ISSN No. 0976-5697

Volume 2, No. 4, July-August 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

An Architecture Insensitive Approach for measuring the Quality of Software Modularization

Vikas Verma*
M.Tech Scholar,
CSE Department, U.I.E.T., Kurukshetra University,
Kurukshetra, India
vik.ver86@gmail.com

Sona Malhotra
Assistant Professor,
CSE Department,
U.I.E.T., Kurukshetra University,
Kurukshetra, India

Abstract: Today most of the software's designed by a user follow the concept of object orientation or the modularization. Software Modularization is more vast term that includes the concept of procedural, object based and objects oriented languages. The notion of modularization uses different concepts like use of classes, objects, procedures, functions etc. The complete software modularization concept is divided in three main components i) Use of API ii) Use of Non API iii) Use of shared variables. In this paper, a tool based representation is proposed where a graphical view is available to select the software project to perform all kind of modularization metrics on it, including cohesion, coupling and interface complexity etc.

Keywords: OOS, MII, APIU, API, Non API , NC, IDI.

I. INTRODUCTION

Modularization refers to the division of software into subunits. These subunits in a programming language can be described as a macro, sub-routine, procedure, module and function. A module is a collection of data structure and functions that together offer a well defined task. Modularization is used in software due to these reasons: modularization provides abstraction, modularization allows multiple programmers to work on a problem, modularization allows you to reuse your work, and modularization makes it easier to identify structures.

Software modularization is the reorganization of the software system as a set of modules with well defined APIs, sticking to the set of modularity principles. A module can be used in different aspects like a function, procedure, class, package, library or the component. The most common use of software modularization is code reusability. Other then this, code reusability gives the better appearance to a program.

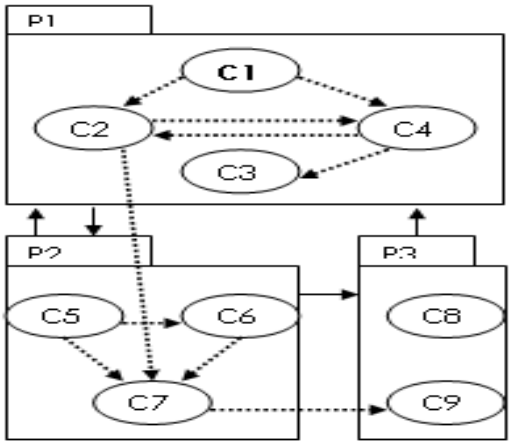

Figure 1 Example of Modularization

Figure1 shows modularization with an example. It consists of nine classes distributed over three packages. A package represents an entity that contains only classes. Every class related to other classes through a set of dependencies. A dependency might be either method call, class access or class inheritance. Every dependency is either internal if it is related to two classes belonging to the same package, or external if not. Figure 1.1 shows dependencies between package and classes. The dotted arrow symbol denote the package internal dependency and package external dependency and solid symbol denote the inter package connection.

II. EXISTING SYSTEM

Till now a lot of work is already done on the same concept of software modularization. But still there are some flaws while working with it. Most of the available system provide modularization for a specific software type i.e. respective of object based, object oriented etc. There also exist some systems that work on the concept of object orientation. But there is no such approach that can work all kind of software modularization including the object based, procedural and object oriented software systems. Theoretical validation implies conformance to a set of agreed upon principles.

III. PROPOSED SYSTEM

In this paper, a tool based solution is proposed, i.e. Code analyzer, which measures the quality of modularizations of an object oriented software system (OOS). It uses any java software code as input. It computes the quality of modularization and gives output in graphical form. Code analyzer uses a set of four object oriented software metrics. These metrics are coupling based structural metrics which provides various measure of the function call interchange through the API of the module in relation to the overall function-call interchange. These metrics are given here module interaction index (MII), Non –API function closeness index (NC), API function usage index (APIU), and Implicit dependency Index (IDI).

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 $\{f_1 \cdots f_n\}$, of which the n1 API functions are given by the subset $\{f_1^a \cdots f_{n1}^a\}$. Also assume that the system S has $m_1 \cdots m_M$ modules. Santonu Sarkar, et.al [1] express Module Interaction Index (MII) for a given modules and for the entire software system S by:

$$MII(m) = \frac{\sum fa \in \{f_1^a .... f_{n1}^a\} Kext(f^a)}{K_{ext}(m)}$$

= 0, when no external calls made to m,

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

.

MII (m) = $\dfrac{\sum fa \in \{fa..... fa\} Kext(fa)}{Kext(m)}$

B. Non-API Function Closedness Index:

Here the function calls from the point of view of non-API functions are analyzed. 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. In reality, however, a module may exist in a semi modularized state where there remain some residual inter-module function calls outside the API's. (This is especially true of large legacy systems that have been partially modularized.) In this intermediate state, there may exist functions that participate in both inter-module and intra-module call traffic. The extent of this traffic is measured using a metric that we call "Non-API Function Closedness Index," or NC. Let $F_{m'}^a$ and $F_{m'}^{na}$ represent the set of all functions, the API functions, and the non-API functions, respectively, in module m. Ideally, $F_m = F_m^a + F_m^{na}$. But since, in reality, we may not be able to conclusively categorize a function as an API function or as a non-API function, this constraint would not be obeyed. The deviation from this constraint is measured by the metric [1].

$$NC(m) = \frac{|F_m^{na}|}{|F_m| - |F_m^a|}$$

= 0 if there are no non – API functions,

$$NC(S) = -\frac{1}{M}\sum_{i=1}^{M} NC(m_i)$$

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. When a big, monolithic module presents a large and versatile collection of API functions offering many different services, any one of the other modules may not need all of its services. That is, any single other module may end up using only a small part of the API. The intent of this index is to discourage the formation of such large, monolithic modules offering services of disparate nature and encourage modules that offer specific functionalities. 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 $m_1 .... m_k$ that call one or more of the API functions of module m. An API function usage index has been formulated in the following manner [1].

$$APIU(m) = \frac{\sum_{j=1}^{k} n_j}{n * k}$$

= 0 if n = 0,

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

,

D. Implicit Dependency Index:

An insidious form of dependency between modules comes into existence when a function in one module writes to a global variable that is read by a function in another module. The same thing can happen if a function in one module writes to a file whose contents are important to the execution of another function in a different module. And the same thing happens when modules interact with one another through database files. Such inter-module dependencies are referred as implicit dependencies.

Detecting implicit dependencies often requires a dynamic runtime analysis of the software. Such analysis is time consuming and difficult to carry out for complex business applications, especially applications that run into millions of lines of code and that involve business scenarios that can run into thousands, each potentially creating a different implicit dependency between the modules. Here, we propose a simple static-analysis-based metric to capture such dependencies. This metric, which is called the Implicit Dependency Index (IDI), is constructed by recording for each module the number of functions that write to global entities (such as variables, files, databases), with the specifications that such global entities are accessed by functions in other modules. The larger this count is in relation to the size of the inter-module traffic consisting of explicit function calls, the greater the insidiousness of implicit dependencies.

For each module mi, the notation $D_g(m_i, m_j), i \neq j,$ is used to denote the number of dependencies created when a function in mi writes to a global entity that is subsequently accessed by some function in m_j let $D_g(m_i, m_j), i \neq j,$ denote the number of explicit calls made by all the functions in mi to any of the functions in m_j.The larger D_g is in relation to D_j, the worse the state of the software system. Therefore the metric is defined as:

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

= 1 when C (m) =θ,

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

Santonu Sarkar, et.al [1] explained that the leaf nodes of the directory hierarchy of the original source code to be the most fine-grained functional modules. All the files (and functions within) inside a leaf level directory were

considered to belong to a single module. In this manner, all leaf level directories formed the module set for the software. After that we apply Coupling-based Structural Metrics as follows [1].

IV. MODULAR DESCRIPTION OF CODE ANALYZER

The modular description of the code analyzer is given here below, it shows the functionality of the tool:

a. List of Modules
b. Getting input
c. Code Parsing.
d. Finding Application metadata.
e. Storing into Database.
f. Applying Metrics.
g. Graphical Representation.

Figure 2: UML diagram of the proposed System

This uml diagram represents step by step working of the Code Analyzer. The Java Net Beans IDE 6.8 is used to run the code analyzer tool.

It accepts the coding as input, then the module parsing is done on the given coding. Once modules are identified then the metrics factors are computed to obtain the API and Non-API functions and the shared data. The computed information is stored in database. The metrics calculations are performed using the formulae given above. Finally the result in graphical form is displayed in the Code Analyzer Window.

V. RESULTS

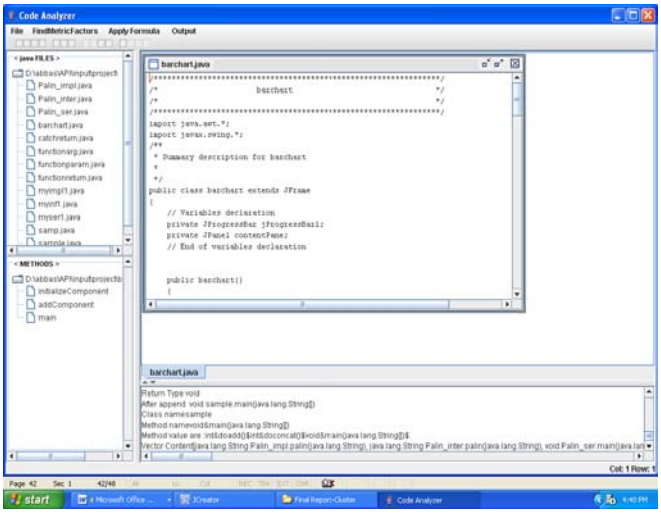

Figure3 The operation of Code Analyzer

The implementation of the Code analyzer can be illustrated using an example,

Example:

```
class Student
{
int rollno;
void getrollno(int r)
{
      rollno=r;
}
void showrollno()
{
      System.out.println("Rollno="+rollno);
}
}
class Test extends Student
{
float part1,part2;
void getmarks(float m1,float m2)
   {
      part1=m1;
      part2=m2;
  }
void showmarks()
   {
      System.out.println("Marks1="+part1);
      System.out.println("Marks2="+part2);
   }
}
interface Sports
{
float wt=6.0f;
void show();
}
class Result extends Test implements Sports
{
float total;
public void show()
{
      System.out.println("Sports weightage ="+wt);
 }
void display()
{
      total=part1+part2+wt;
      showrollno();
```

```
        showmarks();
        show();
        System.out.println("Total Obtain Marks="+total);

 }
 }
public class Main {
public static void main(String[] args)
{
Result r1=new Result();
r1.getrollno(101);
r1.getmarks(55.0f,85.25f);
r1.display();
}
}
```

Corresponding this example code, the output is generated by Code Analyzer, which is shown in the figure below:

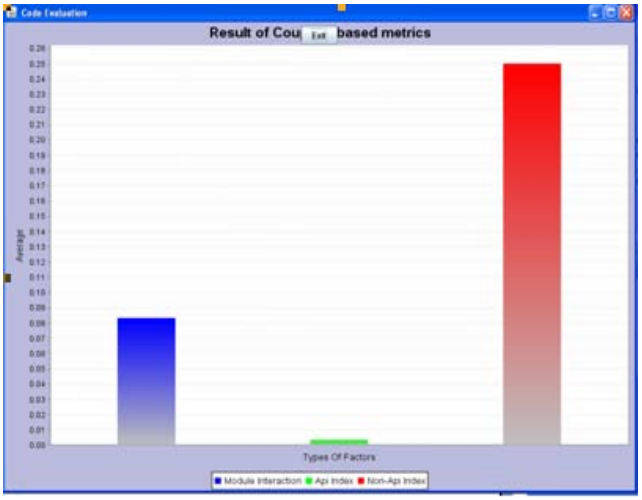

Figure 4. Output corresponding to the example 1

In this output corresponding the example1, we have defined few classes and the interfaces to achieve the modularization. In this application no separate libraries being used because of this the API level is very low. The above application gives the modularization up the level of inheritance and the interfacing because of this modularization index is better.

VI. CONCLUSION AND EXTENSION

The proposed system is capable to estimate the software quality for any software system irrespective to the software architecture. It works for all kind of software systems including the object based system, object oriented systems, procedural system, open source software system etc. The basic complexity estimated are the functional metrics, interface complexity etc.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. and their significance in the development and maintenance of object-oriented systems. The most common use of software modularization is code reusability. The code reusability is responsible for enhancing the quality of the software and also it gives the better appearance to a program.

A. Future Work:

An estimation of the s/w quality is done based on the concept of s/w modularization .In this research extension in term to test individual component used in an application, and use of the component based testing when different component are being used to get the code and object reusability can also be applied.

VII. REFERENCES:

[1] Santonu Sarkar, Girish Maskeri Rama, and Avinash C.kak "API-Based and Information-Theoretic metrics for Measuring the Quality of Software Modularization", IEEE Trans.on Software Eng., Vol.33, No.1 January 2007.

[2] Kaner Cem, Senior Member, IEEE, and Walter P. Bond, "Software Engineering Metrics: What Do They Measure and How Do We Know?"10th International Software Metrics Symposium, 2004.

[3] E.B. Allen, T.M. Khoshgoftaar, and Y. Chen, "Measuring Coupling and Cohesion of Software Modules: An Information- Theory Approach," Proc. Seventh Int'l Software Metrics Symp. (METRICS '01), pp. 124-134, 2001.

[4] Abreu, F. B. and R. Carapuca, 'Candidate Metrics for Object-Oriented Software within a Taxonomy Framework', J. of Systems & Software, 26, pp87-96,

[5] Abreu, F. B. and W. Melo. 'Evaluating the impact of object-oriented design on software quality', in Proc. 3rd International Software Metrics symposium. Berlin, Germany: IEEE Computer Society Press,

[6] Achee, B. L. and D. L. Carver. 'Evaluating the quality of reverse engineered object-oriented designs', in Proc. 2006 IEEE Aerospace Conference. Proceedings. IEEE, New York, USA.

[7] Ammann, M. H. and R. D. Cameron. 'Measuring program structure with inter-module metrics', in Proc. Eighteenth Annual International Computer Software and Applications Conference (Compsac 94). IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 1994.

[8] Jasmine K.S., and Vasandha R., "DRE - A Quality Metric For Component Based Software Products", World Academy Of Science, Engineering And Technology 34, 2007, Pages 48-51.

[9] Kaner Cem, Senior Member, IEEE, and Walter P. Bond, "Software Engineering Metrics: What Do They Measure and How Do We Know?"10th International Software Metrics Symposium, 2004.

[10] Ammann, M. H. and R. D. Cameron. 'Measuring program structure with inter-module metrics', in Proc. Eighteenth Annual International Computer Software and Applications Conference (Compsac 94). IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 1994.