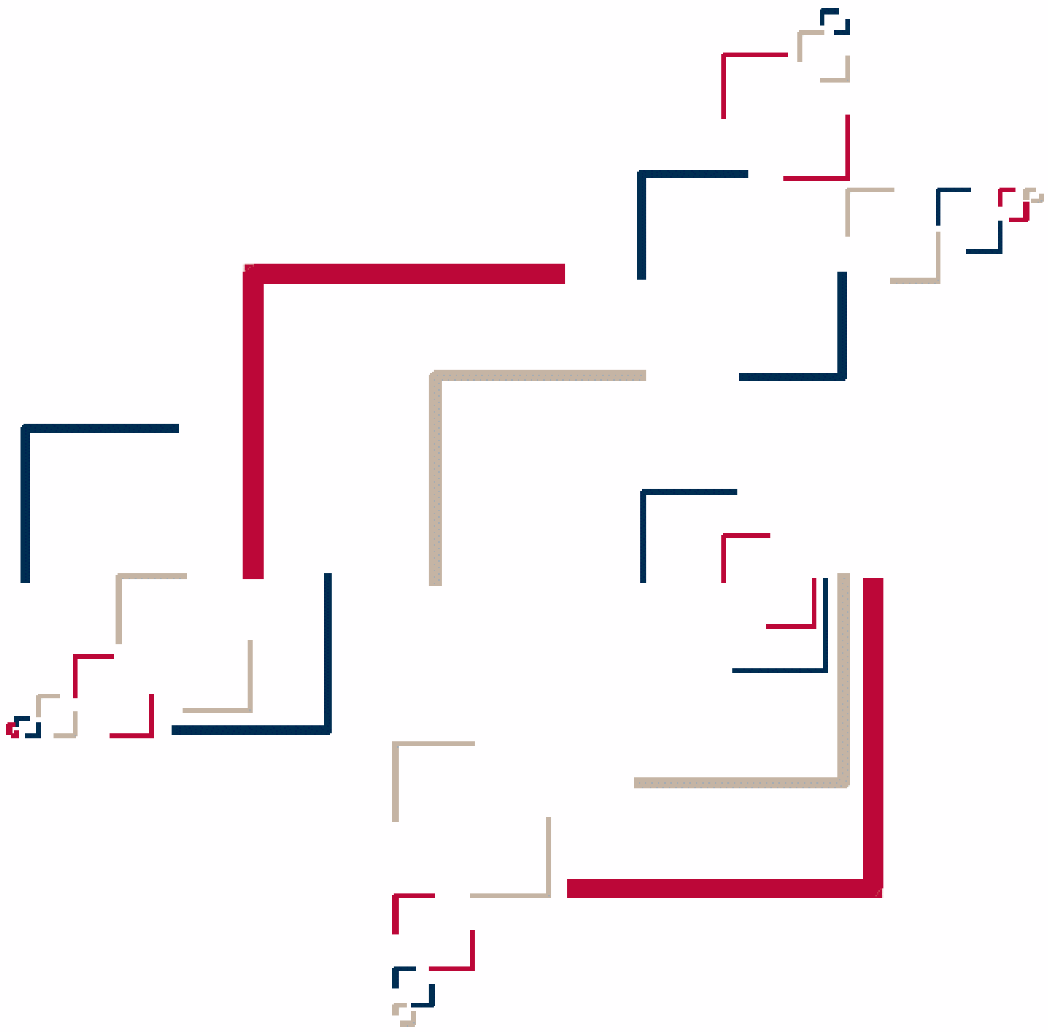

Micro Focus®

Enterprise View™

Measures and Metrics

MICRO® FOCUS

Measures and Metrics

Table of Contents

Measures and Metrics

1.INTRODUCTION

Software metrics are emerging as a powerful tool for the management of the software development process. Software metrics allow to apply engineering principles to software development, providing a quantitative and objective base for process and technology decisions.

The most important reason for establishing a measurement program is to evolve toward an understanding of software and the software engineering processes in order to derive models of those processes and examine relationships among the process parameters. Knowing what an organization does and how it operates is a fundamental requirement for any attempt to plan, manage, or improve.

Measurement provides the only mechanism available for quantifying a set of characteristics about a specific environment or for software in general.

Increased understanding leads to better management of software projects and improvements in the software engineering process. A software organization's objective may be to understand the status of the software engineering process or the implications of introducing a change.

General questions to be addressed might include the following:

- How much are we spending on software development?
- Where do we allocate and use resources throughout the life cycle?
- How much effort do we expend specifically on testing software?
- What types of errors and changes are typical on our projects?

In order to address such issues, an organization must have established a baseline understanding of its current software product and process characteristics, including attributes such as software size, cost, and defects corrected. Once an organization has analyzed that basic information, it can build a software model and examine relationships. For example, the expected level of effort can be computed as a function of estimated software size. Perhaps even more important, understanding processes makes it possible to predict cause and effect relationships, such as the effect on productivity of introducing a particular change into a process [NASA].

2.SOFTWARE METRICS

This chapter explains the main software metrics calculated by Enterprise View. The completed list of measures & metrics calculated by the product is available in this document.

2.1.Lines of codes

Lines of code (blank and comment lines are excluded by count) are an indication of the program size. This is very important for maintenance and reengineering. Program development and maintenance effort is primarily a function of program size in kilo lines of code (KLOC). The length hypothesis states that the number of bugs is proportional to the number of machine language statements. KLOC can be used for cost estimation of future projects, comparison of products and for determining productivity. Most of the standard cost estimation models depend on program size.

2.2.Cyclomatic index (McCabe)

McCabe defines Cyclomatic complexity as the number of independent paths through a program. McCabe defined this metric in support of determining the number of tests required to ensure all program paths are traversed during testing. His research was based on the development of directed graphs for a program. Every control construct that results in the addition of a new program path increments the complexity by one. Studies have shown that increased complexity reflects code with higher instances of defects [McCabe].

2.2.1.Technical details [S.E.I.]

The Cyclomatic complexity of a software module is calculated from a connected graph of the module (that shows the topology of control flow within the program):

- Cyclomatic complexity (CC) = E - N + p
- where E = the number of edges of the graph
- N = the number of nodes of the graph
- p = the number of connected components

To actually count these elements requires establishing a counting convention (tools to count Cyclomatic complexity contain these conventions). The complexity number is generally considered to provide a stronger measure of a program's structural complexity than is provided by counting lines of code.

A large number of programs have been measured, and ranges of complexity have been established that help the software engineer determine a program's inherent risk and stability. The resulting calibrated measure can be used in development, maintenance, and reengineering situations to develop estimates of risk, cost, or program stability. Studies show a correlation between a program's Cyclomatic complexity and its error frequency.

A common application of Cyclomatic complexity is to compare it against a set of threshold values. One such threshold set is:

- 1-10 simple program, without much risk;
- 11-20 more complex, moderate risk;
- 21-50 complex, high risk program;
- >50 un-testable program, very high risk.

2.2.2.Usage considerations [S.E.I.]

Cyclomatic complexity can be applied in several areas, including:

- *Code development risk analysis*. While code is under development, it can be measured for complexity to assess inherent risk or risk buildup.
- *Change risk analysis in maintenance*. Code complexity tends to increase as it is maintained over time. By measuring the complexity before and after a proposed change, this buildup can be monitored and used to help decide how to minimize the risk of the change.
- *Test Planning*. Mathematical analysis has shown that Cyclomatic complexity gives the exact number of tests needed to test every decision point in a program for each outcome. Thus, the analysis can be used for test planning. An excessively complex module will require a prohibitive number of test steps; that number can be reduced to a practical size by breaking the module into smaller, less-complex sub-modules.
- *Reengineering*. Cyclomatic complexity analysis provides knowledge of the structure of the operational code of a system. The risk involved in reengineering a piece of code is related to its complexity. Therefore, cost and risk analysis can benefit from proper application of such an analysis.

Cyclomatic complexity can be calculated manually for small program suites, but automated tools are preferable for most operational environments. For automated graphing and complexity calculation, the technology is language-sensitive; there must be a front-end source parser for each language, with variants for dialectic differences.

Cyclomatic complexity is usually only moderately sensitive to program change.

2.2.3.How Enterprise View calculates the Cyclomatic Complexity

In the Cyclomatic Complexity calculation, Enterprise View counts the following classes of instruction:

- *All conditional expressions (explicit condition)*
  - IF, UNTIL, WHEN, … (COBOL, PL/I)
  - Multiple (TO) cycle condition (PL/I)
- *All exception conditions*
  - ON EXCEPTION, ON SIZE ERROR, … (COBOL)
- *All instructions equivalent to conditional jumps (implicit condition);*
  - INVALID KEY, AT END, ON … GO TO …, … (COBOL)
  - ON <condition> (PL/I)

- *All internal routines*
    - In COBOL we count the number of entry points
    - In PL/I we count the number of procedures

- *CICS Exception conditions and conditional jumps*
  The behavior of the calculation is managed by the CalculateEmbMetrics parameter. This parameter is valid for COBOL and PL/I sources. It can be configured manually from the Configuration Manager – Work With Environment Properties. CalculateEmbMetrics has two values - TRUE and FALSE.

  If its value is FALSE, the metrics for CICS are not taken into consideration.

  If its value is TRUE, the Cyclomatic Complexity will increase according to the following expressions:
    - HANDLE CONDITION – every condition is counted if the (label) option is specified too (EXEC CICS)
    - HANDLE ABEND LABEL – for every (label) defined (you can have more than one label) (EXEC CICS)
    - HANDLE AID – for options ANYKEY, CLEAR, CLRPARTN, ENTER, LIGHTPEN, OPERID, PA1, PA2 , PA3 , PF1 …, PF12, TRIGGER but only if the (label) option is specified too (EXEC CICS);

2.3.Extended Cyclomatic Complexity (Extended McCabe)

The Extended Cyclomatic Complexity is an extension of Cyclomatic Complexity that accounts for the added complexity resulting from compound conditional expressions. A compound conditional expression is defined as an expression composed of multiple conditions separated by a logical OR or AND condition. If an OR or an AND condition is used within a control construct, the level of complexity is increased by one for computation of the Extended Cyclomatic Complexity measure.

2.3.1.How Enterprise View calculates Extended Cyclomatic Complexity

In the Extended Cyclomatic Complexity calculation, Enterprise View makes the same assumptions explained for the Cyclomatic Complexity calculation. Moreover it considers also the following keywords:

- *All keywords that creates combined conditional expressions*;
- AND, OR, ALSO, … (COBOL);
- | |, &&, … in PL/I;

2.4.Halstead Software Science [Halstead]

Maurice Halstead, who claimed they could be used to evaluate the mental effort and time required to create a program, and how compactly a program is expressed, developed this set of metrics.

2.4.1.Technical details [S.E.I.]

The Halstead measures are based on four scalar numbers derived directly from a program's source code:

- n_1 is the number of distinct operators
- n_2 is the number of distinct operands
- N_1 is the total number of operators
- N_2 is the total number of operands

From these numbers, 5 measures are derived:

- Program length $N = N_1 + N_2$
- Program vocabulary $n = n_1 + n_2$
- Volume $V = N * (\log_2 n)$
- Difficulty $D = \left(\frac{n_1}{2}\right) * \left(\frac{N_2}{n_2}\right)$
- Effort $E = D * V$
- Operand Usage $OU = \frac{n_2}{N_2}$
- Estimated Errors $\hat{B} = \frac{V}{EO}$ (3000<EO<3200)

Enterprise View calculates the four scalar numbers, program length and volume.

2.4.2.Usage considerations [S.E.I.]

The Halstead measures are applicable to operational systems and to development efforts once the code has been written. Because maintainability should be a concern during development, the Halstead measures should be considered for use during code development to follow complexity trends. A significant complexity measure increase during testing may be the sign of a brittle or high-risk module.

Halstead measures have been criticized for a variety of reasons, among them the claim that they are a weak measure because they measure lexical and/or textual complexity rather than the structural or logic flow complexity exemplified by Cyclomatic Complexity measures.

However, they have been shown to be a very strong component of the Maintainability Index measurement. In particular, the complexity of code with a high ratio of calculation logic to branch logic may be more accurately assessed by Halstead measures than by Cyclomatic Complexity, which measures structural complexity.

2.4.3.How Enterprise View identifies operands and operators

In Halstead measures calculation, Enterprise View makes the following assumptions:

- All declaration statement are ignored;
- All comments are ignored;
- All embedded language statements are ignored;

- All system-names, function-names, reserved-words, figurative constants, special registers, literals, separators referenced in the source are generically considered as mandatory and consequently are counted.

2.4.3.1.COBOL exceptions

For COBOL language, Enterprise View makes some exceptions to general assumptions:

- All points are ignored. In the COBOL language, the point is used to separate instructions, but only in some specific cases is mandatory (for example before a label) or is necessary to define the program flow (for example after an conditional instruction). Instead the point is mainly used to make clear the code. As their influence is very high, Enterprise View excludes it from the calculation;
- All constants are ignored;
- All keywords that can be defined as optional, independently by the context, are ignored. By following we provide the list of optional keywords: EXIT, IS, SKIP1, SKIP2, SKIP3, SUPPRESS, THAN, THEN and WITH.

2.4.4.Generic rules to identify operands and operators

2.4.4.1.Generic rules to identify an operator

The following rules allow to identify the operators:

- All verbs (main instructions);
  - ACCEPT, MOVE, IF, EVALUATE, … are verbs in COBOL;
- All uses of function;
  - MOVE FUNCTION LN(<variable 1>) TO <variable 2>..is a function call in COBOL;
  - <variable 1> = <procedure-name>(<variable 2>). Is a function call in PL/I;
- Optional keywords (context independent) must be ignored. In general the meaning doesn't change if the developer omits it;
  - 'IS', 'THAN', … in COBOL are optional keywords can be specified only for make the code more similar to English language;
- Only one operator must be considered when two of these must be used at the same time to write an instruction. Generally they identify the opening and the closing of something.
  - Brackets must be consider as only one operators in all languages;
  - IF … END-IF must be considered as only one operator in COBOL;

2.4.4.2.Generic rules to identify an operand

The following rules allow to identify the operands:

- All declared and used variables;
- All declarations of user-functions;
  - <label-name> PROCEDURE <procedure-name>…[PL/I]

- *All referred objects;*
    - CALL <program-name> … (COBOL);
- *All referred labels;*
    - GO TO <label-name> (COBOL);
- *All referred special registers;*
    - … DATE (COBOL);
- *All constant*;

2.5.Maintainability Index [Oman]

In an effort to better quantify software maintainability, several polynomial regression models have been defined by Oman, that predict software maintainability. These models use a combination of predictor variables in a polynomial equation to determine a Maintainability Index (MI). The predictor variables are a combination of a weighted coefficient and an independent metric. Oman's studies have shown strong correlation between Halstead's metrics, McCabe's Cyclomatic complexity, lines of code, and number of comments to the maintainability of the software system.

The original polynomial equations is defined as follows:

2.5.1. 3-metrics MI equation.

Maintainability Index = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln(aveLOC)

- **aveV** is the average Halstead Volume per module
- **aveV(g')** is the average extended Cyclomatic complexity per module
- **aveLOC** is the average lines of code per module.

2.5.2. 4-metrics MI equation

Maintainability Index = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln(aveLOC) + 50 * sin(sqrt(2.46 * percCM))

- **aveV** is the average Halstead Volume per module;
- **aveV(g')** is the average extended Cyclomatic complexity per module;
- **aveLOC** is the average lines of code per module;
- **percCM** is the average percent of lines of comments**.**

To determine which polynomial equation is the most appropriate fit to measure the MI of a given software system, some human analysis of the comments in the code must be performed. If any of the following criteria are true, the *3-metric* MI may be a better fit than the *4-metric* MI for measuring maintainability.

The comments do not accurately match the code. It has been said "the truth is in the code." Unless considerable attention is paid to comments, they can become out of sync with the code and thereby make the code less maintainable. The comments could be so far off as to be of dubious value.

There are large, company-standard comment header blocks, copyrights, and disclaimers. These types of comments provide minimal benefit to software maintainability. As such, the *4-metric* MI will be skewed and will provide an overly optimistic maintainability picture.

There are large sections of code that have been commented out. Code that has been commented out creates maintenance difficulties.

Generally speaking, if it is believed that the comments in the code significantly contribute to maintainability, the *4-metric* MI is the best choice. Otherwise, the *3-metric* MI will be more appropriate.

2.5.3.Integration of MI into Software Development

To maximize utilization of applying the MI to a changing software system, the ability to measure maintainability must be put into the hands of the engineers who are changing the code.

Measurement must be built into the software development environment, and it must be part of the software change process. The metrics must be easy to use, available on demand, and unobtrusive. Maintainability can then be built into the code. Yet, good metrics are not the goal; highly maintainable software is. The metrics serve as a maintainability indicator and still must be applied with common sense and engineering judgment.

When new code is being developed, it is easy to see how having an on demand metrics assessment capability can lead to the development of more maintainable software. After a module has been designed and coded, its maintainability can be measured. Should the MI evaluation predict maintenance difficulty, the module may be redesigned and coded to achieve better maintainability. Again, achieving a good MI is not the goal. Using MI will provide an unbiased second opinion as to the state of the software module, thereby gently reminding and encouraging the software developer to improve software engineering skills that lead to the development of higher quality code.

2.5.4.Metrics-Driven Maintenance

By integrating MI measurement into the software change process, the overall system maintainability will evolve in an organized fashion. This approach provides the appropriate controls to ensure minimal system degradation over time.

In the best case, the software can actually become more maintainable as changes are made. It is also highly cost effective, since modules already targeted for change are the ones being improved. This approach has shown to be an appropriate technique for focusing metrics application in a specific effort to effectively control software entropy.

3.Object-Oriented Software Measures

It is quite clear that measurement is necessary for the software development process to be successful. In addition, the path to controlling and improving the software design process may lie in the use of an object-oriented design approach.

The recent movement toward object-oriented technology must also include the processes that control object-oriented development, namely software measures.

The emphasis of this document is on the design and implementation phases of an object-oriented approach. This chapter explains the main object-oriented software measures calculated by Enterprise View. The completed list of measures & metrics calculated by the product is available in this document.

3.1.Nested and Inner Classes

Nested and inner classes is the number of all classes that are defined within the main class or other classes, defined in the same source file.

3.2.Top Level Classes

Top-level classes is the number of classes which are neither nested not inner. So this is the number of all the classes that are at the level of the main class.

3.3.Number of Attributes

Number of attributes is the number of main class instance variables (class properties).

The number of attributes in a class indicates the amount of data the class must maintain in order to carry out its responsibilities.

3.4.Number of Methods

Number of methods is the number of all the methods in the source file. Number of methods for main class only is provided, too.

3.5.Number of Constructors

This value denotes the number of constructors declared by the main class.

3.6.Method invocations

The number of method invocations performed by the main class is calculated; all invocations are taken into account, which the originating source code (method, constructor, static code).

3.7.DIT – Depth of Inheritance Tree [Chidamber]

DIT Metric is related to the concept of scope of properties, i.e. how far a property extends its influence, by means of inheritance (or, in turn, how far are declared the properties which influence a class).

Yet, deeper hierarchy likely means a greater number of inherited attributes and methods, and therefore more complexity in predicting class behavior.

The depth of Inheritance corresponds to the depth of the class in the inheritance tree. For the calculation are counted only source-defined ancestors. Any jar-defined class is not taken into account.

DIT Metric is useful to define:

- Reuse (higher DIT correspond to more reuse);
- Understandability (higher DIT correspond to more complex);
- Testability (higher DIT correspond to more complex)

3.8.CBO – Coupling Between Object classes [Chidamber]

CBO refers to the degree of interdependence between classes; in this context, only peer to peer (collaborative) coupling is considered, excluding inheritance-related coupling.

Two classes are coupled when methods declared in one class use methods or instance variables of another class.

Software design good-practices calls for minimizing coupling, for different reasons:

- Reuse: excessive coupling prevents reuse (class is no longer independent, or not enough independent to be reused);
- Maintenance: the larger the number of couples, the higher the sensitivity to changes in other parts of the design;
- Testing: likely, the higher the class coupling, the more complex the testing

Measure is given by the number of distinct classes whose methods and attributes are used by the analyzed class. When measuring this metric, ancestor classes are excluded.

3.9.Lack of Cohesion Of Methods

Cohesion refers to the internal consistency within parts of the design. In other words, we could say that it refers to the concept of similarity, given as the intersection of the sets of properties of two things. This concept is applied to pairs of methods, giving the degree of similarity of methods, that can be defined as the intersection of sets of attributes used by each of them.

Two methods whose degree of similarity is not zero are said to be Similar.

The degree of similarity of methods can be viewed as a major aspect of object class cohesiveness. If an object class has different methods performing different operations on the same set of instance variables, the class is cohesive.

Yet, Chidamber and Kemerer highlight that software design good-practice calls for maximizing cohesiveness:

- Lack of cohesion (low value) implies that classes should probably be split into two or more sub-classes;
- Low cohesion increases complexity;
- Disparateness of methods helps identify flaws in the design of classes

A high LCOM (Lack of Cohesion of Methods) value indicates:

- That the class design resulted in attempt to achieve many different objectives (giving less predictable behavior, error-proneness, complex testing);
- A good class subdivision.

3.9.1.LCOM1 – Lack of Cohesion Of Methods [Henderson & Sellers]

Henderson-Sellers gave the following interpretation of LCOM:

- It is the number of pairs/couples of methods having no common attribute

LCOM1 can be simply calculated by counting all the intersections between method attributes, whose result is the empty set.

Consider class C1 with methods M1, M2, M3, .., Mn

For each two methods (Mi, Mj) the common class instance variables (class attributes/properties) are counted:

$$\sum_{common\_class\_instance\_variables} (M_i, M_j)$$

If this sum is equal to zero ($\sum_{common\_class\_instance\_variables} (M_i, M_j) = 0$) than 1 is added to the result.

3.9.2.LCOM2 – Lack of Cohesion Of Methods [Chidamber & Kemerer]

LCOM2 is the count of the number of method pairs whose similarity is 0, minus the count of method pairs whose similarity is not zero.

LCOM2 can be simply calculated by:

- Counting all the intersections between method attributes, whose result is the empty set (P set);
- Counting all the intersections between method attributes, whose result is a non empty set (Q set)

Consider a Class

C1

With *n* methods

M1, M2..., Mn

Let

{Ij} = set of instance variables used by method Mi.

There are *n* such sets

{I1}, ... {In}

Let

$P = \{ (Ii,Ij) \mid Ii \cap Ij = \varnothing \}$

Let

$Q = \{ (Ii,Ij) \mid Ii \cap Ij \_ \varnothing \}$.

$$LCOM2 = \begin{cases} |P| - |Q| & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases}$$

3.9.3.LCOM3 – Lack of Cohesion Of Methods [Hitz & Montazeri]

Hitz and Montazeri proposed a different, graph-theoretic formulation of LCOM metric: it is the number of pairs/couples of methods with at least one common instance variable

LCOM3 is then defined as the number of connected components of G, that is, the number of method "clusters" operating on disjoint sets of instance variables.

LCOM3 can be calculated by:

- Create a graph **G** whose vertex **V** are class methods;
- Add an edge **E** for each pair of similar methods;
- **LCOM3** is the number of connected components of **G**

$E = \{ (m,n) \in V \times V \mid \exists i \in Ix : (m \text{ accesses } i) \wedge (n \text{ accesses } i) \}$

3.9.4.LCOM4 – Lack of Cohesion Of Methods [Hitz & Montazeri]

LCOM3 presents a problem for access methods (methods which provide read or write access to an attribute). As access methods exist, other methods no longer need to directly reference these attributes. In turn, invocation of access methods represent an access to corresponding attributes.

LCOM4 is intended to give a correct representation of cohesion in such a case; the graph is changed, adding edges not only for similar methods, but also for pairs of methods in which one invokes the other.

Such methods are said to be indirectly connected (where similar methods are directly connected).

LCOM4 can be calculated by:

- Create a graph **G** whose vertex **V** are class methods;

- Add an edge **E** for each pair of similar methods;
- Add an edge **E** for each pair of indirectly connected methods;
- **LCOM4** is the number of connected components of **G.**

3.9.5.LCOM5 – Lack of Cohesion Of Methods [Henderson & Sellers]

LCOM5 Metric provides a cohesion Metric expressed as percentage value. Therefore, it has the advantage of falling in a defined range (0 – 1, if each attribute is accessed by at least one method).

Consider a set of methods
$\{M_i\}$ (I= 1, …,m)
Accessing a set of attributes
$\{Aj\}$ (j = 1, …,a)
Let the number of methods which access each attribute be
μA_j.
The simplest formula which gives the properties discussed above (a value of zero with full cohesion and a value of 1 for no cohesion) is:

$$LCOM5 = \frac{\left[\frac{1}{a} * \sum \mu(Aj)\right] - m}{(1 - m)}$$

4.List Of Measures & Metrics Calculated By Enterprise View

4.1.Size metrics

Description	Source	Compiled
Number of sources	Yes	
Number of measured sources	Yes	
Lines of source (blank and commented lines included)	Yes	Yes
Lines of code (without blank and commented lines)	Yes	Yes
Commented lines of code	Yes	Yes
Ratio CLOC/LOC	Yes	

4.2.Complexity metrics

Description	Source	Compiled
Number of distinct operands by Halstead (n2)	Yes	Yes
Cyclomatic Complexity (McCabe)	Yes	Yes
Extended Cyclomatic Complexity (Extended McCabe)	Yes	Yes

4.3.Halstead's product metrics

Description	Source	Compiled
Number of distinct operands by Halstead (n2)	Yes	Yes
Number of total operands by Halstead (N2)	Yes	Yes
Number of distinct operators by Halstead (n1)	Yes	Yes
Number of total operators by Halstead (N1)	Yes	Yes
Halstead Vocabulary	Yes	Yes
Halstead Volume	Yes	Yes
Halstead Length	Yes	Yes

4.4.Quality metrics

Description	Source	Compiled
OMAN 3-metrics MI equation (3-MI)	Yes	Yes
OMAN 4-metrics MI equation (4-MI)	Yes	Yes

4.5.Object Oriented metrics

Description	Source	Main Class
Number of nested/inner classes		Yes
Number of top level classes		Yes
Number of constructor in the main class		Yes
Number of method in source	Yes	
Number of method in main class		Yes
Number of attributes in the main class		Yes
Method invocations		Yes
Chain’s depth from source-defined ancestor		Yes
Coupling between object		Yes
LCOM1 (HENDERSON & SELLERS)		Yes
LCOM2 (CHIDAMBER & KEMERER)		Yes
LCOM3 (HITZ & MONTAZERI)		Yes
LCOM4 (HITZ & MONTAZERI)		Yes
LCOM5 (HENDERSON & SELLERS)		Yes

5.Bibliography

[Halstead] Maurice H. Halstead, Elements of Software Scienc-OP, Elsevier Science Inc., New York, NY, 1977.

[McCabe] McCabe, T., "A Complexity Measure," *IEEE Transactions on Software Engineering*, December 1976.

[NASA] National Aeronautics And Space Administration, *Software Measurement Guidebook.*

[Oman] Oman, p., and J. Hagemeister, "Constructing and Testing of Polynomials Predicting Software Maintainability," *Journal of Systems and Software.*

[S.E.I.] Software Engineering Institute at http://www.sei.cmu.edu/

[Chidamber] Chidamber, Shyam R. & Kemerer, Chris F. "Towards a Metrics Suite For Object Oriented Design," pp. 197-211. Proceedings: OOPSLA'91. Phoenix, AZ, October 6-11, 1991. New York, NY: ACM SIGPLAN Notices, 1991.

Chidamber, Shyam & Kemerer, Chris F. "A Metrics Suite for Object-Oriented Design." IEEE Transactions on Software Engineering 20, 6 (June 1994): 476-493.

[Henderson & Sellers] Henderson-Sellers, Brian. A Book of Object-Oriented Knowledge. Englewood Cliffs, NJ: Prentice-Hall, 1992. ISBN 0-130-59445-8.

Henderson-Sellers, Brian & Edwards, J.M. Book Two of Object-Oriented Knowledge: The Working Object. Riverwood, NSW Australia: Prentice-Hall Ligare Pty Ltd, 1994. ISBN 0-130-93980-3.

[Chidamber & Kemerer] Shyam R. Chidamber, Chris F. Kemerer "Towards a Metrics Suite for Object Oriented Design" Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139

[Hitz & Montazeri] Martin Hitz, Behzad Montazeri "Measuring Product Attributes of Object-Oriented Systems." Proc. 5th European Software Engineering Conf. (ESEC 95)