Coupling and cohesion are two fundamental concepts and their importance is determined by the nature of software and software development. Software systems are, per se, complex that we need to resort to the principle of divide and-conquer for developing software. Software is also intrinsically soft and does not wear out, and there is a constant need of changing software after delivery [71]. If we wish to reduce maintenance effort, divided modules must be as independent as possible. Therefore reducing coupling becomes one of the basic criteria of design and this has now been plainly promoted as elementary knowledge in most of software engineering textbooks ([71], [57], [76], [31], [80], [22]). Although there exist certain guidelines for applying coupling criterion to attain to module decomposition with a good maintainability ([57], [82], [90], [67]), certain experts in software measurement still found that coupling is difficult to apply [78]. Moller and Paulish [60] described the best practices to implement metrics for software development in Europe and North America, but, coupling and its measurement are simply not mentioned. This is not really encouraging, for one of main characters of engineering discipline is being able to assess quantitatively the quality of product and maintainability should be one of the prime qualities of systems design.

Software life cycle costs depend on software quality factors such as complexity, maintainability, reusability, reliability, and portability. The two properties of software that have a great impact on software quality are cohesion and coupling (C&C). Figure 1 shows the 13 software quality factors identified by the Rome Laboratory (Bowen et al., 1983) and 8 of which (highlighted) are dependent on C&C. Thus, the identification, measurement, and management of C & C in software can have a major influence on reducing software costs. **Cohesion** in a module refers to that software property that binds
together the various statements and other smaller modules comprising the module. We define a module to be a compilation unit of code. A module can contain other smaller modules. Therefore, a function, a procedure, or any combination of these is referred to as a module. Cohesion is an intra module property that reflects the design considerations for integrating the various components of the module into one unit. It is the glue that holds a module together, and it is a measure of the logical strength of a software module. The strength and consequently the "quality" of the module increase with increasing cohesion.

_Coupling_ is a measure of the interdependence between two software modules. It is an inter module property.

Because it is desirable that the changes made in a module affect another module as little as possible, the "quality" of a module increases as module coupling decreases. Object-oriented software is a collection of many classes and each class is a collection of attributes and methods. Software can be said to be of good quality if its classes have maximum cohesion and minimum coupling. The diffusion of object-oriented paradigm has led to the need of object-oriented metrics for assessing coupling between classes. Three types of coupling may exist between classes in object-oriented systems: inheritance coupling, interaction coupling, and component coupling. Prediction models using object-oriented design metrics can be used for obtaining assurances about software quality.

In practice, quality estimation means either estimating reliability or maintainability. The estimated number of defects can also be normalized by a size measure to obtain a defect density estimate. Maintainability is typically measured as change effort. Change effort can mean either the average effort to make a change to a class, or the total effort spent on changing a class.
### 2.1. Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Functional cohesion of a module</td>
</tr>
<tr>
<td>i₁</td>
<td>In data parameters</td>
</tr>
<tr>
<td>i₂</td>
<td>In control parameters</td>
</tr>
<tr>
<td>u₁</td>
<td>Out data parameters</td>
</tr>
<tr>
<td>u₂</td>
<td>Out control parameters</td>
</tr>
<tr>
<td>l₁</td>
<td>Number of local variables used as data</td>
</tr>
<tr>
<td>l₂</td>
<td>Number of local variables used as control</td>
</tr>
<tr>
<td>g₁</td>
<td>Number of global variables used as data</td>
</tr>
<tr>
<td>g₂</td>
<td>Number of global variables used as control</td>
</tr>
<tr>
<td>W</td>
<td>Number of modules called</td>
</tr>
<tr>
<td>D</td>
<td>Data flow cohesion in a module</td>
</tr>
<tr>
<td>s</td>
<td>Total number of statements in the module</td>
</tr>
<tr>
<td>j</td>
<td>Number of statements separating pairs of statements being evaluated for data flow cohesion. When the pair of statements are adjacent, then ( j = 1 ). The value of ( j ) goes from 1 to ( s - 1 ).</td>
</tr>
<tr>
<td>dᵢⱼ</td>
<td>Number of pairs of statements that have data flow cohesion when ( j = 1, 2, \ldots, s - 1 ).</td>
</tr>
<tr>
<td>v</td>
<td>Total number of variables in the module, then</td>
</tr>
<tr>
<td>A</td>
<td>Action-bundling cohesion in a module</td>
</tr>
<tr>
<td>S</td>
<td>Action-bundling cohesion in a module</td>
</tr>
</tbody>
</table>

*Fig. 2.1. Hierarchy of software qualities as proposed by Rome Laboratory (Bowen et al. 1983).*
b : Number of statements separating pairs of statements being evaluated for action-bundling cohesion. When the pair of statements are adjacent then \( b = 1 \). The value of \( b \) goes from 1 to \((s-1)\).

\( \text{t}_k \) : Number of pairs of statements that have action-bundling cohesion when \( j = 1, 2, \ldots (s-1) \).

L : Logical bundling cohesion

w : Nesting depth of block in which a block of statements is located

\( t \) : Number of statements in the block under consideration

\( k \) : Total number of blocks in the module

C : Total module coupling

g_1 : Number of global variables used as data

g_2 : Number of global variables used as control

r : Number of modules calling the module under consideration

M(c) : Set of methods of a class \( c \)

A(c) : Represents set of all attributes whether implemented or inherited in a class \( c \).

O(c) : Represents the set of objects of a class \( c \) created during execution of the program.

\( o \) : An object \( o \) is an instance of a class \( c \) created at run-time such that \( o \in O(c) \wedge c \in C \)

c : It represents the set of classes in an object-oriented system.

\( m \) : Total number of attributes i.e. \( |A(c)| = m \) & \( |A(o)| = m \)

\( n \) : Total number of normal methods i.e. \( |MN(c)| = n \) & \( |M(o)| = n \).
2.2. Quantitative Models

Dhama [25], in his paper in 1995, discussed on Quantitative models of Cohesion and coupling (C&C) in software. Qualitative evaluation of C&C has been used to measure the “good” qualities of software. However, the subjective judgment and consequent inconsistency inherent in qualitative assessments have raised questions about the consistency and credibility of such evaluations. In their research, Dhama has developed quantitative models of C&C and then used those models to evaluate C&C for some existing software modules.

The significance of C&C was established by Yourdon, Constantine, and Myers (1976) in the period 1973-1979, and their analysis forms a part of Yourdon and Constantine’s [89] book, *Structured Design*. A large portion of the subsequent work on software metrics has been done by Rome Laboratory-funded research, and in the early 1980s, there were a number of Rome Laboratory publications [20] promoting and outlining the use of software metrics. Although the importance of C&C in software metrics has been well established, there have been limited attempts to quantify C&C. In 1989, Ott and Thuss at Michigan Technological University [66] developed a quantitative model of cohesion using a slice based methodology. In June 1992, Zage et al. [91] built a model of module coupling and showed that the number of module errors increases as coupling increases. In an empirical study done by Card et al. in 1986 [23] at Computer Sciences Corporation, high cohesion values have been related to fewer software errors. Because a correlation between C&C and number of errors has been established, the C&C metric can be used for

- Forecasting testing costs and reliability
- Allocating the testing effort according to the error proneness of the module
- Software quality assurance
- Assessing the quality of reusable software
2.2.1. Cohesion

2.2.1.1. Functional Cohesion

This type of cohesion results from the single-purpose functional design of the module. The more focused the module goal, the greater its functional cohesiveness.

Let \( p = i_1 + q_1i_2 + u_1 + q_2u_2 + l_1 + q_3l_2 + g_1 + q_4g_2 + q_5w \), where \( q_1, q_2, q_3, q_4, \) and \( q_5 \) are constants, and, as a first heuristic estimate, are assumed to be 2 in our calculations.

Then,

\[
F = K_1 / \text{Maximum}(p,1)
\]

where \( K_1 \) is the proportionality constant.

From this Dhama found that the module which has sub modules which does smaller amount of work is more functionally cohesive.

2.2.1.2. Data Flow cohesion

Data flow cohesion describes the interdependencies among the different statements of the module depending on the processing of data. Data flow exists when a piece of data, after undergoing some transformation in a statement, must undergo another transformation in a following statement. This is shown in Figure 4.2, where the actions (statements) of the module are linked together like a chain by the data that flows from one action to the next.
Figure 2.2: Actions of a Module linked together by a data flow

Data flow cohesion in a module increases as the number of statements connected by the data flow increases. The ratio of such statements to the total number of statements is used as a measure of the cohesion of a module.

\[ D \propto d_i \]

and

\[ D \propto 1/j \]

For a single variable, data flow cohesion normalized over the total lines of code \( s \) is given by

\[ D = \frac{K_2}{s} \sum_{j=1}^{s} d_j / j \]

Where \( K_2 \) is a constant. Assuming \( K_2 = 1 \) data flow cohesion becomes

\[ D = \frac{1}{s} \sum_{j=1}^{s} d_j / j \]

The same data flow considerations have to be given for each variable in the module. With assumption, that the contribution of each variable toward the module data flow cohesion is additive, with total number of variables \( v \), becomes:

\[ D = \frac{1}{s} \sum_{i=1}^{v} \sum_{j=1}^{s} d_{ij} / j \]
From this Dhama found that the modules that are portioned into smaller units of code according to the data manipulated has greater data flow cohesion.

2.2.1.3. Action-bundling cohesion

Action-bundling cohesion takes place as a result of the collection of several actions to be performed on a single piece of data (Figure 4). Suppose the task was to read, up-date, and write out an array or a file; if all these actions were gathered together in one unit of code, then this unit of code would exhibit action-bundling cohesion.

Action bundling between two statements is determined by examining the same data being manipulated in the two statements.

For a single variable action bundling cohesion is given as

$$A = \frac{1}{s} \sum_{s=1}^{s-1} t_{sb} / b$$

By summing up the data flow contribution of each variable, the action bundling cohesion becomes:

$$A = \frac{1}{s} \sum_{v=1}^{v-1} \sum_{s=1}^{s-1} t_{vb} / b$$

Where $v$ is total number of variables.

From the experiments Dhama found, the modules which have highest value of Action-bundling cohesion where the main program has the procedures which calls subprograms and coordinates their activities.

2.2.1.4. Logical bundling cohesion

The cohesion exhibited by the statements of the block is categorized as logical bundling cohesion as the statements “bundle” together following a “logical” decision.
Average nesting depth \( = \frac{1}{k} \sum_{i=1}^{k} w_i \)

Average number of statements in a block \( = \frac{1}{k} \sum_{i=1}^{k} l_i \)

\[ L = \frac{\sum_{i=1}^{k} w_i}{\sum_{i=1}^{k} l_i} \]

Therefore, \( L \) is greater when the module has fewer loops.

The module with fewer loops has greater cohesion.

### 2.2.2. Coupling

When modules are linked together, the four categories of coupling exhibited and their causes are as shown below:

1. Data flow coupling caused by the parameters at the interface
2. Control flow coupling also caused by the parameters at the interface
3. Global coupling caused by global variables
4. Environmental coupling caused by calling and being called by other modules

Any parameter or shared variable can be used as data or as a “control variable”; the latter is defined as a variable that determines operational execution, such as when used in a LOOP or IF-THEN type of statement. A variable used as a diagnostic or error recovery variable is also classified as a control variable.

Dhama assumed the following to develop the model:

1) As a start, their premise is that control variables result in twice as tight a coupling as an equal number of data variables.
2) Also assume that the extent of coupling exercised by parameters and global variables is the same.

3) The coupling between modules to be as low as possible

In the model development for coupling, low coupling will be indicated by high numbers because increasing numbers are generally associated with increasing "goodness." This association is also in keeping with our calculations of module cohesion, in which high numbers indicate high cohesion, which is "good."

Now \( \sum \) \( m = i_i + q_0 l_2 + u_1 + q_7 u_2 + g_1 + q_8 g_2 + w + r \)

where \( q_0, q_7 \) and \( q_8 \) are constants, and, as a first heuristic estimate, \( r \) is assumed to be 2 in all calculations. The minimum value of \( r \) is 1, because every module executed, including a main program, must be called by some other program.

\[ C \propto 1/m \]

Therefore, \( C = K_s/m \) where \( K_s \) is a constant. Assuming \( K_s = 1 \), we have \( C = 1/m \)

Dhama suggests that \( C \) & \( C \) need not be the only metric considered in evaluating a software module; software has many facets and \( C \) & \( C \) is but one of them. However, we feel that \( C \) & \( C \) is an important facet because so many of the higher level software quality factors depend on it.

2.3. Inheritance Coupling Measurement

AlGhamadi. Elish, Ahmed in 2002 has created a tool for measuring inheritance coupling. The tool that has been created to measure inheritance coupling first parses object-oriented systems to collect inheritance coupling metrics data. The collected data are then abstracted into a language independent format according to a conceptual model and stored in the system's database. Finally, query engines are applied on the database to calculate different coupling metrics. Java parser has been implemented to enable the
calculation of inheritance coupling for software implemented in Java. Other object-oriented languages parsers can be implemented without changes to the system.

The presented tool consists of three main components as

1) Parsing engines
2) Central metrics repository
3) Query engines.

2.3.1. Implementation

The developed tool was implemented using Javadoc in JDK 1.2, Microsoft Visual Basic 6.0, Enterprise Edition, and Microsoft Access 97. Javadoc is a tool that parses the declarations and documentation comments in a set of Java source files and produces a set of HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields. Javadoc doclets can be used to customize Javadoc output. A doclet is a Java program written with the doclet API that specifies the content and format of the output to be generated by the Javadoc tool. The developed Java parser is a doclet with some modifications customized to parse any Java source code and produce a text file containing the collected inheritance coupling metrics data. The graphical user interface of the tool was designed using Microsoft Visual Basic. Microsoft Access is used as a database engine to connect the central metrics repository.

2.3.1.1. Features

The following are some features and advantages of the presented tool:

1) It simplifies the metrics data collection and ensures the accuracy and consistency of the collected data.
2) It is not limited to a particular language. Adding support for any new object-oriented languages and metrics can be added easily.
3) It provides a platform for analyzing and comparing different object-oriented coupling metrics.
4) It is easy to use through its graphical user interface.
5) It provides chart view of the metrics results.
6) It has class browser to navigate through system classes and their members.

2.3.1.2. Comparison of tools

Most of the existing tools support C++ software measurement, while the presented tool supports Java software measurement. Unlike the presented tool, existing tools do not support inter-class inheritance coupling metrics, i.e., measuring inheritance coupling between classes. They instead support simple inheritance coupling metrics, such as depth of inheritance tree (DIT), which are also supported by the presented tool. The presented tool separates the representation of the system from the calculation procedure. The advantage of such approach is that in order to evaluate the coupling between classes, efforts are only concentrated on how to define weights of attributes and methods in the corresponding classes. The presented tool, like some other existing tools, is not limited to a particular language and support for new object-oriented languages and metrics can be added easily.

2.4. Applications of neural networks for software quality prediction

In 2005 Thwin & Quah has published a paper on Applications of neural networks for software quality prediction. Prediction models using object-oriented design metrics can be used for obtaining assurances about software quality. In practice, quality estimation means either estimating reliability or maintainability. The estimated number of defects can also be normalized by a size measure to obtain a defect density estimate. Maintainability is typically measured as change effort. Change effort can mean either the average effort to make a change to a class, or the total effort spent on changing a class. A variety of statistical techniques are used in software quality modelling. Models are often based on statistical relationships between measures of quality and measures of software metrics. However, relationships between static software metrics and quality factors are often complex and nonlinear, limiting the accuracy of conventional approaches. Artificial
neural networks are adept at modelling nonlinear functional relationships that are difficult to model with other techniques, and thus, are attractive for software quality modelling.

### 2.4.1. Predicting Software Quality

The neural network model aims to predict object-oriented software quality by estimating the number of faults and the number of lines changed per class. Software metrics used include both object-oriented metrics and traditional complexity metrics. Object-oriented metrics used include inheritance related measures, cohesion measures, coupling measures and memory allocation measures. Ward neural network and General Regression neural network has been used to improve prediction result for estimating software quality.

Ward neural network is a back propagation network with different activation functions. They are applied to hidden layer slabs to detect different features in a pattern processed through a network to lead to better prediction. We use a Gaussian function in one hidden slab to detect features in the mid-range of the data and a Gaussian complement in another hidden slab to detect features for the upper and lower extremes of the data. Thus, the output layer will get different “views of the data”. Combining the two feature sets in the output layer leads to a better prediction.

General Regression Neural Network (GRNN), Specht (1991), is a memory-based network that provides estimates of continuous variables and converges to the underlying (linear or nonlinear) regression surface. This is a one-pass learning algorithm with a highly parallel structure. Even with sparse data in a multidimensional measurement space; the algorithm provides smooth transitions from one observed value to another.

In this paper, two kinds of investigation are performed. The first on predicting the number of defects in a class and the second on predicting the number of lines changed per class.
### 2.4.1.1. Predicting number of defects in a class

The applications used in this prediction are three subsystems of HMI (Human Machine Interface) software, which is a fully networked Supervisory Control and Data Acquisition system. Subsystem A is a user interface-oriented program that allows customers to configure the basic product operations and device communications. Subsystem B is a real time data logging process that collects data as needed and logs data into the database, based on the user configuration. Subsystem C is a communication-oriented program that acts as a router, not only delivering messages between processes within the same host but also forwarding messages to other hosts. Preliminary analysis using multiple regressions with all predictors WMC, Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response For a Class (RFC), Inheritance Coupling (IC), Coupling Between Methods (CBM), Weighted Methods per Class (WMC) and Number of Object/Memory Allocations (NOMA). The adjusted R square corrected for the number of predictors equals 0.843. So therefore, 84% of the total variance in the number of faults is accounted for by the metrics in the population. The observed significance level is less than 0.001. We can conclude that the prediction of the number of faults from the above metrics is possible. In the prediction of the number of faults experiment, the correlation between predicted faults and actual faults is 0.9531 in the GRNN network and 0.9476 in the Ward network. The number of observations is 97. The significance level of a cross-validation is indicated by the p value. A commonly accepted p value is 0.05. In our experiment, the two tailed probability p value is less than 0.001 in the both cross-validations. This shows a high degree of confidence for the successful validations. The results clearly indicate a close relationship between metrics (independent variables) and the number of faults per class in software applications (dependent variable).

### 2.4.1.2. Predicting number of lines changed per class

The commercial software product QUES (Quality Evaluation System) data is used in this investigation (Wei and Henry, 1993). To predict the maintenance effort, DIT, RFC, WMC, Message Passing Coupling (MPC) Lack of Cohesion in Methods (LCOM).
Data Abstraction Coupling (DAC), number of local methods (NOM), SIZE1) is the traditional line of code metric, which is calculated by counting the number of semicolons in a class. The second size metric (SIZE2) is the total number of attributes and methods of a class. We got an R square value of 0.734. About 73% of the variation in the criterion variable maintenance effort can be explained by the regression model with all predictors Depth of Inheritance Tree (DIT), MPC, RFC, LCOM, DAC, WMC, NOM, SIZE1 and SIZE2. The correlation between predicted maintenance effort and actual maintenance is 0.8590 in the GRNN network and 0.747 in the Ward network. The number of observations is 71. Two tailed probability p values less than 0.001 in both cross-validations shows a high degree of confidence for the successful validations. We conclude that the impact of model prediction is valid in the population.

M.M.T. Thwin, T. Quah’s future research direction aims to estimate software readiness using neural network models. To estimate readiness, three factors will be considered in our future study:

1) How many faults are remaining in the programs
2) How many changes are required to correct the errors and
3) How much time is required to change the programs.

Software metrics concerning with polymorphism, inheritance, complexity, cohesion, coupling, dynamic memory allocation, database operations and size will be used.

2.5. Cohesion Measures for Object-Oriented Software

Object-oriented software is a collection of many classes and each class is a collection of attributes and methods. Software can be said to be of good quality if its classes have maximum cohesion and minimum coupling. Cohesion of a class is an internal software attribute representing the degree to which its elements are bound together. Most of the class cohesion measures proposed above is static in nature. Only a few attempts have been made to measure cohesion of a class at run-time. However, these run-time cohesion measures are direct extensions of the existing static cohesion measures. V. Gupta, J.K. Chhabra, in their paper “Dynamic cohesion measures for object-oriented software, (2010)” [34], made an attempt to define the dynamic cohesion measures for objects as
well as classes from the scratch and the proposed dynamic cohesion measures take into consideration the features of object-orientation like inheritance, polymorphism and dynamic binding. The inclusion of effect of these features in computation of cohesion makes them different and more accurate than the existing static cohesion measures.

The dynamic cohesion metrics proposed in their paper are based on the executable code from which dynamic behavior of applications is obtained. The method to measure object-level dynamic cohesion is to instrument the source code to log all occurrences of interactions among object-members while the application is being executed. The main difference between static and dynamic cohesion metrics is the scope of measurement at which cohesion is being measured. The scope of static cohesion measurement is always the whole class. On the other hand, scope of dynamic cohesion measurement can even be specific to a single object belonging to a class at run-time. In case of dynamic metrics, cohesion is first measured at object level and then cohesion of corresponding class is obtained by aggregating cohesion values of all objects belonging to that class. Moreover, static cohesion metrics attempt to predict the potential interactions that would take place at run-time, whereas dynamic cohesion metrics measure what is actually happening at run-time rather than predicting.

2.5.1. Dynamic Cohesion measurement

In object-oriented systems, attributes and methods are the basic elements of an object or a class. A well-defined theoretical framework that formally defines these elements and depicts the relationships among the elements is the precondition of a well defined cohesion measure. Here, a novel theoretical framework is proposed for characterizing elements and dependence relationships among elements of an object or class. This framework is used to describe relationships of four types:

Notations Used:

\[
\begin{align*}
M(c) & : \text{Set of methods of a class } c \text{ which may be either inherited, or implemented in the class} \\
M_{\text{inh}}(c) & : \text{Set of methods inherited in class } c
\end{align*}
\]
**Optimization Models for Component Selection in Designing of Modular Software System**

\[ M_{\text{IMP}}(c) : \text{Set of methods implemented (including overridden/re-defined methods) in class } c \]

\[ M_{\text{DEF}}(c) : \text{Set of methods defined in class } c \]

\[ M_{\text{REDEF}}(c) : \text{Set of methods re-defined in class } c \]

\[ M_s(c) : \text{Set of special methods in class } c \]

\[ M_n(c) : \text{Set of normal methods in class } c, \text{which are not special methods.} \]

\[ M_R(o) : \text{Set of methods used by an object } o \text{ at runtime} \]

\[ M^R_s(o) : \text{Set of special methods used by object } o \]

\[ M^R_n(o) : \text{Set of normal methods used by object } o \]

\[ A(c) : \text{Set of all attributes whether implemented or inherited in a class } c \]

\[ O(c) : \text{Set of objects of a class } c \text{ created during execution of the program} \]

\[ A_{\text{INH}}(c) \text{ and } A_{\text{IMP}}(c) : \text{Set of attributes inherited and implemented respectively in class } c. \]

\[ A^R(o) : \text{represents the set of attributes used by object } o \text{ at run-time} \]

\[ E^R : \text{Set of elements (attributes and methods) used by object } o \text{ at run-time}, \]

\[ R^R : \text{Set of relations among elements} \]

\[ c = (E^R, R^R) : \text{A object } o, \text{ in an object-oriented system can be defined as a set of its elements (attributes and methods)} \]

**2.5.1.1. Write dependence relation between attributes and methods**

This type of dynamic cohesion exists between attributes and methods of an object when a method of an object writes an attribute of the object during execution of a specific scenario x.
2.5.1.2. Read dependence relation between methods and attributes

This kind of cohesion exists between methods and attributes of an object when a method of an object reads an attribute during execution of a specific scenario \( x \).

\[
DC_{AM} x(o) = \begin{cases} 
0 & \text{if } m=0 \text{ or } n=0 \\
\sum_{i=1}^{m} \sum_{j=1}^{n} \frac{e_{i}^{n}(e_{j}^{n}, e_{j}^{n})}{n,m,n,i,j} & \text{else}
\end{cases}
\]

Where \( e_{i}^{n} \in A(o) \land e_{j}^{n} \in M_{n}(o) \land o \in O(c) \)

2.5.1.3. Call dependence relation among methods

This type of cohesion exists between a pair of methods of an object when a method \( m_{i} \) calls other method \( m_{j} \) of the object during program execution.
Optimization Models for Component Selection in Designing of Modular Software System

\[ DC_{MM} X(o) = \begin{cases} 
0 & \text{if } m = 0 \\
\sum_{i=1}^{n} \sum_{j=m}^{n} \tau_{ij}^{(i)} e_{ij} & \text{where } e_{ij} \in E_{MM}(o) \wedge e_{ij} \in E_{MM}(o) \wedge e \in O(c) \\
1 & \text{if } m = 1 
\end{cases} \]

2.5.1.4. Reference dependence relation among attributes

This category of cohesion exists between attributes of an object when these attributes are referred together in a method of the object during program execution. The reference of a pair of attributes together in a single method induces some dependency between them.

\[ DC_{AA} X(o) = \begin{cases} 
0 & \text{if } m = 0 \\
\sum_{i=0}^{n-1} \sum_{j=m}^{n} \tau_{ij}^{(i)} e_{ij} & \text{where } e_{ij} \in A(o) \wedge e_{ij} \in A(o) \wedge e \in O(c) \\
1 & \text{if } m = 1 
\end{cases} \]

The proposed measures take into account two types of access relationships between methods and attributes, i.e., read access relations and write access relations between methods and attributes. If a method having some logical error writes an attribute, then the value of attribute may also be incorrect. Thus, value of the attribute is dependent on the behavior of the method during write access relationship between methods and attributes. Similarly, if a method reads an attribute that has incorrect value, then behavior of the method may also be erroneous. Though, if method has a logical error, the value of attribute will not be affected by the method reading it. This fact states that the behavior of method is dependent on the value of attribute during read access relationship between
methods and attributes. The proposed metrics account for inheritance and polymorphism present in object-oriented software.

During dynamic cohesion measurement, class (including inherited features) is treated as a single semantic concept. Thus, set of attributes and set of methods of a class (formally defined in the next section) include set of inherited attributes and set of inherited methods, respectively. The concept of polymorphism is relevant only in method invocation type of connections. Since, cohesion is being measured for an object at run-time; polymorphic method invocations are accounted for automatically instead of static method invocations.

This paper proposes new well-defined dynamic cohesion measures. In the definitions of the measures, the elements as well as the relationships among the elements in a class are precisely characterized and the special methods which do not contribute to the cohesiveness of objects and classes are excluded in the measurement. The proposed measures are well-defined cohesion measures which satisfy the four cohesion properties defined by Briand et al. [19] and in comparison with the existing cohesion measures, the proposed measures have the following advantages:

❖ The proposed dynamic cohesion measures are more accurate as they are defined at run-time and take into consideration the actual interactions taking place rather than the potential interactions which may or may not happen as is the case with static cohesion metrics.
❖ The proposed cohesion metrics take inheritance and polymorphism into consideration as the actual targets of polymorphic invocations can only be determined at run-time due to the presence of inherited members of a class.
❖ The scope of measurement of the proposed dynamic cohesion metrics can be specific to a single object or even a scenario.
❖ The limiting of scope of measurement might be useful in testing and impact analysis [3]. Whereas, other existing cohesion metrics are able to measure cohesion up to class level only.
The proposed dynamic cohesion measures are better indicators of external software quality attributes such as change-proneness than the existing static cohesion metrics as proved by the experimental study.

Gupta et. al. in their paper suggests that in future, relationships between the proposed dynamic cohesion metrics and other quality attributes such as fault-proneness could be explored. Moreover, in future work, an alternative approach for selection of values of weights assigned to different relations could be explored to use the four base metrics in predictive models and estimate coefficients of the predictive models, which could be used as weights.

2.6. Optimization of software components selection for component-based software system development

Kwong & Luo in 2010 has published a paper on optimization of software components selection for component-based software system development. To formulate the problem of selecting software components for CBSS development, the following notations are introduced:

- \( M \): Number of software modules
- \( N \): Number of software components
- \( L \): Number of sets of software components
- \( s_{c_i} \): \( i = 1, \ldots, N \) the \( i^{th} \) software component
- \( m_{j} \): \( j = 1, \ldots, M \) the \( j^{th} \) software module
- \( s_k \): \( k = 1, \ldots, L \) a set of alternative software components for the \( k^{th} \) functional requirement of a CBSS. Only one software component in \( s_k \) is selected to implement the \( k^{th} \) requirement
- \( i \in s_k \) denotes that \( s_{c_i} \) belongs to the \( k^{th} \) set
- \( r_{ii'} \): \( i, i' = 1, \ldots, N \) the number of interactions between \( s_{c_i} \) and \( s_{c_{i'}} \). As the coupling and cohesion are undirected relations, \( r_{ii'} = r_{i'i} \).
Optimization Models for Component Selection in Designing of Modular Software System

\[ f_{ij}, \quad i = 1, \ldots, N, j = 1, \ldots, M : \text{the function rating of } sc, \text{ to } m_i \text{, } f_{ij} \text{ are real numbers ranging from 0 to 1} \]

\[ x_{ij}, \quad i = 1, \ldots, N, j = 1, \ldots, M: x_{ij} \text{ is a binary variable, } x_{ij} = 1 \text{ if } sc_i \text{ is selected for } m_j; \text{ otherwise, } x_{ij} = 0 \]

\[ H : \text{a threshold value of } ICD_j \text{ of each module that needs to be set by decision makers. The constraint, } ICD_j \geq H, j = 1, \ldots, M, \text{ needs to be satisfied} \]

Assumptions:

1) Each component has a distinct, known reliability, cost and functional rating.
2) Number of components is equal in each module.
3) At least one component is supposed to get selected from each module.
4) Interaction data for components is exactly same for each module.
5) Interaction cost associated and functional ratings data is set by the software development team.
6) At least one components is reliable from each module
7) The budget is limited.
8) A threshold value of \( ICD_j \) & reliability are set by the decision makers.

The cohesion within the \( j^{th} \) module, \((CI_{IN})_j\), can be formulated as follows:

\[
(CI_{IN})_j = \sum_{i=1}^{N-1} \sum_{i'=i+1}^{N} r_{ij} x_{ij} x_{ij'}
\]

All interactions including cohesion and coupling associated with the \( j^{th} \) module, \((CI_{OUT})_j\), can be expressed as:

\[
CA_j = (CI_{IN})_j (CI_{OUT})_j = \sum_{i=1}^{N-1} \sum_{i'=i+1}^{N} r_{ij} x_{ij} \sum_{j'=1}^{M} x_{ij'}
\]
All interactions including cohesion and coupling of a software system, CA, can be expressed as shown below:

$$CA = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} r_{ij} \sum_{j'=1}^{M} x_{ij'}$$

The sum of cohesions within all modules, CIIN, can be expressed as shown below:

$$CI_{IN} = \sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{j'=1}^{N} r_{ij} x_{ij} x_{ij'}$$

Two objective functions of the optimization problem based on the criteria identified in the last section can be formulated as shown below:

$$\text{Max } ICD = \frac{CI_{IN} / CI_{IN} + CI_{OUT}}{\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} r_{ij} x_{ij} x_{ij}}$$

$$\text{Max } F = \sum_{j=1}^{M} \sum_{i=1}^{N} f_{ij} x_{ij}$$

The objective function (2.1) is formulated based on minimizing coupling among software modules and maximizing cohesion within software modules. The objective function of F is formulated based on maximizing the functional performance of a software system to be developed. Hence, the optimization model of software components selection for CBSS development can be formulated as follows:
Optimization Models for Component Selection in Designing of Modular Software System

\[
\text{Max } ICD = \sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{m=1}^{j} r_{ij} x_{ij} x_{ij} \frac{1}{\sum_{i=1}^{N} \sum_{j=1}^{M} r_{ij} x_{ij} x_{ij}}.
\]

Max \( F = \sum_{i=1}^{M} \sum_{j=1}^{N} f_{ij} x_{ij} \)

Subject to

\[
\sum_{i=1}^{M} \sum_{j=1}^{N} r_{ij} x_{ij} x_{ij} + 1 \geq H \quad j, j' = 1, \ldots, M \quad \ldots \quad (2.2)
\]

\[
\sum_{i=1}^{M} \sum_{j=1}^{N} r_{ij} x_{ij} x_{ij} \sum_{j=1}^{N} x_{ij} x_{ij} \geq 1 \quad \ldots \quad \ldots \quad (2.3)
\]

\[
\sum_{i=1}^{N} x_{ij} = 1 \quad \ldots \quad \ldots \quad (2.4)
\]

\[
\sum_{j=1}^{M} x_{ij} = 1 \quad j = 1, 2, \ldots, M \quad \ldots \quad (2.5)
\]

Where Equation (2.3) denotes that only one software component can be selected from a set of alternative software components for a particular software module. Constraint (2.4) refers to a software module that contains at least one software component.

2.7. Optimization of software components selection for component-based software system development with cost considerations

In the model discussed in section 2.6 the components are selected without considering the cost of the software modules. In the COTS based development system, acquiring a component either developed in-house or purchased from the market there is a cost associated with that. For any organization, budgets are not unlimited. Developing the software with cost optimization in mind is an important goal of any organization.
When we develop the application with finite budgets, it is necessary to utilize the resources in an optimized way without compromising on quality and reliability. In order to achieve this we need to have optimization models which will minimize the cost and maximize the reliability and also ensures that the application runs without fail.

With this in mind, a new constraint on cost is proposed in this section with an intention of minimizing the cost of selecting the modules is proposed.

The proposed optimization model of software components selection for CBSS development can be formulated as follows:

Let the $c_{ij}$ is the cost of Component and $C$ is the total Budget available. The cost constraint becomes

$$\sum_{i=1}^{N} \sum_{j=1}^{M} c_{ij} x_{ij} \leq C \quad (2.6)$$

In the light of the above discussion, the model described in section 2.6 can be reformulated as follows:

**Model 1**

$$\text{Max ICD} = \frac{\sum_{i=1}^{M} \sum_{j=1}^{N} r_{ij} x_{ij}}{\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} r_{ij} \sum_{j=1}^{M} x_{ij} \sum_{j=1}^{M} x_{ij}}$$

$$\text{Max} \quad F = \sum_{i=1}^{M} \sum_{j=1}^{N} f_{ij} x_{ij}$$
Subject to

\[ X \in S \left\{ x_{ij} \in [0,1] \right\} \quad i = 1, \ldots, N; \quad j = 1, 2, \ldots, M \]

\[
\sum_{i=1}^{N-1} \sum_{j=1}^{M} r_{ii} x_{ij} x_{ij} + 1 \geq H 
\]

\[
\sum_{i=1}^{N-1} \sum_{j=1}^{M} r_{ij} x_{ij} \sum_{j'=1}^{M} x_{ij'} \geq H 
\]

\[
\sum_{i=1}^{N} \sum_{j=1}^{M} x_{ij} = 1
\]

\[
\sum_{i=1}^{N} x_{ij} \geq 1 \quad j = 1, 2, \ldots, M
\]

\[
\sum_{i=1}^{N} \sum_{j=1}^{M} c_{ij} x_{ij} \leq C
\]

Constraint described in equation (2.7) above will ensure that the cost of the selected modules will not exceed the total budget.

Let

\[ f(x) = \sum_{j=1}^{M} \sum_{i=1}^{N} \sum_{i'=i+1}^{N} r_{ii'} x_{ij} x_{i'j} \]

\[ g(x) = \sum_{i=1}^{N-1} \sum_{i=1}^{N} \sum_{j=1}^{M} r_{ij} x_{ij} \sum_{j'=1}^{M} x_{ij'} \]

Further the above problem can equivalently be written as following mathematical programming problem using Bhatia (1997).

\[ \text{Max} \quad F_1(x) = f(x) - g(x) \]

\[ \text{Max} \quad F_2(x) = \sum_{j=1}^{M} \sum_{i=1}^{N} f_{ij} x_{ij} \]

Subject to \( X \in S \)

Vector maximization problem can be written as:
Finding Properly Efficient Solution

**Definition 1 [81]:** A feasible solution \( X^* \in S \) is said to be an efficient solution for the below problem if there exists no \( X \in S \) such that \( F(X) \geq F(X^*) \) and \( F(X) \neq F(X^*) \)

**Definition 2 [81]:** An efficient solution \( X^* \in S \) is said to be a properly efficient solution for the problem (P4.3.1) if there exist \( \alpha > 0 \) such that for each \( r \)

\[
F_i(X) - F_i(X^*) / (F_j(X) - F_j(X^*)) < \alpha \quad \text{for some } j \text{ with } F_j(X) \leq F_j(X^*) \quad \text{and} \quad F_i(X) > F_i(X^*) \quad \text{for } X \in S.
\]

Using Geoffrion’s scalarization [30] the problem reduces to

\[
\text{Maximize} \quad \lambda_1 F_1 - \lambda_2 F_2 \\
\text{Subject to} \quad X \in S \\
\lambda_1 + \lambda_2 = 1 \quad \lambda_1, \lambda_2 \geq 0
\]

The original problem was a Bi-Criteria problem. Using Geoffrion’s scalarization the problem gets converted to a single objective problem by attaching the weights to both the objectives.

### 2.8. Numerical illustrations

In this section, a numerical example is illustrated to illustrate the proposed methodology of optimizing the selection of software components for CBSS development.

In this example a software system is decomposed into three modules, m1, m2 and m3. Ten software components (sc1–sc10) that are available in markets make up four sets of
alternative software component (s1–s4) for each module. Say for example a local software system supplier planned to develop a financial system for small and medium-size enterprises. 4 functional requirements of the system were identified, namely, Facsimile/Fax. (R'1), Encryption (R'2), Credit Card Authorization (R'3), Ecommerce/Accounts (R'4). The software system development team of the company has defined three software modules, business related module (m1), security module (m2) and assistance module (m3), that the financial software system needs to contain. The business-related module mainly provides the functions of e-commerce and financial reporting. The security module mainly provides the functions of encryption and authorization while the assistance module is to provide auxiliary functions for the system such as fax and software automatic updating.

Individual functional requirements and their corresponding alternative software components, as well as the function ratings of software components corresponding to the software modules, are shown in Table 1. The function ratings describe the degrees of functional contributions of the software components towards the software modules. The function ratings range from 0 to 1 where 1 refers to a very high degree of contribution while 0 indicates zero degree of contribution. Table 2 shows the degrees of interaction among the software components. The range of the degrees is 1–10. The degree ‘1’ means a very low degree of interaction while the degree ‘10’ refers to a very high degree of interaction. The degrees of interaction were determined based on the team’s judgment. The detailed data set we are using here is given in Table 1, Table 2, Table 3 & Table 4.
### Functional requirements

<table>
<thead>
<tr>
<th>Sets of alternative software components</th>
<th>Software components</th>
<th>Component</th>
<th>Module m1</th>
<th>Module m2</th>
<th>Module m3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Business</td>
<td>Security</td>
<td>Assistance</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s1</td>
<td>s2</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>Facsimile/Fax,R'1</td>
<td>S1</td>
<td>OpalVOIP</td>
<td>1</td>
<td>0.68</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HylaFAX</td>
<td>1</td>
<td>0.22</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Faxman</td>
<td>0.97</td>
<td>0.77</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HylaPEx</td>
<td>1</td>
<td>0.23</td>
<td>0.57</td>
</tr>
<tr>
<td>Encryption, R'2</td>
<td>S2</td>
<td>CryptoXpress CF</td>
<td>0.55</td>
<td>0.94</td>
<td>1</td>
</tr>
<tr>
<td>Credit Card Authorization, R'3</td>
<td>S3</td>
<td>IBiz E</td>
<td>1</td>
<td>0.22</td>
<td>0.45</td>
</tr>
<tr>
<td>eCommerce/Account, R'4</td>
<td>S4</td>
<td>Symphero</td>
<td>1</td>
<td>0.45</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shopforma t Account manager</td>
<td>0.99</td>
<td>0.25</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PayPal</td>
<td>0.95</td>
<td>0.34</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Account services manager</td>
<td>1</td>
<td>0.57</td>
<td>0.45</td>
</tr>
</tbody>
</table>

**Table 1** Functional ratings of software components

### Sets of Alternative components

<table>
<thead>
<tr>
<th>Sets of Alternative components</th>
<th>Components</th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sc1</td>
<td>sc2</td>
<td>sc3</td>
<td>sc4</td>
</tr>
<tr>
<td>s1</td>
<td>sc1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>sc2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>sc3</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>sc4</td>
<td>10</td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>s2</td>
<td>sc5</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>sc6</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>s3</td>
<td>sc7</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>sc8</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>sc9</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>sc10</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 2** Interactions among software components for all three modules m1, m2 & m3
Optimization Models for Component Selection in Designing of Modular Software System

<table>
<thead>
<tr>
<th>Module</th>
<th>m1</th>
<th>Module</th>
<th>m2</th>
<th>Module</th>
<th>m3</th>
</tr>
</thead>
<tbody>
<tr>
<td>c11</td>
<td>1099</td>
<td>c12</td>
<td>1068</td>
<td>c13</td>
<td>1010</td>
</tr>
<tr>
<td>c21</td>
<td>1060</td>
<td>c22</td>
<td>1022</td>
<td>c23</td>
<td>1063</td>
</tr>
<tr>
<td>c31</td>
<td>1000</td>
<td>c32</td>
<td>1015</td>
<td>c33</td>
<td>1019</td>
</tr>
<tr>
<td>c41</td>
<td>1005</td>
<td>c42</td>
<td>1023</td>
<td>c43</td>
<td>1020</td>
</tr>
<tr>
<td>c51</td>
<td>1055</td>
<td>c52</td>
<td>1094</td>
<td>c53</td>
<td>1093</td>
</tr>
<tr>
<td>c61</td>
<td>1000</td>
<td>c62</td>
<td>1022</td>
<td>c63</td>
<td>1045</td>
</tr>
<tr>
<td>c71</td>
<td>1000</td>
<td>c72</td>
<td>1045</td>
<td>c73</td>
<td>1094</td>
</tr>
<tr>
<td>c81</td>
<td>1094</td>
<td>c82</td>
<td>1025</td>
<td>c83</td>
<td>1000</td>
</tr>
<tr>
<td>c91</td>
<td>1064</td>
<td>c92</td>
<td>1034</td>
<td>c93</td>
<td>1030</td>
</tr>
<tr>
<td>c101</td>
<td>1000</td>
<td>c102</td>
<td>1057</td>
<td>c103</td>
<td>1045</td>
</tr>
</tbody>
</table>

Table 3 Budget Data Set

The normalized data set for fixed weights approach (Geoffrion, 1968) for the objective function ICD is as follows:

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sc2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sc3</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sc4</td>
<td>1</td>
<td>0.8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sc5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sc6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sc7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sc8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sc9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sc10</td>
<td>1</td>
<td>1</td>
<td>0.3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4 Interactions among software components for all three modules m1, m2 & m3

In this example, $W_f$ is the weight of functional performance, $W_i$ is the weight of the ICD. The weights are considered as the value of $W_f = 0.5$, $W_i = 0.5$. H (threshold value) is set as 0 for all the module constraints. Hence, an optimization model of the optimal selection of software components for the system development is established. Solving Problem discussed above with the above mentioned data set. we get the solution as follows:-
Results of selection of software components for modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Sc21, Sc61</th>
<th>Sc52, Sc93</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective function value</td>
<td>2.227508</td>
<td></td>
</tr>
<tr>
<td>ICD</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>6.39</td>
<td></td>
</tr>
</tbody>
</table>

*Table 4 Output*

The solution mentioned is a local optimal solution for the above mentioned Integer Non-Linear Programming Formulation for the given data set, solved on the software Lingo (11). By Table 4 we can conclude that Sc21, Sc61 got selected for Module 1, i.e., Module 1 is able to fulfill all four functional requirements (S1 & S3); similarly Sc52 & Sc93 got selected for Module 2 & Module 3, i.e., these Modules are able to fulfill only one functional requirement (S2 & S4). So over all for fulfilling S1 functional requirement we have selected one components Sc21(j=1); for second functional requirement S2 we have only Sc52(j=2) : for S3 we have only Sc61(j=1) & last but not the least for S4 functional requirement we have Sc93(j=3).

2.9. Conclusion

In this paper, a methodology of selecting software components for CBSS development is described. Compared with the previous studies on CBSS development, a modified way of measuring the cohesion and coupling and the function ratings of various software components for CBSS development is used in this research along with & satisfying budget constraint simultaneously.

Based on the proposed methodology, an optimization model can be formulated to perform the selection of software components for the software modules of a CBSS. To obtain an optimal/near optimal solution for the selection of software components, Bhatia et. al. approach is followed to solve the optimization problem. An example of a financial system design was used to illustrate the methodology in this paper.

Vijaya Mohan Ghantasala  
Sri Krishnadevaraya University
However, this methodology involves some subjective judgments from software development teams, such as the determination of the scores of interaction, and the function ratings under consideration of a fixed budget. In this regard, the fuzzy set theory could be introduced to deal with the fuzziness caused by subjective judgments.