API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization

Santonu Sarkar, Girish Maskeri Rama, and Avinash C. Kak

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.

- *S. Sarkar and G.M. Rama are with SETLabs, Building 19FF, Infosys Technologies Ltd., Electronic City, Hosur Road, Bangalore, Pin 560100, India. E-mail: {santonu_sarkar, girish_rama}@infosys.com.*
- *A.C. Kak is with Purdue University, 1285 EE Building, West Lafayette, IN 47907-1285. E-mail: kak@ecn.purdue.edu.*

Manuscript received 18 May 2006; revised 28 Aug. 2006; accepted 17 Aug. 2006; published online 30 Nov. 2006.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0108-0506.

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 vis-a-vis 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 intermodule 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, Section 5, Section 6, and Section 7, 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. Next, in Section 5, we propose sized-based metrics in support of the other metrics. Section 6 focuses on metrics that apply to layered architectures. We define a set of metrics to measure the cohesiveness of the modules with respect to similarity of service in Section 7. The experiments section (Section 9) presents empirical support for the new metrics. Finally, in Section 10, we conclude the paper by summarizing the described work and making a few brief statements about our future research in this area.

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.

Other contributions that have also used function-call dependencies to characterize software for modularization/clustering include [20], [21], [22]. Variations on the MQ measure have been proposed by Mahdavi et al. [23], Shokoufandeh et al. [24], and Harman et al. [25]. The work reported in [23] uses a modified MQ that is the sum of modularization factors (MF) for each cluster. MF for a cluster is given by $\frac{i}{i+1/2*j}$, where i is the sum of internal function-call dependency weights and j is the sum of external function-call dependency weights. The accuracy of a clustering solution is measured with the modified MQ. On the other hand, the MQ metric in [24] is the sum of the cluster factors for all the clusters; the cluster factor for a module is expressed as the ratio of a weighted sum of the function-call dependencies internal to a module to a weighted sum of the internal as well as the external dependencies.

Coupling and cohesion sort of metrics have also been used by Sartipi and Kontogiannis [26], [27] in the context of software clustering. Their metrics are derived from a relational graph representation of all the software artifacts such as functions, data types, variables, etc., the relations being function-call dependencies, type declaration of variables, macro defines, etc. Their metrics measure the degree of association between the relations included in a candidate module vis-a-vis the associations between a module and all other modules. Another contribution that uses coupling-cohesion sort of metrics in the context of software clustering is by Seng et al. [28]; they have also based their metrics on purely structural information, such as the shared name substrings for functions and variables, function-call dependencies, etc. The contribution by Wen and Tzerpos [29] 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. Their rationale is that the omnipresent objects distort coupling-cohesion analysis, which is indeed true when metrics are heavily dependent on functional-call dependencies.

Another set of metrics has come out of information-theoretic approaches to software clustering [30], [31]. Here, the goal is to characterize modules on the basis of the predictive power (as measured by, say, mutual information in the information-theoretic sense) of the software artifacts vis-a-vis the modules in which the artifacts reside. The lower this predictive power, the more diffuse the attributes of the software artifacts in a module. The predictive power, when averaged over all the modules, then becomes a metric of both cohesion and coupling. The method suggested in [30] is different from this general information-theoretic approach only in the sense that the authors have tried to place the essentially counting-based metrical framework of Briand et al. [8] in a probabilistic setting.

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*, *changeability*, *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

1. That all classes in the java.util package descend from just a couple of interfaces is beside the point. The fact remains that the different classes are meant to be used in different ways. For example, the manner in which the class `ArrayList` is meant to be used is very different from the manner in which the class `HashMap` is meant to be used.

2. In case the reader is wondering why our current work is limited to non-OO software, the inheritance and containment relationships in OO software can create intermodule dependencies that go beyond the purview of the metrics we have formulated. Extending our metrics to OO systems is one of our future goals.

intuitively plausible and what support each derives from the research literature.[3]

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:

- Maximization of Module Coherence on the Basis of Similarity and Singularity of Purpose,
- Minimization of Purpose Dispersion,
- 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:

- Maximization of API-Based Intermodule Call Traffic and
- 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 a software system evolves and as some of the modules begin to seem like utilities to the developers, it is all too easy for such interdependencies 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. Therefore, 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:

- 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 subclassing 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:

- Maximization of the Stand-Alone Module Extendibility.

Module extendibility is a particularly important issue for very large software systems in which the modules are likely to be organized in horizontal layers. This is an issue we will take up in Section 6.

P5. *Principle Related to Module Testability.* Testing is a major part of software development. At the minimum, 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 N^2 inputs. A modularization procedure must therefore strive to fulfill the following principle:

- 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:

- Principle of Minimization of Cyclic Dependencies Amongst Modules, and

3. We believe that it should be possible to state these principles in the goal-driven framework proposed by [13]. We could, for example, treat the principles under *P*2 as the goals, the quality focus as the interaction through the API functions, and hypothesis refinement as consisting of the maximization of the interaction through the API's.

- 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:

- Principle of Observance of Module Size Bounds and
- 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.util` 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], [23], [24], [26], [27]; coupling-cohesion, size, cyclic dependency, software complexity [28] 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], [29] 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.

4. Previously it was believed [37], [38] that an ideal software system would consist of a collection of modules whose sizes would all be roughly the same (and that this size value would equal a prespecified number, usually referred to as the magic number). It was shown that the number of defects rose both as the modules became too small or too large. But now some researchers believe that this conclusion, especially in the assertion that the defect rate goes up even when the modules sizes are reduced, is perhaps statistically flawed. It is believed that the source of the flaw is the observation made by Rosenberg [39] that even when two random variables X and Y are uncorrelated, the variables X and Y/X will exhibit a negative correlation. What that implies is that, even if the number of defects and the modules sizes were to be uncorrelated, the number of defects per unit size would exhibit a correlation with size.

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 $\mathcal{S}$.

- $\mathcal{S}$ 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 $K_{ext}(m) = \sum_{f \in \{f_1 \cdots f_n\}} K_{ext}(f)$.
- $\widehat{K}(f)$ 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 m has n functions $\{f_1 \cdots f_n\}$, of which the $n1$ API functions are given by the subset $\{f^a_1 \cdots f^a_{n1}\}$. Also assume that the system $\mathcal{S}$ has $m_1 \cdots m_M$ modules. We now express Module Interaction Index (MII) for a given module m and for the entire software system $\mathcal{S}$ by

$$\begin{aligned} MII(m) &= \frac{\sum_{f^a \in \{f^a_1 \cdots f^a_{n1}\}} K_{ext}(f^a)}{K_{ext}(m)} \\ &= 0, \text{ when no external calls made to } m, \\ MII(\mathcal{S}) &= \frac{1}{M} \sum_{i=1}^{M} MII(m_i). \end{aligned} \quad (1)$$

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 $\sum_{f^a \in \{f^a_1 \cdots f^a_{n1}\}} K_{ext}(f^a)$ for a module m increases as more and more intermodule calls are routed through the API functions of m. We obviously have $MII(m) \rightarrow 1$ in the ideal case when all the intermodule calls are routed through the API functions only. By the same argument, $MII(\mathcal{S})$ 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 semimodularized 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 $\mathcal{F}_m$, $\mathcal{F}^a_m$, and $\mathcal{F}^{na}_m$ represent the set of all functions, the API functions, and the non-API functions, respectively, in module m. Ideally, $\mathcal{F}_m = \mathcal{F}^a_m + \mathcal{F}^{na}_m$. 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

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

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, $|F_m| - |F_m^a|$ would be equal to $|F_m^{na}|$. 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 n_j number of API functions are called by another module m_j. Also assume that there are k modules $m_1 \cdots m_k$ that call one or more of the API functions of module m. We may now formulate an API function usage index in the following manner:

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

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 m_i, we use the notation $D_g(m_i, m_j)$, $i \neq j$, to denote the number of dependencies created when a function in m_i writes to a global entity that is subsequently accessed by some function in m_j. Let $D_f(m_i, m_j)$, $i \neq j$ denote the number of explicit calls made by all the functions in m_i to any of the functions in m_j. We claim that the larger D_g is in relation to D_f, the worse the state of the software system. We therefore define the metric as follows:

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

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 Size-Based Metrics

5.1 Module Size Uniformity Index

As we mentioned earlier in Section 3, for a freshly started software development effort involving expert programmers, there would ordinarily be no reason to subject

module sizes to any particular constraints in terms of any magic numbers. But for automatic or semiautomatic modularization of chaotic legacy code, we need metrics that would steer the modularization algorithm toward producing modules whose sizes are within acceptable bounds. As stated earlier, placing all of the code in a single module is technically a correct modularization, but obviously not acceptable. A modularization algorithm obviously needs a bias toward producing modules whose sizes do not violate acceptable bounds and whose size variations are minimal, subject, of course, to the fulfillment of other modularization constraints.

The constraint with regard to size uniformity can be expressed in terms of the average value μ and the standard deviation σ associated with the module sizes. We can define a Module Size Uniformity Index (MSUI) as the ratio

$$MSUI(\mathcal{S}) = \frac{\mu}{\mu + \sigma}. \tag{5}$$

Obviously, the closer this index is to 1, the greater the uniformity of the module sizes. The formula for this metric assumes that the standard deviation is not comparable to the average size value.

5.2 Module Size Boundedness Index

While the previous metric will "push" a modularization algorithm to make the modules as nearly uniform in size as possible, it could still result in modules that are too large. We now formulate a metric that measures the extent to which the module sizes differ from some desirable value. We will assume that this "magic number" for the size is Γ.

We start by defining a deviation in module size from the magic size by $\delta_i = |Sz_i - \Gamma|$, where Sz_i is the number of lines in module m_i. Further, let $\delta_{max} = \max_i\{\delta_i\}$. Ideally, $\forall_{i=1\ldots M}, \delta_i = 0$ and a histogram of δ_i values will have a single bar of height M at $\delta_i = 0$. In practice, a histogram of δ_is could be spread over the range $[0, \delta_{max}]$ of values. We need an approximate way to measure as to what extent the deviations depart from the ideal.

Here we will use a rationale based on the Kullback-Leibler (KL) divergence for measuring the "distance" between two probability distributions. If we could assume that every measured size deviation between 0 and δ_{max} has a nonzero probability of occurrence, we can show that the KL "distance" between the ideal and the actual distributions for the deviations would be $-\log(prob\{\delta = 0\})$, where $prob\{\delta = 0\}$ is obviously proportional to the actual number of modules whose size deviations from the magic size are zero. We would, of course, want to weight this measure by δ_{max}/Γ because as δ_{max} becomes a negligible fraction of Γ, we stop caring how the deviations are distributed in the range $[0, \delta_{max}]$—they will all become unimportant. This should explain the following formula to measure the extent to which the distribution of the actual size deviations differs from the ideal distribution:

$$\begin{aligned} d &= -\frac{\delta_{max}}{\Gamma} \times \log\left(\frac{|\{\delta_i \ : \ \delta_i < \tau\}|}{M}\right) \\ &= 0 \text{ when } \delta_{max} = 0, \end{aligned} \tag{6}$$

where the numerator in the argument to the logarithm is the cardinality of the set of δ_is whose values are less than the user-specified parameter τ. Except for the special case of $d = 0$ shown above, the "distance" d is lower-bounded by a value close to 0, which corresponds to the case when the sizes of all modules except one (the one for which the deviation corresponds to δ_{max}) are at the magic number. The value of d would be upper-bounded by some large number depending on the least number of modules whose size deviations can be expected to be within τ. To convert d into a properly normalized metric, we now state[5]

$$MSBI(\mathcal{S}) = e^{-d}. \tag{7}$$

It is evident from (5) that $MSUI$ increases as the standard deviation of the module sizes from the average module size decreases. This obviously addresses the principles related to module size (P7). More specifically, this metric characterizes the software according to the principle *Maximization of Module Size Uniformity*.

Similarly, (7) measures the extent to which the module size bound as specified by Γ is honored by the individual modules. Evidently, this metric also addresses the principles related to module size (P7). More specifically, it characterizes the software according to the principle *Observance of Module Size Bounds*.

We should also mention that there exist several metrics in the literature related to the size of modules. For instance, Halstead length [1], DIT (depth of inheritance), NOC (number of methods), WMC (weighted method count) [7], counts of lines (LOC), modules and procedures [8]. Unlike MSUI and MSBI, these metrics provide numerical counts of various size-related aspects of a system.

6 Architectural Metrics

In the context of this paper, by architecture we mean an organization of the modules into higher-level structures, especially the horizontally layered structures. When a system consisting of millions of lines of code is modularized, it is unlikely that the modules will all exist at the same level of utility. The least organization that will be imposed on the modules will be some sort of horizontal layering in which modules in any one layer are only allowed to call the modules in layers below them. Such layered organization of the modules is considered to be an important architectural principle by the software design community [40], [41].

Imposing such layering on the modules is made complicated by the presence of cyclic dependencies among modules. In the rest of this section, we will present three metrics that relate to layered organization of modules in the presence of possible cyclic dependencies. We will start by first addressing what it means for the modules to be cyclically dependent on one another and we will define a metric that is focused on just this aspect of intermodule relationships. Next, we will address the issue of modules organized in horizontal layers. Here, we will introduce the

5. Note that MSBI is rooted in probabilistic notions. So its properties such as monotonicity, convergence, etc., can only be understood in a probabilistic/statistical framework. We should also mention that we could have used any base for the exponentiation shown in (7).

concept of a *Layer Organization Index* (LOI) and present a related metric that takes into account any cyclic dependencies between the modules that reside in different layers. Finally, our third metric in this section will address the issue of stability/volatility of the modules that reside in different layers of a layered system. In large, layered systems, you want the modules in the lower layers to be relatively more stable with respect to any future changes. Recognizing the difficulty of devising a metric to measure the stability/volatility of a layered system, we will nonetheless follow Martin [35] and present a metric that tries to construct such a measure on the basis of fan-in and fan-out dependencies among the modules.

6.1 Cyclic Dependency Index

We now present a metric that measures the extent of cyclic dependencies between the modules of a system. Following Seng et al. [28], the metric is based on the notion of strongly connected components (SCC) in a directed module dependency graph $MDG = < \mathcal{M}, \mathcal{E} >$, where $\mathcal{M}$ represents the nodes corresponding to the modules and the arcs $\mathcal{E}$ the pairwise dependencies between the modules. An arc connecting two nodes means that the two modules corresponding to the nodes are either explicitly or implicitly dependent on each other. In keeping with our earlier discussion, explicit dependencies come into play when a function in one module calls a function in another module. And implicit dependencies come into a play through shared global variables, a function in one module writing into a file that is read by a function in another module, etc.

An SCC in the graph MDG is a maximal set of vertices (i.e., modules) in which there exists a path from every vertex (in the SCC set) to every other vertex in the same set. (With this definition, an SCC has the desirable property of being able to capture multiple cycles). Furthermore, an SCC has the property that if the module dependency graph does not contain any cycles at all, then the number of strongly connected components is exactly equal to the number of nodes (modules) in the graph. In such a case, for every SCC, we have $|SCC| = 1$.

A cyclic dependency metric can now be formed by comparing the sum of the cardinalities of all the SCCs with the cardinality of the module dependency graph itself. We can write

$$Cyclic(\mathcal{M}) \quad = \quad \frac{number\ of\ SCC\ in\ \mathcal{M}}{|\mathcal{M}|}.$$

This metric evidently caters to the "Principle of Minimization of Cyclic Dependencies Among Modules." The value of $Cyclic(\mathcal{M})$ equals 1 for a system having no cycles. The number of SCCs for an acyclic system is equal to the number of modules.

6.2 Layer Organization Index

A layer provides a set of services that can be used by layers above it. A layer is only aware (function calls are only made to the lower layers) of the layer below it and is not aware of the layer above it [40]. A layer can be thought of as horizontal partitioning of the system. There are two types of layering: 1) Closed, where a function in layer L_i can only make calls to the functions in the immediately lower layer L_{i+1}, and 2) Open, where a function in L_i can make calls to functions in any lower layer.

Suppose $\mathcal{L}$ denotes a mapping from a set of modules $\mathcal{M}$ to an ordered set of layers $\{L_1 \cdots L_p\}$ into which the modules have been organized, with L_1 denoting the topmost layer, L_2 the layer below L_1, and so on. Thus, $\mathcal{L}(m)$ is the layer where the module m resides. Suppose that $K_{ext}^{\overline{L_i}}(m)$ denotes the number of external calls made to a module m (residing in layer L_i) from the modules residing in layers other than L_i. We have used the notation $\overline{L_i}$ to consider the calls from all possible layers except L_i.

Next, let $K_{ext}^{L_{i-1}}(m)$ be the number of external calls made to a module m (residing in layer L_i) from the modules in the immediate upper layer L_{i-1}.

The Layer Organization Index (LOI) for the layer L_i is defined as

$$LOI(L_i) = \frac{\sum_{\forall m \in L_i} K_{ext}^{L_{i-1}}(m)}{\sum_{\forall m \in L_i} K_{ext}^{\overline{L_i}}(m)}.$$

$LOI(L_1) = 0$, i.e., the LOI of the topmost layer is considered to be 0.

Now, we will extend the above metric to the entire system. While formulating the system-level metric, we consider the existence of cycles among modules across layers and within layers. Those of us who are in the thick of dealing with large software systems based on layered architectures often come across situations where peer modules collaborate heavily among themselves to perform a complex business workflow. Such a collaboration can result in cycles. Cycles among peer modules are generally acceptable when they are in the same layer. However, it is highly undesirable if the modules engaged in cyclic dependencies cross the layer boundaries, thereby violating the layering principle. In the LOI metric we present next, the metric will be forgiving of cyclic dependencies within the layers, but will penalize any cyclic dependencies across the layers.

Let us assume that the system consists of q number of SCC components. Obviously, the number of modules in an SCC is ≥ 1, i.e., $|SCC| \geq 1$. When the module dependency graph is acyclic, $q = M$.

Next, let $\mathcal{L}(SCC_i)$ be the set of layers involved in the ith SCC. That is, $\mathcal{L}(SCC_i) = \{\mathcal{L}(m) | m \in SCC_i\}$. Obviously $|\mathcal{L}(SCC_i)| = 1$ for a strongly connected component when all of its modules are in the same layer. On the other hand, $|\mathcal{L}(SCC_i)| > 1$ for the case when a strongly connected component contains modules that populate multiple layers. The condition $|\mathcal{L}(SCC_i)| = 1$ is acceptable, but the condition $|\mathcal{L}(SCC_i)| > 1$ is not. The following formula exhibits this desirability property:

$$1 - \frac{\sum_{i=1}^{q} (|SCC_i| - 1)^{|\mathcal{L}(SCC_i)|}}{(M-1)^p},$$

where $M > 1$ is the total number of modules in the system organized in p layers. This formula has the following characteristics:

- When the module dependency graph is acyclic, $|SCC_i| = 1, i = 1 \cdots q$. In that case, the above formula yields the best result, which is 1.
- At the other extreme, all M modules distributed in p layers exist in one SCC. In such a case,

$$(|SCC_i| - 1)^{|\mathcal{L}(SCC_i)|} = (M-1)^p.$$

  This causes the above formula to yield the worst value of 0.
- The formula $\frac{\sum_{i=1}^{q}(|SCC_i|-1)^{|\mathcal{L}(SCC_i)|}}{(M-1)^p}$ is always bounded between 0 and 1 because of the property that

$$x_1^n + \cdots + x_p^n \leq (x_1 + \cdots + x_p)^n$$

  for nonnegative x_is.

With these observations, we can describe the system level LOI as follows:

$$LOI(\mathcal{S}) = \left(\frac{1}{p-1}\sum_{i=2}^{p} LOI(L_i)\right) * \left(1 - \frac{\sum_{i=1}^{q}(|SCC_i - 1|)^{|L(SCC_i)|}}{(M-1)^p}\right) = 0 \text{ if } p \leq 2. \quad (8)$$

Observe that the system level LOI is impacted by both the interlayer calls and the skipping of layers in the interlayer calls. Also, larger the cycles, lower is the value of LOI.

Clearly, the LOI metric characterizes software according to the principle of Maximization of the Unidirectionality of Control Flow in Layered Architectures (P6) and strongly discourages the existence of cycle across multiple layers.

6.3 Module Interaction Stability Index

As mentioned in the introduction to this section, horizontal layering is the least organization one would want to impose on the modules of large and complex software. A desirable property of such an architecture is that the modules in the lower layer be more stable in relation to the modules in the upper layers and that any changes made to the overall software be more confined to the upper layers and, to the extent possible, not impact the lower layers.

Measuring such characteristics in a quantitative way is challenging, since it is difficult to anticipate a priori whether a given module is likely to change, as changes often depend on some unpredictable external situations such as market pressures and customer requests. (In recent years, there have been efforts to predict and visualize module changes based on the revision histories [42], [43], but these do not provide a mechanism to measure the effect of module changes quantitatively.) Martin [35] has proposed a simplified approach to measure the stability based on the fan-in and fan-out of modules in a module dependency graph. Though this metric in [35] was originally proposed in the context of object-oriented systems, we have adopted its underlying concepts (such as afferent and efferent couplings, and instability) to develop a similar metric for non-object-oriented systems. We call this metric the Module Interaction Stability Index (MISI).

Following the notation in the previous subsection, let $\mathcal{L} : \mathcal{M} \rightarrow \{L_1 \cdots L_p\}$ be the assignment of the modules to the different layers and $\mathcal{L}(m)$ be the layer where m resides. Furthermore, in the stack of layers, let L_1 and L_p be the highest and the lowest layers, respectively. We now rank-order the layers with the $\leq$ operator defined as

$$L_1 \leq L_p \text{ and } L_i \leq L_j \; iff \; 1 \leq i, j \leq p \; \& \; i \leq j.$$

For a given module m, let $fanin(m)$ be the set of modules that depend on m and let $fanout(m)$ be the set of modules that m depends on. Following Martin [35], the instability $\mathcal{I}(m)$ of m is defined as

$$\mathcal{I}(m) = \frac{|fanout(m)|}{|fanin(m)| \; + \; |fanout(m)|}.$$

Let $SD(m) \subseteq fanout(m)$ be the set of stable modules with respect to m, defined as

$$SD(m) = \{m_i \in fanout(m) \mid \mathcal{I}(m) > \mathcal{I}(m_i) \; \& \; \mathcal{L}(m) \leq \mathcal{L}(m_i)\}.$$

The above equation states that $SD(m)$ are only those modules 1) that m depends on 2) that are more stable than m and 3) that reside in layers lower than (or the same as) the layer of the module m.

Now, MISI can be defined as

$$MISI(m) = \frac{|SD(m)|}{|fanout(m)|} = 1 \text{ when } fanout(m) = \emptyset, \quad (9)$$
$$MISI(\mathcal{S}) = \frac{1}{M}\sum_{i=1}^{M} MISI(m_i).$$

This is the same M as in (8). For a well-modularized system, all or most of the modules would depend on modules which are more stable and reside in lower layers. Hence, for a well-modularized system, $|SD(m)| \simeq |fanout(m)|$ and $MISI(m) \rightarrow 1$. Clearly, if a system has a high MISI index, any changes or extensions made to the system will affect only a small number of modules in the upper layers of the system. The metric conforms to the principle of Maximization of Stand-Alone Module Extendibility (P4).

6.4 Testability Index

It goes without saying that a module that has few dependencies on other modules would be easier to test.[6] So, it is useful to measure the extent to which modules depend on one another with regard to testability. Note that just because a module m_1 calls an API function of module m_2 does not imply that m_1 depends on m_2 from the standpoint of testability. A testability-dependence is created if the assessment of the correctness of the input-output relationships during the testing of a module depends on the correctness of the input-output relationship of another module. That is, if the testing of module m_1 requires

6. Comprehensive testing of modules includes both requirements-based testing and testing for the correctness of the logic of a module, the latter also referred to as "structured testing" or "white-box" testing. Structured testing, focussed more on the logic internal to the individual functions of a module, uses cyclomatic complexity of the implementation code for the functions. For example, the number of test cases for a function must equal the number of "basis paths" in the control-flow graph of the function.

another module m_2 to produce intermediate results that then get used in module m_1, we have a testability-dependence between the two modules. We measure the extent of such testability-dependencies with the help of the Normalized Testability-Dependency Metric ($NTDM$) that is defined as described below.[7]

Let $dep(m_i)$ be the number of *other* modules that a given module m_i has a testability-dependence on. We now define a Testability-Dependency Count (TDC) as

$$TDC \quad = \quad \sum_{i=1}^{M} dep(m_i),$$

where M is the total number of modules in the system. We now define our $NTDM$ metric to be

$$NTDM = 1 - \frac{TDC}{M \times (M-1)}. \tag{10}$$

Note that TDC varies from 0 to $M \times (M-1)$ depending on the extent of testability-dependencies between the modules. The upper limit is reached when a module depends on every other module (i.e., $M-1$) with regard to its testability, and the lower limit is reached when every module depends only upon itself and is independent of all other modules from the standpoint of testability. The value of $NTDM$ will be 1 when the modules are independently testable (i.e., when $dep(m) = 0$), and it will be 0 when every module depends on every other module for its testability.

This metric obviously characterizes the software according to the principle of Maximization of the Stand-Alone Testability of Modules (P5).

7 Metrics Based on Similarity of Purpose

In software intended for business applications, it is frequently the case that either the name of a function or the names of some of the variables used in the function, or the comments associated with a function hold some clue to the purpose of the function inside the module and to the purpose of the module itself. For example, this could be the case for a module devoted to interest calculations in banking software; the word "interest," either fully or in some abbreviated form, is likely to show up in various artifacts of the module. The developers of complex software often leave behind such clues because it makes for easier reading of the software and for its debugging.[8] We can refer to these clues as concepts.[9] With regard to software characterization, we may then use the concepts to assess whether or not the software adheres to the principles that are based on the semantics of the module contents.

Metrics proposed in this section are based on the assumption that the domain experts have provided a set of concepts that may appear as keywords and/or their synonyms in the body of the code. Let $\mathcal{C} = \{c_1, \ldots, c_N\}$ denote a set of N concepts for a given system S. Also assume that the system consists of set of functions $\mathcal{F} = \{f_1, \ldots f_F\}$ functions distributed over $\mathcal{M}$ modules $\{m_1, m_2, \ldots, m_M\}$. In the rest of this section, we will also use the symbol f_i as denoting the set of all functions that are in module m_i. For each function, we first search the number of occurrences of each concept (and its synonyms) in the function signature and function body. Let us denote the concept frequency as $\mathcal{H}_f(c)$ to be the frequency of occurrence of the concept $c \in \mathcal{C}$ for a given function $f \in \mathcal{F}$.

TABLE 1
Concept Frequency Table

Function	authentication	platform	caching	protocol	logging
allow_cmd	3	0	0	0	0
check_dir_-access	4	0	0	5	3
create_access_-dir_config	3	0	0	0	0
authenticate-basicuser	21	0	0	3	6
checkuser-access	29	0	0	5	6

As an illustration of the concept content of a body of software, consider the Apache httpd 2.0.53 sources that contain concepts such as `authentication`, `caching`, `protocol`, `logging` and so on. Table 1 shows the frequencies of some of the main concepts and their synonyms in the software. In this example,

$$\mathcal{H}_{allowcmd}(authentication) = 3.$$

Next, we normalize the frequency distribution of these concepts. It may so happen that a particular concept occurs more frequently than other concepts across all functions, thereby causing apparent skewing of the distribution. In order to avoid that, we first find the global maximum of concept frequency. Let

$$\mathcal{H}^{max}(c_i) = \max\{\mathcal{H}_f(c_i), \forall f \in \mathcal{F}\}.$$

We then normalize each frequency as $\widehat{\mathcal{H}}_f(c_i) = \frac{\mathcal{H}_f(c_i)}{\mathcal{H}^{max}(c_i)}$.

7.1 Concept Domination Metric (CDM)

If a module is highly purpose focussed, a probability distribution of the concepts in the module will be peaked at the concept corresponding to the purpose of the module. So we can say that the more nonuniform the probability distribution of the concepts in a module, the more likely that the module conforms to the singularity of purpose. The Concept Domination Metric (CDM) gives us a measure of this non-uniformity of the probability distribution of the concepts. In the worst case, all the concepts will be uniformly distributed and the value of the metric would be expected to go to zero. A convenient measure from probability theory that has such properties is the Kullback-Leibler divergence. We will now show this divergence can be adapted for a metric suited to our purpose.

We start by creating a normalized concept frequency at the module level. For a given module m, the frequency of occurrence of a given concept c_i can be obtained from the

7. Our $NTDM$ metric is an adaptation of the *Cumulative Component Dependency* (CCD) metric of Lakos [44].

8. This is likely to be even more so in the future because of the modern emphasis on software that is self-documenting.

9. We are not using the notion of a concept in the same formal sense as in [45], [46] where a concept is defined over binary relations between objects and attributes. In our work, any artifact that holds a clue as to the purpose of a module is a concept.

normalized concept frequency of all the functions that constitute the module as

$$\widehat{\mathcal{H}}_m(c_i) = \sum_{\forall f \text{ in } m} \widehat{\mathcal{H}}_f(c_i).$$

We next find the concept c^m_{max} that maximizes $\widehat{\mathcal{H}}_m(c_i)$ for each module m, i.e. $\widehat{\mathcal{H}}_m(c^m_{max}) = \max_{c_i} \widehat{\mathcal{H}}_m(c_i)$ Next, we convert the normalized frequency into a discrete probability measure as

$$g_m(c_i) = \frac{\widehat{\mathcal{H}}_m(c_i)}{\sum_{j=1}^{N} \widehat{\mathcal{H}}_m(c_j)}, \tag{11}$$

where N is the total number of concepts as mentioned earlier.

We next measure the Kullback-Leibler divergence of a uniform probability distribution from the probability distribution g_m:

$$KL(g_m) = \log N + \sum_{i=1}^{N} g_m(c_i) \times \log(g_m(c_i)), \tag{12}$$

where the logarithms are to base 2.

We now define the dispersion of the concepts in any given module m relative to all concepts whose normalized frequencies of occurrence are within a predefined threshold θ of $g_m(c^m_{max})$. This permits the purpose of a module to be stated in terms of not just one dominating concept but by a set of concepts that are more or less equally important. The following rule partitions the set $\mathcal{C}$ of concept for a module m into two subsets $\mathcal{C}^1_m$ and $\mathcal{C}^2_m$:

$$\mathcal{C}^1_m = \begin{cases} \{c_i | g_m(c_i) > \theta \cdot g_m(c^m_{max})\}, & if \;\; KL(g_m) > \Psi \\ \emptyset, & otherwise, \end{cases} \tag{13}$$
$$\mathcal{C}^2_m = \mathcal{C} - \mathcal{C}^1_m.$$

Note that the existence of $\mathcal{C}^1_m$ is predicated on the KL-divergence exceeding a certain threshold. If the KL-divergence does not exceed this threshold, the concept distribution is much too uniform to allow for the partitioning of $\mathcal{C}$ into a subset $\mathcal{C}^1_m$ of dominating concepts.

We will now state the following observation to establish some desirable properties for the KL-divergence and the concept subsets $\mathcal{C}^1_m$ and $\mathcal{C}^2_m$:

Observation. *When the total number of concepts satisfies $N = 2^k$, the $KL(g_m)$ divergence of (12) is bounded by $[0, k]$. When a module contains only one concept, the $KL(g_m)$ distance becomes maximum and equal to k.*

Proof. Being nonnegative, the divergence $KL(g_m)$ is bounded on the low side by 0. $KL(g_m)$ becomes zero when the discrete probability distribution g_m is uniform. At the other extreme, when a module m contains only one concept $c \in \mathcal{C}, g_m(c) = 1$ and for all other $c' \in \mathcal{C}, g_m(c') = 0$. In that case, $KL(g_m) = \log N + \log 1$ which is equal to k.□

We define the dispersion for a module m with respect to the concept c^m_{max} and other almost equally frequently occurring concepts in $\mathcal{C}^1_m$ by

$$\mathcal{D}_m = \frac{\sum_{c_i \in \mathcal{C}^1_m} g_m(c_i)}{\sum_{c_i \in \mathcal{C}^1_m} g_m(c_i) + \sum_{c_i \in \mathcal{C}^2_m} g_m(c_i)}. \tag{14}$$

We need to show that the value of $\mathcal{D}_m$ will always be close to 1 when $\mathcal{C}^1_m$ is of low cardinality and has the maximum concept frequency. Note that when $\mathcal{C}^1_m$ is not $\emptyset$, its cardinality is always low. Further, when $\mathcal{C}^1_m$ has maximum concept frequencies, $\sum_{c_i \in \mathcal{C}^1_m} g_m(c_i) \gg \sum_{c_i \in \mathcal{C}^2_m} g_m(c_i)$. This in turn will result in $\mathcal{D}_m \to 1$.

Finally, we define the Concept Domination Metric (CDM) for the system as

$$\mathcal{D} = \frac{1}{M} \sum_{m \in \mathcal{M}} \mathcal{D}_m, \tag{15}$$

where M is the total number of modules.

This metric measures the extent to which the modules in a software system are cohesive with respect to the concepts that contain the largest number of clues regarding the main purpose of a module. This metric characterizes a modularization according to the principle Maximization of Module Coherence on the Basis of Similarity and Singularity of Purpose (P1). Since this metric also takes into account the dispersion of concepts, the metric also characterized the modularization according to the principle Minimization of Purpose Dispersion (also P1).

7.2 Concept Coherency Metric

Using the concept probability function $g_m(c)$ for a module m as given by (11), we will now formulate another metric that is along the lines of the criterion used in [31] for software clustering. We call this metric Concept Coherency Metric. This metric is based on the information-theoretic notion that if a concept is central to the services offered by a module, then the mutual information between the module and the concept should be high. Mutual information between a concept and a module becomes high when each is able to predict the likelihood of the occurrence of the other with a high probability. Obviously, if a concept is orthogonal to the services offered by a module, neither can tell us much about the likelihood of the occurrence of the other. In this case, the mutual information between the two will be close to zero.

To formulate this metric, let $\mathbf{C}$ be a random variable whose value can be any element of the set $\mathcal{C}$ of all concepts. Similarly, let $\mathbf{M}$ be a random variable that can take on any value from the set $\mathcal{M}$ of all modules. The mutual information between the concepts and the modules is given by

$$I(\mathbf{C}; \mathbf{M}) = H(\mathbf{C}) - H(\mathbf{C}|\mathbf{M}),$$

where $H(\mathbf{C})$ denotes the entropy associated with all the concepts in the system, and $H(\mathbf{C}|\mathbf{M})$ denotes the conditional entropy of the concepts vis-a-vis the modules. The entropy $H(\mathbf{C})$ can be expressed as

$$H(\mathbf{C}) = -\sum_{c \in \mathcal{C}} \rho(c) \log \rho(c),$$

where $\rho(c)$ is the discrete probability function over the concepts, the probability function expressing the relative frequencies of all the concepts in the system. The conditional entropy is given by

$$H(\mathbf{C}|\mathbf{M}) = -\sum_{m \in \mathcal{M}} \rho(m) \sum_{c \in \mathcal{C}} g_m(c) \log g_m(c),$$

where $\rho(m)$ denotes the probability with which a particular module m is selected from the set $\mathcal{M}$ of modules. If we assume this probability to be uniform, the expression for conditional entropy reduces to

$$H(\mathbf{C}|\mathbf{M}) = \frac{1}{M}\Bigg(- \sum\nolimits_{c \in \mathcal{C}} g_{m_1}(c) \log g_{m_1}(c) - \cdots - \sum\nolimits_{c \in \mathcal{C}} g_{m_M}(c) \log g_{m_M}(c)\Bigg).$$

We now define the Concept Coherency Metric (CCM) as

$$CCM \quad = \quad \frac{I(\mathbf{C};\mathbf{M})}{\log(|\mathcal{C}|)}. \tag{16}$$

Following the information theoretic notion, we can interpret CCM as the normalized measure of the extent to which one can predict the occurrence of a concept when a module is given. Observe that the denominator, used for normalization, represents the best possible scenario where the conditional entropy $H(\mathbf{C}|\mathbf{M}) = 0$ (i.e., the uncertainty with which one can predict the occurrence of a concept for a given module is 0).

It may further be observed that, in the worst case, one may have all of the software in a single module (a monolithic system having $|\mathcal{M}| = 1$). In the worst case scenario, the $I(\mathbf{C};\mathbf{M})$ becomes 0 as explained below:

$$\begin{aligned} I(\mathbf{C};\mathbf{M})|_{|\mathcal{M}|=1} &= H(\mathbf{C}) - H(\mathbf{C}|\mathbf{M}) \\ &= H(\mathbf{M}) - H(\mathbf{M}|\mathbf{C}) \\ &= -\sum\nolimits_{m \in \mathcal{M}} \rho(m) \log \rho(m) + \\ &\quad \sum\nolimits_{c \in \mathcal{C}} \rho(c) \sum\nolimits_{m \in \mathcal{M}} \rho(m|c) \log(\rho(m|c)) \\ &= 0 \text{ since } \rho(m) = 1 \textit{ and } \rho(m|c) = 1). \end{aligned}$$

Vis-a-vis the CDM metric, the CCM metric adopts an information-theoretic approach to measure the extent to which the modules in the system are cohesive with respect to the main concepts embodied in the modules. Like CDM, this principle also characterizes a modularization according to the principle Maximization of Module Coherence on the Basis of Similarity and Singularity of Purpose (P1).

8 Modularity Principles—Modularity Metrics Conformance

Table 2 summarizes the connection between the principles enunciated in Section 3 and the metrics presented in Section 4 through 7.

9 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+, wxWindows, JDK and many others, are object-oriented. Our metrics are not meant for such software.

TABLE 2
The Metrics-Principles Connection

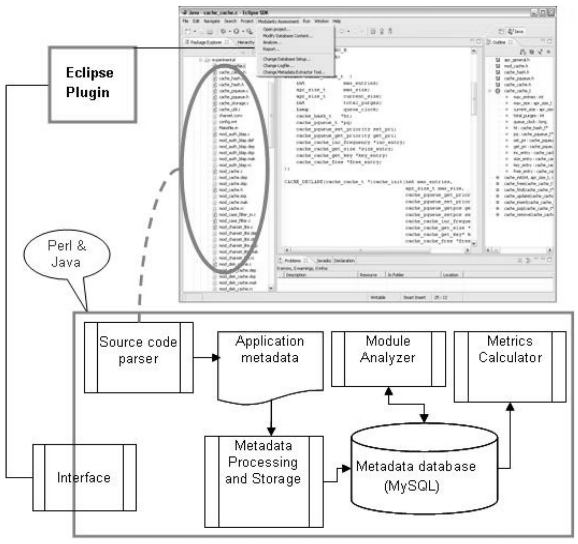
Modularity Principle	Metric
P1: Similarity of Purpose, Minimization of Purpose Dispersion	CDM,CCM
P1: Maximization of Module Coherence on the Basis of Commonality of Goals	APIU
P2: Maximization of API-Based Inter-Module Call Traffic	MII
P2: Minimization of non-API Based Inter-Module Call Traffic	MII, 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.

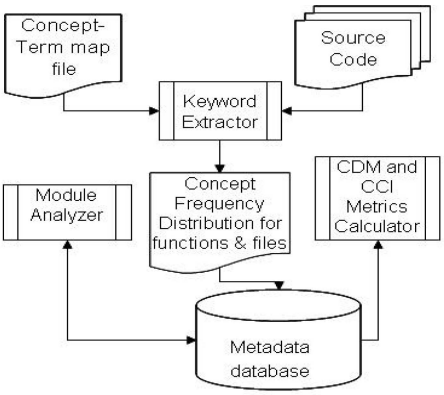
The open source software systems chosen for reporting our experimental results in this section—MySQL, Apache, Mozilla, GCC, the Linux kernel, and Postgresql—are highly regarded in academia and industry for their robustness and for the quality of code. For these software systems, we took the directory 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

10. Note that declaring *every* externally called function to be an API function would defeat the whole point of a modularization exercise. Ideally, our goal in modularization must be to create seed groupings of functions based on the similarity of purpose and/or commonality of goals, declare these seeds to be the initial API's for the respective directories/modules, and then try to assign the other externally called functions in a source module to other modules or to modify the purpose/goal definition of the source module in such way that it can stay in the current module.



(a)



(b)

Fig. 1. Modularity Assessment Tool Architecture. (a) Schematic diagram. (b) Concept extraction scheme.

extract the metrics presented in this paper. These steps are depicted in Fig. 1a. The output produced by Sourcenav is summarized in Table 3.

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

TABLE 3
Software Systems Used for Metrics Validation

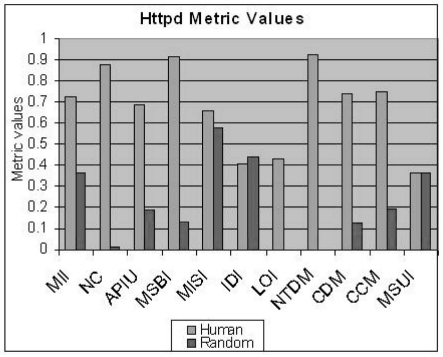
Name	#files	LOC	LOC wo comments	#functions	#Cross Ref	#Leaf directory
apache-2.0.53	940	403918	358982	5811	4567	111
Mysql-4.1.12	2187	1285675	1107722	28457	59415	153
mozilla-1.7	10214	3610671	2777730	97210	144920	703
GCC-3.3	1865	1152077	1006977	17156	37362	58
Linux kernel-2.0.27	1612	700656	608039	12880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

9.1 API Identification Heuristics for Open-Source Software Examples

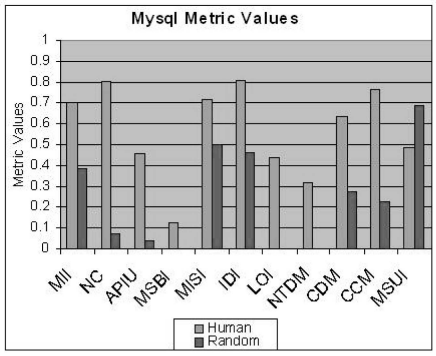
In each of these scenarios, a key challenge was how to identify the API functions vis-a-vis the non-API functions in a given module. For the purpose of this particular experiment, we identify API functions with the heuristic that an ideal API function is mostly called by functions belonging to other modules. Similarly, an ideal non-API function is never exposed, i.e., it is never called by functions from other modules. According to this heuristic, in a given modularized version, a module function was declared to be an API function or a non-API function depending on the relative frequencies of the function being called from other modules vis-a-vis the function being called from within its home module. We defined two threshold parameters α, β, both between 0 and 1, for this purpose. For a function f in a module m having $K(f)$ number of calls made to it, if $K_{ext}(f) > \alpha \cdot K(f)$, we consider f to be an API function for m. Similarly, a function f' in the module m is treated as a NON-API function if $K_{int}(f') > \beta \cdot K(f)$.

9.2 Concept Extraction

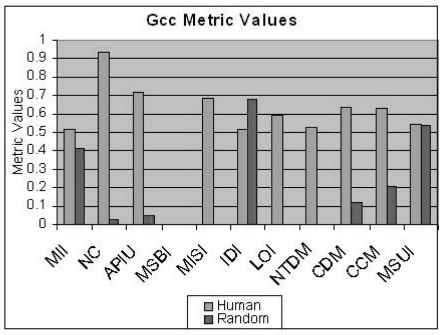
In order to evaluate the CDM and CCM metrics, we must extract various domain concepts from the source code and create a frequency distribution of the concepts for each module. The concept extraction process is shown in Fig. 1b. The extraction process takes an input file containing a set of business concept names and the associated terms or keywords 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



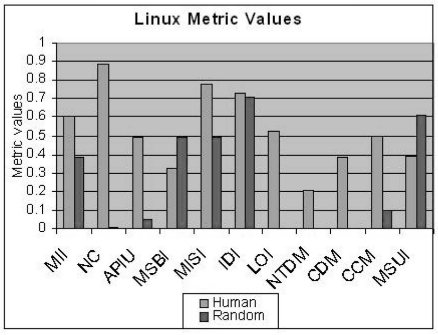
(a)



(b)



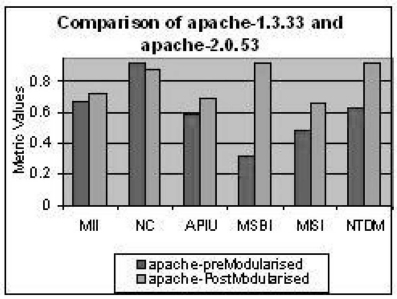
(c)



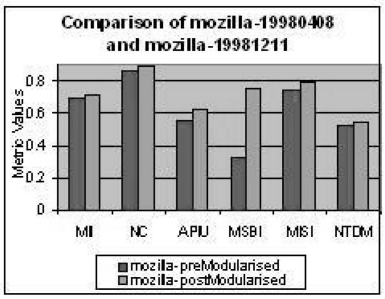
(d)

Fig. 2. Comparison of metric values for Human and random modularization of some systems. (a) httpd-2.0.53 metric values. (b) Mysql metric values. (c) Gcc metric values. (d) Linux metric values.

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



(a)



(b)

Fig. 3. Comparison of metric values for different versions of open source software. (a) Comparison of metric values for Apache-1.3.33 and Apache-2.0.53. (b) Comparison of metric values for Mozilla-19980408 and Mozilla-19981211.

9.3 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 set 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, `userauthentication` 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. 4 and Fig. 5.

9.4 Analysis of Results

As shown in Fig. 2, the core metric values take a turn for the worse when the original modular structure of the software is destroyed. When the modular structure is destroyed by randomly assigning functions and files to modules, the internal coherence of the modules is broken. This, in turn, causes the MII, NC, and APIU values to degrade.

The size-based metrics, on the other hand, often do not degrade when we go from human-supplied modularization to the random scenario. Since the random assignments tend to create modules of roughly the same size, it stands to

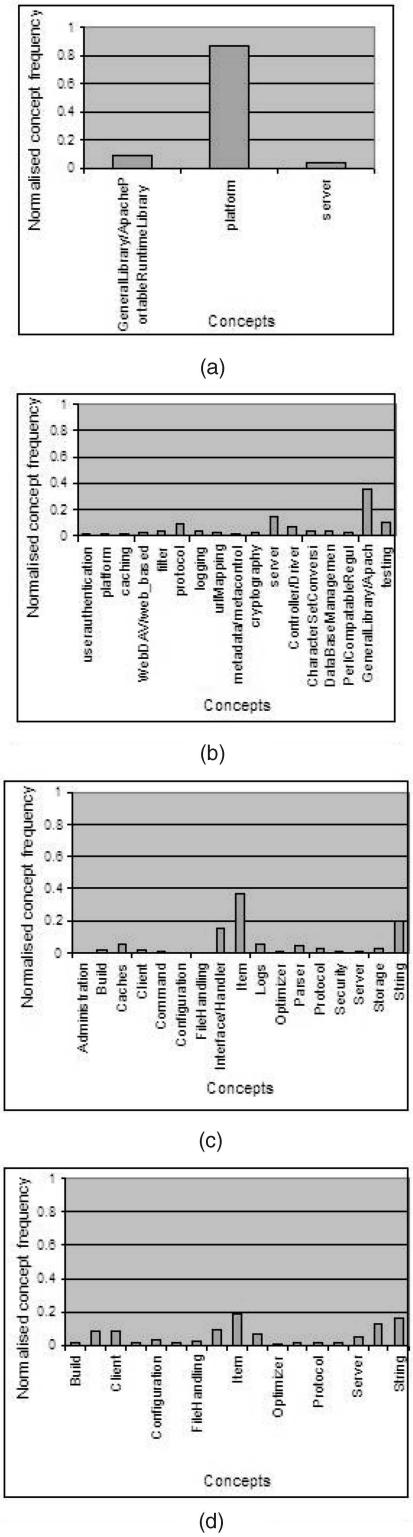


Fig. 4. Dispersion of concept frequencies for the named modules for Human and Random Modularizations. (a) Concept distribution for module os/beos for Apache with human modularization. (b) Concept distribution for module-13 for Apache with random "modularization." (c) Concept distribution for module mysqld for mysql with human modularization. (d) Concept distribution for module-137 for mysql with random "modularization."

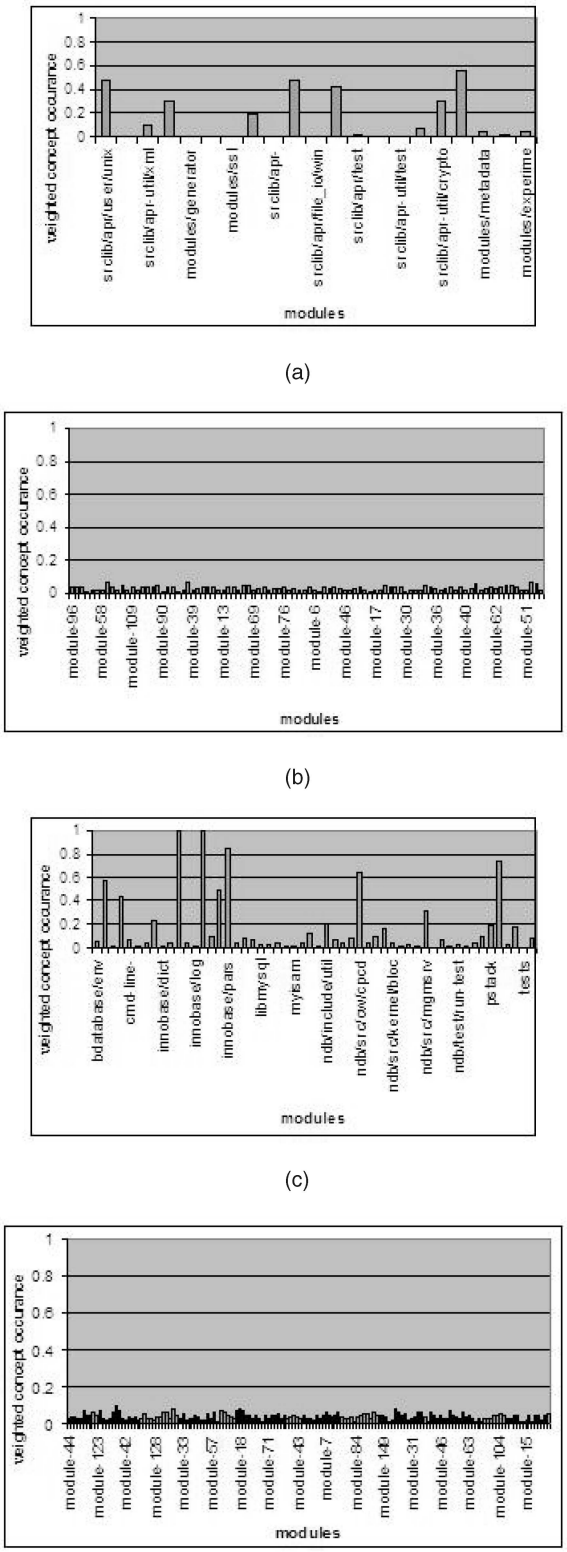


Fig. 5. Concept frequencies for two sample concepts across all modules with human and random modularizations. (a) Distribution of the concept "userauthentication" for httpd with human modularization. (b) Distribution of the concept "userauthentication" for httpd with random "modularization." (c) Distribution for the concept "parser" for mysql with human modularization. (d) Distribution of the concept "parser" for mysql with random "modularization."

reason that MSUI for the random scenario could be higher compared to its value for the original software. About MSBI, it is interesting to note that software created by the expert humans frequently contains modules whose sizes vary considerably. This causes the MSBI values for the original software to be low.

Regarding the IDI metric, recall that the IDI of a module is the ratio of the number of nonimplicit dependency-producing function calls to the total intermodule function call traffic emanating from that module. The randomization process randomly assigns functions to modules, but that does not alter the implicit dependencies of the sort we mentioned in Section 4.4. As a result, the intermodule dependency (through global variables) counts remain mostly unchanged. Furthermore the random assignment of functions to modules in many cases does not have a significant impact on the total count of external function calls.

These observations regarding the various metrics as we go from human modularization to random modularization speak to the fact that any single metric all by itself does not tell us much about the quality of modularization. For a composite picture of the quality of modularization, one has no choice but to examine all the metrics.

9.4.1 Comparison of Different Versions of Software Systems

Regarding the results shown in Fig. 3 for two different versions of the same software, Apache 2.0 seems to be a better-modularized version. For Mozilla, we chose the two versions of the software used by MacCormack et al. [48]. According to the empirical study reported in [48] and the Mozilla development roadmap [49], Mozilla-19981211 is a better-modularized version. The results in Fig. 3 show a marginal improvement of most of the metric values for Mozilla-19981211. As to whether the differences in the modularization quality as recorded by our metrics are proportional to the actual differences in the software, that is obviously open to debate. Our following remarks are pertinent to this comparison for the four different metrics MII, NC, APIU, and MISI:

- MII: One of the goals of the redesign of the Mozilla architecture according to the Mozilla development roadmap [49] and the discussions in the Mozilla-general mailing list is the specification of interfaces for each of the components in Mozilla using XPIDL, an interface definition language similar to the CORBA IDL. The developers wanted to ensure that all intercomponent interactions are through the API functions published in these interfaces. Since this follows the principle "Maximization of API-Based Inter-Module Call Traffic," we expect MII of Mozilla-19981211 to be greater than that of Mozilla-19980408. As shown in Fig. 3b, MII for the postmodularized version of Mozilla is 0.72, whereas it is 0.69 for the premodularized version. Admittedly, the difference between the metric values is not large. The reason, we suspect, is that since in the experiment we considered the leaf-level directories as modules and since there did not exist a direct one-to-one mapping between the leaf level directories and the Mozilla components, some intramodule dependencies were wrongly considered as intermodule dependencies, thus affecting the computation of MII.
- NC: For enhanced modularization, the new version of Mozilla architecture was designed using XPCOM, which is a component object model similar to COM. The XPCOM framework insists that only the functions declared in the interfaces be called by other components. We should therefore expect NC to have a larger value for the newer version of Mozilla. As shown in Fig. 3b, NC for Mozilla-19980408 is 0.86, whereas the value for Mozilla-19981211 is 0.90.
- APIU: The more recent Mozilla-19981211 has 40 "modules" with APIU values greater than 0.5, whereas the older Mozilla-19980408 has 28 "modules" with APIU values equal to 1 (the other "modules" have APIU values less than 0.5). This indicates that the "modules" of the newer version of Mozilla possibly exhibit a greater similarity of purpose. However, we could not find any documentation at the Mozilla Web site to ascertain whether maximizing similarity of purpose was one of the goals of the modularization effort.
- MISI: The source code facts extracted from the two versions reveal that 26 "modules" in Mozilla-19980408 and 50 "modules" in Mozilla-19981211 have stable dependencies. The MISI value value for the former is 0.74 and for the latter 0.79.

As mentioned already, we also tested the metrics on several proprietary business applications and obtained similar results. In particular, we evaluated a pre- and postmodularized version of a large business application. A partial modularization of the code was carried out manually; its focus was the identification of the API functions for each module, enforcement of the requirement that the intermodule calls take place through the API functions, and, to the maximum extent possible, reduction of the number of global-variable-based interactions among the modules. We noticed a significant improvement in the values of the MII, NC, APIU, MISI, and IDI metrics for the modularized version of the business application.

9.4.2 Analysis of Concept Dispersion

To analyze how the concepts might be dispersed/concentrated across the various modules, we examined the frequency distributions for the concepts. Obviously, if a module has roughly equal occurrences of multiple concepts, it becomes difficult to identify the dominating concept for the module and difficult to attribute to the module a singularity of purpose. As shown in Fig. 4a, Apache's `os/beos` module is dominated by the concept `platform`; the other concepts are not present as strongly as `platform` in this module. On the contrary, if we take any sample module from a random modularization of httpd-2.0 in Fig. 4b, we don't see any strong dominance by a single concept. Therefore, $\mathcal{D}_{os/beos}$ becomes 0.8695 but $\mathcal{D}_{module-13}$ is 0.3152. This observation is also true for the MySQL software system.

As shown in Fig. 5, for httpd-2.0.53, the occurrence of the concept `userauthentication` is high only for a small number of modules for the case of human modularization (Fig. 5a) but it appears in almost all the modules with random modularization (Fig. 5b). This conclusively demonstrates why the CDM and CCM values are significantly low for the random case.

10 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-boundedness, 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. 3.)

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 Kitchenham 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.

Acknowledgments

The authors would like to thank N.S. Nagaraja for his support of this project. Thanks are also owed to Kenneth Heafield and Ziad Mohamad Naamani for valuable discussions related to the theoretical aspects of the work reported here and Shubha Ramachandran for her help with issues related to the implementation of the 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 Framework 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 of Source 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 Maintenance (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 Clustering," *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 Omnipresent 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. 2005.
[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.

Santonu Sarkar received the PhD degree in computer science from IIT Kharagpur in the area of object-oriented modeling of VLSI circuits. He is a principal architect at SETLabs, Infosys Technologies Ltd., India. He has nearly 14 years of industrial software development experience in business applications and product development. His current research interests include architecture modeling and analysis, enterprise architecture, program comprehension, and reengineering techniques.

Girish Maskeri Rama received the master's degree from the University of York, UK, and has been carrying out applied research and tool development in the area of software engineering for nearly seven years. He is a research associate at SETLabs, Infosys Technologies Ltd., India. Apart from program comprehension and software reengineering, his other research interests include metamodeling and model transformations. He was part of the QVTP consortium and contributed to the specification of the OMG standard MOFQVT.

Avinash C. Kak is a professor of electrical and computer engineering at Purdue University and a consultant to Infosys. His most recent book, *Programming with Objects* (John Wiley and Sons, 2003), is used by a number of leading universities as a text on object-oriented programming. His forthcoming book *Scripting with Objects* focuses on object-oriented scripting. These are two of the three books for an "Objects Trilogy" he is creating. The last, expected to be finished sometime in 2008, will be titled *Designing with Objects*. In addition to computer languages and software engineering, Dr. Kak's research areas include sensor networks, computer vision, and robotic intelligence. His coauthored book *Principles of Computerized Tomographic Imaging* was republished as a Classic in Applied Mathematics by SIAM (Society of Industrial and Applied Mathematics).