Theoretic Metrics for Measuring the Quality of Software

Ashutosh Lahariya, Aman Jain, Rosedeep Singh, Rachana Nemade

Abstract—We present in this paper a new set of metrics that measure the quality of modularization of a non-object-oriented software system. We have proposed a set of design principles to capture the notion of modularity and defined metrics centered around these principles. These metrics characterize the software from a variety of perspectives: structural, architectural, and notions such as the similarity of purpose and commonality of goals. (By structural, we are referring to intermodule coupling-based notions, and by architectural, we mean the horizontal layering of modules in large software systems.) We employ the notion of API (Application Programming Interface) as the basis for our structural metrics. The rest of the metrics we present are in support of those that are based on API. 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. These metrics are based on information-theoretic principles. We tested our metrics on some popular open-source systems and some large legacy-code business applications. To validate the metrics, we compared the results obtained on human-modularized versions of the software (as created by the developers of the software) with those obtained on randomized versions of the code. For randomized versions, the assignment of the individual functions to modules was randomized.

Index Terms — Metrics/measurement, modules and interfaces, information theory, distribution, maintenance and enhancement, maintainability, coupling, layered architecture.

1 Introduction

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.
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. The work we report here addresses the fundamental issue of how to measure the quality of a given modularization of the software.

- *Ashutosh Lahariya, Aman Jain and Rosedeep Singh is currently pursuing BE in computer engineering in University Of Pune , India, MO- +919049199333.E-mail:ashu_lahariya@yahoo.co.in,amanjainer@gmail .com, singhrosedip@gmail.com*
- *Rachana Nemade is a professor of computer engineering at MIT AOE,Pune and is currently pursuing ME in computer engineering at North Maharashta University , India, MO- +918600025675. E-mail: nemade_rachana@rediffmail.com*

Note that modularization quality is not synonymous with modularization correctness. 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. As a theoretical extreme, retaining all of the software in a single monolithic module is a correct modularization though it is not an acceptable solution. On the other hand, the quality of modularization has more to do with partitioning software into more maintainable (and more easily extendible) modules on the basis of the cohesiveness of the service provided by each module. Ideally, while containing all of the major functions that directly contribute to a specific service vise the other modules, each module would also contain all of the ancillary functions and the data structures if they are only needed in that module. Capturing these "cohesive services" and" ancillary support" criteria into a set of metrics is an important goal of our research. The work that we report here is a step in that direction.

More specifically, we present in this work a set of metrics that measure in different ways the interactions between the different modules of a software system. It is important to realize that metrics that only analyze intermodal interactions cannot exist in isolation from other metrics that measure the quality of a given partitioning of the code. To explain this point, it is not very useful to partition a software system consisting of a couple of million lines of code into two modules, each consisting of a million lines of code, and justify the two large modules purely on the basis of function call routing through the published APIs for the two modules. Each module would still be much too large from the standpoint of code maintenance and code extension. The module interaction metrics must therefore come with a sibling set of metrics that record other desirable properties of the code. The metrics we present in Section 4, while focusing primarily on module interactions, also include other necessary measures of the quality of a given partitioning of code.
The paper is organized as follows: In the next section, we provide a brief review of the literature relevant to our work. In Section 3, we describe the notion of modularity of a system and enunciate a set of design principles that should be adhered to in a well-modularized system. Next, in Section 4, we define a set of metrics based on the structural aspects of intermodule relationships.

2 Previous Work On Software Metrics Relevant To Our Contribution

Some of the earliest contributions to software metrics deal with the measurement of code complexity [1], [2] and maintainability [3] based on the complexity measures proposed in [1], [2]. From the standpoint of code modularization, some of the earliest software metrics are based on the notions of coupling and cohesion [4], [5]. 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. In 1988, Weyuker [6] proposed a set of properties to be satisfied by software complexity metrics. Many subsequent contributions discussed these properties from the standpoint of sufficiency/necessity and whether or not they could be supported by more formal underpinnings. See for example [7], [8], and [9], and the citations contained therein. Other notable software metrics validation frameworks include those by Kitchenham et al. [10] (see also Fenton and Pfleeger [11]), who have borrowed the needed principles from the classical measurement theory. However, this framework was found wanting by Morsaca et al. [12] with regard to how the scale types used for

attribute measurements were constrained.

With regard to modularity, Briand et al. [8] have given us a generic formalization of such fundamental notions as module and system, and such metrical notions as coupling, cohesion, and complexity. Their formalization is generic in the sense that it is not limited to any specific style of programming. The authors have also mapped several well-known metrics such as Halstead length [1] and cyclomatic complexity [2], as well as coupling and cohesion metrics, into this framework. Frameworks such as those proposed by Briand et al. [8] are important because they educate us about the fundamental criteria that must be fulfilled by the various metrics. The work reported in [8] was extended by the authors in [13]; in this more recent work, they have proposed a framework for a goal-driven definition of software measures.

The early work on software metrics was followed by their reformulation for the object-oriented case. Researchers came up with coupling, cohesion, and complexity metrics that measured the various quality attributes of OO software. These measures were primarily at the level of how the individual classes were designed from the standpoint of how many methods were packed into the classes, the depth of the inheritance tree, the inheritance fan-out, couplings between objects (CBO) created by one object invoking a method on another object, etc. [7]. But, then, responding to the observations (such as those made by Churcher and Shepperd [14]) that any counting-based measurements applied to software where objects inherited methods and attributes from other objects were open to interpretation, Briand et al. [15] proposed a framework that formalized how one could exercise different options when applying coupling, cohesion, and complexity metrics to object-oriented software. Recently, Arisholm et al. [16] have used the framework laid out in [15] to propose metrics to measure coupling among classes based on runtime analysis for object-oriented systems.

While the work mentioned above deals primarily with how to measure the quality of a modularized software system through coupling, cohesion, and complexity metrics, many other researchers have proposed metrics, albeit indirectly, in their quest to develop automated tools for software clustering. Clustering obviously depends on the measurement of properties of semiformed modules that, when optimized with respect to those properties, lead (hopefully) to a well-modularized system. For example, in the work on automated software-partitioning by Schwanke [17], modules are quantitatively characterized by the degree to which the functions packaged within the same module contain "shared information." Functions may share information on the basis of, say, the commonality of the names of the data objects used. Schwanke also characterizes modules on the basis of function-call dependencies. If a function A calls function B, then, in the approach used by Schwanke, both A and B presumably belong to the same module.

Along the same lines, meaning along the lines of formulating metrics in the context of developing code modularization algorithms, Mancoridis et al. [18], [19] 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 system level MQ is calculated as the difference between the average cohesion and the average coupling. Finally, an earlier preliminary publication by us [32] mentions the need for API-based metrics for measuring the quality of software modularization and presents some metrics for doing the same. Our present work is a major overhaul, upgrade, and expansion of that earlier contribution.

3 The Notion of Modularity—Enunciation Of The Underlying Principles

Modern software engineering dictates that a large body of software be organized into a set of modules. According to Parnas [33], 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. The difference between the two is subtle but important. An example of a module that represents the first type of grouping is the java.util package of the Java platform. The different classes of this package provide different types of containers for storing and manipulating objects.1 On the other hand, a module such as the java.net package groups software entities on the basis of commonality of goal, the goal being to provide support for networking. The asymmetry between modules based on these two different notions of grouping is perhaps best exemplified by the fact that you are likely to use a java.util class in a java.net-based program, but much less likely to do so the other way around.

In either case, 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 various dimensions along which the quality of the software is improved by the encapsulation provided by modularization include understandability, testability, change- ability, analyzability, and maintainability. These specific traits of software quality were recently articulated by Arevalo in the context of object-oriented software design [34], but they obviously apply to modularization in general.2
So, if modularization is the panacea for the ills of disorganized software, on what design principles should code modularization be based? In what follows, we will enunciate such principles and state what makes them intuitively plausible and what support each derives from the research literature.

P1. Principles Related to Similarity of Purpose. A module groups a set of data structures and functions that together offer a well-defined service. In other words, 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 principles as coming under the "Similarity of Purpose" rubric:

1) Maximization of Module Coherence on the Basis of Similarity and Singularity of Purpose,
2) Minimization of Purpose Dispersion,
3) Maximization of Module Coherence on the Basis Of Commonality of Goals ,and Minimization of Goal Dispersion.

P2. Principles Related to Module Encapsulation. 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:

1) Maximization of API-Based Intermodule Call Traffic.
2) Minimization of non-API-Based Intermodule Call Traffic.

P3. Principle Related to Module Compilability. A common cause of intermodule compilation dependency is that a file from one module requires, through import or include declarations, one or more files from another module. As software system evolves and as some of the modules begin to seem like utilities to the developers, it is all too easy for such inter- dependencies to become circular. For obvious reasons, such compilation interdependencies make it more difficult for modules to grow in parallel and for the modules to be tested independently. There- fore, to the largest extent possible, it must be possible to compile each module independently of all the other modules. When modules can be compiled independently of one another, then, as long as the module APIs do not change, the other modules can be oblivious to the evolution of internal details of any given module. This notion is captured by the following principle:

1) Maximization of the Stand-Alone Module Compilability.

P4. Principle Related to Module Extendibility. One of the most significant reasons for object-oriented software development is that the classes can be easily extended whenever one desires a more specialized functionality. Extending object-oriented software through the notion of sub classing allows for a more organized approach to software development and maintenance since it allows for easier demarcation of code authorship and responsibility. While module- level compartmentalization of code does not lend itself to the types of software extension rules that are easy to enforce in object-oriented approaches, one nonetheless wishes for the modules to exhibit similar properties when it comes to code extension and enhancement. The following principle captures this aspects of code modularization:

1) Maximization of the Stand-Alone Module Extendibility.
2) Module extendibility is a particularly important issue for very large software systems in which the modules are likely to be organized in horizontal layers.

P5. Principle Related to Module Testability. Testing is a major part of software development. At the mini- mum, testing must ensure that software conforms to the prevailing standards and protocols. This is commonly referred to as requirements-based testing. But, even more importantly, testing must ensure that the software behaves as expected for a full range of inputs, both correct and incorrect, from all users and processes, both at the level of the program logic in the individual functions and at the level of module interactions. Testing must take into account the full range of competencies of all other agents that are allowed to interact with the software. Testing procedures can easily run into combinatorial problems when modules cannot be tested independently; meaning that if each module is to be tested for N inputs, then two interdependent modules must be tested for N2 inputs. A modularization procedure must therefore strive to fulfill the following principle:

1) Maximization of the Stand-Alone Testability of

Modules.

P6. Principles Related to Acyclic Dependencies. For obvious reasons (and also as pointed out by Martin [35] and Seng et al. [28]), it is important to minimize the cyclic dependencies between the modules of a body of software. Cyclic dependencies directly negate many of the benefits of modularization. It is obviously more challenging to foresee the consequences of changing a module if it both depends on and is depended upon by other modules. Cyclic dependencies become even more problematic when modules are organized in the form of horizontal layers in a large software system. (Layering serves an important organizational concept for large systems; all the modules in a layer can only seek the services of the layers below.) We therefore state the following two principles:

1) Principle of Minimization of Cyclic Dependencies Amongst Modules.

2) Principle of Maximization of Unidirectionality of Control Flow in Layered Architectures.

P7. Principles Related to Module Size.In light of the findings reported by Emam et al. [36], in a new software development effort started from scratch today, one would not ordinarily insist that the module sizes be roughly the same and equal to some prespecified magic number. Nonetheless, when modularizing legacy code that happens to be in a chaotic state of organization, it would be highly desirable to be able to bias a clustering algorithm toward producing modules that are roughly of the same size, whose value is dictated by considerations related to software maintenance and such.4 As we said earlier in the Introduction, placing all of the code in a single module is technically a correct modularization, albeit not very useful. We therefore need metrics that can steer a modularization algorithm away from producing unacceptably large modules and, to the extent other important considerations are not violated, toward producing modules roughly equal in size. The following two principles address this need:

1) Principle of Observance of Module Size Bounds.

2) Principle of Maximization of Module Size Uniformity.

The metrics we propose in the rest of this paper show how good a given modularization is with respect to the principles we have enunciated in this section.

3.1 Relationship Of The Previous Metrics To The Enunciated Principles

What follows is a summarization of the metrics mentioned in our literature survey in Section 2. This summarization states briefly the extent, if at all, to which the metrics measure the quality of the software from the standpoint of the modularization principles P1 through P7.

1. Halstead [1] and Cyclomatic [2] Measures. These have focused on the control flow complexity at the level of individual functions and subroutines and do not directly relate to any of the modularity principles stated earlier.

2. Maintainability Index [3]. This is a linear expression based on the Halstead and cyclomatic measures as well as module lines of code and module comments. Clearly, this metric does not directly relate to any of the listed principles either.

3. Coupling-cohesion-based metrics [4], [5], [7], [15]. These are measures of modularization quality based on intermodule and intramodule relationships that are derived from the structural dependencies of modules obtained mainly through a static analysis of software.

- These metrics can be broadly related to the principles concerning module compilability, extendibility, and testability (Principles P3, P4, and P5). It is obviously the case that a cohesive module that is just loosely coupled to other modules exhibits less of a dependency on the other modules. Consequently, its compilation, extension and testing will have less of an impact on the other modules.

- In this context, it may be observed that characterizing modules primarily on the basis of cohesion among entities (that constitute a module) derived from structural dependencies (such as function-call and data dependency) penalizes modules in which the entities are grouped on the basis of similarity of purpose. (A case in point would be the java. until package of the Java platform.) The work by [29] supports this observation.

4. The modularity measure of [17]; modularization quality (MQ) measure and its variations [20], [18], [21], [22], [19]; coupling- cohesion, size, cyclic dependency, software complexity [measures for automated clustering. Several of the metrics proposed in [28] relate to principles of acyclic dependency (P6) and module size (P7), in

addition to P3, P4, and P5. The works by [19], consider the notion of omnipresent objects and clustering based on omnipresent objects. These are loosely related to the principles concerning the similarity of service (P1).

5. The coupling-cohesion metric based on information theoretic notions [30]. This metric measures coupling-cohesion patterns among modules on the basis of structural dependencies. This approach is different from the other coupling-cohesion measures listed in items 3 and 4 above since those are count-based, whereas this metric is pattern based.

- Like the other coupling-cohesion-based metrics mentioned in items 3 and 4, this metric is related to principles P3, P4, and P5.

6. Metric based on nonstructural information [31]. This metric measures cohesiveness and coupling of entities on the basis of mutual information in the information-theoretic sense. The authors have used the metric to guide a software clustering algorithm to arrive at a set of cohesive modules.

- This metric is closely related to the Similarity of Purpose principles (P1).

The metrics listed in items 1 though 5 above conform only partly to the principles outlined earlier in this section. As a case in point, the coupling-cohesion-based metrics certainly do not measure the similarity of purpose or the commonality of goals (principles P1). Perhaps the metric that comes closest to fulfilling the spirit of P1 is the one presented in [31]. These prior contributions certainly do not measure how effectively the principle that says that all intermodule function calls should be routed through the API's of the modules (principles P2) is honored. These metrics also do not measure the extent to which a module encapsulates its internal (meaning non-API) functions and keeps them from getting exposed to the external world. All of the metrics listed above also do not provide a good measure of the consequences of cyclic dependencies between the modules, especially when the modules reside in layered architectures.

3.2 Notation

In the rest of this paper, we will denote a software system by the symbol S.

- $\mathcal{S}$ will be considered to consist of a set of modules $\mathcal{M} = \{m_1, m_2, \ldots, m_M\}$. M will stand for the *number* of modules. Thus, $|\mathcal{M}| = M$.
- All of the functions in $\mathcal{S}$ will be denoted by the set $\mathcal{F} = \{f_1, \ldots f_F\}$. These are obviously assumed to be distributed over the set $\mathcal{M}$ of modules. Furthermore, $|\mathcal{F}| = F$.
- An API function for a given module will be denoted by f^a and a non-API function of a module as f^{na}.
- We will use the notation $\mathcal{L}$ to denote the set of layers $\{L_1 \cdots L_p\}$ into which the modules are organized if a horizontally layered architecture is used for the software.
- $K(f)$ will denote the total number of calls made *to* a function f belonging to a module m. Of these, $K_{ext}(f)$ will denote the number of calls coming in from the other modules, and $K_{int}(f)$ the number of calls from within the same module. Obviously, $K(f) = K_{ext}(f) + K_{int}(f)$.
- $K_{ext}(m)$ will denote the total number of external function calls made to the *module* m. If m has $f_1 \cdots f_n$ functions then $K_{ext}(m) = \sum_{f \in \{f_1 \cdots f_n\}} K_{ext}(f)$.
- $\hat{K}(f)$ will denote the total number of calls made *by* a function f.
- For a module m, $K^j_{ext}(f)$ $(K^j_{ext}(f) \leq K_{ext}(f))$ will denote the number of external calls made to f (in module m) from another module m_j.

4. Coupling-Based Structural Metrics

Starting with this section, we will now present a new set of metrics that cater to the principles enunciated in Section 3. 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.

4.1 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 has n functions{f1 ... fn}, of which the n1 API functions are given by the subset{fa1……..fan1}. 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

$$MII(m) = \frac{\sum_{f^a \in \{f_1^a \cdots f_{n1}^a\}} K_{ext}(f^a)}{K_{ext}(m)}$$
$$= 0, \text{ when no external calls made to } m, \quad (1)$$
$$MII(S) = \frac{1}{M} \sum_{i=1}^{M} MII(m_i).$$

MII measures the extent to which a software system adheres to the module encapsulation principles P2 presented in Section 3. Recall that these principles demand a well- designed module should expose a set of API functions through which other modules would interact. These API functions represent the services that the module has to offer. 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. Therefore, a non- API function of a module should not receive external calls to the maximum extent possible. In other words, 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.

Note that $\sum_{f^a \in \{f_1^a \cdots f_{n1}^a\}} K_{ext}(f^a)$ for a module m increases as more and more intermediate module calls are routed through the API functions of m . We obviously have MII(m)->1 in the ideal case when all the intermodule calls are routed through the API functions only. By the same argument, MII(S) should also be close to 1 in the ideal case. Therefore, MII quantitatively measures the extent to which encapsulation related principles have been followed.

Complex software systems sometimes employ what are known as driver modules to orchestrate the other modules. For a driver module, MII(m)=0 will likely be the case. The fact that a modularization effort must allow for such modules does not detract from the fact that the overall goal of a modularization effort should be to achieve as large a value as possible for MII while allowing for the possibility that some modules may not be able to contribute a fair share to the overall index.

4.2 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 intermodule 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 intermodule 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 intermodule and intramodule call traffic. We measure the extent of this traffic using a metric that we call "Non-API Function Closedness Index," or NC.

Extending the notation presented in Section 3.2, let Fm, F am, and F nam represent the set of all functions, the API functions, and the non-API functions, respectively, in module m. Ideally, Fm = Fam +Fnam. 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

$$NC(m) = \frac{|\mathcal{F}_m^{na}|}{|\mathcal{F}_m| - |\mathcal{F}_m^{a}|}$$
$$= 0 \text{ if there are no non-API functions}, \quad (2)$$
$$NC(S) = \frac{1}{M} \sum_{i=1}^{M} NC(m_i).$$

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 functions, |Fm | - |Fam| would be equal to |Fnam| Therefore, NC(m)=1 for a well designed module. Otherwise, the value for this metric will be between 0 and 1.

4.3 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 nj number of API functions are called by another module mj. Also assume that there are k modules m1……mkthat call one or more of the API functions of module m. We may now formulate an API function usage index in the following manner:

$$APIU(m) = \frac{\sum_{j=1}^{k} n_j}{n * k}$$
$$= 0 \text{ if } n = 0, \quad (3)$$
$$APIU(S) = \frac{1}{M_{apiu}} \sum_{i=1}^{M_{apiu}} APIU(m_i),$$

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. For example, maximizing module coherence on the basis of commonality of goals does require that the modules not be monolithic pieces of software and ought not to provide disparate services. 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. However, it must be mentioned that since the metric does not actually analyze the specificity of the API functions, one could indeed conjure up a set of modules that are far from being goal-focused and that nonetheless yield a high value for this metric. So, this metric all by itself will not force a set of modules to become less monolithic. Nonetheless, when considered in conjunction with the other metrics, this metric can be expected to play a desirable role in the characterization of a set of modules.

4.4 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. We refer to such inter-module dependencies 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 we call 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 proviso that such global entities are accessed by functions in other modules. We believe that the larger this count is in relation to the size of the intermodule traffic consisting of explicit function calls, the greater the insidiousness of implicit dependencies.

For each module mi, we use the notation Dg (mi, mj),i not equal to j denote the number of dependencies created when a function in mi writes to a global entity that is subsequently accessed by some function in mj. Let Dg (mi, mj),i not equal to j denote the number of explicit calls made by all the functions in mi to any of the functions in mj. We claim that the larger Dg is in relation to Df, the worse the state of the software system. We therefore define the metric as follows:

$$IDI(m) = \frac{\sum_{m_j \in C(m)} D_f(m, m_j)}{\sum_{m_j \in C(m)} (D_g(m, m_j) + D_f(m, m_j))}$$
$$= 1 \text{ when } C(m) = \emptyset, \quad (4)$$
$$IDI(S) = \frac{1}{M} \sum_{i=1}^{M} IDI(m_i),$$

Where C(m) is the set of all modules that depend on the module m through implicit dependencies of the sort we have described in this section.

With regard to where this metric belongs in the landscape of the principles we presented in Section 3, as we have said before, ideally all the interaction between modules must be through published API functions, implying that the number of implicit dependencies must be few and far between. Therefore, an ideal API-based system will make IDI equal to 1. Clearly, this is in conformance with the principles of module encapsulation (P2) that requires minimization of such implicit, non-API- based communications.

5 Experiments

Our experimental validation of the metrics is made challenging by the fact that it is difficult to find examples of non-object-oriented software that are modularized and that have published APIs for each of the modules. Many of the publicly available software systems with published APIs for the modules, such as Qt, GNOME/GTK+, wx Windows, JDK and many others, are object-oriented. Our metrics are not meant for such software.

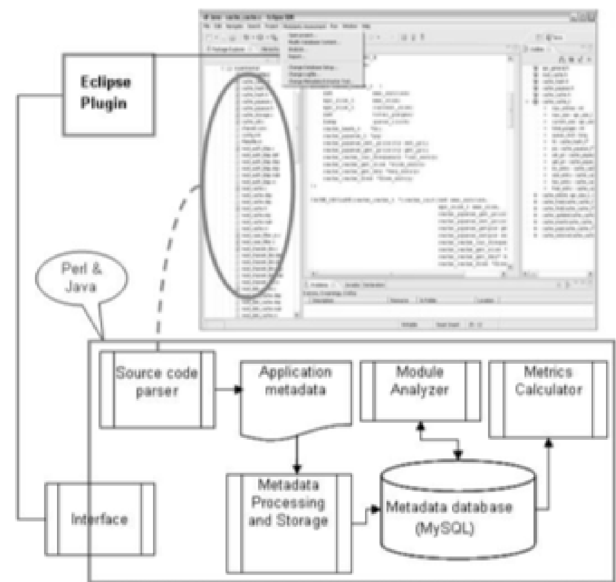
The Metrics-Principles Connection

Modularity Principle	Metric
P1: Similarity of Purpose, Minimization of Purpose Dispersion	CDM,CCM
P1: Maximization of Module Coherence on the Basis of Commonality of Goals	APIU
P2: Maximization of API-Based Inter-Module Call Traffic	MII
P2: Minimization of non-API Based Inter-Module Call Traffic	MII, NC, IDI
P3: Maximization of the Stand-Alone Module Compilability	–
P4: Maximization of the Stand-Alone Module Extendibility	MISI
P5: Maximization of the Stand-Alone Testability of Modules	NTDM
P6: Principle of Minimization of Cyclic Dependencies Amongst Modules	*Cyclic*
P6: Principle of Maximization of Unidirectionality of Control Flow in Layered Architectures	LOI
P7: Observance of Module Size Bounds	MSBI
P8: Maximization of Module Size Uniformity	MSUI

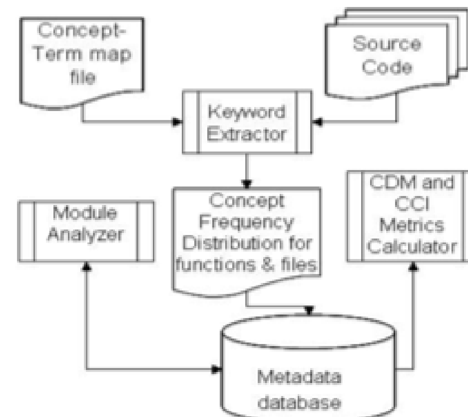
We have therefore resorted to applying our metrics to software systems that are well-organized into directory structures (mostly on the basis of the services offered by the different directories). As we will explain later, it is relatively straightforward to label the functions in the different directories of these 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.10 The software systems we chose included a mix of open source systems and several proprietary business applications. These software systems are medium to large sized, Ranging from 160,000 to several million lines of C and C++ programs. The largest proprietary business application we tested the metrics on ran into 10 million lines of C code. But, for obvious reasons, we will limit our discussion in the rest of this section to the freely available software.

The open source software systems chosen for reporting our experimental results in this section—MySQL, Apache, Mozilla, GCC, the Linux kernel, and PostgreSql—are highly regarded in academia and industry for their robustness and for the quality of code. For these software systems, we took the directory structure as examples of human-delineated modularization. Given the high quality of code organization, it should not come as a surprise that the major directories of these systems correspond to the different specialized services offered by the software systems. In all these systems, the different directories and subdirectories carry mnemonic names that hold clues to the services offered by those directories.

Before calculating the metrics, the code was first analyzed by the open-source tool Sourcenav [47] that yielded a database containing the associations between the function definitions and the corresponding file names and also the function-call dependency information. Subsequently, we ran a set of tools written in Perl and Java to extract the metrics presented in this paper. These steps are depicted in Fig. 1a. The output produced by Sourcenav is summarized in Table .



(a)



Modularity Assessment Tool Architecture. (a) Schematic diagram. (b) Concept extraction scheme.

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. These versions correspond to the following scenarios:

1. Scenario 1 (Human). 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—the module corresponding to the directory itself. In this manner, all leaf level directories formed the module set for the software. We call this module set the Developer Generated Module Set.

2. Scenario 2 (Random). Functions were assigned randomly to modules in such a manner that we ended

Table

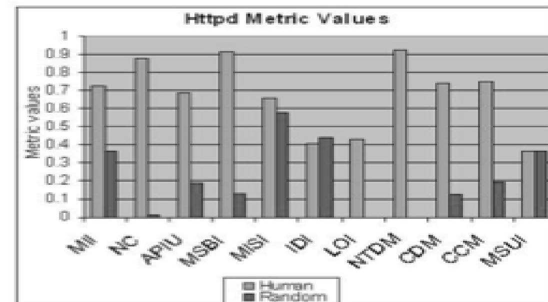
Software Systems Used for Metrics Validation

Name	#files	LOC	LOC wo com-ments	#functions	#Cross Ref	#Leaf direc-tory
apache-2.0.53	940	403918	358982	5811	4567	111
Mysql-4.1.12	2187	1285675	1107722	28457	59415	153
mozilla-1.7	10214	3610671	2777730	97210	144920	703
GCC-3.3	1865	1152077	1006977	17156	37362	58
Linux kernel-2.0.27	1612	700656	608039	12880	33027	94
Postgresql-7.3.12	1012	481468	398762	7479	20162	141

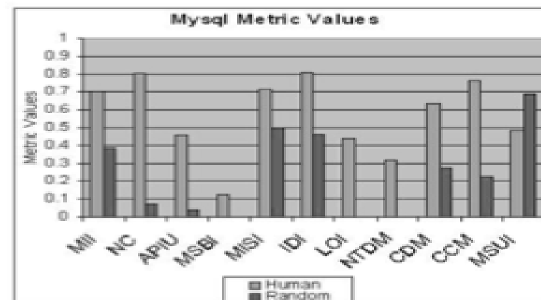
up with the same number of modules as in the developer generated module set. We call this the Randomly Generated Module Set.

6 Concept Extraction

In order to evaluate the CDM and CCM metrics, we must extract various domain concepts from the source code and create a frequency distribution of the concepts for each module. The concept extraction process is shown in Fig. 1b. The extraction process takes an input file containing a set of business concept names and the associated terms or key-words that can uniquely identify a business concept. This file in Fig. 1b is currently created manually. The Keyword Extractor component takes these keywords as regular expressions and computes a concept frequency distribution for each function by counting the occurrences of these keywords from function signature, return types, and data structures used by the function. However, function calls are not considered in the counting process since they can skew the concept distribution frequencies. For example, suppose that a customer-update function $f_{customer}$ calls an account-handling function $f_{account}$ several times. If we look for the concept account in the function call $f_{account}$(called by the function $f_{customer}$), several occurrences of the concept



(a)



(b)

Comparison of metric values for Human and random modularization of some systems. (a) httpd-2.0.53 metric values. (b) Mysql metric values.

account will be found. If the number of occurrences is relatively high, account might appear (incorrectly) as the dominating concept in $f_{customer}$.

7 Experimental Results

The first set of experimental results show how the metric values change when the modularization scenario is changed from the human-supplied to random. Fig. 2 shows the results obtained for the four open-source applications.

The second sets of experimental results are a comparative presentation of the metric values for different versions of the software systems. Fig. 3 shows the metric values for two different versions of Apache and Mozilla software.

The CDM and CCM metrics depend on the relative frequencies of the concepts in the different modules. It is obviously not feasible to show these relative frequencies and how these frequencies change with changes in the modularization for all the concepts. So, we arbitrarily chose a couple of concepts, user authentication for the Apache software and parser for MySQL, to show the distribution of their relative frequencies in the original software and in the randomized version. These are plotted in Fig.

8 Conclusion

We have enunciated a set of design principles for code modularization and proposed a set of metrics that

characterize software in relation to those principles. Although many of the principles carry intuitive plausibility, several of them are supported by the research literature published to date. Our proposed metrics seek to characterize a body of software according to the enunciated principles. The structural metrics are driven by the notion of API—a notion central to modern software development. Other metrics based on notions such as size-roundedness, size-uniformity, operational efficiency in layered architectures, and similarity of purpose play important supporting roles. These supporting metrics are essential since otherwise it would be possible to declare a malformed software system as being well-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 highly unacceptable).

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. (See Fig.)

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.

With regard to our future work, in addition to the empirical support presented in this paper, we would also like to validate them theoretically. As we mentioned in Section 2, theoretical validation implies conformance to a set of agreed-upon principles that are usually stated in the form of a theoretical framework. Again as mentioned in Section 2, the more notable of the frameworks that have been proposed over the years for software metrics validation include those by Kitchen ham et al. [10] (see also Fenton and Pfleeger [11]) and Briand et al. [8], [13]. If we also include the set of desirable properties for metrics proposed by Weyuker [6], that gives us four "frameworks" as possible approaches for the theoretical validation of our metrics.

References

[1] M.H. Halstead, Elements of Software Science, Operating and Programming Systems Series, vol. 7, Elsevier, 1977.

[2] T.J. McCabe and A.H. Watson, "Software Complexity," Crosstalk, J. Defense Software Eng., vol. 7, no. 12, pp. 5-9, Dec. 1994.

[3] P. Oman and J. Hagemeister, "Constructing and Testing of Polynomials Predicting Software Maintainability," J. Systems and Software, vol. 24, no. 3, pp. 251-266, Mar. 1994.

[4] W. Stevens, G. Myers, and L. Constantine, "Structured Design," IBM Systems J., vol. 13, pp. 115-139, 1974.

[5] E. Yourdon and L.L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, 1979.

[6] E. Weyuker, "Evaluating Software Complexity Measures," IEEE Trans. Software Eng., vol. 14, no. 9, pp. 1357-1365, Sept. 1988.

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

[8] L.C. Briand, S. Morasca, and V.R. Basili, "Property-Based Software
Engineering Measurement," IEEE Trans. Software Eng., vol. 22,
no. 1, pp. 68-85, Jan. 1996.

[9] N. Sharma, P. Joshi, and R.K. Joshi, "Applicability of Weyuker's
Property 9 to Object Oriented Metrics," short note, IEEE Trans.
Software Eng., vol. 32, no. 3, pp. 209-211, Mar. 2006.

[10] B. Kitchenham, S. Pfleeger, and N. Fenton, "Towards a Frame- work for Software Validation Measures," IEEE Trans. Software
Eng., vol. 21, no. 12, pp. 929-944, Dec. 1995.

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

[12] S. Morasca, L.C. Briand, V. Basili, E.J. Weyuker, and M. Zelkowitz,
"Comments on 'Towards a Framework for Software Measurement Validation,'" IEEE Trans. Software Eng., vol. 23, no. 3, pp. 187-188, Mar. 1997.

[13] L.C. Briand, S. Morasca, and V. Basili, "An Operational Process for Goal Driven Definition of Measures," IEEE Trans. Software Eng., vol. 28, no. 12, pp. 1106-1125, Dec. 2002.

[14] N. Churcher and M. Shepperd, "Comments on a Metrics Suite for Object-Oriented Design," IEEE Trans. Software Eng., vol. 21, no. 3, pp. 263-265, Mar. 1995.

[15] L.C. Briand, J.W. Daly, and J.K. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," IEEE Trans. Software Eng., vol. 25, no. 1, pp. 91-121, 1999.

[16] E. Arisholm, L.C. Briand, and A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software," IEEE Trans. Software Eng., vol. 30, no. 4, pp. 491-506, Aug. 2004.

[17] R.W. Schwanke, "An Intelligent Tool for Reengineering Software Modularity," Proc. 18th Int'l Conf. Software Eng., pp. 83-92, May 1991.

[18] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner, "Using Automatic Clustering to Produce High-Level System Organizations ofSource Code," Proc. Sixth Int'l Workshop Program Comprehension (IWPC '98), pp. 45-52, 1998.

[19] S. Mancoridis, B.S. Mitchell, Y.-F. Chen, and E.R. Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," Proc. Int'l Conf. Software Main- tenance (ICSM), pp. 50-59, http://citeseer.ist.psu.edu/article/ mancoridis99bunch.html, 1999.

[20] H. Fahmy and R. Holt, "Software Architecture Transformations," Proc. Int'l Conf. Software Maintenance, pp. 88-96, Oct. 2000.

[21] D. Doval, S. Mancoridis, and B.S. Mitchell, "Automatic Clustering of Software Systems Using a Genetic Algorithm," Proc. Int'l Workshop Software Technology and Eng. Practice, 1999.

[22] B.S. Mitchell, S. Mancoridis, and M. Traverso, "Search Based Reverse Engineering," Proc. 14th Int'l Conf. Software Eng. and Knowledge Engineering (SEKE '02), pp. 431-438, 2002.

[23] K. Mahdavi, M. Harman, and R.M. Hierons, "A Multiple Hill Climbing Approach to Software Module Clustering," Proc. 19th Int'l Conf. Software Maintenance (ICSM '03), pp. 315-324, 2003.

[24] A. Shokoufandeh, S. Mancoridis, T. Denton, and M. Maycock, "Spectral and Meta-Heuristic Algorithms for Software Cluster- ing," J. System and Software, vol. 77, no. 3, pp. 213-223, Sept. 2005.

[
25] M. Harman, S. Swift, and K. Mahdavi, "An Empirical Study of the Robustness of Two Module Clustering Fitness Functions," Proc. 2005 Conf. Genetic and Evolutionary Computation, pp. 1029-1036, 2005.

[26] K. Sartipi and K. Kontogiannis, "Component Clustering Based on Maximal Association," Proc. Eighth Working Conf. Reverse Eng. (WCRE '01), pp. 103-114, 2001.

[27] K. Sartipi, "Software Architecture Recovery Based-On Pattern Matching," PhD dissertation, School of Computer Science, Univ. Waterloo, 2003.

[28] O. Seng, M. Bauer, M. Biehl, and G. Pache, "Search-Based Improvement of Subsystem Decompositions," Proc. Conf. Genetic and Evolutionary Computation, pp. 1045-1051, http://doi.acm.org/ 10.1145/1068186, 2005.

[29] Z. Wen and V. Tzerpos, "Software Clustering Based on Omni- present Object Detection," Proc. 13th Int'l Workshop Program Comprehension (IWPC '05), pp. 269-278, 2005.

[30] 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.

[31] P. Andritsos and V. Tzerpos, "Information-Theoretic Software Clustering," IEEE Trans. Software Eng., vol. 31, no. 2, pp. 150-165, Feb.

[32] 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.

[33] D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," Comm. ACM, vol. 15, no. 12, pp. 1053-1058, 1972.

[34] G.B. Arevalo, "High-Level Views in Object-Oriented Systems Using Formal Concept Analysis," PhD dissertation, 2004.

[35] R. Martin, "Design Principles and Design Patterns," http://www.objectmentor.com, 2000,

[36] K.L. Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S.N. Rai, "The Optimal Class Size for Object Oriented Software," IEEE Trans. Software Eng., vol. 28, no. 5, pp.

494-509, May 2002.

[37] L. Hatton, "Reexamining the Fault Density-Component Size Connection," IEEE Software, vol. 14, no. 2, pp. 89-97, 1997.

[38] D.H. Hutchens and V.R. Basili, "System Structure Analysis: Clustering with Data Binding," IEEE Trans. Software Eng., vol. 11, no. 8, pp. 749-757, Aug. 1985.

[39] J. Rosenberg, "Some Misconceptions About Lines of Code," Proc. Fourth Int'l Software Metrics Symp. (METRICS '97), pp. 137-142, 1997.

[40] F. Bachmann, L. Bass, J. Carriere, P. Clements, D. Garlan, J. Ivers, R. Nord, and R. Little, Software Architecture Documentation in Practice: Documenting Architectural Layers, Special Report CMU/ SEI-2000-SR-004, Software Eng. Inst., Carnegie Mellon Univ., 2000.

[41] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, Documenting Software Architecture, Views and Beyond. Addison Wesley, Sept. 2002.

[42] F. Rysselberghe and S. Demeyer, "Studying Software Evolution Information by Visualizing the Change History," Proc. 20th IEEE Int'l Conf. Software Maintenance, pp. 328-337, Sept. 2004.

[43] T. Girba, S. Ducasse, and M. Lanza, "Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes," Proc. Int'l Conf. Software Maintenance, 2004.

[44] J. Lakos, Large Scale C++ Software Design. Addison-Wesley, 1996.

[45] M. Siff and T. Reps, "Identifying Modules via Concept Analysis,"
IEEE Trans. Software Eng., vol. 25, pp. 749-768, 1999.

[46] P. Tonella, "Concept Analysis for Module Restructuring," IEEE
Trans. Software Eng., vol. 27, pp. 351-363, 2001.

[47] Source Navigator 5.4.1,http://sourcenav.sourceforge.net, 2003.

[48] A. MacCormack, J. Rusnak, and C. Baldwin, Exploring the Structure
of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code, Technical Report 05-016, Harvard Business School working paper, 2005.

[49] B. Eich, "Development Roadmap, " Mozilla home page, http:// www.mozilla.org/roadmap/roadmap-26-Oct-1998.html, Oct. 1998.