Comparative Study of the Modularization Techniques used for an OOS System to Achieve Quality

1Ms. Pragati D. Chowhan, 2Prof. S. S. Dhande
1Department of Information Technology Sipna COET, SGBAU, Amravati (MH), India  
2Department of Information Technology Sipna COET, SGBAU, Amravati (MH), India

Abstract- The major issue faced by the developers in today’s environment is how to measure the quality of modularized software. Here a new set of metrics are proposed which measure the quality of modularization of an Object-Oriented Software (OOS) System. A set of design principles are proposed to capture the notion of modularity and metrics are defined based on the principles. For this we have the coupling-based structural metrics which is used to measure the function-call traffic through the API’s of the modules in relation to the overall function-call traffic. It is now widely accepted that the overall quality of a large body of software is enhanced when module interactions are restricted to take place through the published API’s for the modules. The metrics can be tested on some large legacy-code business applications. The metrics can be validated from the results obtained on human-modularized versions of the software.

Keywords— API (Application Programming Interface), Coupling, Function Dependencies, Modularization.

I. INTRODUCTION

Object oriented software is of more recent vintage than the old-style procedural code. Nonetheless, there exist many commercial object-oriented applications that show the same signs of aging as the legacy procedural code. So what may have started out as a well-modularized architecture may have decayed into a system of interdependent modules that are difficult to maintain and extend on an individual basis. As is well known in the literature, whereas the main factor responsible for the disorganization of the procedure a legacy code is the unregulated function-call traffic between what might have started out as well-partitioned modules, for object-oriented legacy code, there exist multiple avenues that can lead to a high state of software disorganization. These include a module extending a class defined in another module, a class in one module using an instance of a class in another module as an attribute or as a parameter in a method definition, the implementation code for a method in a class in one module calling on a method defined for a class in another module, etc. The metrics we propose seek to characterize the quality of a given modularization with respect to each such mechanism for code degradation. To be sure, there exists a significant body of literature dealing with the metrics for object-oriented software. But, as we point out in the next section, much of this previous work considers a module and a class as synonymous concepts. Even when the contributors have not explicitly stated their metrics on a per-class basis, many of the core metrics can often be reformulated straightforwardly on a per-class basis. Our vision of the future is dictated by the fact that software for many commercial applications now runs into millions of lines of code. When such code is object-oriented, as is increasingly the case, the granularity of a class is often at much too low a level to serve as a unit for software modularization. A module in large object-oriented software may contain hundreds of classes. As in the software of the past, the purpose of a module is still to provide some focused functionality to the rest of the world. Nonetheless, due to its complexity, it may take a large number of classes to implement that functionality. Since many of the classes in the sort of modules we have mentioned above are intended solely for the internal functioning of a module, it makes no sense to simply declare all of the public methods for all of the classes as the API of a module.

It makes a lot more sense to construct module-level APIs with only those public methods that are intended to be used by the outside world. The metrics we proposed are for such large object-oriented software. Our metrics measure the quality of the modularization of the software by measuring the extent to which all inter module method call traffic is directed through the APIs of the modules, the extent to which the module are free of the linkages created by inheritance and associations, etc. Our metrics also measure the quality of the APIs from the standpoint of the cohesiveness of the services provided, etc. But the API-based metric is for non-object-oriented systems only. The inter module couplings created by inheritance, containment, access control, polymorphism, encapsulation, etc, are strictly outside the purview of the metrics in what follows, review of the previous work in metrics for object-oriented software. In order to accommodate the increasingly popular notion of plug-in, there is a brief overview of two different types of APIs, one a Service API and the other an Extension API. A Service API is the traditional API for software modules. An Extension API describes the services that must be provided by an external plug in module.

II. RELATED WORK

The earliest contributions to software metrics deal with the measurement of code complexity and maintainability based on the complexity measures. The standpoint of code modularization, some of the earliest software metrics are based on the notions of coupling and cohesion. Low inter-module coupling, high intra-module cohesion and low complexity have always been deemed to be important attributes of any modularized software. With regard to modularity, Briand et al. have given us a generic formalization of such fundamental notions as module and system, and such metrical notions as coupling, cohesion, and complexity. Coupling, cohesion, and complexity metrics that measure the various quality attributes of OO software. Schwanke also characterizes modules on the basis of
function-call dependencies. If a function A calls function B, then, in the approach used by Schwane, both A and B presumably belong to the same module.

The lines of formulating metrics in the context of developing code modularization algorithms, Mancoridis et al. have used a quantitative measure called Modularization Quality (MQ) that is a combination 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. The contribution by Wen and Tzerpos adds a new consideration to the calculation of cohesion-coupling metrics for the purpose of software clustering. These authors first try to isolate what they refer to as “omnipresent objects,” these being heavily used objects and functions in a software system, before the calculation of the more traditional coupling-cohesion metrics.

Coupling in software has been linked with maintainability and existing metrics are used as predictors of external software quality attributes such as fault-proneness, impact analysis, ripple effects of changes, changeability, etc. Many coupling measures for object-oriented (OO) software have been proposed, each of them capturing specific dimensions of coupling. This paper presents a new set of coupling measures for OO systems — named conceptual coupling, based on the semantic information obtained from the source code, encoded in identifiers and comments. The case study shows that the conceptual coupling captures new dimensions of coupling, which are not captured by existing coupling measures; hence it can be used to complement the existing metrics.

Many maintenance tasks require the developer to measure directly or indirectly several attributes and assess properties of the software system under evolution. A variety of measures are proposed by researchers to assist developers in getting more complete views of the software. Coupling is one of the properties with most influence on maintenance as it has a direct effect on maintainability. Proposed coupling measures are used in tasks such as impact analysis [5], [21], assessing the fault-proneness of classes [22], fault prediction [11], [13], re-modularization [1], identifying of software components [16], design patterns [3], assessing software quality [8], etc.

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 endeavors, 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. We proposed a new set of coupling measures, which formulates and captures new dimensions of coupling, i.e., conceptual coupling, based on the semantic information shared between elements of the source code. Our measures can be classified as measuring the strength of conceptual similarities among methods of different classes. The measures are based on using information retrieval (IR) techniques to model and analyze the semantic information embedded in software (i.e., through comments and identifiers). The conceptual coupling can be used to augment existing measures, especially in tasks such as impact analysis and change propagation, as existing models [5] do not capture all the ripple effects of changes in existing software. They also have direct application in reverse engineering tasks like re-modularization.

Coupling measurement is a very rich and interesting body of research work, resulting in many different approaches using structural coupling metrics [7], [9], [10], [17], dynamic coupling measures [4], evolutionary and logical coupling [16], [38], coupling measures based on information entropy approach [2], coupling metrics for specific types of software applications like knowledge-based systems [14], and more recently systems developed using aspect-oriented approach [23]. The structural coupling metrics have received significant attention in the literature. These metrics are comprehensively described and classified within the unified framework for coupling measurement [6]. The best known among these metrics are CBO (coupling between objects) and CBOJ [9], [10], RFC (response for class) [9] and RFC∞ [10], MPC (message passing coupling) [18], DAC (data abstraction coupling) and DAC1 [18], ICP (information-flow-based coupling) [17], the suite of coupling measures by Briand et al. (IFCAIC, ACAIC, OCAIC, FCAEC, etc) [7]. Other structural metrics like Ce (afferent coupling), Ca (afferent coupling), COF (coupling factor), etc. are also overviewed in [6].

Many of the coupling measures listed above are based on method invocations and attribute references. For example, the RFC, MPC, and ICP measures are based on method invocations only. CBO and COF measures count method invocations and references to both methods and attributes. The suite of measures defined by Briand et al. [7] captures several types of interactions between classes like class-attribute, class-method, as well as method-method interactions. The measures from the suite also differentiate between import and export coupling as well as other types of relationships like friends, ancestors, descendants etc. Dynamic coupling measures [4] were introduced as the refinement to existing coupling measures due to gaps in addressing polymorphism, dynamic binding, and the presence of unused code by static structural coupling measures. Another important family of coupling measures derives from the evolution of software system in contrast to structural coupling which is determined by program analysis or dynamic coupling which is obtained by executing the program. These are called evolutionary couplings among parts of the systems which are determined by the common changes or co-changes [24]. Recently, several specialized coupling metrics were proposed for different types of software systems. They are coupling metrics for knowledge-based systems [14] as well as coupling metrics for aspect-oriented programs [23]. Existing work on software clustering [15], [19] uses the concept of semantic similarity between elements of the source code [20], which stands at the foundation of the conceptual coupling. Using information retrieval methods for coupling measurement, our approach to coupling measurement is based on the philosophy that elements (classes) of (OO) software systems are related in
The main objective of this Project is to measure the quality of modularization of object-oriented projects by Coupling-based Structural metrics. Goal is to analyze or measure how the code is framed for the particular software and applying software metrics to show the result. The major intention of the system being developed is to make the computer user enable to measure the quality of object oriented software being developed. It’s a system that modularizes the newly implemented system or software by using several modularizing processes.

A. Relationship of Previous Work to the Present Goals

We believe that many of these previous approaches suffer from shortcomings with regard to the goals we have in mind. The approaches that carry out software partitioning purely on the basis of function call dependencies (or file-dependencies that are derived from function-call dependencies) are obviously not suitable for meeting our goals. Function call dependencies are semantically orthogonal to the groupings on the basis of cohesiveness of service. To elaborate, in code partitioning on the basis of function-call dependencies, if a function A calls a function B, then both A and B must belong together in the same module. But using function call dependencies as the sole basis for modularization runs counter to the very spirit of what is meant by modules in modern code writing. Modules pull together functions not because they call one another, but because they serve similar purposes with respect to the rest of the software. For example, the number of intra-module function calls in the java.util module (referred to as a package in the Java parlance) is minimal. The main reason for why the functions in the java.util module belong together is because they provide very similar services to the rest of the software.

B. Module Interaction and Other Related Metrics

Modern software engineering dictates that large software be organized along the following lines:

1. The software system should consist of a set of modules where each module is a collection of data structures and functions that together offer a well- defined service. Any associated functions in the same module should cohere on the basis of similarity-of-service as opposed to on the basis of function call dependencies.
2. The modules should interact with one another only through the exposed API functions. With regard to code maintenance, this is desirable for isolating faults and rectifying them quickly.
3. Whenever feasible, the modules should be organized in a hierarchical manner in a set of layers. A layer should only be aware of the layers below it (that is, function calls are only made to the lower layers) and should not be aware of the layers above it.
4. Modules should be independently testable and releasable. These considerations have led us to formulate the following metrics for measuring the quality of module interactions. Given the importance of this issue, we certainly plan to take up testability related issues in a future research contribution.

Additionally, as was stated in the introduction, metrics that focus solely on the interactions between the modules cannot exist in isolation from the metrics that measure other qualities of code modularization. Therefore, the set of metrics shown below includes those that are needed to simultaneously report these other attributes.

C. Notion of Modularity

Based on underlying Principles in software engineering parlance, a module groups a set of functions or subprograms and data structures and often implements one or more business concepts. In a well-organized system, only the interface elements are visible to other modules. The modules promote encapsulation (i.e., information hiding) by separating the module’s interface from its implementation. The module interface expresses the elements that are provided by the module for use by other modules. In a well-organized system, only the interface elements are visible to other modules. On the other hand, the implementation contains the working code that corresponds to the elements declared in the interface. In modern parlance, such a module interface is known as its API (Application Programming Interface). It is now widely accepted that the overall quality of a large body of software is enhanced when module interactions are restricted to take place through the published API’s for the modules. The two principles are:

P1. Principle Related to Similarity of Purpose

Principle Related to Similarity of Purpose The structures used for representing knowledge and any associated functions in the same module should cohere on the basis of similarity-of-service as opposed to, say, on the basis of function call dependencies. Obviously, every service is related to a specific purpose. We present the following principle as coming under the “Similarity of Purpose” rubric:

- Maximization of Module Coherence on the Basis of Commonality of Goals.

P2. Module Encapsulation Principles

As mentioned earlier, encapsulating the implementation code of a module and requiring that the external world interact with the module through its published APIs are now a widely accepted design practice. We now state the following modularization principles that capture these notions:

- Maximization of API-Based Inter module Call Traffic and
- Minimization of non-API-Based Inter module Call Traffic.

The Table1 “Relationship between Principles & Metrics” show how the principles & metrics connected to each other.

<table>
<thead>
<tr>
<th>Modularization Principle</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1. Maximization of Module Coherence on the Basis of Commonality of Goals</td>
<td>APIU</td>
</tr>
<tr>
<td>P2. Maximization of API-Based Intermodule Call Traffic</td>
<td>MII</td>
</tr>
<tr>
<td>P2. Minimization of non-API-Based Intermodule Call Traffic</td>
<td>MII, NC</td>
</tr>
</tbody>
</table>

Table1: Relationship between Principles & Metrics
D. Coupling-Based Structural Metrics

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 are:

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 \( \{ f_1^a, ..., f_{n1}^a \} \). Also assume that the system S has m1...m M modules. We now express Module Interaction Index (MII) for a given module m and for the entire software system S by

\[
MII(m) = \frac{\sum f_1^a \cdot f_1 \cdot f_m \cdot K_{ext}(f_1^a)}{K_{ext}(m)}
\]

\[= 0, \text{when no external calls made to m} \]

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

MII measures the extent to which a software system adheres to the module encapsulation principles P2. 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)

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. In this intermediate state, there may exist functions that participate in both inter module and intra module call traffic. We measure the extent of this traffic using a metric that we call “Non-API Function Closeness Index,” or NC.

Let \( F_m \), \( F_m^a \) & \( F_m^o \) represent the set of all functions, the API functions, and the non-API functions, respectively, in module m. Ideally, \( F_m = F_m^a + F_m^o \).

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,

\[
NC(m) = \frac{|F_m^o|}{|F_m| - |F_m^a|}
\]

\[= 0, \text{if there are no non-API functions} \]

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

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 \[ |F_m^a| \] - \[ |F_m^o| \] would be equal to 0. Hence, 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 nj number of API functions are called by another module mj. Also assume that there are k modules m1 ... mk 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:

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

\[= 0, \text{if n = 0} \]

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

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 (P1) 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.

Experimental validation of metrics

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. 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.

Advantages of proposed System

- For easy maintenance and analyzing each file is divided into single module.
- Metrics is applied to different version of a same software system.
- Any object oriented software can be tested. It is standalone application so no need of internet connection.
IV. CONCLUSION
Here we have set of design principles for code modularization and a set of proposed metrics that characterize software in relation to those principles. These metrics are essential since otherwise it would be possible to declare a malformed software system as being modularized. As an extreme case in point, putting all of the code in a single module would yield high values for some of the API-based metrics, since the modularization achieved would be functionally correct but on the other hand it would be highly unacceptable.

REFERENCES

¹Ms. Pragati D. Chowhan student at Sinpa College of Engineering & Technology, Amravati. B.E. (IT), M.E. (IT) pursuing, Sinpa College of Engineering & Technology, Amravati. Sant Gadge Baba Amravati University, Amravati (MH), India.

²Prof. S. S. Dhande Associate Professor at Sinpa College of Engineering & Technology, Amravati. B.E. (CSE) M.E. (CSE) from Sant Gadge Baba Amravati University, Amravati. Currently pursuing PHD in faculty of Engineering & Technology in the area of OODatabase Management System from Sant Gadge Baba Amravati University, Amravati (MH), India. She has published many papers at National as well as International level. She is a member of ISTE, IE, and IETE India.