Quality Measurement of Software By Using Coupling Metrics

G. Deepti, G.Deepali
G.I.M.T, Department of Computer science & engg,Kurukshetra,India
Email: deepti_taylor17@yahoo.co.in
G.I.M.T, Department of Computer science & engg,Kurukshetra,India
Email:deepali_gupta2000@yahoo.com

Abstract— Software Modularization includes the concept of procedural, object based and objects oriented languages. The metrics presented and evaluated in this paper are both ‘pure’ object–oriented metrics (Metrics for pure object oriented programming) and metrics proposed for object–oriented programming (Object-based programming). Concept of software modularization is divided in three components i) Use of API ii) Use of Non API iii) Use of shared variables.

Index Terms— Metrics /measurement, Software Modularization, API (Application Programming Interface), Coupling, Function Call Traffic.

I. INTRODUCTION
Modularization refers to the division of software into subunits. According to the programming language use subunits can be defined as a macro, sub-routine, procedure, function or a module.[5]

A. Software Modularization
Software modularization deal with different modules. These modules can be integrated directly or indirectly[6] The number of intensity between two modules affects its quality. Modularization i.e. API or non API also assigned by different weight ages. These relations are represented in the form of coupling and cohesion in an object oriented language.[3]

B. S/W Modularization Principles
As Software modularization is an important characteristic to Software Engineering. These are [4].
Principle 1: Associated to Purpose of Similarity
• Module Coherence maximization on the Basis of Similarity and Singularity Purpose.
• Module Coherence maximization on the basis of Commonality of goals.
Principle 2: Associated to Module Encapsulation.
• API-Based Inter-module Call Traffic maximization .
• Non-API Based Inter-module Call Traffic minimization.

II. OBJECTIVES
1. The main objective of this Project is to measure the quality of modularization of object-oriented projects by Coupling-based Structural metrics.
2. Goal is to analyses or measure how the code is framed for the particular software and Applying

DOI: 03.AETS.2013.3.137_33
© Association of Computer Electronics and Electrical Engineers, 2013
Software metrics to show the result.

III. PRESENT WORK

We studied various types of software metrics[1] as well as coupling metrics[2] and at the present time a new set of metrics that cater to the principles is defined. Coupling-based structural metrics provide various measures of the function-call traffic through the API’s of the modules that relates to the overall function-call traffic. For that we have find the following four factors.

A. Module Interaction Index

This metric calculates how effectively a module’s API functions are used by the other modules in the system. Assume that a module m has n functions \( \{ f_1, \ldots, f_n \} \), of which the n1 API functions are given by the subset \( \{ f_1^a, \ldots, f_{n1}^a \} \). Also assume that the system S has \( m_1 \cdots m_M \) modules. We now express Module Interaction Index (MII) for a given module and for the entire software system S by

\[
MII(m) = \sum_{fa \in \{ f_1^a, \ldots, f_{n1}^a \}} K_{ext}(f^a) = 0, \text{ when no external calls made to m}, \quad (1)
\]

\[
MII(S) = \frac{1}{M} \sum_{i=1}^{M} MII(m_i).
\]

B. Non-API Function Closedness Index

We now analyze the function calls from the point of view of non-API functions. Recall that the module encapsulation principles P2 also require minimization of non-API-based inter-module call traffic. Ideally, the non-API functions of a module should not expose themselves to the external world. We measure the amount of this traffic using a metric “Non-API Function Closedness Index,” or NC. Consent to \( F_m^a \) and \( F_m^{na} \) represent the set of all functions, the API functions, and the non-API functions, respectively, in module m. Ideally, \( F_m = F_m^a + F_m^{na} \). The deviation from this constraint is measured by the metric.[4]

\[
NC(m) = \frac{|F_m^{na}|}{|F_m| - |F_m^a|} = 0 \text{ if there are no non-API functions} \quad (2)
\]

\[
NC(S) = \frac{1}{M} \sum_{i=1}^{M} NC(m_i).
\]

C. API Function Usage Index

This index determines what fraction of the API functions exposed by a module is being used by the other modules. Suppose that m has n API functions and let us say that \( n_j \) number of API functions is called by another module \( m_j \). Also assume that there are k modules \( m_1, \ldots, m_k \) that call one or more of the API functions of module m. We may now formulate an API function usage index in the following manner.[4]

\[
APIU(m) = \frac{\sum_{j=1}^{k} n_j}{n \times k} = 0 \text{ if } n = 0, \quad (3)
\]

\[
APIU(S) = \frac{1}{M_{apiu}} \sum_{i=1}^{M_{apiu}} APIU(m_i).
\]

D. Implicit Dependency Index

Implicit Dependency Index (IDI), is constructed by recording for each module the number of functions that write to global entities, with the condition that such global entities are accessed by functions in other...
modules. Larger this count is in relation to the size of the inter-module traffic which consists of explicit function calls, greater the effect of implicit dependencies.\[4\]

For each module $m_i$, we use the notation $D_g(m_i, m_j), i \neq j$, to denote the number of dependencies created when a function in $m_i$ writes to a global entity that is subsequently accessed by some function in $m_j$. Let $D_f(m_i, m_j), i \neq j$, denote the number of explicit calls made by all the functions in $m_i$ to any of the functions in $m_j$. We claim that the larger $D_g$ is in relation to $D_f$, the worse the state of the software system. So the metric is defined as follows:\[4\]:

$$\text{IDI}(m) = \frac{\sum_{m \in C(m)} D_g(m, m_j)}{\sum_{m \in C(m)} (D_g(m, m_j) + D_f(m, m_j))}$$

$$\text{IDI}(S) = \frac{1}{M} \sum_{i=1}^{M} \text{IDI}(m_i)$$

IV. RESULTS AND DISCUSSION

Implementation is the stage of the project when the theoretical design is turned out into a working system. So it can be measured to be the most significant stage in achieving a successful new system and in giving the user, assurance that the new system will work and be effective. We develop a tool in java in this work which explain as follows:

A. Code Analyser

Code Analyzer is a software tool which reads any java code which uses the modulization notion. In this code analyzer tool we create four steps in menu bar as shown in figure below the steps are: file, FindMetricFactor, Apply Formula and output the working simulation result of the following code analyzer tool is in details shown step by step. It is shown as under in the figure 1.

![Figure 1: Code Analyzer](image)

B. Opening the Input Project

The below fig.2 shows the input of the project

![Figure 2: Input Project Data](image)
C. Input Project After Parsing
The below fig.3 shows the output of the project after the parsing of the code.

D. Finding Metrics Factors
The below fig.4 shows the metrics factor
E. Applying Formula
The below fig.5 shows how to apply the formula.

F. Resulting Metrics
The below fig.6 shows how the resulting metrics. In this we have designed three process classes and these all are integrated in the main class. The modularization achieve here is the symphony type in which independent classes are accessed in the same program. In the above application we are interacting with in the application as well as the library is being used. It means together the API and non-API modularization subsist in the above representation. As we can feel this type of sonata where there is weak bounding between these classes. The given chart also represents the same where the modularization index show low bounding of modularization. But still both the API and non API interfacing is being used in the application [4].

V. CONCLUSION
The proposed system is capable to estimate the software quality of the modules of the software. The proposed work provides a conceptual and practical framework for the measurement of various attributes like inheritance, polymorphism, coupling, cohesion, and depth of inheritance etc. In this work set of principle for modularization of code are designed and metrics is proposed that characterize software in relation to those principles. The structural metrics are driven by the use of API. These metrics are required since otherwise it would be possible to declare a software system as malformed but well modularized. This work can be extended by:
1. Applied to distributed testing
2. Use of the component based testing and Designing.
3. Used to check the level of Integration
4. Object reusability can also be applied.

REFERENCES