Metrics for measuring the quality of object oriented software modularization

Priya Walde
Student
M.I.T. College of Engineering,
Aurangabad

S.V.Kulkarni
Asst. Professor
M.I.T. College of Engineering,
Aurangabad

ABSTRACT

Our ongoing effort, from which we draw the work reported here, is focused on the case of reorganization of legacy software, consisting of millions of line of non-object oriented code, that was never modularized or poorly modularized to begin with. We can think of the problem as reorganization of millions of lines of code residing in thousands of files in hundreds of directories into modules, where each module is formed by grouping a set of entities such as files, functions, data structures and variables into a logically cohesive unit. Furthermore, each module makes itself available to the other modules (and to the rest of the world) through a published API.

Keywords- Application Programming Interface(API), Modularization,Function Call Traffic, Non API,Metrics.

1. INTRODUCTION

1.1 Objective:

- The main objective of this paper is to measure the quality of modularization of object-oriented projects by Coupling-based Structural metrics.
- Goal is to analyses or measure how the code is framed for the particular software and Applying Software metrics to show the result[1].

1.2. Methodology:

Much work has been done during the last several years on automatic approaches for code reorganization. Fundamental to any attempt at code reorganization is the division of the software into modules, publication of the API (Application Programming Interface) for the modules, and then requiring that the modules access each other's resources only through the published interfaces[2,3].

Our ongoing effort, from which we draw the work reported here, is focused on the case of reorganization of legacy software, consisting of millions of line of non-object oriented code, that was never modularized or poorly modularized to begin with. We can think of the problem as reorganization of millions of lines of code residing in thousands of files in hundreds of directories into modules, where each module is formed by grouping a set of entities such as files, functions, data structures and variables into a logically cohesive unit. Furthermore, each module makes itself available to the other modules (and to the rest of the world) through a published API[1,2]. The work we report here addresses the fundamental issue of how to measure the quality of a given modularization of the software.

1.3. Modularization:

In this context "module" is considered to be a responsibility assignment rather than a subprogram. The modularizations include the design decisions which must be made before the work on independent modules can begin. Quite different decisions are included for each alternative, but in all cases the intention is to describe all "system level" decisions (i.e. decisions which affect more than one module).

2. SYSTEM ANALYSIS

2.1 Analysis of Existing System:

In the existing system large number of coding are divided into only two modules, so each module contains large number of coding. So in the existing system performance analysis takes more time as well as not more accurate.

Some of the earliest contributions to software metrics deal with the measurement of code complexity and maintainability . From the standpoint of code modularization, some of the earliest software metrics are based on the notions of coupling and cohesion . Low intermodule coupling, high intramodule cohesion, and low complexity have always been deemed to be important attributes of any modularized software. The above-mentioned early developments in software metrics naturally led several researchers to question their theoretical validity. Theoretical validation implies conformance to a set of agreed-upon principles and these principles are usually stated in the form of a theoretical framework**.**

2.2 Process of Proposed System:

Modern software engineering dictates that a large body of software be organized into a set of modules. A module captures a set of design decisions which are hidden from other modules and the interaction among the modules should primarily be through module interfaces. In software engineering parlance, a module groups a set of functions or subprograms and data structures and often implements one or more business concepts. This grouping may take place on the basis of similarity of purpose or on the basis of commonality of goal.

In the Proposed system we considered 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

2.2.1. Coupling-Based Structural Metrics

Starting with this section, we will now present a new set of metrics that cater to the principles. We will begin with coupling-based structural metrics that provide various measures of the function-call traffic through the API's of the modules in relation to the overall function-call traffic. For that we have find the following four factors.[4,5]

1. Module interaction index

2. Non-API Function Closed ness Index

3. API Function Usage Index

4. Implicit Dependency Index

2.2.2 Plagiarism Detection:

Plagiarism detection is the process of locating instances of plagiarism[9] within a work or document. due to this technique, if any user wants to cut&paste the source code from the website that website is immediately identified by using plagiarism detection technique.For this technique we use axmedis framework[7,8].

3. IMPLEMENTATION

The implementation stage involves careful planning, investigation of the existing system and it's constraints on implementation, designing of methods to achieve changeover and evaluation of changeover methods.

Implementation is the process of converting a new system design into operation. It is the phase that focuses on user training, site preparation and file conversion for installing a candidate system

3.1 Modular Description:

List of Modules

1. Getting input.
2. Code Parsing.
3. Finding Application metadata.
4. Storing into Database.
5. Applying Metrics.
6. Graphical Representation.

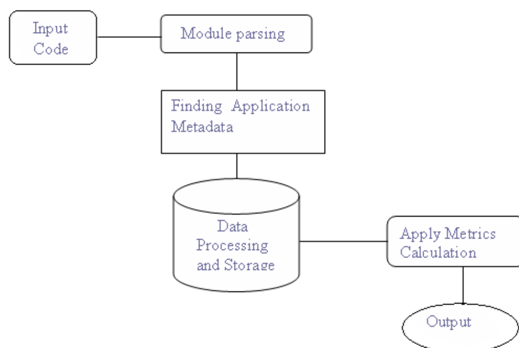

Fig 1. Data Flow Diagram

3.2 *Mathematical Hypothesis*:

3.2.1 IFAC (Index factor for API function calls):

This metric calculates how effectively a Module's API functions are used by the other Modules in the system. Suppose module has n API functions and let's say that nj numbers of API functions are called by another module mj. Also assume that there are z numbers of modules from module1 to module z that calls one or more API functions of module[1,4].

IFAC (module) = (n1+n2+…nz) / (n * z)

= 0, if n i. e. number of API function is zero.

If we assume that module api is the total number of modules having more than zero API functions. Then

IFAC (system) = SUM [IFAC (module)i] / module api,

Where i = 1 to module api…………………………….……..…..(1)

The maximum value of this metric IFAC (system) will be 1, depending on the focus and nature of the modules with similar purpose.

2. IFNC (Index factor for non API function calls):
Let us represent API function as function api and non API functions as function napi for given module.

Then total function will be function = function api+ function napi

Total number of modules is M.

IFNC (module) = function napi/ (function - function api)

= 0, if the non API functions are zero.

IFNC (system) = SUM [IFNC (module)i] / M,

Where i =1 to M……………………… (2)

In good modularized object oriented software, functions will be either API or non API type of functions. And non API functions are not used outside the module.

Then function - function api will be equal to functionnapi. So that IFNC (module) = 1.

Here sometimes the value of the IFNC (module) can be between 0 and 1.

3.2.2 IFMC (Index factor for non API function calls):

This metric calculates the index factor for module communication and how well API functions of modules are used by the other modules in the system for communication. Assume that a module has n functions from 1 to n, of which the n1 API functions are given by the subset {f1 api ……. F n1 api}. Cext is used to denote the total number of external calls coming from the other modules. It is a java file as module. Also assume that system has m1 to mi modules. Total number of modules is M. Index Factor for module communication (IFMC) for a given module and for the entire software system by[1,2]

IFMC (module) = {SUM [Cext (fapi)]} / Cext (module),

Where fapi is in range from f1 api to fn1 api

= 0, when there are No external calls made to the particular module

IFMC (system) = {SUM [IFMC (module)i]} / M,

Where i is in range from 1 to M……………………….. (3)

3.2.3 IFID (Index factor for Implied Dependency):

When function in one module is writing to a global variable that is in use by another module then there is indirect dependency. There can be many events where this kind of dependency occurs in program. Generally in large enterprise application made in object oriented language may have complex source for the hidden dependency between the modules.Let us say that dependency is denoted by Dglobal (modulea, moduleb) where a≠b. In which dependency will be there when module a tries to write in to global entity (e.g. files, variables etc) at the same time moduleb is also trying to work on the same entity.

Let us say that D function (module a, module b) where a≠b. In which the calls are made by the functions in module a to functions in module b. Then the Index factor for the implied dependency for module will be given by[2],

IFID (module) = SUM [Dfunction (modulea, moduleb)] / SUM [Dfunction (modulea, moduleb) + D global (modulea,moduleb)],for all implied dependencies

= 1, when D global (modulea, moduleb) = 0

IFID (system) = IFID (module) / M,

Where M is total number of module from 1 to M……….(4)

From this metric we can say that there should be very less or none implied dependencies in the system.

Generally the value of IFID (system) is equal to 3.

4. CONCLUSION

We reported on two types of experiments to validate the metrics. In one type, we applied the metrics to two different versions of the same software system. Our experiments confirmed that our metrics were able to detect the improvement in modularization in keeping with the opinions expressed in the literature as to which version is considered to be better. The other type of experimental validation consisted of randomizing a well-modularized body of software and seeing how the value of the metrics changed. This randomization very roughly simulated what sometimes can happen to a large industrial software system as new features are added to it and as it evolves to meet the changing hardware requirements. For these experiments, we chose open-source software systems. For these systems, we took for modularization the directory structures created by the developers of the software. It was interesting to see how the changes in the values of the metrics confirmed this process of code disorganization.

5. ACKNOWLEDGEMENT

I take this opportunity to thank a number of individuals whose guidance and encouragement were of enormous help to us while working on this paper. First and foremost i would like to thank our project guide and mentor Prof S.V.Kulkarni for her valuable advice, guidance and help in searching the topic and innovative suggestions for the improvement of the same. I also highly thankful to our Head of Department **Prof K.V.Bhosale** and our Principal **Dr.R.G.Tated** for their constant support and encouragement

6. REFERENCES

[1] S. Sarkar, A.C. Kak, and N.S. Nagaraja, “Metrics for Analyzing Module Interactions in Large Software Systems,” Proc. 12th Asia- Pacific Software Eng. Conf. (APSEC ’05), pp. 264-271, 2005.

[2] Java 2: The Complete Reference, Fifth Edition by HerbertSchildt

[3] Sarkar S., Kak A. C. and Rama G. M, “API-Based and Information-Theoretic Metrics for measuring the Quality of SoftwareModularization” IEEE Trans. Software Eng., vol. 33, no. 1, pp.14-30.

[4] Chidamber S. R. and Kemerer C. F.,“A Metrics Suite for Object Oriented Design,” IEEE Trans. Software Eng., vol. 20, no. 6, pp.476-493, June 1994.

[5] Pfleeger S. and Fenton N., Software Metrics: A Rigorous and Practical Approach. Int’l Thomson Computer Press, 1997.

[6] Pressman R. S., Software Engineering: A Practitioners Approach, 6/e, TATA McGRAW HILL, 2005, pp 461-670.

[7] Stein, Benno; Koppel, Moshe; Stamatatos, Efstathios "Plagiarism Analysis, Authorship Identification, and Near-Duplicate Detection PAN’07"

[8] Potthast, Martin; Stein, Benno; Eiselt, Andreas; Barrón-Cedeño, Alberto; Rosso, Paolo (2009), "Overview of the 1st International Competition on Plagiarism Detection"

[9] Stein, Benno; Meyer zu Eissen, Sven; Potthast, Martin (2007), "Strategies for Retrieving Plagiarized Documents", Proceedings 30th Annual International ACM SIGIR Conference