An overview of Object Oriented Design Metrics

Master Thesis

Department of Computer Science, Umeå University, Sweden

June 23, 2005

Author: Muktamyee Sarker
Email: ens03msr@cs.umu.se

Supervisor: Jürgen Börstler
Email: jubo@cs.umu.se

ACKNOWLEDGMENT

I would like to thank Mr.. Jürgen Börstler who supervised my thesis work with his advices and suggestions in the fulfilments of this thesis. Without him, this study would never exist. My special thanks to Mr. Per Lindström, for giving an opportunity to carry out my studies. Thanks to my parents for financial and mental support**.**

ABSTRACT

Object oriented design is becoming more popular in software development environment and object oriented design metrics is an essential part of software environment. This study focus on a set of object oriented metrics that can be used to measure the quality of an object oriented design.

The metrics for object oriented design focus on measurements that are applied to the class and design characteristics. These measurements permit designers to access the software early in process, making changes that will reduce complexity and improve the continuing capability of the design.

This report summarizes the existing metrics, which will guide the designers to support their design. We have categorized metrics and discussed in such a way that novice designers can apply metrics in their design as needed.

Table of contents

1 Introduction

It is widely accepted that object oriented development requires a different way of thinking than traditional structured development[1] and software projects are shifting to object oriented design. The main advantage of object oriented design is its modularity and reusability. Object oriented metrics are used to measure properties of object oriented designs.

Metrics are a means for attaining more accurate estimations of project milestones, and developing a software system that contains minimal faults [7]. Project based metrics keep track of project maintenance, budgeting etc. Design based metrics describe the complexity, size and robustness of object oriented and keep track of design performance.

Compared to structural development, object oriented design is a comparatively new technology. The metrics, which were useful for evaluating structural development, may perhaps not affect the design using OO language. As for example, the "Lines of Code" metric is used in structural development whereas it is not so much used in object oriented design. Very few existing metrics (so called traditional metrics) can measure object oriented design properly. As discussed by Bellin [7], Vessey et al. [40] claim that "metrics such as Line of Code used on conventional source code are generally criticized for being without solid theoretical basis".

One study estimated corrective maintenance cost saving of 42% by using object oriented metrics [21]. There are many object oriented metrics models available and several authors have proposed ways to measure object oriented design. The motivation of this thesis is to give an overview of object oriented design metrics.
This report is organised in the following way. The next section will discuss object oriented design in the context of metrics. Section 3 discusses metrics and their quality. Section 4 focuses on the Goal Question Metrics approach. Section 5 describes different metrics models. Evaluations of metrics are discussed in section 6. In this section we will show some of metrics analysis result. Section 7 discusses the summary of this study.

[1] Jürgen Börstler: Teaching and Learning OO, Extended Abstract, Department of Computing Science Umeå University, SE–901 87 Umeå, Sweden

2 Object Oriented Design

Object oriented design is concerned with developing an object-oriented module of a software system to apply the identified requirements. Designer will use OOD because it is a faster development process, module based architecture, contains high reusable features, increases design quality and so on.

> "*Object-oriented design is a method of design encompassing the process of object-oriented decomposing and a notation for depicting both logical and physical as well as static and dynamic models of the system under design"*[9].

Objects are the basic units of object oriented design. Identity, states and behaviors are the main characteristics of any object. A class is a collection of objects which have common behaviors.

> "*A class represents a template for several objects and describe how these objects are structured internally. Objects of the same class have the same definition both for their operation and for their information structure*" [19].

There are several essential themes in object oriented design. These themes are mostly support object oriented design in the context of measuring. These are discussing in next sub section.

2.1 Internal quality of OOD

Cohesion

Cohesion refers to the internal consistency within the parts of the design. Cohesion is centred on data that is encapsulated within an object and on how methods interact with data to provide well-bounded behaviour. A class is cohesive when its parts are highly correlated. It should be difficult to split a cohesive class. Cohesion can be used to identify the poorly designed classes.

> "*Cohesion measures the degree of connectivity among the elements of a single class or object"* [9].

Coupling

Coupling indicates the relationship or interdependency between modules. For example, object X is coupled to object Y if and only if X sends a message to Y that means the number of collaboration between classes or the number of messages passed between objects. Coupling is a measure of interconnecting among modules in a software structure.

Inheritance

Inheritance is a mechanism whereby one object acquires characteristics from one, or more other objects. Inheritance occurs in all levels of a class hierarchy.

> "*Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship".*[2]

In general, conventional software does not support this characteristic because it is a pivotal characteristic in many object oriented systems as well as many object oriented metrics focus on it. (See chapter 5.3 for more information)

Encapsulation

Encapsulation is a mechanism to realize data abstraction and information hiding. Encapsulation hides internal specification of an object and show only external interface.

> "*The process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation"* [9].

Encapsulation influences metrics by changing the focus of measurement from a single module to a package of data.

Information Hiding

Booch [9] States that, information hiding is the process of hiding all the secrets of an object that do not contribute to its essential characteristics. An object has a public interface and a private representation; these two elements are kept distinct. Information hiding acts a direct role in such metrics as object coupling and the degree of information hiding.

[2] Rumbaugh, J.,Blaha, M., Premerlani,W., Eddy F. And Lorenses, W: Object oriented modeling and design, Prentice Hall, 1991.

> **"***All information about a module should be private to the module unless it is specifically declared public".*[3]

Localization

In object oriented design approach localization is based on objects. In a design, if there is some changes in the localization approach, the total plan will be violated, because one function may involve several objects, and one object may provide many functions.

> *"Localization is the process of gathering and placing things in close physical proximity to each other".*[4]

Metrics should apply to the class as a complete entity. Even the relationship between functions and classes is not necessarily one-to-one. For that reason, metrics that reflect the manner in which classes collaborate must be capable of accommodating one-to-many and many-to-one relationships [34].

2.2 Principles of OOD

This section shows some OO design principles, which are used for support in OO design. Object oriented principles advise the designers what to support and what to avoid. We categorized all design principles into three groups in the context of design metrics. These are general principles, cohesion principles, and coupling principles. These principles are collected by Martin [33]. Some of the principles are measure in section 6. The following discussion is a summary of his principles according to our categories.

2.2.1 General Principles

The Open/Closed Principle (OCP): Open close principle states a module should be open for extension but closed for modification i.e. Classes should be written so that they can be extended without requiring the classes to be modified.

[3] Meyer, B.: Object-oriented Software Construction, Prentice Hall, 1998.
[4] Edward V. Berard, The Object Agency, Inc, http://www.toa.com/pub/moose.htm

The Liskov Substitution Principle (LSP): Liskov Substitution Principle mention subclasses should be substitutable for their base classes i.e. a user of a base class instance should still function if given an instance of a derived class instead.

The Dependency Inversion Principle (DIP): Dependency Inversion Principle state high level classes should not depend on low level classes i.e. abstractions should not depend upon the details. If the high level abstractions depend on the low level implementation, the dependency is inverted from what it should be, [32].

The Interface Segregation Principle (ISP): Interface Segregation Principle state Clients should not be forced to depend upon interfaces that they do not use. Many client-specific interfaces are better than one general purpose interface.

2.2.2 Cohesion Principles

Reuse/Release Equivalency Principle (REP): The granule of reuse is the granule of release. Only components that are released through a tracking system can be efficiently reused. A reusable software element cannot really be reused in practice unless it is managed by a release system of some kind of release numbers. All related classes must be released together.

Common Reuse Principle (CRP): All classes in a package should be reused together. If reuse one of the classes in the package, reuse them all. Classes are usually reused in groups based on collaborations between library classes.

Common Closure Principle (CCP): The classes in a package should be closed against the same kinds of changes. A change that affects a package affects all the classes in that package. The main Goal of this principle is to limit the dispersion of changes among released packages i.e. changes must affect the smallest number of released packages. Classes within a package must be cohesive. Given a particular kind of change, either all classes or no class in a component needs to be modified.

2.2.3 Coupling Principles

Acyclic Dependencies Principle (ADP): The dependency structure for a released component must be a Directed Acyclic Graph (DAG) and there can be no cycles.

Stable Dependencies Principle (SDP): The dependencies between components in a design should be in the direction of stability. A component should only depend upon components that are more stable than it is.

Stable Abstractions Principle (SAP): The abstraction of a package should be proportional to its stability. Packages that are maximally stable should be maximally abstract. Instable packages should be concrete.

2.3 Symptoms of bad design

Designers can perform a good OO design by following the OOD principles discussed above (sec 2.2). If designers know the reasons for and symptoms of bad design then it is helpful for them to avoid the bad design. There are some reasons for bad design, as for example: changing technology, domain complexity, lack of design skills and design practices and so on.

Technology is “constantly changing”. So for a good design, it is usual to adapt with new technologies. Now it is the era of OOD, because various properties of OOD (Inheritance, modularity etc) support the modification without changing the previous or existing modules. But one should always be careful about some properties of OOD, which can make the design more complex, for example “inheritance” property. Designers cannot be able to use OOD in such a way that it will help him in case of later with the change of technologies but will not make the program more complex. Too much method makes a system complex. We will discuss more about complexity in section 5.2. Martin [32] proposes four primary symptoms tell whether designs are rotting. They are not orthogonal, but are related to each other in ways that will become obvious. They are: rigidity, fragility, immobility, and viscosity. The following is a summary of his work

Rigidity

The concept of rigidity is if the design change in simple way the entire design will be change, i.e. a design is rigid if a single change causes a cascade of subsequent change in dependent modules. More module changes in a design indicates more rigid the system.

Fragility

The idea of fragility is that changes cause new bugs i.e. the tendency of a program to break in many places when a single change made. Martin [32] states the new problem are in areas that have no conceptual relationship with the area that was changed, fixing those problems leads to even more problem and the development team begins to resemble a dog chasing its tail.

Immobility

Immobility means unsuccessful to reuse software from different or same design. Sometimes it happens that one designer will find out that he needs a module which is already written by another designer. It means similar module in a design makes immobile.

Viscosity

Martin [32] states viscosity comes in two forms: viscosity of the design and viscosity of the environment. Designers always look for more options to make changes their design if they need to change something. In any cases designers maintain their design. According to Martin [32], viscosity of design indicates, "when the design preserving methods are harder to employ than the hacks, and then the viscosity of the design is high". It is easy to do the wrong thing, but hard to do the right thing. Viscosity of environment indicates slow and inefficient environment in a design.

Object oriented design is fundamentally different from software developed using conventional methods (procedural methods). The purposes of design principles are to mark poor use of inheritance and poor dependencies of design structure, along with among other kinds of design errors. The knowledge of Bad Design Symptom assists to the designer to perform better. The metrics for object oriented system focus on measurements that are applied to the class and the design characteristics, for example encapsulation, information hiding, inheritances, localization, etc. So Object oriented metrics are usually used to assess the quality of software designs. Next section we will discuss metrics and their quality.

3 Metrics and Quality

This section focuses on measurements and corresponding measurement criteria. Different kinds of metrics and their quality are also discussed in this subsection.

3.1 Introduction

Since object oriented system is becoming more pervasive, it is necessary that software engineers have quantitative measurements for accessing the quality of designs at both the architectural and components level. These measures allow to designer to access the software early in the process, making changes that will reduce complexity and improve the continuing capability of the product. The measurement process is to drive the software measures and metrics that are appropriate for the representation of software that is being measured. Suitable metrics are analysed based on pre-established guidelines and past data [34].

3.2 Metrics

We categorized metrics into two groups: project based metrics and design based metrics. Project based metrics contain process, product and resources; these are discussed in next sub section. Design based metrics contain traditional metrics and object oriented metrics. In traditional metrics, we will discuss complexity metrics, SLOC (Source lines of code), and CP (Comment percentage) metric, see section 5.2.1. Object oriented metrics are discussed in section (5.5.2 to 5.2.4). The following figure shows metrics hierarchy according to our categorization.

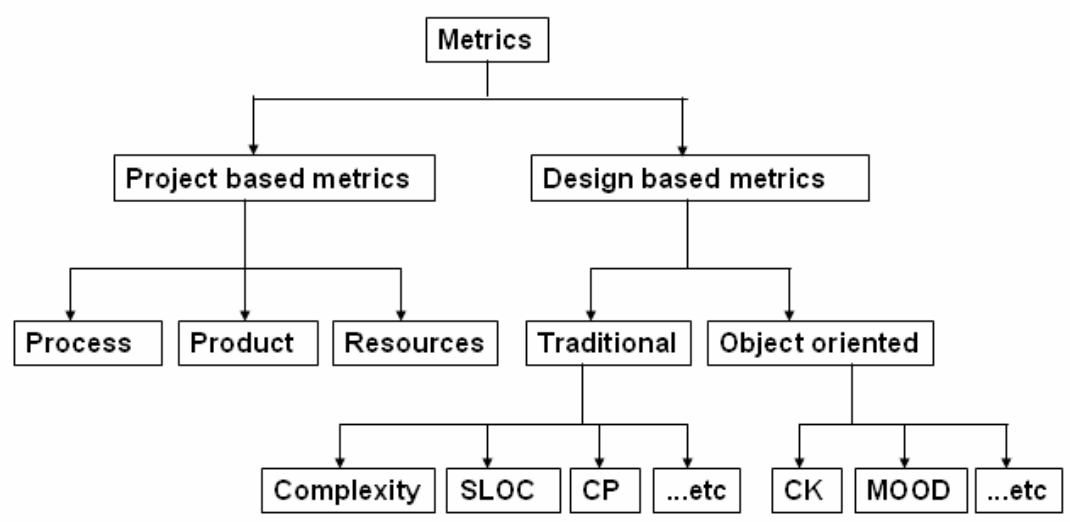


Figure 1: Metrics hierarchy

Norman E. Fenton et al. [14] propose three kinds of entities and attributes to measure in software design. The entities are process, product, resources and attributes are internal and external attributes. The following is a summary of his discussion

3.2.1 Process

Processes are set of software related activates which are used to measure the status and progress of the system design and to predict future effects. A process is usually related with some timescale. The timing can be explicit, as when an activity must be finished by a specific date, or implicit, as when one activity must be finished before another can begin. The following examples of a process related metrics that it is proposed to collect when working with object oriented software engineering (OOSE) [19].

- Total development time,
- Development time in each process and subprocess,
- Time spent to modify models from previous processes,
- Time spent in all kinds of subprocess, such as use case specification, object specification, use case design, block design, block testing and use case testing for each particular object,
- Number of different kind of fault found during reviews,
- Number of change proposals on previous models,
- Cost for quality assurance,
- Cost for introducing new development process and tools.

3.2.2 Products

Product metrics are used to control the quality of the software product. These metrics are applied to incomplete software products in order to measure their complexity and to predict properties of the final product. Products are any artefacts, deliverables or documents that result from a process activity. Products are not restricted to the items that management is committed to deliver to the customer. Any artefact or document produced during the software life cycle can be measured. Various kinds of product related metrics are proposed. None of these have been demonstrated to be generally useful as overall quality predictor. However, some quality criteria can be used to predict a certain quality property [19] as follow:

- Number, width and height of the inheritances hierarchies,
- Number of classes inheriting a specific operation,

- Number of classes that a specific class is dependent on,
- Number of classes that are dependent on a specific class,
- Number of direct users of a class or operation.

3.2.3 Resources

Resources are entities required by a process activity. The resources that we want to measure include any input for software production. Thus, personnel, materials, tools, and methods are candidates for measurement. According to internal and external attribute each class of entity can be distinguish.

Internal attributes

Internal attributes of a product, process or resource are those that can be measured purely in terms of the product, process, or resource itself. In other words, an internal attribute can be measured by examining the product, process or resource on its own.

External attributes

External attributes of a product, process or resource are those that can be measured only with respect to how the produce process or resource, relates to its environment. Here, the behavior of the process, product or resource is important, rather than the entity itself.

Table 1 represents a classification of software metrics [14]. Essentially any software metrics is an attempt to measure or predict some internal or external attribute of some product, process, or resource. The table provides a feel for the board scope of software metrics, and clarifies the distinguished between the attributes [37].

Entities	**Attributes**	
	Internal	**External**
Products		
Specification	Size, reuse, modularity, redundancy, functionality, syntactic correctness.	Comprehensibility, maintainability,
Design	Size, reuse, modularity, coupling, cohesiveness, inheritance, functionality.	Quality, complexity, maintainability.

Code	Size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness.	Reliability, usability, maintainability, reusability.
Test data	Size, coverage level.	Quality, reusability.
Process		
Constructing Specification	Time, effort, number of requirements changes.	Quality, cost, stability.
Detailed design	Time effort, number of specification faults found.	Cost, cost-effectiveness.
Testing	Time, effort, number of coding faults found.	Cost, cost-effectiveness, stability, …
Resources		
Personnel	Age, price.	Productivity, experience, intelligence.
Teams	Size, communication level, structuredness.	Productivity, quality.
Organization	Size, ISO Certification, CMM level	Maturity, profitability.
Software	price, size.	Usability, reliability.
Hardware	Price, speed, memory size.	Reliability.
Offices	Size, temperature, light.	Comfort, quality.

Table 1: Components of software measurements (taken from [14])

3.3 Measuring quality

Measurement enables to improve the software process, assist in the planning, tracking the control of a design. A good software engineer uses measurements to asses the quality of the analysis and design model, the source code, the test cases, etc. What does quality mean?

"Quality refers to the inherent or distinctive characteristics or property of object, process or other thing. Such characteristics or properties may set things apart from other things, or may denote some degree of achievement or excellence"[5].

Many quality measures can be collected from literature, the main goal of metrics is to measure errors and defects. The following quality factor should have every metrics [11, 20, 35]:

- Efficiency - Are the constructs efficiently designed?
  The amount of computing resource and code required by a program to perform its function.
- Complexity - Could the constructs be used more effectively to decrease the architectural complexity?
  ….
- Understandability - Does the design increase the psychological complexity?
  ….
- Reusability - Does the design quality support possible reuse?
  Extent to which a program or part of a program can be reused in other application , related to the packaging and scope of the functions that the program performs.
- Testability/Maintainability - Does the structure support ease of testing and changes?
  Effort required locating and fixing an error in a program, as well as effort required to test a program to ensure that it performs its intended function.

How do we know that our metrics measure the desired design qualities? We should establish the objectives of measurements before data collection begins and then we should define each and every metrics in a way that measure the quality of a design. Next section is discussing the widely known GQM approach.

[5] This definition is taken from "http://en.wikipedia.org/wiki/Quality"

4 GQM

Basili et. al. [5] developed GQM (Goal Question Metric) approach. This approach was originally defined for evaluating defects for a set of projects in the NASA Goddard Space Flight Center environment. It provides a framework involving three steps:

1. List major goals of the development or maintenance project.
2. Derive from each goal the questions that must be answered to determine if the goals are being met.
3. Decide what must be measured in order to be able to answer the questions adequately.

He has also provided a series of templates which are useful for designers. The goals of GQM can be expressed by means of a template which covers purpose, perspective and environment; a set of guidelines also proposed for driving question and metrics. As discussed in [14, 34] the following discussion is a summary of basili's discussions.

Purpose

The purpose template is to articulate what is being analyzed, for example it is used to characterize, evaluate, predict, motivate from the process, product, model, and metric. This template also expresses what purpose it will be used. For example, a designer might want to evaluate the maintenance process in order to improve.

Perspective

The perspective template focuses on the factors which are important within the process or product that is being evaluated, for example cost, effectiveness, correctness, defects, changes, product measures, maintainability, testability, usability. Customers and developers are the main two perspective of software development process. A developer might examine the cost from the viewpoint of the manager.

Environment

The environment template consists of the process factors, people factors, problem factors, methods, tools constraints as for example the type of the computer system that is being used, the skills of the stuff involves, the amount of trained resource available. For example, the maintenance staffs are poorly motivated programmers who have limited access to tools.

When the purpose, perspective and environment of a goal have been specified, the process of questioning and metric development can begin. As for example, an application of the template for the goal definition is as follow.

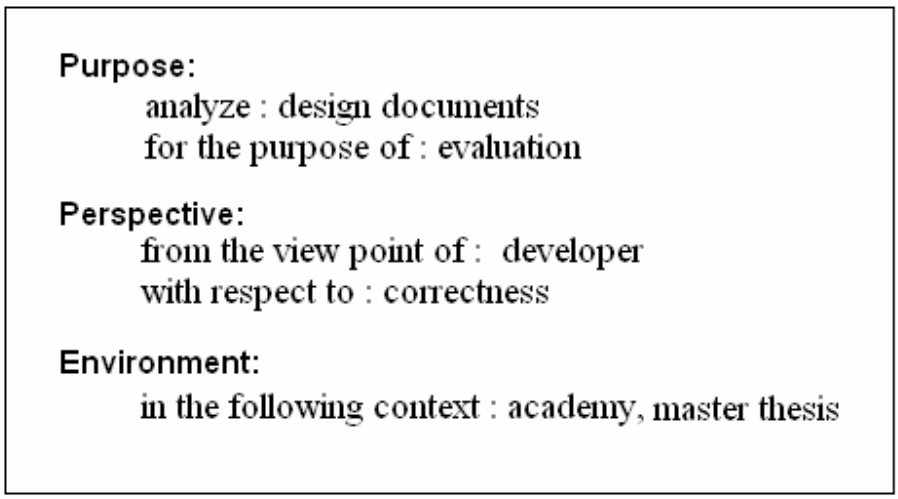


Figure 2: Goal template [6]

The result of the application of the GQM approach application is the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data [6]. The GQM approach has three levels. The following is a summery of [6] discussion.

1. **GOAL** (Conceptual level): A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment. Objects of measurement are products, processes and resources (these are discussed in section 4.2).

2. **QUESTION** (Operational level): A set of questions is used to characterize the way the assessment/achievement of a specific goal is going to be performed based on some characterizing model.

3. **METRIC** (Quantitative level): A set of data is associated with every question in order to answer it in a quantitative way. The data can be objectives and subjective.

[6] As discussed Annabella Loconsole: ” Measuring the requirements management key process area”

- This data is said to be objective if they depend only on the object that is being measured and not on the viewpoint from which they are taken. For example, number of versions of a document, staff hours spent on a task, size of a program.
- The data is said to be subjective if they depend on both the object that is being measured and the viewpoint from which they are taken. For example, readability of a text, level of user satisfaction.

The GQM approach define some goals, refine those goals into a set of questions, and the questions are further refined into metrics. Consider the following figure, for a particular question; G1 and G2 are two goals, Q2 in common for both of these goals. Metric M2 is required by all three questions. The main idea of GQM is that each metric identified is placed within a context, so metric M1 is collected in order to answer question Q1 to help achieve the goal G1.

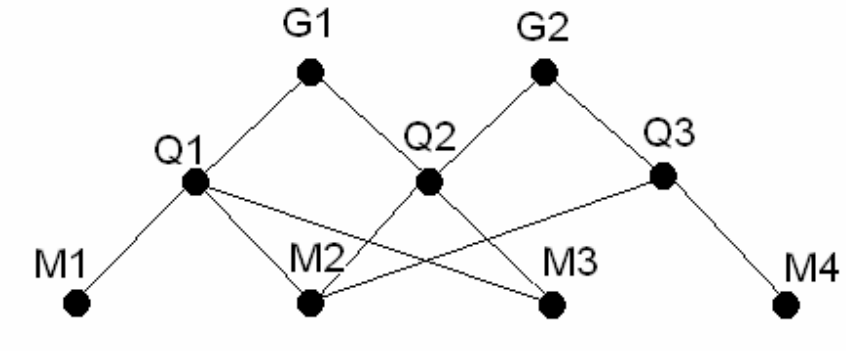


Figure 2: Goal-Question-Metrics hierarchy

Consider a goal[7] is to evaluate the effectiveness of using a coding standard. To decide if the standard is effective, we have to check some questions. A question might be 'who is using the standard' because it is important to know what proportion of coders is using the standard. The metric might be the proportion of coders using the standard, and so on. A number of measurements may be needed to answer a single question; on the other hand, a single measurement may be applied to more than one question. The following figure shows how different metrics might be generated from a single goal.

[7] This example is taken from Fenton [14]

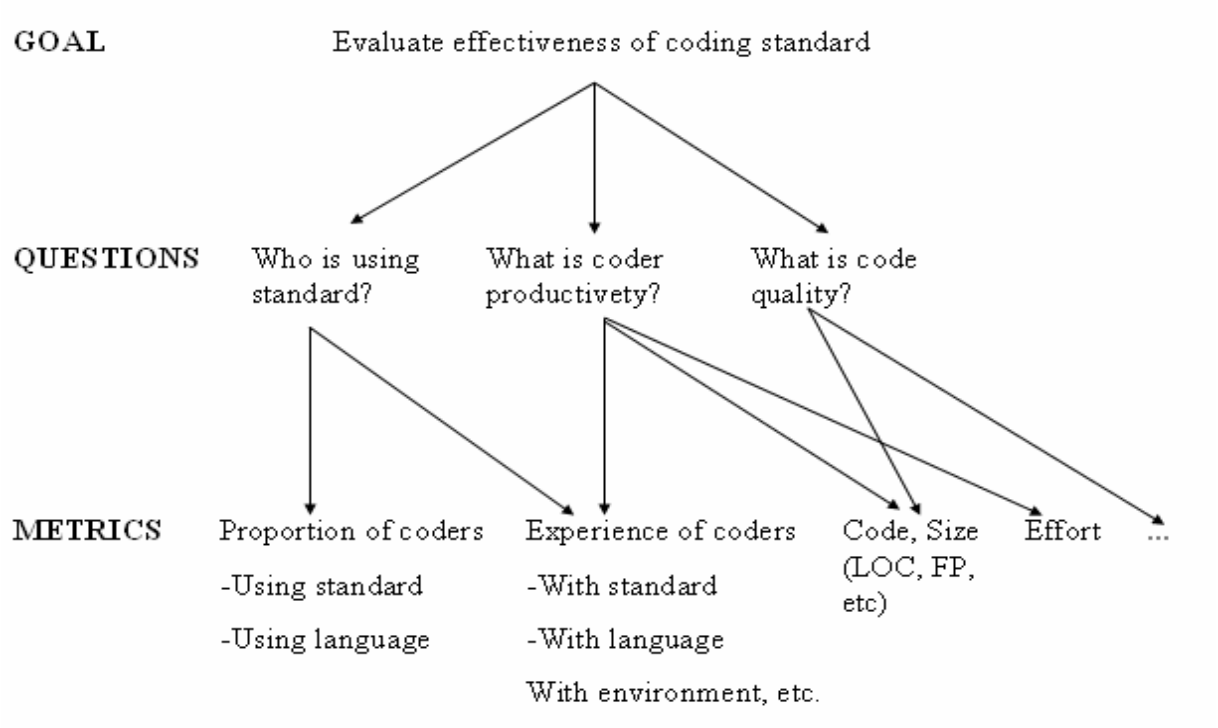


Figure 3: Example of deriving metrics from goal and questions (taken from [14]).

5 Metrics for OO Design

5.1 Introduction

A significant number of object oriented metrics have been developed in literature. For example, metrics proposed by Abreu [1], C.K metrics [12], Li and Henry [26] metircs, MOOD metrics [1b], Lorenz and Kidd [27] metrics etcs. C.K metrics are the most popular (used) among them. Another comprehensive set of metrics is MOOD metrics. This subsection will focus on traditional metrics and above mention metrics (mainly C.K and MOOD metrics).

5.2 Metrics Design Model

5.2.1 Traditional Metrics

In an object-oriented system, traditional metrics are generally applied to the methods that comprise the operations of a class. Methods reflect how a problem is broken into segments [36]. Traditional metrics have been applied for the measurement of software complexity of structured systems since 1976 [28]. The following discussion shows three popular traditional metrics.

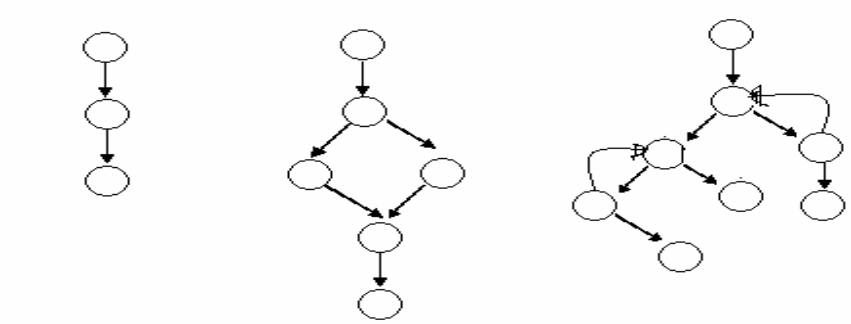
McCabe Cyclomatic Complexity (CC)

Complexity metrics can be used to calculate essential information about constancy and maintainability of software system from source code. It also provides advice during the software project to help control the design. In the testing and maintain phase, complexity metrics provide detail information about software module to identify the areas of possible instability.

Cyclomatic complexity (McCabe) can be used to evaluate the complexity of a method [36]. This metric measures the complexity of a the control flow graph[8] of a method or procedure. The idea is to draw the sequence a program may take as a graph with all possible paths. The complexity is calculated as "connections - nodes + 2" and will give a number denoting how complex the method is. See the following figure. Since complexity will increase the possibility of errors, a too high[9] McCabe number should be avoided [19].

[8] A graph is a representation of nodes and edges. When the edges are directed, the graph is said to be direct graph.

[9] Some standard require that no module should have a higher McCabe number than 10 [19]



N= 2-3+2 = 1 N= 6-6+2 = 2 N= 11-8+2 = 5

Figure 4: The McCabe complexity metrics (see [19])

As described in Laing et al. [23], McCabe et al. [28] mention cyclomatic complexity is a measure of a module control flow complexity based on graph theory. Cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class [36]. A high[10] cyclomatic complexity indicates that the code may be of low quality and difficult to test and maintain [23].

Source Lines of Code (SLOC)

SLOC is used to estimate the total effort that will be needed to develop a program, as well as to calculate approximate productivity. The SLOC metric measures the number of physical lines of active code, that is, no blank or commented lines code [27]. Logical SLOC measures the number of statements, but their specific definitions are fixed to specific language for example, in C programming language logical SLOC measure the terminating semicolon.

Since functionality is not as much interconnected with SLOC, expert developers may be capable to develop the same functionality with less code. So one program with less SLOC may show more functionalities than another similar program. Programs with larger SLOC values usually take more time to develop. Therefore, SLOC can be very effective in estimating effort. Thresholds for evaluating the SLOC measures vary depending on the coding language used and the complexity of the method [36].

[10] More than 10

Comment Percentage (CP)

The CP metric is defined as the number of commented lines of code divided by the number of non-blank lines of code [23]. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. The SATC[11] has found a comment percentage of about 30% is most effective [36].

5.2.2 C.K. Metrics Model

Chidamber and Kemerer define the so called CK metric suite [12]. This metric suite offers informative insight into whether developers are following object oriented principles (see section 2.2) in their design [26]. They claim that using several of their metrics collectively helps managers and designers to make better design decision. CK metrics have generated a significant amount of interest and are currently the most well known suite of measurements for OO software [14]. Chidamber and Kemerer proposed six metrics; the following discussion shows their metrics.

Weighted Method per Class (WMC)

WMC measures the complexity of a class. Complexity of a class can for example be calculated by the cyclomatic complexities (sec 5.2.1) of its methods. High value of WMC indicates the class is more complex than that of low values. So class with less WMC is better. As WMC is complexity measurement metric, we can get an idea of required effort to maintain a particular class.

Depth of Inheritance Tree (DIT)

DIT metric is the length of the maximum path from the node to the root of the tree. So this metric calculates how far down a class is declared in the inheritance hierarchy. The following figure shows the value of DIT for a simple class hierarchy. This metric also measures how many ancestor classes can potentially affect this class. DIT represents the complexity of the behaviour of a class, the complexity of design of a class and potential reuse.

[11] Software Assurance Technology Center

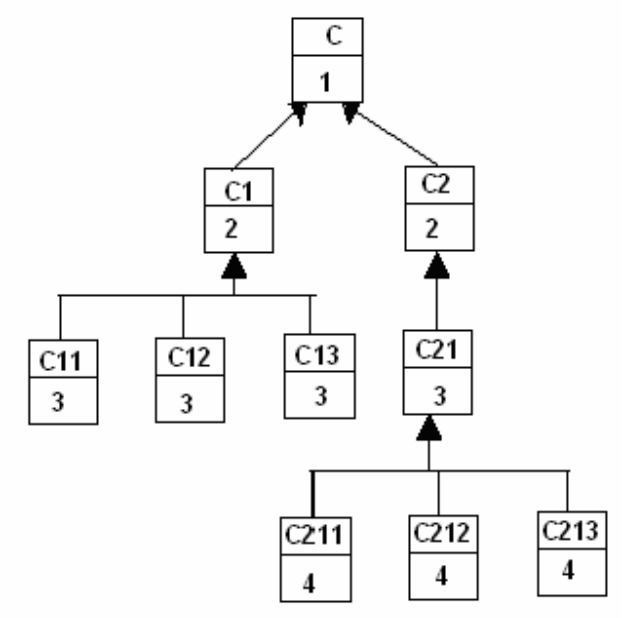


Figure 5: The value of DIT for the class hierarchy

If DIT increases, it means that more methods are to be expected to be inherited, which makes it more difficult to calculate a class's behavior. Thus it can be hard to understand a system with many inheritance layers. On the other hand, a large DIT value indicates that many methods might be reused.

Number of children (NOC)

This metric measures how many sub-classes are going to inherit the methods of the parent class. As shown in above figure, class C1 has three children, subclasses C11, C12, and C13. The size of NOC approximately indicates the level of reuse in an application . If NOC grows it means reuse increases. On the other hand, as NOC increases, the amount of testing will also increase because more children in a class indicate more responsibility. So, NOC represents the effort required to test the class and reuse.

Coupling between objects (CBO)

The idea of this metrics is that an object is coupled to another object if two object act upon each other. A class is coupled with another if the methods of one class use the methods or attributes of the other class. An increase of CBO indicates the reusability of a class will decrease. Thus, the CBO values for each class should be kept as low as possible. CBO metric measure the required effort to test the class [17].

Response for a Class (RFC)

RFC is the number of methods that can be invoked in response to a message in a class. Pressman [34] States, since RFC increases, the effort required for testing also increases because the test sequence grows. If RFC increases, the overall design complexity of the class increases and becomes hard to understand. On the other hand lower values indicate greater polymorphism. The value of RFC can be from 0 to 50 for a class[12], some cases the higher value can be 100- it depends on project to project.

Lack of Cohesion in Methods (LCOM)

This metric uses the notion of degree of similarity of methods. LCOM measures the amount of cohesiveness present, how well a system has been designed and how complex a class is [17]. LCOM is a count of the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero.

Raymond [31b] discussed for example, a class C with 3 methods M_1, M_2, and M_3. Let I_1= {a, b, c, d, e}, I_2= {a, b, e}, and I_3= {x, y, z}, where I_1 is the set of instance variables used by method M_1. So two disjoint set can be found: $I_1 \cap I_2$ (= {a, b, e}) and I_3. Here, one pair of methods who share at least one instance variable (I_1 and I_2). So LCOM = 2-1 =1. Riel[13] states "Most of the methods defined on a class should be using most of the data members most of the time". If LCOM is high, methods may be coupled to one another via attributes and then class design will be complex. So, designers should keep cohesion high, that is, keep LCOM low.

5.2.3 MOOD Metrics Model

Abreu et at.[1c] defined MOOD (Metrics for Object Oriented Design) metrics. MOOD refers to a basic structural mechanism of the object-oriented paradigm as encapsulation (MHF, AHF), inheritance (MIF, AIF), polymorphism (POF), and message passing (COF). Each metrics is expressed as a measure where the numerator represents the actual use of one of those feature for a given design [1d]. In MOOD metrics model, two main features are used in every metrics; they are methods and attributes. Methods are used to perform operations of several kinds such as obtaining or modifying the status of objects.

[12] RefactorIT tool suggest the value of RFC. See more detail in Metrics measurement tool at: http://www.refactorit.com/

[13] Arthur J. Riel: " Object oriented design heuristics", heuristics # 4.6, Addison-Wesley, 1996.

Attributes are used to represent the status of each object in the system. Each feature (methods and attributes) is either visible or hidden from a given class [1b, 1c, 1d].

We will discuss MOOD metrics in the context of encapsulation, inheritance, polymorphism, and coupling. These are discussed below.

Encapsulation

The Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) were proposed together as measure of encapsulation[14] [1b]. MHF and AHF represent the average amount of hiding between all classes in the system.

Method Hiding Factor (MHF)

The MHF metric states the sum of the invisibilities of all methods in all classes. The invisibility of a method is the percentage of the total class from which the method is hidden. Abreu et al. [1a] States, the MHF denominator is the total number of methods defined in the system under consideration. The MHF metric is defined as follows

$$MHF = \sum_{i=1}^{TC} M_h(C_i) / \sum_{i=1}^{TC} M_d(C_i)$$

Here, $M_d(C_i) = M_v(C_i) + M_h(C_i)$

$M_d(C_i)$ = the number of methods defined in class C_i

$M_v(C_i)$ = the number of methods that visible in the class C_i

$M_h(C_i)$ = the number of methods hidden in C_i

Where the summation occurs over i=1 to TC. TC is defined as total number of classes.

If the value of MHF is high (100%), it means all methods are private which indicates very little functionality. Thus it is not possible to reuse methods with high MHF. MHF with low (0%) value indicate all methods are public that means most of the methods are unprotected.

[14] Encapsulation is the process of hiding all the details of an object that do not contribute to its essential characteristics [9].

Attribute Hiding Factor (AHF)

The AHF metric shows the sum of the invisibilities of all attributes in all classes. The invisibility of an attribute is the percentage of the total classes from which this attribute is hidden. MHF and AHF represent the average amount of hiding among all classes in the system. The AHF metric is defined as follows.

$$AHF = \sum_{i=1}^{TC} A_h(C_i) / \sum_{i=1}^{TC} A_d(C_i)$$

Here, $A_d(C_i) = A_v(C_i) + A_h(C_i)$

$A_d(C_i)$ = the number of attributes defined in class Ci

$A_v(C_i)$ = the number of attributes that visible in the class Ci

$A_h(C_i)$ = the number of attributes hidden in Ci

Where the summation occurs over i=1 to TC. TC is defined as total number of classes.

If the value of AHF is high (100%), it means all attributes are private. AHF with low (0%) value indicate all attributes are public.

Inheritance

Inherited[15] features in a class are those which are inherited and not overridden in that class. Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) are proposed to measure inheritance.

Method Inheritance Factor (MIF)

The MIF metric states the sum of inherited methods in all classes of the system under consideration. The degree to which the class architecture of an object oriented system makes use of inheritance for both methods and attributes [34]. MIF is defined as the ratio of the sum of the inherited methods in all classes of the system as follow.

[15] Inheritance is the process by which objects of one class acquire the properties of the objects of another class.

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

Here, $M_a(C_i) = M_d(C_i) + M_i(C_i)$

$M_a(C_i)$ = the number of methods defined in class Ci

$M_d(C_i)$ = the number of methods declared in the class Ci

$M_i(C_i)$ = the number of methods inherited in Ci

Where the summation occurs over i=1 to TC. TC is defined as total number of classes.

If the value of MIF is low (0%), it means that there is no methods exists in the class as well as the class lacking an inheritance statement.

Attribute Inheritance Factor (AIF)

AIF is defined as the ratio of the sum of inherited attributes in all classes of the system. AIF denominator is the total number of available attributes for all classes. It is defined in an analogous manner and provides an indication of the impact of inheritance in the object oriented software [34]. AIF is defined as follows

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

Here, $A_a(C_i) = A_d(C_i) + A_i(C_i)$

$A_a(C_i)$ = the number of available attributes defined in class Ci

$A_d(C_i)$ = the number of attributes that declared in the class Ci

$A_i(C_i)$ = the number of inherited attributes in Ci

Where the summation occurs over i=1 to TC. TC is defined as total number of classes.

If the value of AIF is low (0%), it means that there is no attribute exists in the class as well as the class lacking an inheritance statement.

Polymorphism

Polymorphism[16] is an important characteristic in object oriented paradigm. Polymorphism measure the degree of overriding in the class inheritance tree.

Polymorphism Factor (POF)

The POF represents the actual number of possible different polymorphic situation. It also represents the maximum number of possible distinct polymorphic situation for class Ci. The POF is defined as follows.

$$POF = \sum_{i=1}^{TC} M_0(C_i) / \sum_{i=1}^{TC} \left[M_n(C_i) * DC(C_i) \right]$$

Here, $M_0(C_i) = M_n(C_i) + M_0(C_i)$

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

$M_0(C_i)$ = the number of overriding methods in the class Ci

$DC(C_i)$ = the descendants count in Ci

The numerator represents the actual number of possible different polymorphic situation and the denominator represents the maximum number of possible distinct polymorphic situation for class Ci [1d]. The value of POF can be varies between 0% and 100%. If a project have 0% POF, it indicates the project uses no polymorphism and 100% POF indicates that all methods are overridden in all derived classes[17].

Coupling

Coupling shows the relationship between module. A class is coupled to another class if it calls methods of another class.

Coupling Factor (COF)

The COF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of coupling is not imputable to inheritance [31b]. The COF is defined as follows.

$$COF = \sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is\_client(C_i, C_j) \right] / (TC^2 - TC^2)$$

[16] Polymorphism means the ability to take more than one form.

[17] These values can be found at "http://www.aivosto.com/project/help/pm-oo-mood.html"

Here $is\_client(C_i, C_j)$ = 1 if and only if, a relationship exists between the client class, Cc and the server class Cs and Cc not equal to Cs.

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

Where the summation occurs over i=1 to TC. TC is defined as total number of classes.

Pressman [34] argue that, although many factors affect software complexity, understandability, and maintainability. It is reasonable to conclude that as "the COF value" increases, the complexity of object oriented design will also increase, and as a result the understandability, maintainability, and the potential for reuse may suffer. The value of COF can be varies between 0% and 100%. 0%COF indicates no class are coupled and 100% COF indicates all class are coupled with all other classes. High values of COF should be avoided.

The idea in COF metric is as same idea used in CBO metrics because they both use coupling factor. The main difference between COF and CBO is, in COF metric all variable accesses are counted whereas CBO metric does not count variables[18].

5.2.4 Other Metrics Models

Several researchers propose object oriented metrics from different point of view. These metrics helps the designers to know which metrics are found at which level of decision. [7, 1, 11] proposes their metrics in different categories[19]. The following is very brief discussion[20] of them.

Lorenz and Kidd [27] proposed metrics are focused on size, inheritance, internal, and external measurements. Size metrics for the object oriented class focus on counts of attributes and operations for an individual class. Inheritance based metrics focus on the method in which operations are reused through the class hierarchy. Internal metrics are focus on cohesion and code oriented issue. External metrics observe coupling and reuse.

[18] "http://www.aivosto.com/project/help/pm-oo-mood.html"
[19] They categories their metrics in different context and different point of views.
[20] Most of the metrics I found in different source: articles, books, journals and internet. Since I don't have proper evidence, I discuss this section in very brief.

Belin et at. [7] categorized metrics in three groups. Group A consists of “number of methods” metric, “number of classes” metric, and “number of levels” metric in the class hierarchy tree. Group B focus on “code reuse” metric, “number of classes reused” metric, and “percent of reused” classes modified metric. Group C discusses coupling metric, cohesion metric, sufficiency metric, completeness metric and primitiveness metric. These metrics are deal with the quality of an abstraction in an OO system

Brito e Abrue et al. categorized metrics are: design, size, complexity, reuse, productivity, quality, method, class and system levels. They provide a catalogue for object oriented design metrics [1]. That taxonomy is based on a Cartesian product of the two vectors: (design, size, complexity, reuse, productivity, quality) and (method, class, system). His proposed metrics are CC2 (Class Complexity), CR1 (Class Reuse), CC3 (Class Complexity), CR2 (Class Reuse), CR3 (Class Reuse). In his measure, class and system quality metrics that the authors suggest are based on counts of observed defects, failures, and time between failures.

5.2.5 Other OO Metrics

Chen et al.[11] proposed metrics are 1.CCM (Class Coupling Metric), 2.OXM (Operating Complexity Metric), 3.OACM (Operating Argument Complexity Metric), 4.ACM (Attribute Complexity Metric), 5.OCM (Operating Coupling Metric), 6.CM (Cohesion Metric), 7.CHM (Class Hierarchy of Method) and 8.RM (Reuse Metric). Metrics 1 through 3 are subjective in nature; metrics 4 through 7 involve counts of features; and metric 8 is a boolean (0 or 1) indicator metric. To validate these metrics, the authors conduct an experiment involving six "experts" whose subjective class scores are regressed against the eight metrics. The resulting regression equation is used to score future object classes [2].

Li, Wei, Henry, Salley et al. state that metrics for the object-oriented paradigm have yet to be studied [26]. Since terminology varies among object oriented programming languages, the authors consider the basic components of the paradigm as objects, classes, attributes, inheritance, method, and message passing. They propose that each object-oriented basic concept implies a programming behaviour. They assembled metrics are: Data Abstraction Coupling (DAC), Number of methods (NOM), Message Passing

Coupling (MPC), and Number of semicolons per class (Size1), Number of methods per attributes (Size2). There is no individual breakdown of which of these metrics is significant in the prediction [2].

5.3 Similarity of OO Metrics

Object oriented metrics can be collected in different ways. Although different writers have described different metrics, according to object oriented design, there are some similarities found in different metrics model. The following table shows similar OO metrics. We have categorized metrics in class, attribute, method, cohesion, coupling, and inheritance category because most of the object oriented metrics are defined in above mention categories. In this table we will discuss only CK metrics suite, MOOD metrics model, and metrics defined by Chen & Lu, Li & Henry. Since other[21] metrics are defined from different context and different point of views, we have not considered those metrics in our table.

Category	**Class**	**Attribute**	**Method**	**Cohesion/ Coupling**	**Inheritance**
MOOD [1b]	MHF, AHF, MIF, AIF, POF, COF	AHF, AIF	MHF, MIF, POF		MIF, AIF
Chidamber & Kemerar [12]	WMC, RFC, LCOM	LCOM	WMC, RFC, LCOM	CBO	DIT, NOC
Chen & Lu [11]	OXM, RM, OACM			CCM, OCM	CHM
Li & Henry [26]	DAC, MPC, NOM,	Size2	MPC, NOM, Size1, Size2	MPC	

Table 2: Similar object oriented metrics

[21] Other metrics indicates section 5.2.4 which are categorized from different context and different point of views.

We categorized these metrics (shown above table) to find out the right metrics to measure class, methods, attributes, etc. Since we did not collect all proposed metrics and we did not categorize all of them, this table is not complete to give clear recommendation as to which metrics should be used. But these categorizations focus on common metrics which will be helpful for novice designers to support their design measurements. The following is a brief discussion of that categorization.

Class

The class is the fundamental unit of object oriented design. Therefore the metrics are used to measure a class to access design quality. For example, MPC (Message Passing Coupling) measures the complexity of message passing among classes. Although messages are passed between objects, the types of messages passed are defined in classes. So that, message passing is calculated at the class level instead of the object level. WMC (Weighted Methods per Class) discuss the complexity of the methods. In general methods are small enough so that the complexity of each could be considered as equal to unity. RFC (Response For Class) metrics states the response set of a class consists of the set M of methods of the class, and the set of methods invoked directly by the methods in M. LCOM (Lack of Cohesion in Methods) measures the number of pairs of methods in the class that have no attributes in common i.e. similarity is zero, minus the number of pairs of methods whose similarity is not zero. If the difference is negative, the metrics value is set to zero.

Attribute

Attributes define the properties of data object and take an instance of the data object, describe the instance as well as make reference to another instance in another table. For example, the AHF metric is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes for all classes[22]. LCOM metric counts the sets of methods that are not related through the sharing of some of the class's instance variables.

Method

A message is a request that an object makes of another object to perform an operation. The operation executed as a result of receiving a message is called a method. For example, WMC metric is the sum of the complexities of all class methods. It calculates

[22] Software Measurement Page: http://yunus.hun.edu.tr/~sencer/oom.html

all declared methods and constructors of class. The RFC metric uses a number of methods to review a combination of a class's complexity and the amount of communication with other classes. The LCOM metrics uses data input variables or attributes to measure the degree of similarity between methods. The MPC metric define a class which sends a number of statements. This send statement is a message sent out from a method in a class to a method in another class. Size1 is defined as the number of noncommand lines of source code and Size2 defined as the total count of the number of data attributes and the number of external local methods in a class[23].

Coupling/Cohesion

The most potential outcome with object oriented metrics is obtained using coupling metrics. In the context of design metrics, coupling and cohesion are used to measure a systems structural complexity. These are also used to asses design. A class is coupled with one more classes if the methods of one class use the methods or attributes of the other classes. CK metrics suite includes measures for coupling and cohesion, the suite provide descriptive power for administrative concern. Mainly high level of coupling and low level of cohesion were associated with problems and maintainability. For example, CBO (Coupling between Object Classes) is the number of other class with which a class is coupled. CCM (Class Coupling Metrics) measures the coupling between class and other class; MPC (Message Passing Coupling) measures the complexity of message passing between classes as well as objects. Although messages are passed among objects, the types of messages passed are defined is class.

A class is cohesive when its parts are highly correlated. It should be difficult to split a cohesive class. Cohesion can be used to identify the poorly designed classes. High functional cohesion as existing when the elements of a component all work together to provide some well-bounded behavior [9]. High cohesion indicates good class subdivision. Low cohesion increases complexity, thereby increasing the likelihood of errors during development. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion [36].

Inheritance

Inheritance shows the relationship among classes and reuse earlier defined objects as well as variables and operators. Inheritance decreases complexity by reducing the number of

[23] IEEE transaction on Software Engineering, Jan 2005, Vol 31, Number 1

operations and operators. There are some metrics used to measure the amount of inheritance. For example NOC metric measures the number of direct subclasses of a class. The size of NOC approximately indicates how an application reuses itself. DIT metric calculates how far behind a class is declared in the inheritance hierarchy. MIF and AIF allows expressing similarity between classes; the portrayal of generalization and specialization relations; and simplification of the definition of inheriting classes, by means of reuse [1c].

6 Evaluation of OO Metrics

Metrics have a number of interesting characteristics for providing development support. Some of them are simple, precise, general and scalable to large size software systems [30]. Abreu et al. state a set of metrics (see section 5.2.3) for evaluating the use of the mechanism that support the main concepts of the object-oriented paradigm and the consequent emphasis on reuse, that are believed to be responsible for the increasing in software quality and development productivity [1a].

In this report, we analyzed some metrics by using RefactorIT (See Appendix 9.1) tool. In our analysis we use two java packages to measure object oriented metrics. Package1 contains 25 classes, 103 methods and the total line of code (LOC) is 1023. Package2 contains 20 classes, 134 methods, and LOC is 1729. The main reason to choose those packages is both of the packages have less then 2000 line of code and it is faster to execute[24]. In this paper, we focused mainly LOC, WMC, RFC, DIT, NOC and DIP metrics, because this tool support those metrics

Table 1 and table 2 represent package1 and package2 metrics respectively. In this analysis we analyze WMC, RFC and DIT metrics elaborately, NOC and DIP metrics discuss briefly.

Report created at May 14, 2005 12:20:44 AM

Type	LOC	WMC	RFC	DIT	NOC	DIP
Class 1	41	7	22	3	1	0
Class 2	166	22	61	2	1	0
Class 3	12	2	6	3	0	0
Class 4	27	7	6	1	0	
Class 5	295	25	58	4	0	0.2
Class 6	18	4	9	4	0	0.25
Class 7	3	1	0	1	0	0
Class 8	7	5	3	1	0	0
Class 9	7	6	4	1	0	0
Class 10	11	5	6	4	0	0.2

[24] We tried to analyze other packages which have more them 2000 line of code by this tool and it makes hang the system.

Class 11	17	4	12	2	0	0
Class12	35	6	13	3	0	0
Class 13	52	12	16	3	0	0
Class 14	6	3	3	1	0	0
Class 15	32	7	5	3	0	0
Class 16	19	8	5	3	0	0
Class 17	101	10	23	3	0	0
Class 18	54	30	19	1	0	
Class 19	81	22	31	4	0	0.333
Class 20	18	2	7	1	0	
Class 21	5	1	1	3	0	0
Class 22	68	10	20	3	0	0
Class 23	79	10	23	4	0	0.5
Class 24	17	4	2	1	0	0
Class 25	5	2	3	3	0	0

Table 3: Package1 detail information

Report created at May 15, 2005 10:15:17 AM

Type	**LOC**	**WMC**	**RFC**	**DIT**	**NOC**	**DIP**
Class 1	25	1	2	1	0	0
Class 2	86	16	3	1	0	
Class 3	69	7	4	2	0	0
Class 4	90	14	21	1	0	0.5
Class 5	108	10	26	3	0	0.167
Class 6	4	1	1	2	0	0
Class 7	73	9	25	1	0	0
Class 8	175	31	60	1	0	0.063
Class 9	32	4	11	1	0	0.286
Class 10	187	21	43	1	1	0.333
Class 11	40	3	12	2	0	0.273
Class 12	9	2	2	2	0	0.4
Class 13	51	7	10	2	0	0.143
Class 14	25	3	5	1	0	0.333

Class 15	174	15	43	3	0	0.067
Class 16	13	2	6	2	0	0
Class 17	163	28	48	1	0	0.333
Class 18	26	5	3	2	0	0
Class 19	137	21	38	1	0	0
Class 20	38	3	8	3	0	0.167

Table 4: Package2 detail information

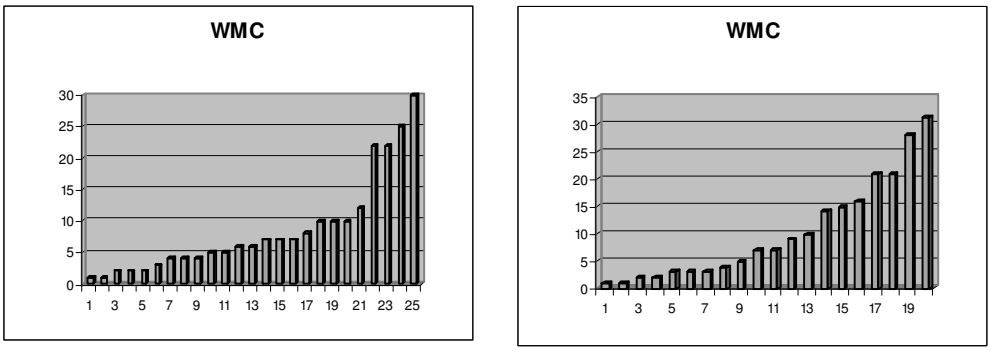


Figure 6: Graphical representation of WMC from table 1 and table2

By the WMC metric we can observe cyclomatic complexity of methods of a class. Since WMC metric can be found by the sum of complexity of all method. In our analysis we found in package1, 20 classes WMC is 10 and only 1 class have WMC is 30. This result indicates that most of the classes have more polymorphism and less complexity. In package2, 13 classes have WMC 10 out of 20 classes; only 2 classes have WMC is more then 25. Since a class consists at least one function, so the lower limit of WMC is 1 and higher limit of WMC is 50[25]. Low WMC indicates greater polymorphism in a class and high WMC indicates more complexity in the class. The WMC figures look quite similar for both packages.

[25] RefactorIT tool suggests the value of WMC. See more detail in Metrics measurement tool at: http://www.refactorit.com/

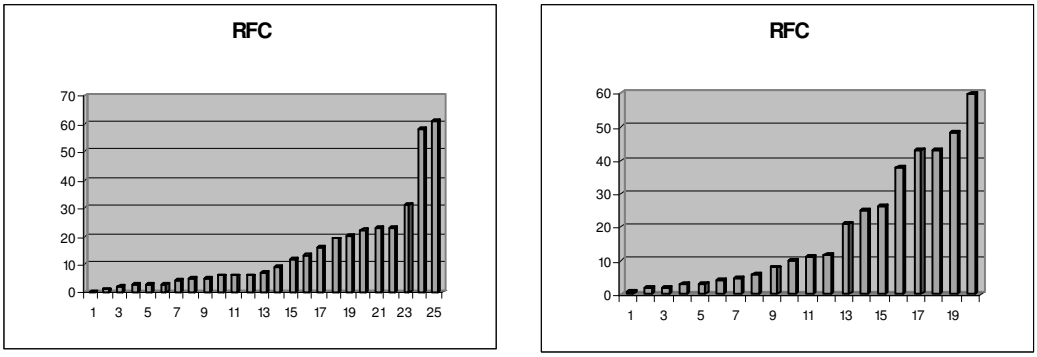


Figure 7: Graphical representation of RFC from table 1 and table 2

In RefactorIT tool, RFC metric measured by the set of all methods and constructors that can be invoked as a result of a message sent to an object of the class[26] . RefactorIT tool suggests the range of RFC should be 0 to 50. A class with large RFC indicates the class is more complex and it's harder to maintain. In our analysis, package1 have 25 classes, 20 of them have RFC threshold is around 20. Only two classes contain RFC threshold more then 50. This result indicates that only 2 classes have to be modified to reduce complexity. In package2, 13 classes have RFC with threshold 20 and other 7 classes have RFC with threshold more then 30.

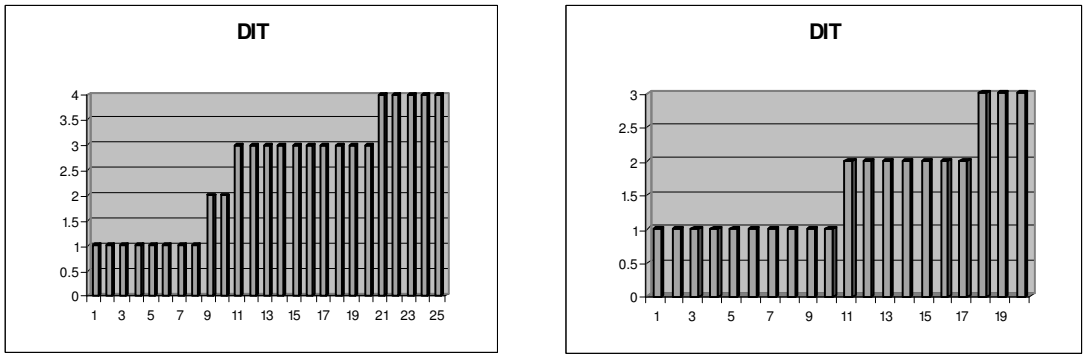


Figure 8: Graphical representation of DIT from table 1 and table2

[26] RefactorIT manual: http://www.refactorit.com/

DIT metrics is the length of the maximum path from the node to the root of the tree. A DIT value of 0 indicates a root. Since deeper trees constitute greater design complexity as more methods and classes are involved, so maximum DIT value of 5. DIT value of 2 and 3 indicates a higher degree of reuse. If there is a majority of DIT values bellow 2, it may represent poor exploitation of the advantages of OO design and inheritance [RefactorIT]. In our analysis, packages 1 have 25 classes, 13 of them have DIT value is 2 to 3 and 8 classes have DIT value is 1 and rest of DIT value is 4. This result indicates, classes of package 1 are a higher degree of reuse and fewer complexes. In package 2, 50% classes have DIT value is 2 and 50% classes have DIT value is 1.

NOC metric measures the number of direct subclass of a class. Since more children in a class have more responsibility, thus it is harder to modify the class and requires more testing. So NOC with less value is better and more NOC may indicate a misuse of subclassing. In our analysis, both package1 and package2 have 0 and 1 NOC respectively.

Now we will discuss the measurements of object oriented principles. We already discussed DIP in section 2.2. DIP principles are mainly used for avoid developing software which have bad symptom (see section 2.4 for more detail). The DIP metric measure the ratio of dependencies that have abstract classes. In our analysis, most of the class from package1 indicates 0.0 DIP whereas most of the classes from packag2 indicate 0.067 to 0.50 DIP. This result shows the classes from package1 are more depend on abstract classes than package2's classes.

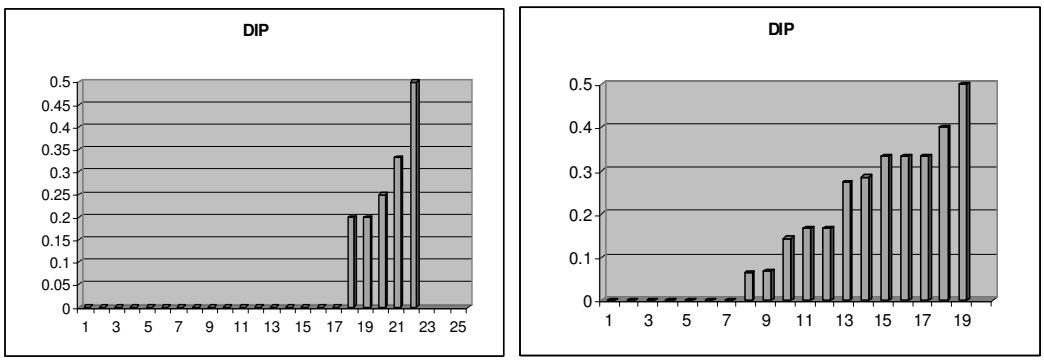


Figure 9: Graphical representations DIP from table 1 and table 2.

Cyclic dependency principle is used for quality measurements. Cyclic dependencies calculate how many cycle involved in a package. According to ADP principle (see section 2.2.3) designers should avoid package dependencies graph[27]. It will be very difficult to maintain the design if one packages involved in many cycle, because a single change in one package of a cycle may affect the other packages of a same cycle. In our analysis there was no cycle among package1 and package2.

Software Assurance Technology Center (SATC) analysis C.K. metrics [23]. In their analysis they use three applications to validate the reduced object-oriented metrics. They discuss the applications as: System A, System B, and System C. System A was the commercial software implemented in Java and consisted of approximately 50,000 lines of code and had 46 classes. System B was implemented by NASA software which applications was also implemented in Java and consisted of approximately 300,000 lines of code and contained 1,000 classes. The last application, System C, was also a NASA product implemented in the C++ programming language and approximately consisting of 500,000 lines of code distributed over 1,617 classes.

The distributions for all the metrics are similar between systems except for the DIT metric. The DIT metric for Systems B and C are similar. However, System A exhibits a different distribution from both Systems B and C. The distribution for System A shows that over 60% of the classes in that system had a DIT metric of 0, suggesting a lack of reuse via inheritance [35]. The reduced metrics set approach was able to classify the software systems with respect to the level of code quality. Both the reduced metrics set approach and the full metrics set (CK metrics suite and traditional) approach resulted in the same software quality system classification. System A was low quality software, System B was high, and System C was medium [36]. From this analysis we can get, system A should be modified to get high quality software.

[27] A graph is representation of nodes and link.

7 Summary

Several measures have been defined so far in order to estimate object oriented design. Coupling and cohesion are used to measure a system's structural complexity, and can be used to assess design quality and to guide improvement efforts. The application of object oriented design principles for example modularity, abstraction lead to better maintainability and reusability. Designers always have to be considering the bad symptoms of designs. Because, design with bad symptom needs more measurements.

Measurement can help to improve the software process, assist in the tracking and control of a project and asses the quality of a product. By analyzing metrics, a developer can correct those areas of software process that are the cause of software defects. The GQM idea is a useful approach for deciding what to measure. It creates a hierarchy of goals; questions that should be answered in order to know if the goal satisfy; and metrics that must be made in order to answer the question. Thus, the GQM approach provides guidelines for find out metrics.

A wide variety of object oriented metrics have been proposed to assess the testability of an object oriented system. Most of the metrics focus on encapsulation, inheritance, class complexity and polymorphism. CK metrics suite is a set of six metrics which capture different aspects of an OO design; these metrics mainly focus on the class and the class hierarchy. It includes complexity, coupling and cohesion as well. On the other hand MOOD metrics focus on system level which includes encapsulation, inheritance, polymorphism, and massage passing.

Many metrics have been adapted from CK metrics suite. In this literature we discussed CK metrics elaborately and we also analysed some of the CK metrics. In our analysis we found some result which are similar to the result of SATC's [23] analysis. Basili et al. [5] presented the results of an empirical validation of CK's metrics. Their results suggest that five of the six of CK's metrics (WMC, DIT, RFC, NOC, and CBO) are useful quality indicators for predicting fault-prone classes. We discussed some similar metrics for example class, attributes, cohesion, coupling, etc categories. These categories will assist to find out for a particular metrics.

Since very few object oriented metrics are empirically validated to measure object oriented design and this report is not complete for suggesting which metrics should be used. This report suggest that, only those metrics should be used which are empirically validated. This study also advice to metrics developers that, metrics should be simple, computable and programming language independent. There will be always something new to measure and metrics developers have to make new metrics to satisfy them.

8 References

[1a] Abreu, Fernando B. ,Carapuca, Rogerio.: “Candidate Metrics for Object-Oriented Software within a Taxonomy Framework.", Journal of systems software 26, 1(July 1994)

[1b] Abreu, Fernando B: "The MOOD Metrics Set," *Proc. ECOOP'95 Workshop on Metrics,* 1995.

[1c] Abreu, Fernando B: “Design metrics for OO software system”, ECOOP’95, Quantitative Methods Workshop, 1995

[1d] Abreu, Fernando B, Rita, E., Miguel, G. : “The Design of Eiffel Program: Quantitative Evaluation Using the MOOD metrics”, Proceeding of TOOLS’96 USA, Santa Barbara, California, July 1996

[2] Archer C.,Stinson M.: “Objece Oriented Software Measure”, Technical report CMU/SEI-95-TR-002, ESC-TR-95-002, 1995

[3] Balasubramanian NV.: “Object oriented metrics”, Proceedings 3rd Asia-Pacific Software Engineering Conference (APSEC’96). IEEE Computer Society, 1996; 30-34.

[4] Banker, Rajiv D., Kauffman, Robert J.,Kumar, Rachina.: "An Empirical Test of Object-based Output Measurement Metrics in a CASE Environment." Journal of Management Information Systems 8,3 (Winter 1991): 127-150.

[5] Basili VR, Briand LC, Melo WL.: “A validation of object-oriented design metrics as quality indicators”, Technical Report, University of Maryland, Department of Computer Science, 1995; 1-24

[6] Basili, V.R., Gianluigi Caldiera, H. Dieter Rombach: “THE GOAL QUESTION METRIC APPROACH.” http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/gqm.pdf

[7] Bellin, D., Manish Tyagi, Maurice Tyler: "Object-Oriented Metrics:An Overview", Computer Science Department,North Carolina A ,T state University,Greensboro,Nc 27411-0002.

[8] Briand LC, Morasoa S.: “Defining and validating measures for object-based high level design”, IEEE Transactions on Software Engineering 1999; 25(5): 722-743.

[9] Booch, G: “Object-Oriented Analysis and Design with Applications”, 2nd ed., Benjamin Cummings, 1994

[10] Chapin N., Hale J., Khan K., Ramil J.: “Type of software evolution and

software maintenance". Journal of software maintenance and evolution,2001

[11] Chen, J-Y., Lum, J-F.: "A New Metrics for Object-Oriented Design." Information of Software Technology 35,4(April 1993):232-240.

[12] Chidamber, Shyam , Kemerer, Chris F. "A Metrics Suite for Object-Oriented Design." M.I.T. Sloan School of Management E53-315, 1993

[13] Demeyer, S., Ducasse, S. and Nierstrasz, O: "Refactorings via change metrics". In Proc. Int. Conf. 2000. ACM Press,

[14] Fenton, N., S.L. Pfleeger: "Software Metrics: A Rigorous and Practical Approach", PWS Publishing Co.

[15] Frederick T. Sheldon, Kshmata Jerath, and Hong Chung: "Metrices for maintainability of class inheritance hierarchies", Journal of software maintenance and evaluation: Reachers and practice, 2002

[16] Harrison R., Steve J.: "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 24, NO. 6, JUNE 1998

[17] Harrison, R., Counsell, S.J. Nithi, R.V: "An Investigation into the Applicability and Validity of Object-Oriented Design Metrics", technical report

[18] Harrison, R., Samaraweera, L.G., Dobie, M.R., and Lewis, P.H: "Comparing Programming Paradigms: An Evaluation of Functional and Object-Oriented Programs," *Software Eng. J.*, vol. 11, pp. 247-254, July 1996.

[19] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard G.: "Object-Oriented Software Engineering: A Use-Case Driven Approach", Addison-Wesley, 1992

[20] Jon Avotins: "Defining and Designing a Quality OO Metrics Suite", Depertment of Software Development,Monash University,Australia 3145

[21] Khaled El Emam,: "A Primer on OO Measurement", 1530-1435/05 IEEE, Proceeding of the Seventh International Software Metrics Symposium (METRICS'01)

[22] Kitchenham B, Pfleeger SL, Fenton NE.: "Towards a framework for software measurement validation". IEEE Trans. On Software Engineering 1995; 21(12): 929-944

[23] Laing V.,Coleman C., : "Principal Components of Orthogonal OO Metrics", Software Assurance Technology Center (SATC),2001

[24] Lee, Y., Liang, B., Wang, F.: "Some Complexity Metrics for Object Oriented Programs Based on Information Flow", Proceedings: CompEuro, March, 1993, pp. 302-310

[25] Li W.: "Another metric suite for object-oriented programming". The Journal of Systems and Software 1998; 44(2):155-162

[26] Li,Wei , Henry, Salley.: "Maintenance Metrics for the Object Oriented Paradigm", First International Software Metrics Symposium. Baltimore,Maryland, May 21-22, 1993. Los Alamitos, California: IEEE Computer Society Press, 1993.

[27] Lorenz, Mark & Kidd Jeff: "Object-Oriented Software Metrics", Prentice Hall, 1994.

[28] McCabe and Associates,: "Using McCabe" QA *7.0*, 1999, 9861 Broken Land Parkway 4th Floor Columbia, MD 21046

[29] McCall, J., P., Richards, and G., Walters: "Factors in software quality", NTIS AD-A049-014,015,055, 1977

[30] Mens T., Demeyer S.: "Future Trends in software evolution metrics" ACM 2002 1-58113-508-4

[31a] Ramil, J.E and Lehman, M.M: "Metrics of evolution as effort predictors - a case study". In Conf. Software Maintenance, pages 163-172, October

[31b] Raymond, J. A, Alex, D.L: "Adata model for object oriented design metrics", Technical Report 1997, ISBN 0836 0227.

[32] Robert C. Martin , www.objectmentor.com

[33] Robert C. Martin:" Agile Software Development":Principles,Patterns and Practices,2002.

[34] Roger S. Pressman: "Software Engineering", Fifth edition, ISBN 0077096770

[35] Rosenberg, H Linda: "Applying and Interpreting Object Oriented Metrics" Software Assurance Technology Office (SATO)

[36] Rosenberg, H. Linda, Lawrence E. Hyatt: "Software Quality Metrics for Object-Oriented Environments", Crosstalk Jounal,1997

[37] Scotto M., Sillitti A., Succi G., Vernazza T.: "A relational approach to software metrics", 2004 ACM Symposium on Applied Computing.

[38] Steven C., Doug Lea.: "Process and Metrics for Object-Oriented Software Development". OOPSLA 1993.

[39] Tang MH,Kao MH, Chen MH.: "An empirical study on object-oriented metrics". Proceedings 23rd Annual International Computer Software and

Application Conference. IEEE Computer Society, 1999;

[40] Vessey, I. and Weber R.: "Research on Structured Programming: An Empiricist's Evaluation". IEEE Transaction on Software Engineering 10,4, 394-407

9 Appendix

9.1 RefactorIT Tool

Version: 2.5.0.7
Date: April 18, 2005
Company: Aqris Software AS, Ravala puiestee 5, 10143 Tallinn, ESTONIA
Web: http://www.refactorit.com

RefactorIT Evaluation license - a fully functional version of RefactorIT for 30 day trial period, without charge.RefactorIT makes possible to efficiency analyzing information from source code. It supports java and C language.

9.2 Metrics Collection

(We found these metrics from different sources. In this study, most of them are discussed very briefly)

ACM - Attribute Complexity Metric
AHF- Attribute Hiding Factor
AIF - Attribute Inheritance Factor
CBC - Count of Base Classes
CBO - Coupling Between Object classes
CC - Class Complexity
CC2 - Class Complexity (progeny count)

CC3 - Class Complexity (parent count)
CCM - Class Coupling Metric
CCR - Count of number of Contains Relationships
CHM - Class Hierarchy of Method
CM - Cohesion Metric
COF - Coupling Factor
COU - Count Of Uses
CR1 - Class Reuse (% of inherited methods that are overloaded)
CR2 - Class Reuse (number of times class is reused "as is")
CR3 - Class Reuse (number of times class is reused with adaptation)
CSC - Count of Standalone Classes
DAC - Data Abstraction Coupling (Number of abstract data types)
DIT - Depth of Inheritance Tree
GSDM - Graph of Source and Destination of Messages (no measure given)
HC - Hierarchy Complexity of system
IL - Inheritance Lattice (stated, but no measure indicated)
LCOM - Lack of Cohesion Of Methods
LOC - Lines Of Code
MC - Method Complexity
MCC - McCabe's Cyclomatic Complexity metric
MHF- Method Hiding Factor
MIF - Method Inheritance Factor
MPC - Message Passing Coupling (number of send statements in a class)
NOC - Number Of Children
NOM - Number Of local Methods
NOT - Number of Tramps (count of extraneous parameters)
OACM - Operation Argument Complexity Metric
OC - Object Counts (count of classes)
OCM - Operation Coupling Metric
OP - Object Points
OXM - Operation Complexity Metric (within a class)
PC - Program Complexity
POF - Polymorphism Factor
RFC - Raw Function Counts
RFC - Response For a Class
RFC = |RS| , where RS = response set for the class.

RL - Reuse Leverage
RM - Reuse Metric (of classes)
SC1 - System Complexity (total length of inheritance chain)
Size - Size of Object-Oriented system
Size1 - number of semi-colons in a class
Size2 - number of attributes + number of local methods
SR1 - System Reuse (% reused "as is" classes)
SR2 - System Reuse (% reused classes with adaptation)
SR3 - System Reuse (library quality factor)
SSM - Software Science Metrics (Halstead)
SSM - Software Science Metrics (Halstead)
SSM - Software Science Metrics (Halstead)
VOD - Violations Of the law of Demeter
WAC - Weighted Attributes per Class
WMC - Weighted Methods per Class