Software Reuse Metrics: Measuring Component Independence and its applicability in Software Reuse

DR. P. K. SURI[1] , NEERAJ GARG[2]
1. Professor, Department of Computer Science & Applications,
Kurukshetra university, Kurukshetra
INDIA

2. Associate Professor, Department of IT Engineering,
M. M. University, Mullana, Ambala
INDIA

Summary
In this paper we have enumerated the various metrics of software to evaluate the reusability of the modules. We now introduce a new metric for evaluating the independence of a software component which will in turn accesses the degree of reusability of that component. The more independent the component is the more it is reusable.

Keywords
Software metrics-coupling-component-independence-software reusability

Introduction

THE aim of Object Oriented (OO) Metrics is to predict the quality of the object oriented software products. Various attributes, which determine the quality of the software, include maintainability, defect density, normalized rework, understandability, reusability etc. The requirement nowadays is to explore the relation of the reusability attributes with the metrics and to find how these metrics collectively determine the reusability of the software component. To achieve both the quality and productivity objectives, it is always recommended to go for the software reuse that not only saves the time taken to develop the product from scratch but also delivers the almost error free code, as the code is already tested many times during its earlier reuse.

Metrics have been developed in software engineering to quantitatively measure these factors and such metrics have been used to assess software modules for reusability. In this research, the focus is whether or not coupling affects database module reuse.

In this paper we have enumerated the various metrics of software to evaluate the reusability of the modules. We had introduced a new metric for evaluating the independence of a software component which will in turn accesses the degree of reusability of that component. The more independent the component is the more it is reusable.

Manuscript received May 5, 2009
Manuscript revised May 20, 2009

Software Reusability & its Measurement

Software reuse is the use of existing software components to construct new systems . Reuse is the application of existing solutions to new problems. Reuse can reduce the time spent in creating solutions by avoiding duplicated efforts. In software engineering the concept of reuse has been explored and has been reported to be very beneficial. Frakes, for example, notes that "using reusable software generally results in higher overall productivity" [1]. The benefits are not only realized in productivity but also in quality; software developed using existing components can be more reliable than those developed from scratch because the reused components are usually well tested and have been used in several developments. However, the reusable components must exist before they can be reused. Reusing existing parts or components is a standard part of software engineering and human problem solving in general. However, reuse in software development is more effective if practice formally [2]. Formal reuse implies that reuse must be viewed as a goal to strive for, not just a result that happens by chance. Before reuse can take place, the reusable components must exist in some form, and designers must be aware of their existence and the functionality they provide.
If formal reuse is part of an organization's overall development goals, then the software construction process is different; not only are developers tasked to find and use existing artifacts, they also have to assure that the final product can also be reused in future development.

Characteristics of Reusability

The reusability assets are different in different contexts. However, there are some characteristics that generally contribute to the reusability of assets. Although many of these characteristics apply to assets in general, we focus in this section on components as assets. [31].

Reusability = Usability + Usefulness

Usability is the degree to which an asset is 'easy' to use in the sense of the amount of effort that is needed to use an asset. Usability as such is independent of functionality of the component. Sub characteristics of usability are shown in

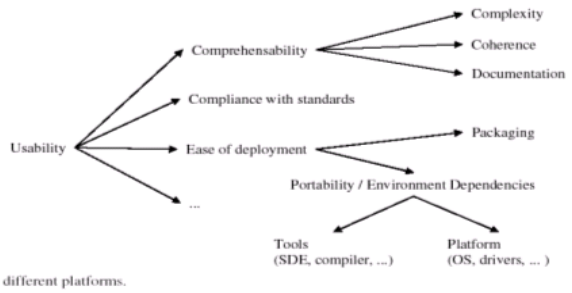

Fig1: Characteristics of usability

Reusable Assets

Set of artifacts that can be considered reusable asset are Requirements, Architectures, Design, Implementation, Program code and Data. The use of commodity software such as operating systems or database system is typically not considered reuse. As a rule of thumb, if a component is not considered as part of the design of a system, it is not considered as being reuse.

Factors Affecting Reusability

Figure 2 shows a "fishbone diagram" that represents the factors affecting reusability. It can be observed from this figure that reusability depends on *Usefulness*, *Costs* and *Quality*. Each of these factors is explained below.

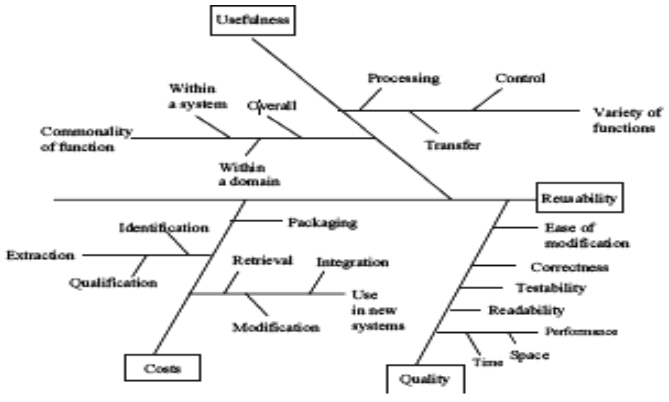

Figure 2: Factors affecting reusability [27]

Usefulness

To be reused, a prerequisite is that the component implements functionality that is useful for the new system. It is extremely hard to decide in an automated way whether or not a component will be useful in a new system, since this decision is based on domain knowledge and the requirements of the new system. However, an indirect automatable measure of usefulness was developed to measure the reusability of the existing component within the analyzed system itself (i.e., its origin). The assumption is that the highly used components within a system are a good candidate for reuse in a new context. There is also a limitation because of our assumption: We tend to exclude those domain specific components that are not frequently used in the existing system. It is important to note that the domain expert is crucial to decide about the usefulness of a component candidate.

Cost

Reuse cost includes cost of identifying a component from the existing system, modifying and integrating them into a new system. Measures of size and complexity of a component provide a partial indication of difficulty in adapting it to reuse in a new system. The cost to reuse the component is influenced by the readability of its code, a characteristics that can again be partially evaluated using size and complexity measures. That is, small and simple code fragments are usually easier to read and adapt than larger and complex fragments.

Quality

The quality of the component is important in order to succeed in reuse-driven development. Several qualities that are important for component reuse are correctness, readability, testability, ease of modification, and performance, but most of them are not directly measurable. Measures of size and complexity of a component however provide a partial indication of the presence of these qualities in it.

The Factor, Criteria, Measurement (FCM) Model

In software engineering, several measures have been used to evaluate software quality. At minimum, for a component to be considered for reuse, it must be of good quality. Measuring quality quantitatively is not a simple task. As stated by Fenton et al., "quality is multi-dimensional; it does not reflect a single aspect of a particular product" [3]. Many software metrics text and papers [3,4] give models for measuring software quality. One of these models, proposed by Fenton and colleagues [2], define factor, criteria, and metric (FCM) for each measurement. FCM is a tree- like structure where the top level lists the factors—items that are known to be the major indicators in the evaluation of the attribute in question. For instance, in evaluating quality, one may look at usability, testability, and portability as factors giving indication of the quality of a product. The second level in FCM consists of a list of criteria for each factor. These lower level items are easy to understand and

measure. The last level comprises of the actual metrics that define the specific measurements for each criterion. For instance the criteria comment ratio may be defined as criteria for evaluating understandability.

Metrics

Metrics may play an important role in quality assurance, specially in the acquisition of components and in deciding whether they should be used or not. Metrics should provide a basis for deciding whether reuse is sensible , whether it is cost effective to adapt existing component or build a component from scratch. In short, metric which address cost savings on component basis are needed. Metrics can seen as part of the topics acquisition and usage.

Software Reuse Impacts

Empirical studies, in both industry and academia, with the aim of assessing the relationship between software reuse and different quality and cost metrics have been reported in the literature [5,6,7,8]. All of the reported studies dealt with a very limited number of projects, which made their results inconclusive, but the general notion that software reuse and software quality are intrinsically related held true for all cases, while the inverse relation between software reuse and development cost failed to hold for some of the studies. Table 1 summarizes the measurable impacts of software reuse.

Table 1. Measurable impacts of software reuse.

Aspect	Measurable Impacts
Quality	• **Error Density** • **Fault Density** • **Ratio of Major errors to total faults** • **Rework effort** • **Module deltas** • **Developers perception**
Productivity	• **Lines of code per effort**
Time –to – Market	• **Development cycle time**

Error density is the average number of severe errors a piece of software presents per line of code, while **fault density** accounts for less severe errors. The studies show that projects with higher reuse activity tend to have lower error density. The reason is that a reused piece of software has been tested and debugged in previous systems, thus leading to fewer errors. Besides having fewer errors, the **ratio between major errors and total number of faults** tends to be smaller for projects that reuse more software. As direct consequences, the overall **rework effort** and the number of **module deltas** tend to be smaller. Since there are fewer errors, less effort must be spent fixing errors and fewer changes (deltas) will be necessary. The **software quality as perceived by developers** is a subjective measure based on the experience of the developers during the development process. Developers fill out forms describing their impressions of the quality of the software built and the difficulties they had to deal with and the results are compared between projects that considered reuse and projects that did not consider reuse during the entire development cycle. Although there is no definitive conclusion about the actual impacts software reuse has on different aspects such as quality and cost, studies have shown that there is a correspondence between them.

Software Structure Oriented Metrics

The whole point of software reuse is achieving the same or better results at the same or smaller cost when compared to a non-reuse oriented software development approach. From this perspective, the previous sections on economically oriented metrics and software reuse impacts would be enough for the reuse metrics field. The problem with these metrics is that they rely on a set of basic observable data that in some cases may lead to incorrect results. Such metrics are concerned on *how much* was reused versus how much was
developed from scratch, but fail to help on the analysis of *what* was reused and *how* it was reused. Software structure oriented metrics aim to fill this gap by providing more elaborate ways of analyzing the relationship between reused and new code on a software system. The software structure oriented metrics are divided into two main categories: the **amount of reuse metrics** and the **reusability assessment metrics**. The former target assessing the reuse of existing items, while the later aim to assess, based on a set of quality attributes, how reusable items are. Table 2 summarizes the main amount of reuse metrics .

Object oriented Structures

A brief description of the structure is given in this section using the pictorial description in Figure:3.The new object-oriented development methods have their own terminology to reflect the new structural concepts.
Referencing Figure 3, an object-oriented system starts by defining a *class* that contains related or similar *attributes* and *operations* (some operations are *methods*)
The classes are used as the basis for *objects* forming *hierarchical trees*. An object *inherits* all of the attributes and operations from its parent class, in addition to having its own attributes and operations.. An object can also become a class for other objects .When an object is applied and contains data or information, it is an *instantiation* of the object. Objects interact or communicate by passing *messages* . When a message is passed between two objects, the object are *coupled*. The degree to which the methods within a class are related to one another . Object X is coupled to Y if and only if X sends message to Y . Inheritance is a relationship among classes, wherein one class shares the structure or methods

defined in one or more other class . Instantiation , a process of creating an instance if object and binding or adding the specific data.. Message , a request that an object makes of another object to perform an operation.

Table 2. Amount of reuse metrics.

Metric	Definition
Reuse level (RP)	**Ratio of the number of reused lines of code to the total number of lines of code**
Reuse Level (RL)	**Ratio of the number of reused items to the total number of items.**
Reuse Frequency(RF)	**Ratio of the references to reused items to the tota number of references**
Reuse size & Frequency(RSF)	**Similar to Reuse Frequency , but also considers the size of items in the number of lines of code**
Reuse Ratio(RR)	**Similar to Reuse percent, but also considers partially changed items as reused .**
Reuse Density	**Ratio of the number of reused parts to the total number of lines of code**

Object-Oriented Specific Metrics

The object-oriented metrics that were chosen measure principle structures that, if they are improperly designed, negatively affect the design and code quality attributes. The selected object-oriented metrics are primarily applied to the concepts of classes, coupling, and inheritance. Preceding each metric, a brief description of the object-oriented structure.

We make use of the Overview Pyramid is a metrics-based means to both describe and characterize the structure of an object-oriented system by quantifying its complexity, coupling and usage of inheritance

The Overview Pyramid [28]

The overview of an object-oriented system must necessarily include metrics that reflect three main aspects:

1. Size and complexity. We want to understand how big and how complex a system is.

2. Coupling. The core of the object-oriented paradigm is objects that encapsulate data and that collaborate at run-time with each other to make the system perform its functionalities. We want to know to which extent classes (the creators of the objects) are coupled with each other.

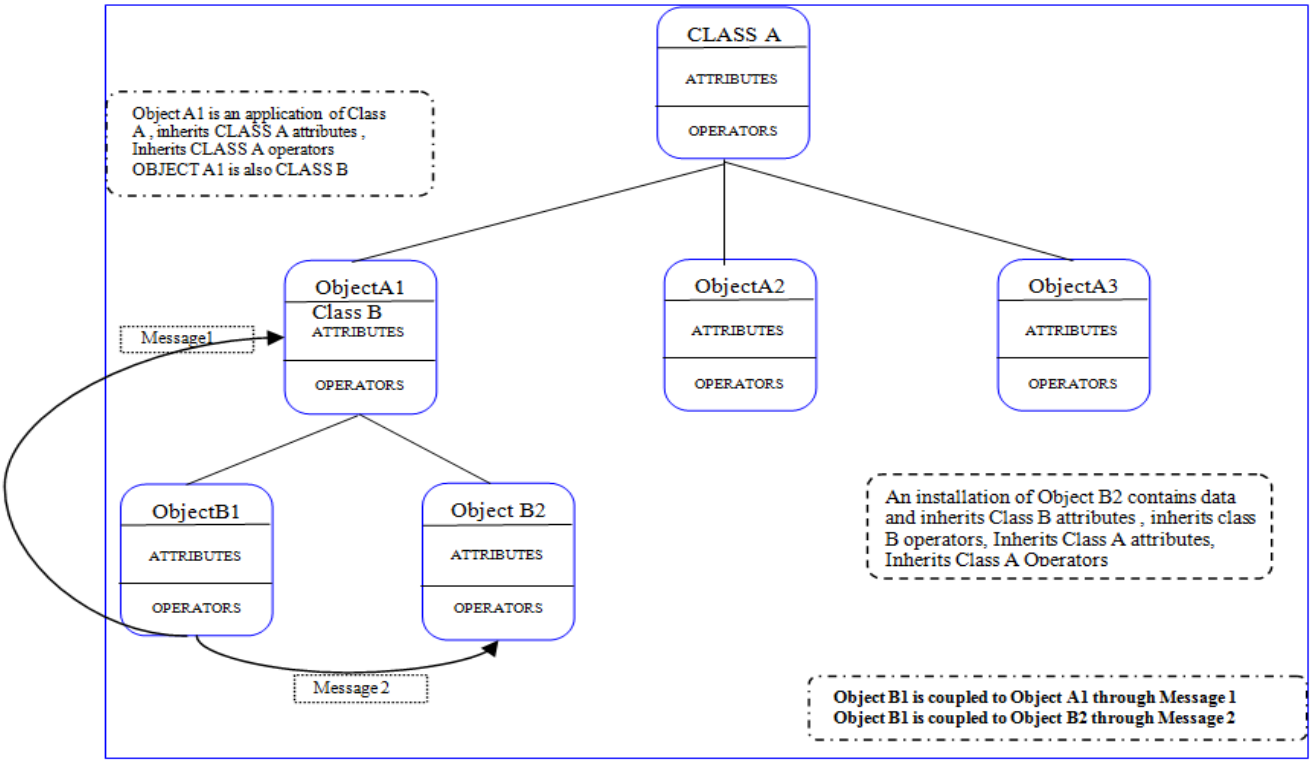

Figure 3- Object oriented Structures

Table 3: Object Oriented Metrics

Metric		Object oriented Feature	Concept	Measurement Method	Interpretation
CC	Cyclomatic complexity	Method	Complexity	Algorithmic test paths	Low => decisions deferred through message passing Low not necessarily less complex
SIZE	Lines of code	Method	Complexity	Physical lines , statements , and/or comments	Should be small
COM	Comment percentage	Method	Usability Reusability	Components divided by the total line count less blank lines	20 to 30 %
WMC	Weighted methods per class	Class/ method	Complexity Usability Reusability	1)Methods implemented within a class 2)Sum of complexity of methods	Larger => greater complexity and decreased understandability ; testing and debugging more complicated
LCOM	Lack of cohesion of methods	Class/ Cohesion	Design Reusability	Similarity of methods within a class by attributes	High=> good class subdivision Low=> Increased complexity – subdivide
CBO	Coupling between Objects	Coupling	Design Reusability	Distinct non-inherited related classes inherited	High=> poor design , difficult to understand , decreased reuse , increased maintenance
DIT	Depth of Inheritance tree	Inheritance	Reusability Understandability Testability	Maximum length from class node to root	Higher=> more complex , more reuse
NOC	Number of children	Inheritance	Design	Immediate Subclass	Higher=> more reuse ; poor design increasing testing

3. Inheritance. A major asset of object-oriented languages is the ease of code reuse that is possible by creating classes that inherit functionality from their super classes. We want to understand how much the concept of inheritance is used and how well it is used.

To understand these three aspects we use Overview Pyramid, which is an integrated, metrics-based means to both describe and characterize the overall structure of an object-oriented system, by quantifying the aspects of complexity, coupling and usage of inheritance.

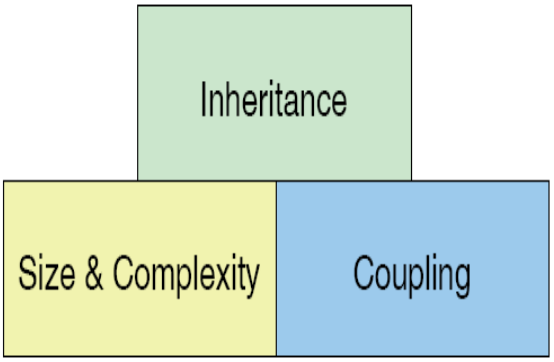

Figure 4: The Overview Pyramid[28]

lowest level units (i.e., code lines and independent functionality blocks). For each unit there is one metric in the Overview Pyramid that measures it. The metrics are placed one per line in a top-down manner, from a measure for the highest level unit (i.e., Number of Packages (NOP)) down to a complexity measure counting the number of independent paths in an operation (i.e., the Cyclomatic complexity (CYCLO)). We use the following metrics for the size and complexity side of the Overview Pyramid:

Size and complexity: direct metrics. We need a set of direct metrics (i.e., metrics computed directly from the source code) to describe a system in simple, absolute terms. The metrics describing the size and complexity are probably some of the simplest and widely used metrics. They count the most significant modularity units of an object-oriented system, from the highest level (i.e., packages or namespaces), down to the there is one metric in the overview pyramid that measures it. The metrics are placed one per line in a top-down manner.

• **NOP — Number of Packages**, i.e., the number of high-level packaging mechanisms, e.g., packages in Java, namespaces in C++, etc.

• **NOC — Number of Classes**, i.e., the number of classes defined in the system, not counting library classes.

• **NOM — Number of Operations,** 1 i.e., the total number of user defined operations within the system, including both methods and global functions (in programming languages that allow such constructs).

• **LOC — Lines of Code**, i.e., the lines of all user-defined operations. In the Overview Pyramid only the code lines containing functionality (i.e., lines of code belonging to methods) are counted.

• **CYCLO — Cyclomatic Number**, i.e., the total number of possible program paths summed from all the operations in the system. It is the sum of McCabe's Cyclomatic number [[29] for all operations.

The Right Part: System Coupling

The second part of the Overview Pyramid provides an overview with information about the level of coupling in the system (see Fig. 3.3),by means of operation invocations.

System coupling: direct metrics. The key questions when trying to characterize the level of coupling in a software system are: How intensive and how dispersed is coupling in the system? The two direct metrics that we use are:

• **CALLS — Number of Operation Calls**, i.e., this metric counts the total number of distinct operation calls (invocations) in the project, by summing the number of operations called by all the user-defined operations. If an operation fo () is called three times by a method f1() it will be counted only once. If it is called by methods

f1(), f2() and f3(), three calls will be counted for this metric.

• **FANOUT — Number of Called Classes**, this is computed as a sum of the FANOUT [30] metric (i.e., classes from which operations call methods) for all user-defined operations. This metric provides raw information about how dispersed operation calls are in classes.

System coupling: computed proportions. Again, the numbers above describe the total coupling amount of a system, but it is difficult to use those numbers to characterize a system with respect to coupling. We can compute, using the number of operations (NOM), two proportions that better characterize the coupling of a system.

• **Coupling intensity (CALLS/Operation)**. This proportion denotes the level of collaboration (coupling) between the operations, i.e., how many other operations are called on average from each operation. Very high values suggest that there is excessive coupling among operations, i.e., a sign that the calling operation does not "talk" with the right "counterpart".

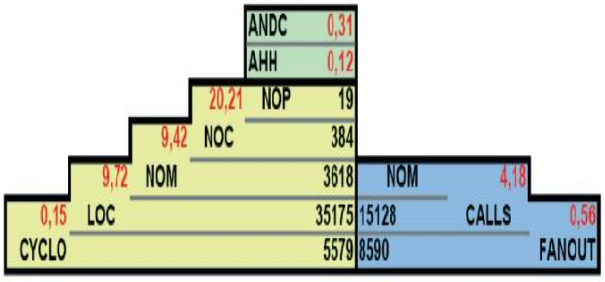

Fig. 5. Characterizing a system's coupling.[28]

• **Coupling dispersion (FANOUT /Operation Call)**. This proportion is an indicator of how much the coupling involves many classes (e.g., 0.5 means that every two operation calls involve another class).

Top Part: System Inheritance

The top part of the Overview Pyramid is not a adder as in the previous cases; it is composed of two metrics that provide an overall characterization of inheritance usage. These proportion metrics reveal how much inheritance is used in the system, as a first sign of how much "object-oriented ness" (i.e., usage of class hierarchies and polymorphism) to expect in the system.

Measuring Coupling & Independence

Coupling is defined as physical connections between elements of the Object Oriented (OO) design e.g. the number of collaborations between classes or the number of messages passed between objects represent coupling within an OO system

Measuring Software Coupling [27]

Coupling refers to the degree of interdependence between software system components. It is a software internal attribute that has been correlated to important software quality attributes such as maintainability, traceability, and robustness [10]. Coupling metrics can be used to assist managerial decisions because high levels of coupling were associated with lower productivity, greater rework, and greater design effort [11]. They can be used to assist in design decisions, where high coupling was associated with fault proneness of classes [I12, I13] and can also be used to aid in program re-factoring [10]. Therefore,

software designers are expected to determine, trace, and manage the factors that contribute to coupling, as a means of developing reliable and maintainable software and reducing costs. This paper introduces a technique that was successfully used in document information retrieval into software coupling measurement. This technique makes the coupling measurement in two steps and provides a systematic procedure for each step. The first step captures information about the elements of each component of the system into a description matrix without considering interactions between different components. The second step calculates coupling between components from the description matrix directly, according to a coupling formula.

Related work is discussed briefly in the next section. Then two examples of coupling metrics are provided, followed by a broad classification of coupling measures. Our coupling measure is then presented. Some results and comparisons of various metrics are then presented, followed by some concluding remarks and observations.

Related Work

In their seminal work, Stevens, Myers, and Constantine introduced the concept of coupling in procedural programming [14]. Six levels of coupling based on the Myers classification were then defined in [16]. We provide formal definition of these coupling classifications as binary relations on a pair of system components, *x* and *y*; these classifications are shown here in order from worst to best:

- Content coupling relation *R*5: (*x*,*y*) €*R*5 if *x* refers to the internals of *y* , *i.e.*, it branches into, changes data, or alters a statement in *y* .
- Common coupling relation *R4* : (*x*,*y*) €*R* if *x* and *y* refer to the same global variable.
- Control coupling relation *R3* : (*x*,*y*) €*R* if *x* passes a parameter to *y* that controls its behavior.
- Stamp coupling relation *R2* : (*x*,*y*) €*R* if *x* passes a variable of a record type as a parameter to *y* , and *y* uses only a subset of that record.
- Data coupling relation *R1* : (*x*,*y*) €*R* if *x* and *y* communicate by parameters, each one being either a single data item or a homogeneous set of data items that does not incorporate any control element.
- No coupling relation *R0* : (*x*,*y*) €*R* if *x* and *y* have no communication, *i.e.*, are totally independent.

This ordered classification has obtained general acceptance and has formed the basis for several software metrics such as the coupling metrics proposed by Fenton and Melton [16] and by [17]-Dhama, which we describe briefly.

Fenton and Melton Software Metric

Fenton and Melton [16] have proposed the following metric as a measure of coupling between two components *x* and *y* :

$C(x,y) = i + n/(n+1)$ where,

n = number of interconnections between *x* and *y* , and
i = level of highest (worst) coupling type found between *x* and *y* .

Table 4 : Fenton and Melton Modified definition for Myers Coupling levels.

Coupling Type	Coupling Level	Modified Definition between components x and y
Content	5	Components x refers to the internals of components y i.e. it changes data or alerts a statement in y.
Common	4	Components x and y refer to same global data.
Control	3	Components x passes a control parameter to y.
Stamp	2	Component x passes a record type variable as parameter to y.
Data	1	Components x and y communicate by parameters , each of which is either a single data item or a homogenous structure that does not incorporate a control element.
No Coupling	0	Components x and y have no connection ,i.e. are totally independent.

The level of coupling type is based on the Myers classification and is assigned a numeric
value, as shown in Table above.

Dhama Coupling Metric

Coupling is a measure of how closely tied are two or more modules or class. I particular, a coupling should indicate how likely would be that a change to another module would affect this module .. The basic form of coupling metric is to establish a list of items that cause one module to be tied to the internal working of another module. One of the metric to measure coupling is Dharma's Module coupling [25]

Dhama [17] proposed a coupling metric that measures the coupling of an individual component *C* , which is equal to:

1/(i1 + q612+u1+q2u2+g1+q8g2+w+r) where q6,q7,q8 are constants assigned a value of 2 as a heuristic estimate, and
i1 is the number of in data parameters,

i2 is the number of in control parameters,
u1 is the number of out data parameters, and
u2 is the number of out control parameters.
For global coupling:
g1 is the number of global variables used as data, and
g2 is the number of global variables used for control.

For environment coupling:

w is the number of other components called from component *C*, and *r* is the number of components calling component *C*; it has a minimum value of 1.

Coupling Metric Proposed by Alghamdi S. jarallah[27]

This metric proposes a framework that can be applied to both of the above paradigms. Each paradigm requires a different process to deal with the first step of collecting coupling data from either the system design or the code. The second step, which calculates the actual coupling values, operates in an identical manner regardless of the paradigm used.

The general approach of other coupling metrics is to calculate the coupling values for a system in one step. This metric involves breaking the calculation of coupling into two steps. The first step is to generate a description matrix that captures the factors that affect coupling in a system. The second step is to calculate the coupling between each two components of the system from the description matrix to produce a coupling matrix. The objective of generating a description matrix is to create a structure that captures all of the characteristics of a software system that relate to coupling, which can then be used to calculate coupling information for that system. Each component of the software
system is represented by a row of the description matrix. Components are classes in an object-oriented system, or functions, procedures, and subroutines in a procedural system. Columns of the description matrix represent elements. Elements are methods and instance variables in an object-oriented system, or variables and parameters in a procedural system.

Limitations of these metrics

There are two difficulties with these metrics . One is that an inverse means that the greater the number of situations that are counted , the greater the coupling that this module hs with other modules and smaller will be the value of mc. The other issue is that the parameters and calling counts offer potential for problems but do not guarantee that this module is linked to the inner working of the other modules , The use of global variables almost guarantees that this module is tied to the other modules that access the same global variables .

The following observations can be made concerning these coupling metrics:

1. The Fenton and Melton metric is a direct quantification of the Myers coupling levels, whereas the Dhama metric considers the number of variables or parameters belonging to categories that are less directly influenced by the Myers classification.

2. The highest coupling level between two components is the main determinant of their coupling value in the Fenton and Melton metric. The coupling value approaches the value of next coupling level as the number of interconnections between the two components increases.
3. The Fenton and Melton metric considers all types of interconnections between components to have the same complexities and have the same effects on coupling.
4. The Dhama metric considers the effect on coupling of a parameter to be the same as the effect of a global variable, which is a major deviation from the Myers classification scheme.
5. The Fenton and Melton metric is an example of an inter-modular coupling metric, which calculates the coupling between each pair of components in the system. The Dhama metric is an example of an intrinsic coupling metric, which calculates the coupling value of each component individually.

Classifications of Coupling Measures

Existing coupling measures can be broadly classified into the following two groups:

1. Procedural programming coupling measures: these measure the coupling of software components that are implemented in procedural programming languages; examples include metrics proposed by Lohse and Zweben [18], Huches and Basili [19], Fenton and Melton [16], Offut, Harold and Kotle [20], and Dhama [17]. This class of metrics
is heavily influenced by the Myers classification of coupling levels.

2. Object-oriented coupling measures: these measure the coupling of software components that are implemented in object-oriented programming languages; examples include metrics proposed by Henry and Li [21], Tegarden and Sheetz [22], J-Y Chen and J-F La [23], Lorenz and Kidd [24], and Chidamber and Kemerer [26].
All the above stated metrics focus on the measuring the coupling of the objects but not the independence .

Independence is an important quality of a reusable module. The more independent the module is , more it is usable and is not dependent on the other modules.

Proposed Metrics for the evaluation of the independence of the Functionalities in a module for reusability

This metric evaluates the reusability of the component by checking its independence. A component with more independence can be treated as more reusable.

We can have the different types of combinations with components treated as independent

Null Hypothesis

Hypothesis is as follows

Ho; Ri ~ Independent

H1; Ri not~ independent

Ho, reads that the components are independent and can be reused. Failure to reject hypothesis means that no evidence of non – Independence has been detected on the basis of this test. This does not imply that further testing of the components for independence is unnecessary.

Level of significance α must be stated. The level α is the probability of rejecting the null hypothesis given that null hypothesis is true or

α = P (reject Ho/ Ho true)

α can be set to 0.01 to 0.5

This metric is based on the poker test for independence, which is based on the frequency which certain digits are repeated in a series of numbers.

The following example shows an unusual amount of repetition

0.255, 0.577, 0.331. 0.414, 0.828, 0.909, 0.303, 0.001, …

In each case, a pair of like digits appears that was generated. In three-digit numbers there are only three possibilities as follows

1) The individual numbers can all be different

2) The individual numbers can all be the same

3) There can be one pair of like digits

The probability of drawing one ball from the bag of balls is applicable as the base of this metric or the expected value of the independent metric. The probability associated with each o these possibilities is given by the following

P (three different digits) = P (second different from the first) X P (Third different form the first and second)

= (0.9) (0.8) = 0.72

P (three like digits) = P (second different from first) X p (Third different from the first) = (0.1) (0.1) = 0.01

P (exactly one pair) = 3C2 (0.1)(0.90) = 0.27

Lets explain this by an example : a sequence of three digit numbers has been generated and an analysis indicates that 680 have different digits , 289 contain exactly one pair of like digits and 31 contain three like digits . Based on the poker test , are these numbers independent ? let α =0.05.

Chi-square Test

The chi-square test is a very important and useful test for determining how well certain observed data fir the theoretically expected data. The testing is performed by first dividing the observed data into 'k' non-overlapping classes (in our case it will be the various occurrence pattern); 'k' must be 3 or more. Then count the Oi, the number of times the observed data falls in each class i, for i = 1, 2, 3, …,k . Next , we determine the expected number of occurrences Ei in each class i . Then to measure how far the observed frequency deviates from the expected we compute the chi-square test defined by

$$\text{Chi}^2 = \sum_{i=1}^{K} (Oi-Ei)^2/Ei$$

Now read the chi –square tables and find the values in the table corresponding the different probabilities.

eg. Read the values of chi-square for degree of freedom v=9

.995	.99	.95	090	.75	.50	.25
1.73	2.09	3.33	4.17	5..90	8..34	11.4
.10	.05	.01	.005			
14.7	16.9	21.7	23.6			

This means that there is a 99.5% probability of chi-square exceeding 1.73; a 99% probability of chi-square exceeding 2.09 ,….., and .5 % probability of chi-square exceeding 23.6. Thus the probability of chi – square (for v = 9) falling below 1.73 and above 23.6% is only 1%. That is 990 sequences out of 1000 sequences perfectly independent occurrences would have given

1.73 <= chi-square <= 23.6

Like wise , if we take 1 as cutoff point , we would reject all sequences below 2.09 and above 21.7 .

The test is summarized in the table as below

Table 5: Observed and Expected values

Combinations	Observed Frequency, Oi	Expected frequency, Ei	(Oi-Ei) 2 /Ei
Three different digits	680	720	2.22
Three like digits	31	10	44.10
Exactly one pair	289	270	1.33
Total	1000	1000	47.65

The appropriate degrees of freedom are one less than the number of intervals . Since 47.65> X2 0.05,2 = 5.99. The independence of the numbers is rejected on the basis of this test.

Example 2: This is based on poker game in which the cards are drawn from the deck of cards . Suppose we have five independent functionalities viz, a,b,c,d,e and we treat these as a hand of poker in card game and classify accordingly

Five of a kind	(a a a a a)
Four of a kind	(a a a a b)
Full House	(a a a b b)
Three of a Kind	(a a a b c)
Two Pairs	(a a b b c)
One pair	(a a b c d)
Bust	(a b c d e)

The associated probabilities associated with these seven hands are

Five of a kind	(a a a a a)	0.0001
Four of a kind	(a a a a b)	0.0045
Full House	(a a a b b)	0.0090
Three of a Kind	(a a a b c)	0.0720
Two Pairs	(a a b b c)	0.1080
One pair	(a a b c d)	0.5040
Bust	(a b c d e)	0.3024

If we generated 5n random digits we can form n random poker hands and then compare the observed frequencies of these seven types of poker hands with the expected distribution . To measure the amount of deviation between the expected and the actual distribution we once again use the chi-square test with degree of freedom v=6.

Now this test of independence can be applied to reusable components of a software.

The functionalities to be included in software for reusability are dependent or independent of each other.

For example say we have one functionality a, which is incorporated in the software and it may be dependent or may call for functionality b in the software.

If a is in the influence of b up to certain extent then a and b must co-exist.

Thus if the software developed for reuse includes only the functionality a and does not include the functionality b will not be a good reusable package. To make a perfect reusable software it must contain all the general functionalities so that it may be used across all several domains.

There are a number of metrics available for measuring various parameters of a component or software but still these metrics lack the capability to calculate the independence of a component.

If we have to design software or we have to check the software on the terms of this metric the following steps are involved

1) Firstly, find out the sequential occurrence of a component or functionality across several software. These are termed as observed occurrences. (O)
2) Then compare the occurrences with then compare these occurrence with the expected (E), based on poker test.
3) perform the chi-square test with given degrees of freedom and values of α.
4) Then conduct the chi-square test on the O and E and draw inferences as per the values of the chi-square.

For example we now take the example of 10 components and used in different software with five at a time:

The different combinations of components can be

C1 C2 C3 C8 C9 - five different kinds
C1 C1 C7 C9 C8 - a pair
C2 C2 C5 C5 C9- two pair
C6 C6 C6 C1 C8 – three of a kind
C8 C8 C8 C8 C7 – four of a kind
C1 C1 C1 C1 C1 – five of a kind
The occurrence of five of a kind is rare.

Lets take an example to explain this metric

In a software which is using 10000 components of 10-different kinds (a, b, c, d, e, f, g, h, i, j, k).
1) The observed sequences of components in the software are

The expected combinations for 10000 poker hands are as below

Table: Expected values

Combination Distribution	Expected (Ei)	percentage
Five different components	3024	30.24%
Pairs	5040	50.40%
Two Pairs	1080	10.80%
Three of a kind	720	7.20%
Full Houses	90	0.90%
Four of a kind	45	0.45%
Five of a kind	01	0.01%

Table: Expected and Observed values

Combination Distribution	Observed Distribution	Expected (Ei)	(Oi-Ei)2/E
Five different components	3033	3024	0.0268
Pairs	4945	5040	1.7906
Two Pairs	1098	1080	0.3
Three of a kind	667	720	3.9
Full Houses	101	90	1.3444
Four of a kind	52	45	1.8
Five of a kind	01	01	0
Total	10000	10000	9.1619

The degrees of freedom in this case is 6, which is one less than the number of cases i.e .7 Tabulated value of Chi – Square for v=6 is 0.675727 for probability 0.995 ,i.e.,α=0.05 and the calculated value is 9.1618.

Thus the component combinations are independent and the software can be designed to have different components with independent functionalities.

Discussion and Conclusion

In the above example the no of components were 10 and the possible combinations were taken as 5. We can have any number of combinations and components. All we have to calculate the expected values by probability rules and then the observed values form the system. The chi-square test will then be done to calculate the independence. Thus we have seen that the components can be used across various software with dependency and non dependency on other components. The test of independence evaluates how much independent the component is . The more independent the component , the more reusable it is.

References

[1] Briand et al. A Unified Framework for coupling Measurement. *IEEE* Transactions on Software Engineering. vol. 25, no. 1, Jan-Feb 1998 1999.
[2] Etzkorn, Letha, Bansiya, Jagdish, Davis, Carl. Design and Code Complexity Metrics for Object-Oriented Classes. Quality Metric for Object-Oriented Design. Journal of Object-Oriented Programming. April 1999.
[3] Fenton, Norman E., Pfleeger, Shari L. Software Metrics: A Rigorous & Practical Approach. 2 nd ed. PWS, 1997.
[4] Gillibrand, David, Liu, Kecheng. Quality Metric for Object-Oriented Design. Journal of Object-Oriented programming. Jan 1998.
[5] Lim W., Effects of Reuse on Quality, Productivity, and Economics. In: *IEEE Software*, Vol. 11, No. 05, September, 1994, pp. 23-30.
[6] Henry E. and Faller B., Large-Scale Industrial Reuse to Reduce Cost and Cycle Time. In: *IEEE Software*, Vol. 12, No. 05, September, 1995, pp. 47-53.
[7] Basili, V. R.; Briand, L. C.; Melo, W. L. How Reuse Influences Productivity in Object-Oriented Systems. In: *Communications of the ACM*, Vol. 39, No. 10, October, 1996, pp. 104-116
[8] Devanbu, P. T.; Karstu, S.; Melo, W. L.; Thomas, W. Analytical and Empirical Evaluation of Software Reuse Metrics. In: *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, Berlin, Germany, pp. 189-199, 1996.
[9] Frakes, W. & Succi, G. An Industrial Study of Reuse, Quality, and Productivity. In: *Journal of Systems and Software*, Vol. 57, No. 02, June, 2001, pp. 99-106.
[10] P. Joshi and R.K. Joshi, "Microscopic Coupling Metrics for Refactoring", *Proceedings of the Conference on Software Maintenance and Reengineering CSMR 2006*, 22-24 March 2006, pp.145–152.

[11] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An exploratory analysis". *IEEE Transactions on Software Engineering*, 24(1998), pp. 629–639.
[12] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, "Prediction of Fault-Proneness at Early Phase in Object-Oriented Development", *Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 1999, pp. 253–258.
[13] Mie Mie Thet Thwin and Tong-Seng Quah, "Application of Neural Networks for Software Quality Prediction Using Object-Oriented Metrics", *Journal of Systems and Software*, 76(2)(2005), pp. 147–156.
[14] G. Myers, *Composite/Structured Design*. Van Nostrand Reinhold, 1978.
[15] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*. 2nd edn. Reading, 1997.
[16] Norman Fenton and Austin Melton, "Deriving Structurally Based Software Measures", *J. System Software*, (12) 1990, pp. 177–187.
[17] H. Dhama, "Quantitative Models of Cohesion and Coupling in Software", *Journal of System and Software*, 9(1)(1995), pp. 65–74.
[18] J. B. Lohse and S. H. Zweben, "Experimental Evaluation of Software Design Principles: An Investigation into the Effect of Module Coupling on System Modifiability", *Journal of System and Software*, 4(1)(1984), pp. 301–308.
[19] D. H. Hutches and V. R. Basili, "System Structure Analysis: Clustering with Data Bindings", *IEEE Transactions on Software Engineering*, 11(8)(1985), pp. 749–757.
[20] A. J. Offut, M.J. Harrold, and P. Kotle, "A Software Metric System for Module Coupling", *Journal of System and Software*, 20(3)(1993), pp. 295–308. *Jarallah S. Alghamdi April 2008 The Arabian Journal for Science and Engineering, Volume 33, Number 1B* 129
[21] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability", *Journal of Systems and Software*, 23(2) (1993), pp. 111–122.
[22] D.P. Tegarden, S.D. Sheetz, and D.E. Monarchi, "The Effectiveness of Traditional Software Metrics for Object-Oriented Systems", ed., in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, ed. J.F. Nunamaker, Jr. and R.H. Sprague, (1992), pp. 359–368.
[23] J. Chen, and J. Lu, "A New Metric for Object-Oriented, Design", *Information and Software Technology*, 5(4)(1992), pp. 232–239.
[24] Brian Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. New York: Prentice Hall PTR, 1996.
[25] H. Dharama, 'Quantitative Models of Cohesion and Coupling in Software', Journal of Systems and Software, 29:4, April 1995.]
[26] Shyam R. Chidamber, and Chsis F. Kemerer, "A Metrics Suite For Object Oriented Design," *IEEE Transactions On Software Engineering*, 20(6)(1994), pp. 476– 493.
[27] Ganesn Dharmalingam, Knodel Jens, Identifying Domain-Specific Reusable Components from Existing OO Systems to Support Product line Migration; *Fraunhofer Institute for Experimental Software Engineering*
[28] Lanza Michele, Marinescu Radu ; Object – Oriented Metrics in Practice , Springer.
[29] T.J. McCabe . A measure of complexity. IEEE Transactions on Software Engineering ,2(4) : 308 -320, December 1976.
[30] Mark Lorenz and Jeff Kidd. Object- Oriented Software Metrics : A practical guide . Prentice-Hall,1994.
[31] Jarallah S. Alghamdi , Measuring Software Coupling, *Information & Computer Science Department King Fahd University of Petroleum & Minerals* Dhahran, 31261, Saudi Arabia.

P.K. Suri received his Ph.D.degree from Faculty Of Engineering Kurukshetra University, Kurukshetra, India and Master's degree from Indian Institute of Technology, Roorkee (formerly known as Roorkee University), India. He is working as Professor in the Department of Computer Science & Applications, Kurukshetra University, Kurukshetra - 136119 (Haryana), India since Oct. 1993. He has earlier worked as Reader, Computer Sc. & Applications, at Bhopal University, Bhopal from 1985-90. He has supervised five Ph.D.'s in Computer Science and thirteen students are working under his supervision. He has more than 100 publications in International / National Journals and Conferences. He is recipient of ''THE GEORGE OOMAN MEMORIAL PRIZE' for the year 1991-92 and a RESEARCH AWAWD – "The Certificate of Merit-2000" for the paper entitles ESMD- An Expert System for Medical Diagnosis from the Institution of Engineers, India. His Teaching and Research include Simulation and Modeling, SQA, Software Reliability , Software Testing & Software Engineering Process, Temporal Databases, Ad Hoc Networks, Grid Computing , and Biomechanics.

Neeraj Garg received his B.E. Degree and Masters in Computer Science and Applications (MCA) Panajb University , Chandigarh and Kurukshetra University, Kurukshetra in the year 1992 and 2001 respectively. Currently he is pursuing Ph.D. in Computer Science from the Department of Computer Science & Applications , Kurukshetra University , Kurukshetra , India. He had served as Head of the Department of MCA department at Maharaja Agresen Institute of Management and Technology , Jagadhri, Haryana, India and Department of IT Engg. M. M University, Mullana , India. Currently he is Associate Professor in the Department if IT Engg. M. M. University Mullana, India . He had also worked with various organizations including C-DOT where he had carried out research work in SS#7 Protocol of Telephone Networks. He is co- editor of MAIMT- Journal of IT and Management. His research areas include Simulation and Modeling, Software engineering , System Programming and Networks.