measure a component’s quality factor and there are no good metrics available to validate their effectiveness, when components are integrated. Currently, the success of projects based on the CBSE methodology relies on experts who assess software components; however, their evaluation process involves parameters that may not be measured in practice.
Existing traditional metrics are inappropriate since CBSE is aimed at improving interoperability and re-usability. Size metrics based on lines of code are not applicable as component sizes may not be known \textit{a priori}. Furthermore, complexities that arise due to varying nature of facets and interfaces are not addressed by traditional metrics. This thesis addresses the evaluation of a series of metrics based on complexity, criticality and dynamic behavior, in order that component integration performance can be assessed.

Three suites of metrics defined by various authors have been considered for evaluation so that one could choose the best metrics to measure an integrated environment. A suite of metrics proposed by Narasimhan and Hendradjaya are classified based on the attributes of: complexity, criticality and dynamic aspects. These metrics use graph-based connectivity to represent a system of integrated components. While the complexity metrics consider the packing density of integrated components and the interaction density among the components, criticality metrics reveal the extent of binding within each component in the system. Dynamic metrics have also been collected during the execution of an application and aid the process involved in testing and maintenance. Metric related data sets have been from several benchmark programs using instrumentation programs and key inferences have been obtained; these inferences include a systematic evaluation of quality of the various metrics.

Two new metrics have also been provided towards assessing the stability of the application: one metric, namely CRIT_{instability}, calculates the instability of each component, while the second new metric, namely CRIT_{inheritance}, counts the number of components whose children exceed a threshold value. Both these metrics are useful to assess the stability of the application and, in addition, to determine the components in a
given application that needs to be redesigned. Future work will focus on the development of a metric evaluation suite to assess the system’s stability as a whole, considering the role of each component in an application.
Chapter 1

Introduction

This chapter begins with the definitions for Software Engineering, components and Component Based Software Engineering (CBSE). It is followed by a description of the goals and issues in CBSE. The drawbacks of the existing traditional metrics and the reason of motivation for this thesis have also been provided. An overview of the thesis, along with a brief description of each chapter of the thesis has also been provided. Original contributions of the author for the thesis are mentioned and their relevance to CBSE have been highlighted.

1.1 Overview of Software Engineering

Software Engineering (SE) has been described as the discipline devoted to the design, development, and use of computer software covering not only the technical aspects of building software systems, but also management issues, such as, directing programming teams, scheduling, budgeting and whole complement of related issues. Several professional organizations have defined SE in many different ways, which include the following:

i) Definition #1 – from the IEEE

The Institute of Electrical and Electronics Engineers (IEEE) [R.12] defines Software Engineering as: “Software engineering is the application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of software; that is, the application of engineering principles to software”.

ii) Definition #2 – from NATO
"The establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines." [Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels: Scientific Affairs Division, NATO.]

It can be seen from the two definitions, that Software Engineering covers a wide range of activities based on its role and how it is defined in various contexts. In the following, a quick overview of activities that comprise SE are provided [R.15]:

**Software Engineering as a modeling activity:** Software engineers deal with complexity through modeling, and by focusing on only the relevant issues (called *abstraction*) and ignoring everything else.

**Software Engineering as a problem – solving activity:** Models are used to search for an acceptable solution for the given software problem. This style of development, where the primary software artifacts are models, is called Model-driven development.

**Software Engineering as a knowledge acquisition activity:** In modeling an application domain, software engineers collect data, organize into information and formalize into knowledge.

**Software Engineering as a rationale – driven activity:** Software engineers need to capture the context in which decisions are made and the rationale behind these decisions in order to understand the implications of a proposed change, when revisiting a decision.

1.2 **What is a component based software engineering?**

Component based software engineering is a process that emphasizes the design and construction of computer–based systems using reusable software components. This
principle embodies the “buy, don’t build” philosophy that shifts the emphasis from programming software to composing software systems [R.12].

**Definition**

Component-Based Software Engineering, CBSE in short, is a branch of the software engineering discipline, which emphasizes on decomposition systems into functional and/or logical components with well-defined interfaces that can be used for communication between two or more components. It is an approach to software development that relies on software reuse and it emerged from the failure of object-oriented development to support effective reuse.

1.2.1 Goals of CBSE

CBSE is based on building software systems from reusable components so that system developmental costs can be minimized. A software component is defined as a unit of composition, which can be independently exchanged in the form of an object code. In general, the internal structure of the component is not available to the public.

The principles of CBSE are listed below:

1) *Reuse but not re-invent:* New code will not be developed unless it is not available in the market.

2) Assemble only the relevant pre-built components rather than code line-by-line

3) Maintenance made easy due to the availability of well-documented process to make any changes to an existing application if needed, or in assessing the impact a change in a component can cause on the application.

4) Well-defined facets and interfaces that can be readily searched; their callable parameters and their data types are clearly defined.
5) Availability of a suitable ontology so that searchability and navigability of components can be facilitated.

1.2.2 What is a component?

A component is an identifiable part of a larger program. In the words of Szyperski and Messerschmitt [R.14], a software component is a system element offering a predefined service that is able to communicate with other components possessing criteria such as, reuse, ability to compose with other components without integrations problems, non-context specific and encapsulated.

The content of a component is information that is hidden from casual users and need to be known only to those who intend to modify or test the component. The software engineer chooses the set of components that has to be integrated to form an application that caters to the user requirements. Well-defined ontology with clear metadata\(^1\) is required for this purpose.

1.3 Why measure quality?

The behavior and the stability of an application cannot be assessed unless it is tested. The quality of the application is high, when it yields the expected results, is stable and adaptable and leads to reduced maintenance costs. If a change has been introduced in a component integrated in an application, the impact of the change on the whole application has to be determined by the developer to assess the stability of the application. Hence, there is certainly a need to measure the quality to assess the component's value. Metrics are needed to measure the quality.

\(^1\) Metadata provides a comprehensive description of various aspects of a component in a generic manner.
The need for metrics and their role

The purpose of software metrics is to study the characteristics of a given software system under different scenarios. Based on the metric values, the stability and behavior of applications can be assessed and there are a number of popular metrics currently in use. The following sections deal with the rationale for selecting a particular suite of metrics; note that the validity of a metric depends on the characteristics of the component.

1.4 Existing metrics & their roles

Most of the existing metrics are applicable to small programs or components, while the objective of having metrics is to test the behavior and reliability of the component when placed in a large system. According to Prather and Weyuker [R.3], the lack of appropriate mathematical properties fails quality metrics. Metrics that have a sound theoretical basis become applicable to real life organizations [R.8]. Some of the metrics rely on parameters that could never be measured or are too difficult to measure in practice. Since a component's internal structure may not be available, there is a need for black box testing and a number of existing metrics may not be applicable directly. A brief description of the limitations of the metrics based on cyclometry and complexity are provided below.

i) Limitations of LOC (Lines of Code)

The traditional size metrics based on the lines of code are not useful because a component size may not be known to the developers. In the LOC based metric, we need to differentiate structure parts of the code from the executable parts. COCOMO model excludes paragraphs of comments, comment lines in between code, blank lines and spaces but takes into account, executable statements, compiler directives, and declaration
statements of the non-executable part [R.2]. Assessing the software based on its size as
given by LOC is unpredictable as it depends on the programming language used. For
example, the actual code is less when compared to the icons and graphics in a visual
programming environment as observed by Pfleeger [R.8]; instead a count of the number
of objects and methods would be more useful than simple LOC.

ii) Limitations of structural complexity metrics

Software metrics are applicable to small programs whereas component based
metrics depend on granularity and interoperability. The complexity of the problem,
structural complexity to assess the design, algorithmic complexity to make programming
simple, and understandable, as well as cognitive complexity are some of the different
complexities that can be considered for assessing the complexity of code as discussed by
Pfleeger [R.8]. The structural complexity metrics cannot be applied because of their
different inherent structures and hence are not relevant to the integrated components
scenario. Furthermore, software metrics do not address interface complexities.

iii) Limitations of Halstead’s Software metrics

The software science metrics consider the volume of the program based on the
count of tokens categorized as, number of unique operators, number of unique operands,
and total number of operands and operators. The Halstead’s software metric suite cannot
be applied directly, because information such as, the operator and operand counts, is
difficult to obtain for component-based systems. Further, Halstead metrics have been
heavily criticized by a number of researchers [R.7] as being inappropriate for most
software quality measurement applications.
1.5 Motivation for this work

A software component is a coherent package of software implementation that offers well-defined and published interfaces, is reusable and that can be independently developed and delivered; such components are put together to form an application. However, there are no good metrics available to validate their effectiveness, when components are integrated together to form a complete system. Due to the inherent differences in the development of component based and non-component based systems, the traditional software metrics prove to be inappropriate for component-based systems. The component metrics alone are not sufficient for an integrated environment, because there is a need to measure the stability and adaptability of each component when it is integrated with other components.

i) Time-to-test as a factor

One of the features of CBSE is time-to-test. While product metrics deal with the size, complexity performance and quality, process metrics takes into account the time taken to process bugs and errors in the code [R.10]. We use the static behaviors (given by the number of classes, methods, etc.) and dynamic behaviors (given by the number of interactions and interfaces) evaluated during run-time of each component to assess its quality.

ii) Reusability as a factor

A number of studies have been carried out on the reusability of components developed for various projects. In general, such components must be able to accommodate minor changes, enhancement features [R.3] with minimal effort. The important part of the development process is to define consistent and complete
specification of a component. Reusable components are components that possess the ability to be used in different applications that has some common requirements which can is satisfied by the component.

iii) Maintenance as a factor

Reduction of maintenance costs is one of the goals of CBSE. While the application maintenance cost decreases, the maintenance cost of each component integrated might be high as it must respond to different requirements of different applications running on different environments requiring a different level of maintenance support. A well- designed component keeps the maintenance costs of the component as an individual, and as well as that of the application very low.

Narasimhan and Hendradjaya [R.7] noticed the lack of metrics that aids in reducing the maintenance costs and defined metrics whose values are collected during the execution phase. Such metrics are useful for assessing the maintenance cost of individual components and that of the application in which the component is integrated.

1.6 An overview of the thesis

This thesis supports and critiques the ideas of Narasimhan & Hendradjaya [R.7] in providing metrics for the integration of software components. A component when deployed and executed may yield on its own the expected results, but its behavior and functionality when integrated with other components to make a complete application may be different to the expected. Therefore, there is a need for metrics to assess the functionality of each component when integrated with other components and functionality of the application on the whole.
The thesis analyzes the crisis in software engineering and its adverse effects on application development and maintenance. It emphasizes on the need for component-based software engineering and the need for metrics, besides highlighting the inefficiency of existing metrics. A comparison of various metrics has been provided to support the views of the author on how far the traditional metrics and the metrics proposed by Narasimhan & Hendradjiya [R.7] are useful in assessing the quality of components in an integrated application. Benchmarks software programs have been used as inputs to instrumentation programs and metric values have been collected. A systematic analysis of the values for various metrics has been carried out and several key inferences have been drawn from them. Furthermore, I have also proposed two new metrics that can be added to the metrics suite. As part of the future work, the development of a metrics suite solely for the integrated components has been proposed.

1.7 Outline of the Thesis

This thesis has seven chapters whose outlines are provided below:

Chapter one provides the motivation for this thesis beginning from the definitions for software engineering, components and component-based software engineering (CBSE) and its goals and issues. It highlights the drawbacks of the existing traditional metrics, thus necessitating a metric suite requirement solely for integrated components.

Chapter two is a literature survey, which provides an explanation of the various terminologies as assumed by the author of the Thesis. In addition, various metrics defined by a number of authors that have been used in further chapters have also been explained.
Chapter three provides the rationale for comparing the metrics chosen for the purpose of CBSE-based integration. The relevance of metrics to measure various factors is analyzed based on the definitions provided by the authors of the corresponding metrics.

Chapter four describes the basis for seeking good benchmarks, the basis for the selection of benchmark programs for which the metric values have been collected, the instrumentation programs and the values of metrics collected. The metric values collected, has been used in making inferences in the further chapters.

Chapter five provides two new metrics called the CRIT\_Instability and CRIT\_Children. The rationale for the metrics, a step-by-step algorithm, limitations and necessity for revisiting the metrics are dealt with in this chapter.

Chapter six provides a list of inferences obtained from this research, their practical applicability and limitations.

Chapter seven outlines the ways to overcome the limitations of the various metrics by validating them by using empirical data to calculate the threshold values. In addition, a comprehensive summary of the thesis, along with pointers for future work have also been provided.

1.8 Contributions to Thesis

The author contributed the following to the thesis:

(i) Two new metrics are proposed to the existing metric suite as an augmentation to the work of Narsimhan and Hendradjaya [R.7]. The two new metrics namely CRIT\_Instability and CRIT\_Children measure the stability of an integrated software system.

(ii) A new algorithm has been proposed to determine CRIT\_Instability
The new component based software architecture of the metric evaluation system has been proposed and successfully employed in this thesis.

1.9 Summary

This chapter provides an overview of the thesis. The crisis in software engineering, concepts in CBSE, and the need to measure the quality of software produced through the CBSE methodology have been provided. The role of existing metrics and their limitations have also been provided.
Chapter 2

Literature survey

This chapter provides a comprehensive literature survey of the definitions, formulas, and the rationale for various metrics as defined by the respective authors; the need for metadata in the context of metrics has also been provided. The advantages and disadvantages of various metrics based on their properties are also listed.

2.1 The Definition of Metrics

Three suites of metrics defined by various authors are considered in this thesis for comparison. The definitions and formulas for each of the metrics are listed, as defined by the respective authors. The relative value for each metric for every quality factor has been analyzed and the implication of the metric has been tabulated in the next chapter based on the definitions and rationale provided by the authors of the metrics.

2.1.1 Survey of Metrics that relate to “Suite of Metrics for the integration of Software Components”

Narasimhan and Hendradjaya [R.7] classified their metrics into complexity, criticality, triangular and dynamic metrics. The definitions and the formulas of the metrics are listed below:

i) Complexity Metrics

i) Metric 1: Component Packing Density

Definition: Component Packing Density (CPD) metric is in the form of a ratio of the number of constituents in overall component assembly to the number of components in a sub-component assembly as

\[
CPD_{\text{constituent type}} = \frac{\# \ < \text{constituent} >}{\# \text{components}}
\]
Disadvantage: As density increases, complexity increases which means the developer has to spend more effort on analyzing the module and locating the risks.

**ii) Metric 2: Component Interaction Density**

Definition: Component Interaction Density (CID) metric is the ratio of the actual number of interactions to the available number of interactions in a component.

\[
CID = \frac{#I}{#I_{\text{max}}}
\]

where, \(#I\): number of actual interactions and \(#I_{\text{max}}\): number of maximum available interactions.

If a component provides interface and other use it or if a component submits an event and others receive it, then it is called an interaction. When the density of interaction increases, complexity increases.

**iii) Metric 3: Component Incoming Interaction Density**

Definition: Component Incoming Interaction Density (CIID) metric is the ratio of the actual number of incoming interactions\(^1\) to the available incoming interactions in a component.

\[
\text{CIID} = \frac{#I_{\text{in}}}{#I_{\text{max in}}}
\]

where, \(#I_{\text{in}}\): number of incoming interactions used and \(#I_{\text{max in}}\): number of available incoming interactions.

---

\(^1\) Incoming interaction is defined as a received interface that is required in a component or a received event that arrives at a component.
If density increases => a particular component requires so many interfaces.

**iv) Metric 4: Component Outgoing Interaction Density**

Definition: Component Outgoing Interaction Density (COID) metric is the ratio of the actual numbers of outgoing interactions to the available outgoing interactions in a component.

\[
\text{COID} = \frac{\#I_{\text{out}}}{\#I_{\text{max out}}}
\]

where, \(\#I_{\text{out}}\): no. of outgoing interactions used and \(\#I_{\text{max out}}\): no. of outgoing interactions available

Outgoing interaction: any provided interface used, a source of event consumed.

If density increases, a particular component needs so many interfaces

**v) Metric 5: Component Average Interaction Density**

Definition: Component Average Interaction Density (CAID) metric is a sum of Interaction densities for each component divided by the number of components.

\[
\text{CAID} = \frac{\sum_{n=1}^{n} \text{CID}_n}{\# \text{components}}
\]

where, \(\sum_{n=1}^{n} \text{CID}_n\): sum of interaction densities for components \(1\ldots n\) and

\(\# \text{components}\): no. of existing components in the actual system

**ii) Criticality Metrics**

**i) Metric 6: Link Criticality Metric**

Definition: Link Criticality Metric (CRIT\(_{\text{link}}\)) metric is defined as the number component in which their links are more than a threshold value.
where, \#componentlinks is the number components, with their links more than a critical value. The threshold is considered as 8 links. Links are created from the facets of other components. If facets increases, criticality of that component increases.

**ii) Metric 7: Bridge Criticality Metrics**

Definition: Bridge criticality metric (CRITbridge) is defined as the number of bridge components in a component assembly.

\[
\text{CRIT}_{\text{bridge}} = \# \text{bridge}\_\text{component}
\]

where, \#bridgecomponent is the number of bridge components.

A bridge component links two or more components/ application. If there is defect in bridge, the entire application might malfunction. The more the number of bridges, the more are the chances of failure. All the links provided by a bridge component are assigned a similar weight in order to represent that they belong to the same bridge component.

**iii) Metric 8: Inheritance Criticality Metric**

Definition: Inheritance criticality metric (CRIT\textit{inheritance}) metric is defined as the number of components, which become root or base for other inherited components.

\[
\text{CRIT}_{\text{inheritance}} = \#\text{root}\_\text{component}
\]

Where, \#root_component is the number of root components which has inheritance.

It is the number of components which act as parent/root/base for other components.

**iv) Metric 9: Size Criticality Metric**

Definition: Size Criticality metric (CRIT\textit{size}) metric is defined below.

\[
\text{CRIT}_{\text{size}} = \#\text{size}\_\text{component}
\]
where, \(#size\_component\) is the number of component which exceeds a given critical value. Size is determined by considering factors like Loc, number of classes, operations and modules in the application. Narasimhan and Hendradjaya defined the threshold value as 1000 lines of code or 50 classes. So, the value for this metric is given as 1 if it exceeds the threshold value.

\(v) \text{ Metric 10: \#Criticality Metrics}\)

Definition: \(#\text{Criticality Metric (CRITall)}\) metric is the sum all critical metrics

\[ \text{CRITall} = \text{CRITlink} + \text{CRITbridge} + \text{CRITinheritance} + \text{CRITsize} \]

\(iii) \text{Triangular Metrics}\)

Component Packing Density (CPD), Average Interaction Density (CAID), Component Criticality (\(\text{CRIT}_{\text{all}}\)) are considered as 3 axes which can be further modified as 2 axes diagrams with CPD and CAID. For different values varying as high and low for the 2 axes, different cases are considered as the behaviors vary for different systems based on real time, business type etc. [1].

\(iv) \text{Dynamic Metrics}\)

These are the metrics collected during the execution time. This is not available during the design phase as they are collected dynamically and so, are used for maintenance purposes.

\(i) \text{Metric 12: Number of Cycle (NC)}\)

\textbf{Definition}: The Number of Cycle metric (NC) is the number of cycles within an integrated component in a graph representation.

\[ \text{NC} = \# \text{cycles} \]

Where, \(#\text{cycles}\) are the number of cycles within the graph.
**ii) Metrics 13: Active Component (AC)**

Definition: Active Component metric, (AC) can be described as follows: A Component is active when its provided interface is used by other components or when it requires an interface from other components during run-time. Such a component is called an Active Component. A component becomes inactive when its interface is no longer in use during run-time. Some candidate metrics are considered for the Active component like the density of the AC, the average number of Ac’s per time interval (ACD), The ACD per execution time interval that might vary from the execution time of a function, a module or an entire program, Peak number that is the maximum number of components during the entire execution time.

**2.1.2 Survey of Metrics that relate to “A Metrics Suite for Object-oriented Design”**

Chidamber & Kemerer [R.10] came up with metrics that focused on measuring the reusability quality factor of a component. Moreover, the metrics considers the components in an isolated environment. The metrics in their own words are listed below:

**i) Metric 1: Weighted Methods per Class (WMC)**

*Definition:* Weighted Methods per Class metric (WMC) is the count of methods implemented within a class

\[ WMC = \sum_{i=1}^{p} C_i \]

where, \( C_i \) = count of methods in class \( i \). It considers measuring the complexity of methods which is rather a difficult task due to inheritance and that limits the possibility of reuse if there is more number of methods available. It collects the sum of complexities.
Disadvantage: measuring complexity is difficult due to inheritance. If methods increase in a class, the sub classes of that class become complex. More methods limits reuse.

**ii) Metric 2: Depth of Inheritance Tree (DIT)**

Definition: Depth of Inheritance Tree metric (DIT) is the maximum length from the node to the root of the tree is the DIT defined for multiple inheritance. The higher the depth of inheritance, the more it is involved in inheritance which increases the complexity due to difficulty in tracking the methods inherited, but the more it is reusable. 

Advantage: An increase in hierarchy implies a high reusability. 

Disadvantage: An increase in hierarchy implies that the count of number of methods increases and hence, complexity increases.

**iii) Metric 3: Number of Children (NOC)**

Definition: Number of Children metric (NOC) is the number of immediate sub classes subordinate to a class in hierarchy. The higher the number of children, the higher is the inheritance which increases the reuse concept. But, there are chances of misusing the inheritance referred as “improper abstraction of the parent” by the authors. It demands more testing as each and every inherited class, the methods inherited and used has to be tested properly to avoid any misuse. A high value of NOC implies a high reusability and higher sub-classing that demands more testing, and implies a higher complexity.

**iv) Metric 4: Coupling between Object Classes (CBO)**

Definition: Coupling between Object Classes (CBO) is the number of distinct non-inheritance related class hierarchies on which a class depends.
Disadvantage: The higher the coupling, the higher is it interrelated with other classes and complexity increases and it is difficult to make any changes as all other classes related with this one will have an impact of the change. The higher the coupling, the lesser independent a class becomes, and complexity increases with dependability.

v) Metric 5: Response for a Class (RFC)

Definition: Response for a Class metric (RFC) is the count of set of all methods that can be invoked in response to a message to an object of the class. Complexity increases, as number of methods invoked in response is more.

Disadvantage: The higher the number of methods invoked in response, the more difficult it is for the tester to debug as it demands a greater level of understanding due to increase in the complexity.

vi) Metric 6: Lack of Cohesion (LCOM)

Definition: Lack of cohesion is the dissimilarity of methods in a class. A highly cohesive module is highly reusable due to a good deal of modularity. So, it can be called a stand alone which has lesser complexity due to the sub division of a class into sub classes. Highly cohesive module implies a stand – alone and represents a good class sub division, simple and more reusable as stand – alone. A low value of cohesion implies a high value of complexity due to lesser sub-division.

2.1.3 Survey of Metrics that relate to “Component Metrics to Measure Component Quality”

Cho & Kim [R. 4] classified their metrics into design and implementation types. Most of their metrics are useful to measure the complexity of the components and the formulas
defined for the metrics require the entire source code to determine the metric values for that component.

**vii) Metric 1: Component Plain Complexity (CPC)**

*Definition:* Component Plain Complexity is a metric used to measure the component complexity by calculating the sum of classes, interfaces, abstract classes and complexity of classes and methods.

\[
\text{CPC}(C) = CmpC + \sum_{i=1}^{n} CC_i + \sum_{j=1}^{n} MC_j
\]

Where, \(CmpC\) is a count of classes, abstract classes and interfaces, \(CC_i\) is the complexity of each class and \(MC_j\) is the complexity of each method present in a class. CPC focuses on the count of classes, methods, interfaces and parameters declared in a component. \(CmpC\) considers the count of classes and weight of each class (the value of importance) in each component and similarly in turn, for methods in each class. \(CC_i\) is calculated as the count of single attribute and complex attribute with the value of complexity (the level of complexity assigned).

*Disadvantage:* measuring this attribute is difficult. A high value of CPC implies more methods which implies a large code size.

**viii) Metric 2: Component Static Complexity (CSC)**

*Definition:* Component Static Complexity metric (CSC) measures the complexity of internal structure in a component with a static view.

\[
\text{CSC} = \text{sum} (\text{count} (R) * W(R))
\]

Where, \(R\) is called the relationship between two classes or a class and an interface. \(CSC\) is the sum of count of relationships between classes and the level of importance given to each relationship. The relationship between classes for each component is
counted, and weighted values are given for each count considering factors like dependability, composition and aggregation. Thus, this metric focuses on how complex the internal structure is.

*Advantage:* Though, if there are many relationships between classes, it is reduced to binary.

*Disadvantage:* Higher the relationships more interweaved are the classes =&gt; difficulty to restructure / modify design.

**ix) Metric 3: Component Dynamic Complexity (CDC)**

*Definition:* Component Dynamic Complexity metric (CDC) measures the complexity of internal message passing in a component with dynamic view. The count of messages passed between classes gives the dynamic view.

\[
CDC = \text{sum} \ (DC \ (IM))
\]

\(DC\) is the complexity of each interface method and it is calculated by the number of messages passed between classes, their frequency of occurrence and the complexity of each message.

**x) Metric 4: Component Cyclomatic Complexity (CCC)**

Component Cyclomatic Complexity metric (CCC) is used in the implementation phase. The metric is computed using developed source code. It considers the sum of classes, interfaces and methods along with the complexity of each class in the component and each interface.

\[
CCC = CmpC + \sum_{i=1}^{n} CCI + \sum_{j=1}^{n} MCj + \sum_{k=1}^{n} CCMk
\]
The terms here are same as defined for metric 1, except that we consider an additional sum CCM, which is the cyclomatic complexity method. CCM is defined as a sum of edges – nodes + 2.

2.2 Metadata

Metadata is value-enhanced information about a particular data set, which describes the definitions of the data and how it can be interpreted. In the words of Taylor [R.16], "Metadata can be defined as structured data that describes the characteristics of a resource. The term "meta" derives from the Greek word denoting a nature of a higher order or more fundamental kind". A metadata record consists of a number of pre-defined elements representing specific attributes of a resource, and each element can have one or more values. The metadata is categorized into i) list of symbol vocabulary containing the list of terms, which has to be interpreted into a meaning as defined by the author and, ii) list of variable and/or synonymous vocabulary.

Symbol Vocabulary

1) The words redesign and remodeled terms are in this thesis implies making changes to an already existing code.

Variable and/or Synonymous Vocabulary

i) Commonality among CPC – CPD – WMC Metrics

Component Plain Complexity proposed by Cho et al. [R.4], CPC in short is used to measure the component complexity by calculating the sum of classes, interfaces, abstract classes and complexity of classes and methods. CPC considers the complexity by the sum of classes and interfaces. Component Packing Density proposed by Narasimhan et al. [R.7], CPD in short, considers the complexity density of classes and components.
Weighted Methods per Class, WMC in short, considers the count of methods in a class and their complexities. All the three metrics consider the total complexities of the methods/classes/sub-components of a given module similarly, but the difference is that, the formula required to calculate CPC needs the whole source code to get the required data and CPD considers the complexity of sub-components which are finally averaged to get the value as an entire component.

ii) Commonality between CSC – CBO Metrics

Coupling between Object Classes (CBO) defined by Chidamber [R.10], is the number of distinct non-inheritance related class hierarchies on which a class depends. It is the count of other classes to which this class is coupled. Component Static Complexity metric defined by Cho [R.4], CSC in short, measures the complexity of internal structure in a component with a static view.

2.3 Summary

This chapter outlines the metrics considered for evaluation proposed by various authors are listed, along with their definitions, formulas and rationale. The requirement for metadata in the context of metrics has also been provided.
Chapter 3

Comparison of Metrics

This chapter explains the need for the comparison of metrics and their actual comparison. The expected values of metrics under different factors are tabulated based on the definitions provided by the authors of the corresponding metrics.

3.1 The need for comparison

Since this thesis relates to metrics that measure the quality for integrated components, there is a necessity to check the various metrics available so that one can choose the metrics that measures the required aspects considered in this thesis such as, reusability, complexity, size, testing time and maintenance.

The factors considered for comparison are:

1) Reusability
2) Complexity
3) Testability
4) Maintenance
5) Modularity

Procedure of comparison

The metric values are compared by collecting the values of metrics for benchmarks software programs which are discussed in the following chapters. While many authors have provided a comparison of metrics, their focus has been on collecting the metric values for a component considered as a stand-alone entity. I tried to collect metric values for sub-components that make a component and evaluated the best suite of metrics that suit a given context.
3.2 Tabular form of metrics used for comparison

Various popular traditional metrics that are currently in use are considered, whose source details are tabulated below:

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Author(s)</th>
<th>Paper Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPD, CID, CIID, COID, CAID, CRIT, CRIT link, CRIT bridge, CRIT inheritance, CRIT size, CRIT all, Triangular metrics, ANAC, ACD, AACD, PNAC</td>
<td>V.L. Narasimhan, B. Hendradjaya</td>
<td>Some Theoretical considerations for a Suite of Metrics for the integration of Software components, Transactions on Engineering, Computing &amp; Technology V10 Dec 2005</td>
</tr>
</tbody>
</table>

Table: 3.2 Table of metrics used for comparison

3.3 Analysis of behaviors of metrics used in comparison

The behavior of the metric is theoretically analyzed based on their definitions provided by the corresponding authors. A metric is considered as suitable to a given
quality factor if its value is significant to that factor. The ideal relative values for each of the quality factors of the component are listed below:

1) Reusability: A high reusability indicates a well-designed component.
2) Complexity: A high value for complexity indicates a poorly designed component.
3) Testability: Low test-to-time indicates a highly modular component.
4) Maintenance: Less maintenance costs indicates a highly stable component.
5) Modularity: Appropriate size indicates a high reusability.

3.3.1 Metrics behavior under the criteria: Reusability

<table>
<thead>
<tr>
<th>Name of the metric</th>
<th>Relative value of metric</th>
<th>Implication for Reusability</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCOM</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>WMC</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>CBO</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>NOC</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>DIT</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CSC</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>CPD</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>CID</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>CAID</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>CRIT Size</td>
<td>↓</td>
<td>↑</td>
</tr>
<tr>
<td>CRIT Link</td>
<td>↓</td>
<td>↑</td>
</tr>
</tbody>
</table>

Table 3.3.1: Relative values for metrics ideal for the quality factor: reusability.

The metrics mentioned in the Table 3.1 measure reusability of a component. A high LCOM, NOC and DIT, implies that the corresponding components are highly reusable. A high CID, CPD, WMC, CSC, and CBO, implies that the corresponding components are less reusable. A low CRIT Size, CRIT Link implies that the corresponding components are highly reusable. It is noted that a component is considered ideal if it is highly reusable.
3.3.2 Metrics behavior under the criteria: Complexity

<table>
<thead>
<tr>
<th>Name of the metric</th>
<th>Relative value of metric</th>
<th>Implication for Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFC</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CBO</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>LCOM</td>
<td>↓</td>
<td>↑</td>
</tr>
<tr>
<td>DIT</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CPC</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CPD</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CID</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CAID</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CIID</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>COID</td>
<td>↑</td>
<td>↑</td>
</tr>
</tbody>
</table>

Table 3.3.2: Relative values for metrics ideal for the quality factor: Complexity

The metrics mentioned in the Table 3.2 measure the complexity of a component. A high value for the metrics RFC, CBO, DIT, CPD, CIID, COID, and CPC, implies that the corresponding component is considered to be highly complex. A low value for the metric LCOM implies that the corresponding component is considered highly complex. It is noted that a component is considered ideal if it is less complex and hence the values of the metrics like RFC, CPC, CIID, and COID are to be very low.
3.3.3 Metrics behavior under the criteria: Testability

<table>
<thead>
<tr>
<th>Name of the metric</th>
<th>Relative value of the metric</th>
<th>Implication for Testability</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOC</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>RCC</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>CID</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>CRIT Bridge</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>CRIT Link</td>
<td>†</td>
<td>†</td>
</tr>
</tbody>
</table>

Table 3.3: Relative values for metrics ideal for the quality factor: Testability

The metrics provided in the Table 3.3 measure the testability of a component. A high value for the metrics NOC, CID, CRIT Bridge, CRIT Link and RCC imply high testability. An ideal application made up of components should take less time-to-test.

3.3.4 Metrics behavior under the criteria: Maintenance

The list of metrics, whose values can be used to infer the maintenance effort required for a given application, is: ANAC, CCC, NC and ACD. The metrics values for these metrics are collected during run-time which implies that the development phase of the components has been completed and that, these metrics are being collected for maintenance purposes. Narasimhan and Hendradjaya [R.7] have proposed a series of dynamic metrics for the purpose of maintenance.
3.3.5 Metrics behavior under the criteria: Modularity

<table>
<thead>
<tr>
<th>Name of the metric</th>
<th>Relative value of the metric</th>
<th>Implication for Modularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CPC</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CRIT inheritance</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>CRIT Size</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>AC</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>NOC</td>
<td>↑</td>
<td>↓</td>
</tr>
</tbody>
</table>

Table 3.4: Relative values for metrics ideal for the quality factor: Modularity

The metrics mentioned in the Table 3.4 measure the aspect of size of a component. A high value for the metrics WMC, and CPC, implies a huge component size. If CRIT Size is high, it means the component is less modular. A high value for the metrics CRIT inheritance and NOC, imply that the corresponding component is considered to less reusable. It is noted that a component is considered ideal if it is has an appropriate size such that, it makes the component less complex and highly reusable.

3.4 Summary

This chapter provides a comparison of various metrics used in this thesis. The analysis of metric behaviors is determined on the basis of theoretical definitions and the relative value and their implication towards several quality factors of a component have been tabulated.
Chapter 4

Description of Benchmarks used

This chapter describes the basis for benchmark software programs, the basis for the selection of software used in data collection called as instrumentation process, and the implications of the metric values collected.

4.1 Need for benchmarks

In the previous chapter [Chapter 3], different metrics proposed by various authors have been considered for comparison. Each of the metrics has its own significance and usefulness towards measuring the quality of a component. Though the purpose of all the metrics is same, there are variations in their definitions as they measure various quality factors in various environments. The suitability and appropriateness of these metrics to measure quality are inferred by analyzing the outputs.

What is a good benchmark?

A good benchmark has been defined as a software containing at least 50 classes, and 15,000 lines of code and further, the code has is to be available over any object-oriented language. In this thesis, the emphasis is on preserving the properties of CBSE in an integrated environment such that the application yields the expected results. Therefore, if the value of a metric determines the component for which the metric value has been calculated as stable, reusable, more abstract and less complex, then that benchmark is considered stable. Otherwise, some of the components in the benchmark may require being re-designed.
4.2 Overview of the software architecture of the metric evaluation suite

The software architecture of metric evaluation system used in this thesis is provided in Fig. 4.1. The system has six major components:

i) The benchmark suite contains programs, whose source codes are used for metric generation

ii) The instrumentation program suite facilitates collection metric values from the benchmark programs

iii) The compiler that takes the benchmark suite as input for the instrumentation program, compiles and executes

iv) The metric values generator which is the output of the instrumentation program that gives the metric values for the benchmark software

v) Inferences engine where inferences are made from the outputs

vi) The metrics visualization environment
Fig 4.1 Software architecture of metric evaluation system used in this thesis
**Initial stage** | **Compiler** | **Inferences Engine** | **Visualization Environment**
---|---|---|---
Appropriate instrumentation program is chosen
Benchmark software is provided as input
The resulting metric values are provided as input

Inferences are made based on the metric values

---

Fig: 4.2 UML sequence diagram of the metric evaluation system in this thesis

Various benchmark software selected on the basis of some criteria is given as inputs to the instrumentation program which when compiled and executed by the compiler, gives the metric values as outputs. Inferences are made based on the outputs and the theoretical analysis and the best matched metrics suite for a given context is concluded.

**4.2.1 Criteria for Selecting Instrumentation Programs**

The instrumentation programs provide output in the form of data units, which might be a direct or indirect representation of some of the metrics of the three suites considered. If the data units are in indirect form, the required calculations are performed by the author.
4.2.2 List of instrumentation programs used in Data Collection

jDepend and Metrics 1.3.6 are the instrumentation programs used in this thesis to facilitate data collection.

i. JDepend

JDepend software is used to collect data for the following metrics: CRIT Inheritance, CRIT Size, COID, CAID, and CIID. In the developer’s own words, “JDepend traverses Java class file directories and generates design quality metrics for each Java package”. JDepend allows you to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively. The software can be obtained from the website:

http://clarkware.com/software/JDepend.html

The output of the software is the following units. The corresponding metric values obtained from the units are also mentioned.

(i) **Number of Classes and Interfaces**: The number of concrete and abstract classes and interfaces in the package of which the number of concrete classes provides values for CRIT Inheritance. The total count of classes proves helpful to calculate CRIT Size.

(ii) **Afferent Couplings (Ca)**: The number of other packages that depend upon classes within a given package and is used for calculating the value for COID.

(iii) **Efferent Couplings (Ce)**: The number of other packages that classes in a given package depend upon and is used to calculate the value for CAID.
ii. Metrics 1.3.6

Metrics 1.3.6 software is used to collect values for the following metrics directly or indirectly: NOC, WMC, DIT, CPD and LCOM. Metrics 1.3.6 provides metrics calculation and dependency analyzer plug-in for the Eclipse platform which can be obtained from the website http://metrics.sourceforge.net/. It measures various metrics with average and standard deviation, detects cycles in package and type dependencies and provides a graphical visualization. This package is operating system independent software developed over Java programming language. Installation procedure, implementation and documentation can be found at:

http://sourceforge.net/projects/metrics/

The output of the Metrics 1.3.6 is the following units:

(i) **Number of Children:** the number of children implies the total number of direct subclasses of a class. Note that a class implementing an interface counts as a direct child of that interface.

(ii) **Depth of Inheritance Tree (DIT):** Depth of the Inheritance Tree measures the distance of a class from its root in the inheritance hierarchy.

(iii) **McCabe Cyclomatic Complexity:** McCabe Cyclomatic Complexity counts the number of flows through a piece of code (in methods alone).

(iv) **Weighted Methods per Class (WMC):** Weighted Methods per Class is the sum of the McCabe Cyclomatic Complexity for all methods in a class.
(v) **Lack of Cohesion of Methods (LCOM):** Lack of Cohesion of Methods is a measure for the Cohesiveness of a class.

I adopted the following procedure to collect metric values for the metrics from the output of the instrumentation programs:

i. The outputs of the JDepend software provides values for the metrics like COID, CIID, CRIT\textsubscript{Inheritance}, CRIT\textsubscript{size} over a (Java collection) data structure.

ii. Values for the metric CAID, CID and COID are calculated manually based on the definition of the CAID. The indirect values for CIID and COID are obtained from the software outputs.

iii. From the output of Metrics 1.3.6, values for the metrics NOC, DIT, LCOM and WMC have been collected directly.

iv. CPD is calculated by considering the mean value of the *Number of Classes* for each benchmark program.

### 4.3 Overview of Benchmarks

This section provides an overview of the software packages which are used as source inputs for collecting values of the various metrics. Several benchmark packages of different sizes and varying modules have also been considered.

#### 4.3.1 Selection Criteria

The following criteria have been used in the selection benchmark software:

i. **Code should be object-oriented:** The package code is to be written in any object-oriented language.
ii. The size of package: The size of the package should be large enough to depict a practical scenario, i.e., the packages with at least 20,000 lines of code are considered. The number of classes should be at least 50.

iii. Transparency of source code: Packages for which the source code is not transparent are selected for black-box testing and reuse. By the definitions of CBSE, the complete source code of a component may not be available for any developer while reusing the component. Therefore, to depict the real-life scenario, packages with object codes have only been considered.

4.3.2 List of Benchmarks

The details of the benchmark software used as inputs to obtain metric values are detailed below:

i) JUnit

JUnit is an Open Source Software, where programmers can read, redistribute, and modify the source code. JUnit is a simple, open source framework to write and run repeatable tests and is an instance of the JUnit architecture for unit testing frameworks.

ii) Mouse Gestures

Mouse Gestures is an open source pure Java library for recognition and processing mouse gestures and is a product of Smardec. The user just holds down a mouse button and moves the mouse in a certain predefined way in order to execute certain commands without the help of the keyboard. This software is considered as a benchmark program in this thesis for which metric values have been collected. More details are available at: http://www.smardec.com/products/mouse-gestures.html
iii) jGRASP

jGRASP is a lightweight development environment, created specifically to provide automatic generation of software visualizations to improve the comprehensibility of software. jGRASP is implemented in Java, and runs on all platforms with a Java Virtual Machine (Java version 1.3 or higher). More details are available at: http://www.jgrasp.org/

iv) Element

The Element package provides a class that supports a window for generic drawing, a class that supports a window for performing input and output, and a hierarchy of classes that support graphics in a drawing window. More details are available at: http://www.cs.williams.edu/~bailey/JavaElements/source/

v) JCIFS

JCIFS is an Open Source client library that implements the CIFS/SMB networking protocol in pure Java. CIFS is the standard file sharing protocol on the Microsoft Windows platform (Ex: Map Network Drive) where, this client is used extensively in production on large Intranets. More details are available at: http://jcifs.samba.org/src/docs/api/

vi) IDap

The IDap package contains classes and interfaces for using features that are specific to the LDAP v3, Lightweight Directory Access Protocol (v3). This package is primarily for those applications that need to use "extended" operations, controls, or unsolicited notifications.
Table 4.1 provides a snapshot of the characteristics of the chosen benchmark software programs.

<table>
<thead>
<tr>
<th>Benchmark programs</th>
<th>No. of classes</th>
<th>No. of sub-components</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDap</td>
<td>339</td>
<td>16</td>
</tr>
<tr>
<td>JCIFS</td>
<td>141</td>
<td>8</td>
</tr>
<tr>
<td>jGrasp</td>
<td>1265</td>
<td>18</td>
</tr>
<tr>
<td>jUnit</td>
<td>107</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.1: Characteristics of the benchmark software programs

4.4 Usefulness of benchmark software

The metric values have been useful to make inferences as listed in Chapter 7. The behavior of the metrics under various quality factors such as, reusability, size, complexity, testing–time and maintenance have been measured and inferences are based on the assessment of these metric values obtained for the benchmarks.

4.5 Summary

This chapter outlines the need for benchmarks and the details of the benchmarks. The instrumentation programs used in the collection of metric values and their role of benchmarks in the thesis have also been discussed. The architecture of the testing suite has been presented in this chapter.
Chapter 5

New Metrics for Component Stability

There are no direct metrics which calculates the instability value of an application and I have proposed two new metrics that calculate the instability by considering the instability of each component inside the application. The metrics and their relevance are discussed in detail in this chapter.

5.1 Limitations of current metrics

Narasimhan and Hendrajaya [R.7] considered several metrics for integrated components and classified them. Their criticality metrics consider measuring the various links and bridges that assess the amount of abstraction, size and inheritance in view of the number of components that act as parents. However, no metric directly measures the instability of the application, which plays a major role in determining the quality of the functioning of an application and its maintenance costs.

Definition for Instability

The instability of a component is defined as the amount of change that a component can introduce to the system, when its nature and/or interfaces (and facets) are changed. Assessing the impact of the change beforehand is necessary to prevent the application from behaving unpredictably due to the introduced change. In an integrated environment, the change in a component affects the entire application depending on the amount of change [R1]. The component, in which a change occurs, should be able to be composed with other components giving the expected results.
By assessing the instability value, one can estimate the amount of testing to be done and the number of components in the application apart from the component in which a change is introduced, that need to be tested. The components with high instability values in an application needed to be tested for integration in an exhaustive manner.

5.2 Definitions of new metrics

I have proposed two metrics that can be added to the list of “criticality metrics” of Narasimhan and Hendradjaya [R.7]. These metrics facilitate the assessment of the stability of an application, containing several components. The CRIT\_Instability considers in calculating the metric using the formula of Instability. The averages of the instability values of components are measured and if the resulting value exceeds a threshold value, the application is considered unstable.

The CRIT\_Children measures the count of children for each component and if the number of children is higher than a given threshold value for a component, the component is unstable. If the number of such components is high in an application, the application is considered unstable. This is in contrast to the CRIT\_Inheritance metric proposed by Narasimhan and Hendradjaya [R.7] that consider the number of components that act as a single unit.

New Metrics and their rationale

i) Instability Criticality Metric (CRIT\_Instability)

The Instability Criticality metric is defined as the number of components for which the instability value exceeds a given threshold value.

\[
\text{CRIT\_Instability} = \# \text{ of Components that exceeds the Instability value}
\]

Algorithm to calculate the Instability Criticality Metric value for an application:
1) Say there are n components in an application
2) The instability of each component is calculated using the formula

\[ I = \frac{C_e}{(C_a + C_e)} \]

where I = instability value of that component
3) Count the number of components for which I < threshold value (0.6)
4) The final count is the CRIT instability value of that application
5) If the CRIT instability value is greater than n/2, then that application is considered unstable and the components for which I < threshold, need to be redesigned.

Fig: 5.1 Step-by-step algorithm for CRIT instability

Martin [R.13] has proposed that the instability of a component can be calculated using the formula: \( C_e / (C_a + C_e) \). Using the formula, the instability of each component in an application is calculated and the number of components in that application whose instability value is below a given threshold value (I assume the threshold value to be 0.6 intuitively) are counted. The final count is assigned to the CRIT instability metric.

Rationale for the metric

If the Instability Criticality Metric value is high, the system contains many unstable components, thereby leading to a higher chance of failure. The threshold value has been defined intuitively as half of the number of components in the application, which if exceeded, make the component highly unstable, thus, requiring the application to be redesigned. The higher the value of the instability value, the more stable the component is; hence, components whose instability value is less than 0.6 only are counted towards calculating the final value for this metric. If the count reaches more than half of the total number of components in that application, it needs to be redesigned to
keep the application stable. The values for CRIT\textsubscript{Instability} for the benchmarks calculated using the formula as defined by Martin is given in Table 5.1:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Value for the metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDap</td>
<td>2</td>
</tr>
<tr>
<td>jGrasp</td>
<td>7</td>
</tr>
<tr>
<td>JUnit</td>
<td>6</td>
</tr>
<tr>
<td>JCIFS</td>
<td>2</td>
</tr>
<tr>
<td>mouseGestures</td>
<td>0</td>
</tr>
<tr>
<td>Element</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Values of CRIT\textsubscript{Instability} for different benchmark software programs

From the above values, it is inferred that some of the components of jGrasp and JUnit needs to be redesigned.

**ii) Bind Criticality Metric (CRIT\textsubscript{Children})**

I have defined Bind Criticality Metric as the number of components that exceeds the threshold count of children.

\[
\text{CRIT Children} = \# \text{ of Components that exceeds the threshold count of Children}
\]

If the *Number of Children* value for each component exceeds a threshold value, then that component is counted for CRIT Children.

**Rationale for the metric**

If the CRIT Children metric value increases, complexity increases and hence reusability decreases. Further work needs to be done on this metric to analyze usefulness of this metric.
5.3 Limitations of the new metrics

**Limitation of CRIT Instability and CRIT Children metrics due to lack of threshold value**

The threshold value for the CRIT Instability, which if exceeded, considers the application unstable, is not defined. Hence, this metric is not practically applicable unless the threshold value is fixed based on empirical results. The threshold value for the CRIT Children, which if exceeded, considers the application unstable, is not defined. Hence, this metric is not practically applicable unless the threshold value is fixed based on empirical results.

5.4 Summary

This chapter outlines two new metrics proposed by the author and provides their relevance towards measuring application quality. It is followed by the definition, rationale and limitations of two new metrics proposed.
Chapter 6

Inferences & Limitations on the Metrics

This chapter lists the inferences made by the author on the basis of the metric values collected by instrumenting the benchmark software. Analysis of the implication of the metrics for each quality factor have been tabulated on the basis of their definitions, which has been supported by the metric values obtained through a data collection process. The limitations of several metrics are compared with those of Narasimhan and Hendradjaya metrics [R.7]. The limitations of the best matched suite of metrics are then listed.

6.1 Basis of inferences

Inferences are listed below based on the theoretical definitions and the metric values collected. The best suite of metrics that matches the context of measuring the integrated components has also been provided.

Basis of inferences

Chapter 2 presented a suite of metrics considered in this thesis along with their definitions, rationale and limitations as defined by the authors of the metrics. I have made inferences based on the definitions of the metrics which are supported by practical results. The probable relative value of the metric in terms of various quality factors such as, reusability, complexity, testability, maintainability and modularity have also been carried out in this chapter. Inferences are provided on the best suite of metrics to test an integrated environment.
<table>
<thead>
<tr>
<th>Metrics</th>
<th>Junit</th>
<th>Element</th>
<th>mouseGestures</th>
<th>IDap</th>
<th>JCIFS</th>
<th>jGrasp</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPD</td>
<td>13.375</td>
<td>19</td>
<td>4.5</td>
<td>6.25</td>
<td>0.5</td>
<td>70.7</td>
</tr>
<tr>
<td>CID</td>
<td>61</td>
<td>6</td>
<td>11</td>
<td>114</td>
<td>70</td>
<td>204</td>
</tr>
<tr>
<td>CIID (Ce)</td>
<td>315</td>
<td>6</td>
<td>10</td>
<td>89</td>
<td>51</td>
<td>159</td>
</tr>
<tr>
<td>COID (Ca)</td>
<td>91</td>
<td>1</td>
<td>1</td>
<td>25</td>
<td>19</td>
<td>45</td>
</tr>
<tr>
<td>CAID (CID/8)</td>
<td>7.625</td>
<td>6</td>
<td>5.5</td>
<td>7.125</td>
<td>8.75</td>
<td>11.34</td>
</tr>
<tr>
<td>CRIT Inheritance</td>
<td>93</td>
<td>19</td>
<td>8</td>
<td>91</td>
<td>4</td>
<td>1142</td>
</tr>
<tr>
<td>CRIT size</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>AC (=CID)</td>
<td>61</td>
<td>6</td>
<td>11</td>
<td>114</td>
<td>70</td>
<td>204</td>
</tr>
<tr>
<td>NOC</td>
<td>16</td>
<td>4</td>
<td>0</td>
<td>22</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>LCOM</td>
<td>0.91</td>
<td>0.855</td>
<td>0.778</td>
<td>0.627</td>
<td>0.753</td>
<td>0</td>
</tr>
<tr>
<td>DIT</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>WMC</td>
<td>822</td>
<td>407</td>
<td>46</td>
<td>763</td>
<td>639</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 6: Table of metric values for benchmark software
Analysis of the metric values chart

Among the three suite of metrics used for comparison purpose, the author chose the best one matching the context of measuring the integrated components in order to measure various values. The inferences are made from Table 6.

Inferences from the CPD metric: From the theoretical analysis, if a high value for the CPD metric implies that the reusability decreases. Among the considered benchmarks, IDap, JCIFS and mouseGestures have low CPD values and hence highly reusable.

Inferences from the CID metric: A high value for the CID metric implies that the reusability decreases. Further, the time taken for testing and component complexity is high. MouseGestures and Element packages have low CID values, thereby having high reusability and less complexity.

Inferences from the CUD metric: A high value for the CUD metric implies that the complexity is high. IDap, JCIFS, mouseGestures and Element packages have comparably low values.

Inferences from the COID metric: A high value for the COID metric implies that the complexity of the component is relatively high. mouseGestures, IDap, JCIFS and jGrasp have low COID values, which implies that their complexity is less.

Inferences from the CAID metric: A high value for the CAID metric implies that the reusability property of the component decreases. The complexity of the component is considered high thus increasing the effort of testability. CAID metric value is relatively less for all the benchmarks considered.
**Inferences from the CRIT\textsubscript{Inheritance} metric:** A high value for the CRIT\textsubscript{Inheritance} metric implies a highly modular component; high modularity makes a component more reusable. IDap, jUnit and jGrasp are considered to be highly modular.

**Inferences from the CRIT\textsubscript{Size} metric:** If this value is high, it means it is less modular and so less reusable. For the benchmarks, the metric value is within the threshold value except for jGrasp.

**Inferences from the AC metric:** A high value for the AC metric implies that the component is highly modular. IDap and jGrasp are considered highly modular.

**Inferences from the NOC metric:** A high value for the NOC metric implies that the component is highly reusable but testability effort too increases [R10]. IDap, JCIFS, and jUnit are relatively highly reusable.

**Inferences from the LCOM metric:** A high value for the LCOM metric implies that the reusability of that component is high and the component is relatively less complex. jUnit, Element and JCIFS have very high values. It is inferred that all the packages are relatively reusable except jGrasp.

**Inferences from the DIT metric:** if DIT is high, reusability is high and complexity is high [R10]. IDap, JCIFS, mouseGestures and jUnit are highly reusable and highly complex.

This metric value and the inferences indicate that these kinds of metrics are not efficient at measuring the CBSE qualities, as they consider the value of the entire application as a single component.

**Inferences from the WMC metric:** if WMC is high, reusability is considered low. The packages IDap, JCIFS, jUnit are less reusable. This metric value and the inferences
indicate that these kinds of metrics are not efficient at measuring the CBSE qualities, as they consider the value of the entire application as a single component.

6.2 Implications of the inferences

6.2.1 Deficiency of Chidamber metrics used in comparison

The metrics defined by Chidamber and Kemerer and considered in this thesis are WMC, NOC, DIT and LCOM and according to them, metrics do not measure quality of integrated components. The metrics may be applicable to analyze reusability, complexity and size indirectly, but they are not sufficient to measure testing time and maintenance. The metrics proposed by me, considers the whole package as an assembly of sub-components rather than a single component. The authors of these metrics basically defined them in view of components in an isolated environment and consequently, these metrics prove to be deficient for integrated component testing.

6.2.2 Deficiency of Cho’s metrics used in comparison

The metrics defined by Cho and Kim [R.4] are CPC, CSC, CDC, and CCC. These metrics prove deficient for black-box testing. These metrics deal with the complexity of the code which requires the availability of the entire source code.

In the words of Narasimhan and Hendradjaya [R.7], Cho metrics calculate the complexity of metrics by using the combination of the number of classes, and interfaces. The calculation of cyclomatic complexity with the sum of classes and interfaces needs information from the source code, which is a shortcoming. This proves successful only if the developer has access to the source code. Furthermore, there is a need for black box testing of integrated components. The benchmarks for these metrics are not collected, because black-box testing approach is not possible with Cho metrics.
6.2.3 Best matched suite of metrics for Integrated Components

Narasimhan and Hendradjaya [R.7] defined metrics solely for integrated components. They classified their metrics to measure various aspects such as criticality, dynamic (for maintenance), and complexity. CID, CPD, CIID, COID, CAID are metrics that measure complexity of all the sub-components present in an application. These metrics measure both the packing density and interaction density among the components. CRIT \(_{\text{Link}}\), CRIT \(_{\text{Bridge}}\), CRIT \(_{\text{Inheritance}}\), CRIT \(_{\text{Size}}\), CRIT \(_{\text{all}}\) are the metrics categorized under criticality. These metrics test to check if any incorrect operations are not inherited by the subcomponents. Dynamic metrics measure maintenance and testing issues as a consequence of execution of the code.

The metrics of Narasimhan & Hendradjaya[R.7] are able to measure several aspects such as, reusability, complexity, testing-time, size and maintenance. The formulae given for each metric considers the average of the subcomponents rather than a single component, thus extending them to an integrated environment also.

6.3 Limitations & Criticality of the metrics

The suite of metrics proposed by Narasimhan and Hendradjaya prove to be efficient at measuring the quality of integrated components. However, there are some limitations that restrict the use of this suite of metrics, which are discussed in detail below:

6.3.1 Lack of threshold values

The lack of threshold values restricts the suite of metrics theoretically hindering its use practically. Narasimhan and Hendradjya have intuitively defined the values, but an
accurate threshold value calculated by quantifying and testing more empirical data is necessary.

6.3.2 Lack of metrics to measure Instability

While Narasimhan and Hendradjaya consider several aspects to assess the integrated component quality, there are no metrics to directly measure the aspect of instability directly. The author of this thesis suggested a few metrics that can be added to this suite of metrics to cover aspects of instability.

6.3.3 Criticality of the metrics

Criticality of the metrics means the limitations that stop the use of metrics for practical purposes and requires immediate solutions. Threshold values for the best suite of metrics have been defined intuitively which has to be revised at later stages and threshold values based on empirical data testing can be provided. Direct metrics to measure instability can be provided at later stages. The suite of metrics proposed by Narasimhan and Hendradjaya does not have critical problems that require immediate attention and there are no issues that restrict the metrics from applying to real time scenarios.

6.4 Summary

This chapter provides several inferences, based on this thesis work. The limitations and their criticalities are also discussed in detail.
Chapter 7

Conclusions & Future Research

This chapter summarizes the various chapters and offers several conclusions. It also provides pointers for further research on the selection of a good metrics suite solely to measure the quality of integrated components.

7.1 Summary of the Thesis

This section provides a review of the thesis and summarizes the key points.

Chapter one provides an overview of the thesis starting with an introduction to software engineering, software crisis, concepts in CBSE, and the need to measure quality of software. The role of existing metrics and their limitations, and the reason for the motivation of this thesis are provided.

Chapter two provides the metadata required for this thesis. The metrics considered for evaluation proposed by various authors are listed along with their definitions and rationale.

Chapter three compares the various metrics used and an analysis of metric behaviors is determined on the basis of their theoretical definitions. The metric value relative to the quality factor and their implications towards each quality of the component are tabulated.

Chapter four outlines the need for benchmarks and the details on the various benchmarks considered in this thesis. The criteria of choosing the benchmarks have been provided. The instrumentation programs used in the collection of metric values are also discussed in detail.
Chapter five outlines two new metrics proposed by the author beginning from the need for new metrics due to the deficiency of current metrics to measure certain quality attributes, and the necessity to measure them. It is followed by the definition, rationale and limitations of the new metrics proposed.

Chapter six outlines the basis of inferences, and lists the inferences made based on the observations. The limitations and the criticality of the best suite of metrics that suits a given context which might limit using the metrics are also discussed in detail.

7.2 Conclusion

The author contributed two new metrics to the existing metrics that improves the measure of the quality of the integrated components. The comparison chart facilitates any user who wants to measure the quality of integrated components to choose the best applicable metric based on their particular requirements. The metric values provided are helpful to study the behavior of metrics under various quality factors.

Possibility of overcoming the limitations

Considering the possibility of overcoming the limitations, more benchmarks have to be collected under various quality factors to study the behavior of metrics under all possible conditions. By setting up appropriate threshold values, the metrics can be used in several practical scenarios.

7.3 Future research

Future research lists the topics where further research can be carried out in the future. I define both short-term goals and long-term goals that involve resolving the degree of crisis in software engineering.
7.3.1 Short term goals

(1) **Collecting metric values for further benchmarks to study metric behavior:** The threshold values for several metrics have been set based on the metric values obtained from the benchmark software. The threshold value is an important in that it decides the stability and usefulness of an application and whether the application adheres to the principles of CBSE.

(2) **Revising the formulae used for calculation of metrics for greater accuracy:** Most of the current metric proposals consider the average of the component densities, irrespective of the interactions of each component. There is a scope for revising such metrics, taking into account importance-based weighted value of each component in their integration. The weighted value of component reflects its interaction in the given application.

(3) **Setting the Threshold values:** Threshold values for most of the metrics are not yet deduced precisely, which hinders the use of the metrics for integrated systems in practice. Further, empirical studies are therefore required.

7.3.2 Long-term goals

Future work will focus on a metric evaluation suite solely for the integrated environment, e.g., to assess a software system’s stability as a whole, considering the role of each component in a given application. Designing a metric evaluation environment will be another substantial undertaking for future considerations.
REFERENCES


Design”, IEEE Transaction on Software Engineering, Vol. 20 No. 6, pp. 476-493,
June 1994.

R.11) Martin, R.: Designing Object-Oriented C++ Applications Using the Booch


R.14) Szyperski ,“Component Software: Beyond Object-Oriented Programming”,
Addison-Wesley Professional, 2nd edition, 2002

2003.

R.16) Taylor, “An Introduction to Metadata”, University of Queensland, Australia,

9.2 Annotated Bibliography (not necessarily cited in the thesis)

AB.1) Basic knowledge of software metrics


AB.2) Applications

1) S.S. Ali & A. Ghafoor, “Metrics -Guided Quality Management for Component-