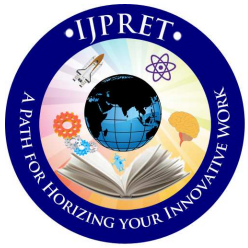
Research Article *ISSN: 2319-507X*
Pragati Chowhan, IJPRET, 2013; Volume 1(8): 723-732 *IJPRET*



INTERNATIONAL JOURNAL OF PURE AND APPLIED RESEARCH IN ENGINEERING AND TECHNOLOGY

A PATH FOR HORIZING YOUR INNOVATIVE WORK

QUALITY MEASUREMENT OF OBJECT ORIENTED SOFTWARE MODULARIZATION

PRAGATI D. CHOWHAN[1], PROF. S. S. DHANDE[2]

1. Department of Information Technology Sipna COET, SGBAU, Amravati (MH), India.
2. Department of Information Technology Sipna COET, SGBAU, Amravati (MH), India.

Accepted Date:

27/02/2013

Publish Date:

01/04/2013

Keywords

Application Programming Interface;
Modularization;
Dependency;
Function-call.

Corresponding Author

Ms. Pragati D. Chowhan[1]

Abstract

In today's environment how to measure the quality of software modularization is main issue. This paper proposed a new set of metrics which measure the quality of modularization of an Object Oriented Software System. These metrics characterize the software from a variety of perspectives such as structural, architectural and notions like similarity of purposes and Module Encapsulation Principles. Structural refers to intermodule coupling-based notions. The notion of API (Application Programming Interface) is employed as the basis for the structural metrics. Some of the important support metrics include those that characterize each module on the basis of the similarity of purpose of the services offered by the module. Here coupling-based structural metrics are used that provide various measures of the function-call traffic through the API of the modules in relation to the overall function-call traffic. The functional-call traffic refers to the inter-modular interaction. It is universally accepted that quality of the software modularization is improved when interaction between modules of system is through published APIs only. We tested our metrics on some large legacy-code business applications. The metrics can be validated from the results obtained on human-modularized versions of the software.

Introduction

The software development process is quite difficult and modularization can make it more complicated. It is a challenge to measure the quality of object oriented software modularization. Software developers develop their software with some standard specification, but important issue is how to measure the quality of software modularization. Modularization of object oriented code is distribution of the software into modules and these modules should communicate with each other through some API. Furthermore, each module makes itself available to the other modules (and to the rest of the world) through a published API. Obviously, after software has been modularized and the API of each of the modules published, the correctness can be established by checking function-call dependencies at compile time and at runtime. If all intermodule function-calls are routed through the published API, the modularization is correct. The quality of modularization has more to do with partitioning software into more maintainable modules on the basis of the cohesiveness of the service provided by each module. More properly modularized software is also easy for maintenance work and it can help the developer.

In our work we are considering the object oriented java language code for defining metrics and modularization. In java the code is residing in directories of some modules and java files are considered as modules. One java file may consist of number of classes but one public class [4]. Classes contain the elements such as methods, interfaces, variables etc. This code contains many files with thousands of line of code so we are using code parser to analyze the modularity of the code. Here a set of metrics are proposed which measure the interactions between the different modules of a software system. It is important to realize that metrics that only analyze intermodule interactions cannot exist in isolation from other metrics that measure the quality of a given partitioning of the code. Each module would still be much too large from the standpoint of code maintenance and code extension. Here the main objective is to measure the quality of modularization of object-oriented projects by Coupling-based Structural metrics.

Need of Metrics

Now days much software's are developed by the developers which are very big in code size. So generally to maintain the quality of the code, developers need to distribute the code in small pieces. But dividing the software is a crucial task as it can lead to various problems of intermodule communication therefore this modularized code should also be checked for the quality. There are problems in removing the errors of non modularized code. Particularly in object oriented software development developer needs to use a lots of object oriented concepts which may introduced the inter dependency of the various units of the software like Inheritance. Software metric is a measure of some property of a piece of software or its specifications. Therefore software metrics suite is needed [2]. We are concentrating on the same issue and providing the software metrics for this modularized object oriented code.

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. Cohesion is measured as the ratio of the number of internal function-call dependencies that actually exist to the maximum possible internal dependencies, and Coupling is measured as the ratio of the number of actual external function-call dependencies between the two subsystems to the maximum possible number of such external dependencies. Low intermodule coupling, high intramodule cohesion, and low complexity have always been deemed to be important attributes of any modularized software. In general, one of the goals of the software designers is to keep the coupling in an OO system as low as possible. Classes of the system that are strongly coupled are most likely to be affected by changes and bugs from other

classes; these classes tend to have an increased architectural importance and thus need to be identified. Coupling measures help in such endeavours, and most of them are based on some form of dependency analysis, based on the available source code or design information. The number of dimensions captured by the measures is lower than the number of proposed coupling measures [8], which reflects the fact that many of these measures are based on comparable hypothesis and use similar information for computation.

Proposed System Architecture

The project aims to developing a software modularization quality testing using Java Technology. Object oriented software can be tested for its modularization quality. Modules of the proposed system are explained below:

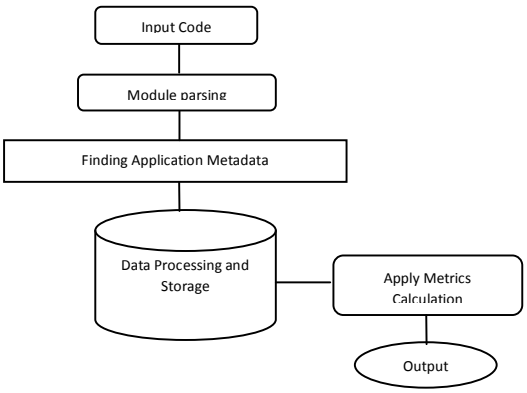


Figure1: Flow diagram of Proposed System

Getting Input: User or tester will import file/project to our tool. The desired project which is to be tested is given as an input to this module. This is done in Java using File Input Stream.

Code Parsing: The tool will partition the source code by its file type. Also in this module the file will be partition in to the form which is easy for applying our metrics. String Tokenizer and File Name Filter are used for this purpose.

Finding Application metadata: In this module the tool will find the size/total number of lines in the project. After that calculate what are the functions/methods are involved in this project. How many methods call from other modules, how many modules call other modules and what are all the functions from other module and find how many classes and modules in a given file. **Applying Metrics:** This module is heart of our project. Here we are going to calculate the quality of software based on the modules, function and size. There are

three types of metrics used to calculate the quality of software.

Proposed Metrics

The following metrics are proposed based on OOP's concepts which are largely used for the software development. The non-object oriented metrics given by Sarkar et. al. [1] is a base for our work. API functions are the functions only which can be get called outside the module and non API functions are not called outside the module. In our implementation we are going to check if a function calling is found in another module or class then it will be API function and if not found then such functions will be considered as isolated and non API functions. The measurement technique is applying the metrics [3]. The proposed metrics for object oriented code are as follows:

Module Interaction Index (MII)

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 $\{f_1 .... f_n\}$ of which the n1 API functions are given by the subset $\{f_1^a .... ,f_n^a\}$. Also assume that the system S has $m_1 \ldots m_M$ modules. We now express Module Interaction Index (MII) for a given module m and for the entire software system S by,

$$MII(m) = \frac{\sum f^a \varepsilon \{f_1^a \ldots f_{n1}^a\} K_{ext}(f^a)}{K_{ext}(m)}$$

= 0, when no external calls made to m

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

MII measures the extent to which a software system adheres to the module encapsulation principles. Since these API functions are meant to be used by the other modules, the internal functions of a module typically would not call the API functions of the module. Ideally, all the external calls made to a module should be routed through the API functions only and the API functions should receive only external calls. For a module m increases as more and more inter module calls are routed through the API functions of m. We obviously have MII(m)→1 in the ideal case when all the inter module calls are routed through the API functions only. By the same argument,

MII(S) should also be close to 1 in the ideal case.

Non-API Function Closeness Index (NC)

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 intermodule function calls outside the API's. In this intermediate state, there may exist functions that participate in both intermodule and intramodule call traffic. We measure the extent of this traffic using a metric that we call Non-API Function Closeness Index, or NC.

Let F_m , F^a_m & F^{na}_m represent the set of all functions, the API functions, and the non-API functions, respectively, in module m. Ideally, $F_m = F^a_m + F^{na}_m$. 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,

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

= 0, if there are no non-API functions

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

Since a well-designed module does not expose the non-API functions to the external world and all functions are either API functions or non-API | F_m | - | F^a_m | would be equal to | F^{na}_m |.Therefore, NC (m) = 1 for a well- designed module. Otherwise, the value for this metric will be between 0 and 1.

API Function Usage Index (APIU)

This index determines what fraction of the API functions exposed by a module is being used by the other modules. 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 a large, monolithic module 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 calls one or more of the API functions of module m. We

may now formulate an API function usage index in the following manner:

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

= 0, if n = 0

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

where we assume that there are M_{apiu} number of modules that have nonzero number of API functions. This metric characterizes, albeit indirectly and only partially, the software in accordance with the principles that come under the Similarity of Purpose rubric. So making the modules more focused with regard to nature of services provided by the API functions would push the value of this metric close to its maximum, which is 1.

Experiment

The experimental validation of the metrics is made challenging by the fact that it is difficult to find examples of object-oriented software that are modularized and that have published APIs for each of the modules. It is relatively straightforward to label the functions in the different directories of the software systems as API or non-API functions on the basis of the relative frequencies of the call traffic from within a directory and from the other directories. To verify the usefulness of our metrics, we not only need to show that the numbers look good for well-written code; we also need to demonstrate that the numbers yielded by the metrics become progressively worse as the code becomes increasingly disorganized. In order to make such a demonstration, starting from the original code, we created different modularized versions of the software. The Figure1 Flow Diagram of the Proposed System shows how the experiments done in the proposed system. In the proposed system the coding is given as input then the module parsing is done on the given coding. Once modules are identified then the metrics factors are found in order to obtain the API and Non API functions and the shared data. Now the identified information is stored in the database. The metric calculations are performed using the above formula. Finally the output chart is generated for the metrics calculation.

Experimental Results

The following figures are the snap shots of the proposed system. Figure1 Snapshot of Module Parsing is used to find the number of modules involved in the software. Then the functions in the different directories of the software systems are labelled as API or non-API functions.

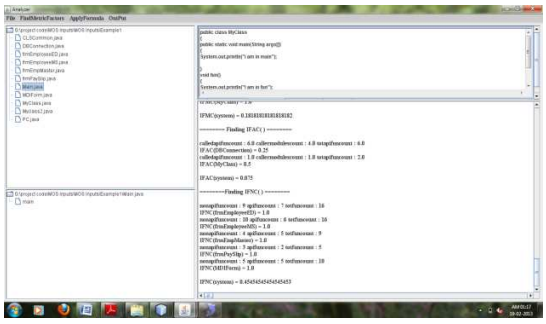


Figure1: Snapshot of Module Parsing

Once the functions are identified then the metrics calculations are performed using the above metrics formulas. The following Figure 2. Snapshot of Module Description in the DB shows how the data stored in the database which gives the details about the

module description.

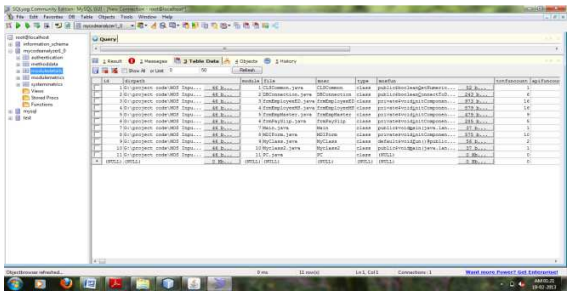


Figure2: Snapshot of Module Description in the DB

Conclusion

In the above we have given a set of design principles for code modularization and proposed a set of metrics that characterize software in relation to those principles. The structural metrics are driven by the notion of API - a notion central to modern software development. These metrics are essential since otherwise it would be possible to declare a malformed software system as being well-modularized. Here more emphasis is given on Module 1 of the project. Defining metrics for object oriented software modularization. We have proposed three metrics for calculating the Module Interaction Index, API Function Usage Index and Non-API Function Closeness Index**.** Hence it helps to test modularization quality of object oriented software. Also test whether how much

improvement in new version and previous version of same software. It can be used while developing any software.

References:

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

2. Allen, E. B., Khoshgoftaar, T. M., and Chen, Y., "Measuring coupling and cohesion of software modules: an information-theory approach", in Proceedings of 7th International Software Metrics Symposium (METRICS'01), April 4-6 2001, pp. 124-134.

3. Pfleeger S. and Fenton N., Software Metrics: A Rigorous and Practical Approach. Int'l Thomson Computer Press, 1997.

4. Schildt H., The Complete Reference Java2, Fifth Edition, TATA McGRAW

5. HILL, 2002, pp 13-54.

6. Briand, L., Wust, J., and Louinis, H., "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems", in Proc. of IEEE International Conf. on Software Maintenance, Aug. 30 - Sept. 03 1999, pp. 475-482.

7. Briand, L. C., Daly, J., and Wust, J., "A Unified Framework for Coupling Measurement in Object Oriented Systems", IEEE Transactions on Software Engineering, vol. 25, no. 1, January 1999, pp. 91-121.

8. Briand, L. C., Devanbu, P., and Melo, W. L., "An investigation into coupling measures for C++", in Proc. of International Conference on Software engineering (ICSE'97), Boston, MA,

9. May 17-23 1997, pp. 412 - 421.

10. Briand, L. C., Wust, J., Daly, J. W., and Porter, V. D., "Exploring the relationship between design measures and software quality in object-oriented systems", Journal of Systems and Software, vol.51, no. 3, May 2000, pp. 245-273.

11. Chidamber, S. R. and Kemerer, C. F., "Towards a Metrics Suite for Object Oriented Design", in Proceedings of OOPSLA'91, 1991, pp. 197-211.

12. Chidamber, S. R. and Kemerer, C. F., "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, vol. 20, no. 6,

13. 1994, pp. 476-493.

14. El-Emam, K. and Melo, K., "The Prediction of Faulty Classes Using Object-Oriented Design Metrics", NRC/ERB-1064, vol. NRC 43609, November 1999.

15. Gall, H., Jazayeri, M., Krajewski, J., "CVS Release History Data for Detecting Logical Couplings", 6th International Workshop on Principles of Software Evolution (IWPSE'03) Sept. 1 - 2, 2003, pp. 13 - 23.