Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software

Santonu Sarkar, *Member*, *IEEE*, Avinash C. Kak, and Girish Maskeri Rama

Abstract—The metrics formulated to date for characterizing the modularization quality of object-oriented software have considered *module* and *class* to be synonymous concepts. But, a typical class in object-oriented programming exists at too low a level of granularity in large object-oriented software consisting of millions of lines of code. A typical module (sometimes referred to as a *superpackage*) in a large object-oriented software system will typically consist of a large number of classes. Even when the access discipline encoded in each class makes for "clean" class-level partitioning of the code, the intermodule dependencies created by associational, inheritance-based, and method invocations may still make it difficult to maintain and extend the software. The goal of this paper is to provide a set of metrics that characterizes large object-oriented software systems with regard to such dependencies. Our metrics characterize the quality of modularization with respect to the APIs of the modules, on the one hand, and, on the other, with respect to such object-oriented intermodule dependencies as caused by inheritance, associational relationships, state access violations, fragile base-class design, etc. Using a two-pronged approach, we validate the metrics by applying them to popular open-source software systems.

Index Terms—Superpackage, modularization, software metrics/measurement, module, interface, object-oriented software, large-scale software, maintenance and enhancement, maintainability, coupling.

1 Introduction

OBJECT-ORIENTED software is of more recent vintage than the old-style procedural code. Nonetheless, there now exist many commercial object-oriented applications that show the same signs of aging as the legacy procedural code. So what may have started out as a well-modularized architecture may have decayed into a system of interdependent modules that are difficult to maintain and extend on an individual basis.

As is well known in the literature, whereas the main factor responsible for the disorganization of the procedural legacy code is the unregulated function-call traffic between what might have started out as well-partitioned modules, for object-oriented legacy code, there exist multiple avenues that can lead to a high state of software disorganization. These include a module extending a class defined in another module, a class in one module using an instance of a class in another module as an attribute or as a parameter in a method definition, the implementation code for a method in a class in one module calling on a method defined for a class in another module, etc. The metrics we propose in this paper seek to characterize the quality of a given modularization with respect to each such mechanism for code degradation.

To be sure, there does exist a significant body of literature dealing with the metrics for object-oriented software. But, as we point out in the next section, much of this previous work considers a module and a class as synonymous concepts. Even when the contributors have not explicitly stated their metrics on a per-class basis, many of the core metrics can often be reformulated straightforwardly on a per-class basis.

With an eye to the future, our goal in this paper is to go beyond the OO metrics formulations of the past. Our vision of the future is dictated by the fact that software for many commercial applications now runs into millions of lines of code. When such code is object-oriented, as is increasingly the case, the granularity of a class is often at much too low a level to serve as a unit for software modularization. A module in large object-oriented software may contain hundreds of classes. As in the software of the past, the purpose of a module is still to provide some focused functionality to the rest of the world. Nonetheless, due to its complexity, it may take a large number of classes to implement that functionality.

Since many of the classes in the sort of modules we have mentioned above are intended solely for the internal functioning of a module, it makes no sense to simply declare all of the public methods for all of the classes as the API of a module. It makes a lot more sense to construct module-level APIs with only those public methods that are

- S. Sarkar is with Accenture Technology Labs, IBC Knowledge Park, 4/1 Banerghatta Road, Bangalore, India, Pin-560029. E-mail: santonu.sarkar@accenture.com.
- A.C. Kak is with Purdue University, 1285 EE Building, West Lafayette, IN 47907-1285. E-mail: kak@ecn.purdue.edu.
- G.M. Rama is with SETLabs, Building 19FF, Infosys Technologies Ltd., Electronic City, Hosur Road, Bangalore, India, Pin-560100. E-mail: girish_rama@infosys.com.

Manuscript received 6 Sept. 2007; revised 22 Feb. 2008; accepted 20 May 2008; published online 5 June 2008.
Recommended for acceptance by K. Inoue.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2007-09-0261.
Digital Object Identifier no. 10.1109/TSE.2008.43.

intended to be used by the outside world.[1] The metrics we propose in this paper are for such large object-oriented software. Our metrics measure the quality of the modularization of the software by measuring the extent to which all intermodule method call traffic is directed through the APIs of the modules, the extent to which the module are free of the linkages created by inheritance and associations, etc. Our metrics also measure the quality of the APIs from the standpoint of the cohesiveness of the services provided, etc.

Since the metrics we propose in this contribution are API-centric, we must bring to the reader's attention our previous contribution [4], which also has API-based metrics as its primary focus. But note that the API-based metric suite proposed in [4] is for non-object-oriented systems only. The intermodule couplings created by inheritance, containment, access control, polymorphism, encapsulation, etc., are strictly outside the purview of the metrics proposed in [4].

In what follows, Section 2 presents a review of the previous work in metrics for object-oriented software. In order to accommodate the increasingly popular notion of *plugins*, Section 3 provides a brief overview of two different types of APIs, one a Service API and the other an Extension API. A Service API is the traditional API for software modules. An Extension API describes the services that must be provided by an external plugin module. To get ready for the presentation of our metrics, Section 4 presents the notation we have used in the rest of this paper. Our metrics are presented in Sections 5 through 13. As a convenience for the reader, Section 14 tabulates all of the metrics and states succinctly the main rationale for each metric. Section 15 presents an exhaustive experimental validation of the new metrics. Finally, we conclude in Section 16.

2 Previous Work on OO Software Metrics Relevant to Our Contribution

Object-oriented software is based on the notions of class, encapsulation, inheritance, and polymorphism. These notions make it more challenging to design metrics for the characterization of OO-based software vis-a-vis what it takes to do the same for the purely procedural code [5], [6].

An early work by Coppick and Cheatham [7] attempted to extend the then popular program-complexity metrics, such as the Halstead [8] and the McCabe and Watson complexity measures [9], to OO software. Subsequently, other works on OO software metrics focused mostly on the issue of how to characterize a single class with regard to its own complexity and its linkages with other classes. This "one class at a time" focus can be considered to apply even when we take into account interclass couplings induced by the methods of one class calling the methods of other classes. Major exemplars of this early work are the contributions by Brito e Abreu and Carapuca [10], Chen and Lum [11], Lee et al. [12], Chidamber and Kemerer (CK) [5], Lorenz and Kidd [13], Li and Henry [14], [15], Henderson-Sellers [16], and Briand et al. [17]. These researchers proposed that OO software be characterized by *per-class criteria* such as the average number of attributes, average number of methods, average number of ancestor classes, average number of abstract attributes, coupling between objects (CBO) as measured by the average number of other-class methods called by all the methods of a given class and by the average number of other-class attributes used by all of the code in a given class, and so on. Other metrics proposed during the same period—metrics that are straightforwardly reformulated on a per-class basis—include the MOOD metrics of Brito e Abreu et al. [18], [19] and Harrison et al. [20]. In particular, we should mention the following MOOD metrics: the Attribute Hiding Factor (AHF) and the Method Hiding Factor (MHF) metrics for measuring the extent of encapsulation, both defined as the ratio of the attributes and methods that are visible in a class vis-a-vis the total number of the same; the Method Inheritance Factor (MIF) metric, which is a ratio of the total number of inherited methods to the total number of the same; and the Coupling Factor (CF) metric that measures the frequency with which a class references an attribute or a method in another class. Recently, Counsell et al. [21] have carried out a rigorous mathematical analysis of two previously known cohesion metrics, *cohesion among methods in a class* (CAMC) [22] and *normalized Hamming distance* (NHD) [23], to understand their behaviors and evaluate their usefulness in measuring class cohesion. The prior work that we have cited also includes some *non-per-class* metrics. These include the depth of the inheritance tree in a software system, the inheritance fan-out, number of ancestor classes, etc. Another previously proposed non-per-class metric is the system-level coupling factor (COF) proposed by Ghassemi and Mourant [24].

There has been significant debate in the literature about the merits of the aforementioned metrics, especially with regard to the extent to which they capture the subtleties introduced by features that are peculiar to OO software. Consider, for example, when, in a purely count-based approach to software characterization, the number of attributes and methods defined for a class is supposed to tell us something about the complexity of that class. In OO software, even when a class is explicitly devoid of its own attributes and methods, it may nonetheless possess a rich set of the same through inheritance. By the same token, when the code in a class makes polymorphic method calls, it becomes very difficult to figure out through static analysis as to which piece of code is actually being called for execution. This has caused some researchers [25], [26], [27], [28], [29], [30] to argue that the quality measures produced by the previously mentioned metrics may be open to interpretation.

There is also a body of work in the literature that has focused on OO metrics from the standpoint of their ability to predict software maintainability [31], [27], [32] and design flaws [33], [34], [35]. Much of the work on using metrics to predict design flaws has focused on the CK metrics. Researchers have also analyzed whether the fault tolerance of software can be predicted by the same or similar metrics [36], [37], [38], [39]. Recently, Olague et al. [40] have carried out an empirical analysis of the CK and

1. This vision of large object-oriented software of the future is also the subject of a proposal currently under review in the Java Community Process (JCP) [1], [2]. The Java Specification Request JSR-294 [2] and Strnisa et al. [3] propose the notion of a *superpackage* that coincides with how we talk about modules in this paper.

the MOOD metric suites on six versions of an open source software to evaluate their ability to predict fault proneness. Similarly, researchers have reported an empirical study of previously known cohesion metrics on a large corpus of software [41] that revealed a bimodal behavior by most of these metrics.

There has also been work [42] on using metrics to determine a need for code refactoring (such as moving a method or an attribute from one class to another, derivation of a new class, and so on). We should also note the work of Alshayeb and Li [43] who, through an empirical study on the Java Development Kit (JDK), showed that the same metrics can reasonably predict the needed refactoring and error correction efforts, especially toward the end of the software development cycle (and specifically when the design cycle is short). Recently, Carey and Gannod [44] have used existing OO metrics [5], [16] and machine learning techniques to identify domain concepts from source code.

This brings us to the subject of frameworks for analyzing the metrics for their theoretical validity. Theoretical validation of a set of metrics means that the metrics conform to a set of agreed-upon principles. It is the principles that then constitute a framework. Briand et al. [29] have proposed a framework that formalizes how one can exercise different options when applying coupling, cohesion, and complexity metrics to object-oriented software. This framework was used by Arisholm et al. [45] to propose metrics to measure coupling among classes based on runtime analysis for object-oriented systems.

3 Two Types of Interfaces for a Module

The work we present in this paper assumes that the relationship of a module to the rest of the world can be characterized with two different kinds of APIs.

Service API: This is a module interface as it is most commonly used and understood. A Service API (S-API) declares the services that the module provides to the rest of the software system. An S-API exposes a set of methods that can be called by other modules.

Extension API: An Extension API (E-API) is a declaration of what functionality needs to be provided by an external plugin for the module. Over the years, plugins have become an important approach to adding versatility to the workings of a module without changing the source code in the module itself. With a properly declared E-API for a module, a third-party developer would not need access to the source code of the module in order to create a plugin for it. A module that a plugin is intended for is sometimes referred to as the *host application* vis-a-vis the plugin. For object-oriented host applications and plugins, an E-API commonly consists of the declaration of an abstract class and its abstract methods, with the expectation that the plugin will provide the implementation code for the abstract methods. Such software systems are based on a design principle commonly referred to as the *inversion of control* [46]. With inversion of control, the main body of the code calls the plugin but the plugin never calls the main body of the code. In other words, control is inverted and the main body of the code is responsible for control flow.

It is important to note that a module may possess multiple APIs of both types mentioned above. If a module possesses multiple S-APIs, each S-API may be intended for a specific usage of that module. As a case in point, an interest-calculation module in a financial software system may contain multiple user-directed S-APIs. For example, there could be an S-API designed specifically for regular banking applications, another for co-op banking applications, yet another for banking-like financial services provided by mutual-fund operators, and so on. While the basics of interest calculations for all of these related applications are likely to be the same, each different usage of the software may entail using a function name vocabulary that is peculiar to that application. Obviously, in addition to vocabulary differences, the different S-APIs for the same module may also differ with regard to the functionality offered.

As the reader will recall, we are talking about very large-scale software where there can be hundreds of classes in each module. In such a case, it is not necessary for all of the public methods of the classes in the module to belong to an S-API. A published S-API for a module would ordinarily consist of a small subset of all of the public classes and a small subset of the public methods of those classes. The role of the non-API classes and/or non-API methods would be to support the intramodule development of the software.

4 Notation

We will use the following notation in the rest of this paper. Our notation is similar to that used by Briand et al. [29] in their framework for the measurement of coupling metrics in object-oriented software.

4.1 System, Classes, Methods, Modules, and API

We represent an OO software system by $\mathcal{S} = < Ent, Rel >$, where Ent is a set of entities and $Rel \subseteq Ent \times Ent$ is a set of pairwise relationships among the entities. An entity can be a class, an attribute of a class, a method defined in a class, a module, and so on. Formally, $Ent = \mathcal{C} \cup \mathcal{A} \cup \mathcal{M} \cup \mathcal{I} \cup \mathcal{P}$, where $\mathcal{C}$ is the set of all classes, $\mathcal{A}$ the set of all attributes, $\mathcal{M}$ the set of all methods, $\mathcal{P}$ the set of all modules, and $\mathcal{I}$ the set of all APIs in the system.

Classes and methods. A class $c \in \mathcal{C}$ possesses a set $M(c) \subseteq \mathcal{M}$ of methods and a set $A(c) \subseteq \mathcal{A}$ of attributes. A method $m \in M(c)$ may constitute either a new method definition for the class c or may be an override definition for a method of an ancestor class or an implementation of an abstract method in a parent class. Similarly, $a \in A(c)$ is either a newly defined attribute for class c or hides the definition of the same attribute in an ancestor class.

Regarding the access discipline that applies to the methods in a class, we assume that any given method of a class is either public or private. A public method of a class c, denoted m_{pub}, is assumed to be accessible to the rest of the software system. The set of all public methods in a class c is denoted $M_{pub}(c)$. Taking liberties with the $M_{pub}(c)$ notation, we use $M_{pub}(p)$ to denote the set of all public methods in module p.

An abstract method of a class c is denoted by m_{abs}. In order to keep the notation simple, we assume without loss of generality that the access discipline for an abstract method is public. When needed, we use the notation m_{conc} to indicate a concrete method, that is, a method that is not abstract.

Modules and API. We assume that the system $\mathcal{S}$ can be partitioned into a set $\mathcal{P}$ of modules. For a module $p \in \mathcal{P}$, we use $C(p)$ to denote the set of classes in module p. Extending further the notation $M(c)$, we use $M(p)$ to denote the set of methods in module p. An API $\imath \in \mathcal{I}$ is a set of selected methods. We will take liberties with what $M(c)$ stands for and use the notation $M(\imath)$ to denote the set of methods that constitutes an API $\imath$. Further, we use the notation $I(p)$ to denote the set of APIs for a module p. Obviously, for a module API $\imath \in I(p)$, $M(\imath) \subseteq M(p)$. For a module p, we use $I^S(p)$ to denote the set of S-APIs and $I^E(p)$ the set of E-APIs. We define $Module(c)$ to be the module to which class c belongs, $Module(m)$ to be the module to which the method m belongs, and $Module(\imath)$ be the module to which the API $\imath$ belongs.

4.2 Relations

Having defined the entities, we now define various types of relationships among them. For our purpose, we define $Rel = \{Par \cup Ovride \cup Imp \cup Call \cup Uses\}$. These relationships are as follows:

Class relations. The main inheritance relationship among classes is the predicate $Par(c, d)$ that is true if class d is a parent of class c. For convenience, we define additional predicates, $Anc(c, d)$ and $Chld(c, d)$, that can be inferred straightforwardly from $Par(c, d)$. $Anc(c, d)$ is true when class d is an ancestor of class c and $Chld(c, d)$ is true when d is a child of class c. To economize on notation, we use the one-argument version of Par, as in $Par(c)$, to denote the set of all parent classes of c. We use one-argument versions of the other two predicates, Anc and $Chld$, in the same manner. That is, $Anc(c)$ denotes the set of all ancestor classes of c and $Chld(c)$ denotes the set of all child classes of c. We use $Depth(c)$ to denote the depth of a class c in the inheritance graph. That is, $Depth(c)$ is the number of ancestors of c along the shortest path from the root class to class c.

Method relations. The relation $Ovride(m, m_o)$ is a predicate that is true when a method m_o, defined for some class c, overrides method m defined for a class in $Anc(c)$. We use the notation $Call(m_f, m_t)$ as a predicate that is true when a method m_f calls another method m_t. The predicate is evaluated through static analysis of the code. For economy of notation, we use the one-argument version of the predicate $Ovride$, that is, $Ovride(m)$, to represent the set of all methods that override a given method m. With regard to the predicate $Call(m_f, m_t)$, we extend it by defining two functions, $Callto(m)$ and $Callby(m)$, the former representing the set of methods that call a given method m and the latter the set of methods called by m.

Uses relation. The notation $Uses(c, d)$ represents a predicate that is true when class c uses class d in the sense that a method directly implemented in c either calls a directly implemented method of d or refers to (reads/writes) an attribute defined directly for d. The $Uses$ predicate is determined statically. Like other predicates, we extend the $Uses$ predicate to $Uses(c)$, which returns a set of classes that c uses. For an attribute a of class c, the predicate $Uses(a, d)$ is true if class d uses the attribute a. Similarly, $Uses(a)$ denotes a set of all classes that use the attribute a.

5 Coupling-Based Structural Metrics

Starting with this section, we will now present a new set of metrics for large object-oriented software. We will begin with coupling-based structural metrics that provide different measures of the method-call traffic through the S-APIs of the modules in relation to the overall method call traffic.

5.1 Module Interaction Index

This metric calculates how frequently the methods listed in a module's APIs (both S-APIs and E-APIs) are used by the other modules in the system. As stated in Section 4, $I(p)$ denotes all of the APIs for a module p. For an API $\imath \in I(p)$, let $ExtCallRel(\imath)$ be the set of calls made to any of the methods of $\imath$ from methods outside p. That is,

$$ExtCallRel(\imath) = \{< m_f, m_t > | Call(m_f, m_t) \wedge m_t \in M(\imath) \\ \wedge m_f \notin M(p)\}.$$

Now, let $ExtCallRel(p)$ denote the set of all external calls made to all of the methods of all classes in module p. Thus,

$$ExtCallRel(p) = \{< m_f, m_t > | Call(m_f, m_t) \\ \wedge m_t \in M(p) \wedge m_f \notin M(p)\}.$$

We express the Module Interaction Index by

$$MII(p) = \frac{\left|\bigcup_{\imath \in I(p)} ExtCallRel(\imath)\right|}{|ExtCallRel(p)|}, \\ MII(\mathcal{S}) = \frac{1}{\mathcal{P}} \sum_{p \in \mathcal{P}} MII(p). \quad (1)$$

In an ideal state, all of the external calls to a module p should take place only through its officially designated S-API methods. The $MII(\mathcal{S})$ value ranges from 0 to 1. A max $MII(\mathcal{S})$ value of 1 indicates an ideal system where all intermodule interaction is only through the officially designated S-API methods. A min $MII(\mathcal{S})$ value of 0 is indicative of a system with very bad intermodule interaction.

5.2 Non-API Method Closedness Index

We now analyze the function calls from the point of view of the methods other than those listed in the APIs of the modules. In our earlier paper [4], we had argued that, ideally, such 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 APIs. (This is especially true of large systems that have been only partially modularized.) In this intermediate state, there may exist non-API public methods that participate in both intermodule and intramodule call traffic.

As stated in Section 4, let $I(p)$ be the current officially designated APIs of a module p. Let $M_{na}(p)$ refer to the actual non-API public methods in a module that are definitely not being called by other modules. Ideally, of course, these two sets would be mutually exclusive and their union would be the set of all public methods $M_{pub}(p)$

in a module p. But, in practice, the set of *actual non-API methods* would be smaller than the set difference $\{M_{pub}(p) - Declared\ API\ methods\ of\ p\}$. We use the following metric to measure this disparity between the declared API methods for a module and the methods that are actually participating in intermodule call traffic. We call this metric the "Non-API Method Closedness Index" and represent it by NC:

$$NC(p) = \frac{|M_{na}(p)|}{\left|M_{pub}(p) - \left\{\bigcup_{\imath \in I(p)} M(\imath)\right\}\right|}, \qquad (2)$$
$$NC(\mathcal{S}) = \frac{1}{\mathcal{P}} \sum_{p \in \mathcal{P}} NC(p).$$

Note that this metric calls for a zero-by-zero division when all of the public methods in a module are declared to be the API methods for the module. This is a common occurrence for a certain class of software systems, such as the Java platform. If we think of each package of the platform as constituting a module, then, in general, all of the public methods for all of the classes in the package would be considered to constitute the module's API. What the value of $NC(p)$ should be under this condition depends entirely on the nature of the software system. If it was the intent of the software designers that all of the public methods for all the classes in a module constitute that module's API, then $NC(p)$ must be set to 1 when both the numerator and the denominator go to 0. On the other hand, if we are talking about a very large business application in which each module may consist of hundreds of classes (that interact with one another to provide the functionality of the module) and for which only a subset of all of the public methods is meant to constitute the module's API, then we set $NC(p)$ to 0 when the denominator goes to 0. In the absence of division by zero, NC for a module p becomes 1 (best case) when all of the public methods of a module p that are not "Declared API methods of p" are *actual non-API* methods, i.e., they never participate in intermodule call traffic.

6 Metrics to Measure Inheritance-Based Coupling between Modules

It is not uncommon for a client of vendor-supplied software to customize a class by deriving from it and embedding client-specific functionality in the extended class. If the software created by the client resides in a separate module, this would naturally create dependencies between the vendor-supplied modules and the client-created modules. In the rest of this section, we will first elaborate on some specific forms of these dependencies that make it more difficult to maintain the code and to extend it further. Subsequently, we will provide metrics to measure these different types of inheritance-induced dependencies between the modules.

6.1 Inheritance-Induced "Fragile Base-Class" Problem

When a class in module B extends a class in module A, that naturally introduces a dependency between the two modules. But, one would expect that as long as the subclass in module B used only the information made public in the relevant section of module A's API, it would be possible to change module A without affecting B and vice versa. But, as is now well known, a phenomenon known as the *fragile base-class problem* may nevertheless introduce dependencies between the two modules that may cause module B to break when A is modified [47], [48]. The following two cases, depicted pictorially in Figs. 1a and 1b, illustrate the fragile base-class problem.

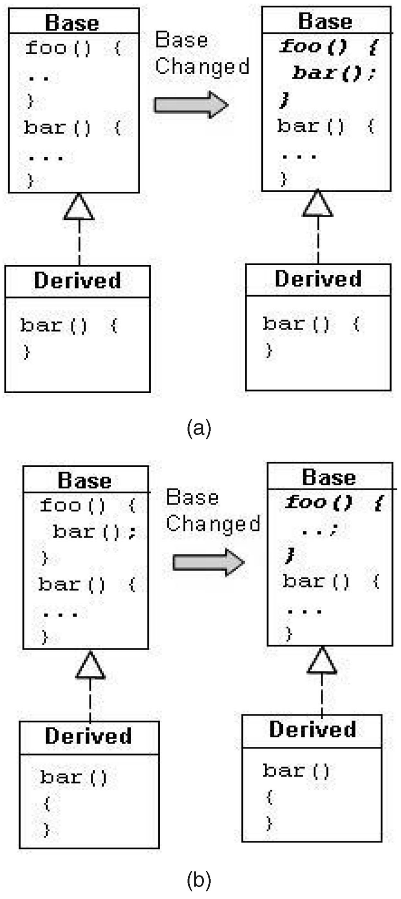


Fig. 1. Two cases of the fragile base-class problem. (a) Case 1. (b) Case 2.

Case 1. The base class, `Base`, contains two methods, $foo()$ and $bar()$, that stand on their own. The derived class, `Derived`, inherits both but is provided with an override definition for $bar()$ that does NOT use `Base`'s $foo()$. At some future time, `Base`'s vendor modifies $foo()$ so that it directly calls on $bar()$ for some of its functionality and this is done without changing the base class's public interface. But, due to polymorphism, for a client of the derived class, the inherited $foo()$ will behave differently from what the base class's vendor had in mind. When the inherited $foo()$ is invoked on an instance of `Derived`, it will call on `Derived`'s $bar()$ as opposed to `Base`'s $bar()$.

Case 2. The second case is the opposite of the previous case. Now, the designer of the base class writes a method, $foo()$, that explicitly calls another of base's methods, $bar()$. The derived class inherits both but provides an override definition for $bar()$. So, when the client of the derived class invokes

$foo()$ on an instance of `Derived`, its base-class definition will use the `Derived`'s $bar()$ due to polymorphism. This is the class behavior that `Derived`'s clients get accustomed to. Now, assume that, at some later date, the base class's vendor rewrites $foo()$ and $bar()$ in such a way that they become standalone functions. This the vendor does without changing the base class's public interface. Subsequently, however, `Base`'s $foo()$ as experienced by a client of `Derived` will behave differently because it would no longer call `Derived`'s $bar()$.

In both of these cases, the base-class fragility was introduced by one method of the base class directly calling another method of the same class either before or after code revision and the called method being overridden in the subclass. Both of these cases reflect the normal semantics of polymorphism. To the extent that polymorphism is a cornerstone of the object-oriented paradigm, we do want a call to $foo()$ of the base class to use $bar()$ of the derived class if the former needs the latter for a part of its functionality and if $foo()$ is invoked on an instance of the derived class. This situation arises all the time in object-oriented application programs. For example, you may write a software package for GUI programming in which a high-level method of a widget class needs to make calls to low-level methods of the same class that interact directly with the operating system for the purpose of determining the various attributes of the display on which the GUI is to be drawn. Now, someone else may try to adapt your software package to a different operating system by extending your widget class and providing override definitions for the low-level methods that interact directly with the operating system. Your inherited high-level methods should work for the new operating system because any calls to them would use the subclass override definitions designed specifically for the new operating system.

However, as demonstrated by the two cases shown above, this polymorphic behavior of a class hierarchy will elicit a price when a class and its subclass exist in two separate modules—it can make the module containing the subclass dependent on the implementation details of the module containing the base class even when the subclass is honoring the public interface of the base class.

So, whereas, on the one hand, polymorphism allows us to write compact code that can easily be extended to adapt to new application circumstances, on the other hand, it can make modularization more challenging. That naturally leads to the question of whether it is possible to detect polymorphic invocations of methods that do and that do not create modularization difficulties. When a method $foo()$ of a base class calls $bar()$ of the same class and when $bar()$ is overridden in a derived class, under what conditions should we construe that as an acceptable recourse to polymorphism and under what conditions as a situation that may create a modularization headache?

We take the position that all such base-class/derived-class relationships constitute acceptable usages of polymorphism when both classes reside in the same module. However, when such relationships cross module boundaries, we consider them harmful in the sense of their limiting the independent further development of the two modules involved. Our position is based on the assumption that it is usually the same team of programmers that would be in charge of any single module. We can expect the programmers to be intimately familiar with the intent underlying each method of a base class and whether or not it would be safe to extend it in a derived class. Obviously, the same cannot be said of two different modules. The module containing the base class may belong to a vendor-supplied library and the module containing the subclass may belong to a client who is trying to adapt the vendor-supplied library to a different application context. Now it would be much more difficult for the base-class programmer's intent to come through when the subclass client examines the base class's public interface.[2] For example, if the base-class programmer wrote up a method $foo()$ to call the base's $bar()$ without expecting this call to translate into an invocation of a derived class's $bar()$, that fact would not normally be in the base class's public interface. But, when both the base class and the subclass are within the same module, there is a greater likelihood that this information concerning the base class would be apparent to whoever is writing the derived class. The same cannot be hoped for when the two classes are in two different modules.

We therefore need a metric that can measure the extent of such base-class/derived-class relationships when the two classes are in different modules. Such a metric is presented in the next section.

6.2 Metric to Measure the Extent of the Fragile Base-Class Problem

Based on the above observations, we now propose a metric to quantitatively measure the extent to which the fragile base-class problem *may* exist in a software system. We assume that the code was initially written by programmers cognizant of the fragile base problem and that, therefore, at the beginning, the code did not exhibit instances of polymorphism mentioned in our Cases 1 and 2 when a base class and a subclass derived from it reside in different modules. The fragile base-class problem ceases to exist if no methods of a class call any of the other methods of the same class or the ancestor classes. We may refer to this as the ideal state of the software since it ensures the absence of the fragile base-class problem. Any departures from this ideal *may* indicate the presence of the fragile base-class problem. So, it is worthwhile to measure the extent of the departure from the ideal as an indication of the possibility of the existence of the fragile base-class problem. If a software system exhibits a high value for the departure from the ideal or if a software system suddenly shows a jump for this value after the system is extended with subclasses, it means that the users of the software need to more carefully test the extensions for potential instances of the fragile base-class problem. The extent to which a given software system conforms to the ideal with regard to base-class fragility will be measured by a metric we call the Base-Class Fragility Index ($BCFI$).

As defined in Section 4, let $Anc(c)$ be the set of all ancestor classes of a class c. We define $M_{anc}(c)$ to be the set

2. Unless, of course, one extends the programming language as advocated by Aldrich and Donnelly [48].

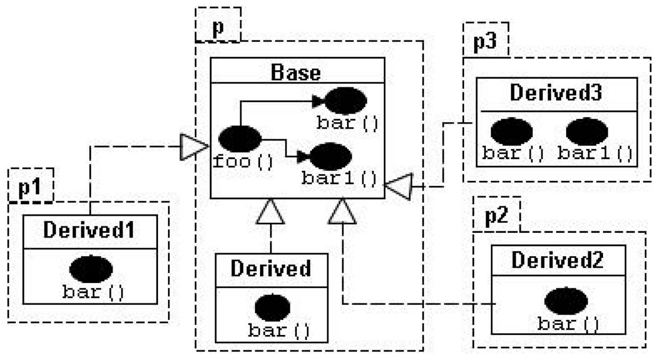


Fig. 2. The figure illustrates the idea of base-class violation. Module p contains the two classes named `Base` and `Derived`, module $p1$ just the class `Derived1`, module $p2$ just the class `Derived2`, and, finally, module $p3$ just the class `Derived3`. The arrows from $foo()$ to $bar()$ and $bar1()$ mean that $foo()$ calls on these two methods for a part of its functionality.

of concrete methods inherited from the ancestor classes of c that are not overridden in class c. Thus,

$$M_{anc}(c) = \{m'_{conc} \in M(c') | c' \in Anc(c) \\ \wedge (\forall_{m \in M(c)} \neg Ovride(m'_{conc}, m))\}.$$

For a method m of class c, let $UpCallby(m)$ be the set of concrete methods called by m that are either defined in a class c or inherited from its ancestor classes but not overridden. That is,

$$UpCallby(m) = \{m'_{conc} | Call(m, m'_{conc}) \\ \wedge (m'_{conc} \in M(c) \vee m'_{conc} \in M_{anc}(c))\}.$$

In order to define $BCFI$, we first define the notion of base-class violation $BCVio(m)$ for a method m that measures the extent to which the methods that have overridden m exist in other modules:

$$BCVio(m') \\ = \frac{|\{m'_o \in Ovride(m') | Module(m') \neq Module(m'_o)\}|}{|Ovride(m')|} \\ = 0 \text{ when } Ovride(m') = \emptyset.$$

$BCVio(m')$ is proportional to the number of times m' is overridden outside of the module in which m' resides. The idea of this violation is illustrated in Fig. 2. In this figure, the method `bar()` in module `p` is overridden 3 times outside `p`. Therefore, $BCVio(\texttt{bar}) = \frac{3}{4}$.

Having defined $BCVio$, we now compute the maximum of $BCVio$ values of all of the concrete methods in $UpCallby(m)$, for a method m. We denote this as $BCVMax$ as follows:

$$BCVMax(m) = \max_{m' \in UpCallby(m)}\{BCVio(m')\} \\ = 0 \text{ when } UpCallby(m) = \emptyset.$$

At the class level, $BCVMax(c)$ for a class c is the maximum of the $BCVMax(m)$ of all of the methods in class c, i.e., $BCVMax(c) = \max_{m \in M(c)} BCVMax(m)$.

To measure the violation at the module level, we first need to determine the set of classes in the module for which $BCVMax$ is applicable. This set, denoted by $BCVSet(p)$ for a given module p, is the set of classes that contains at least one method with nonempty $UpCalledby$ set.

Thus, $BCVSet(p) = \{c \in \mathcal{C}_p | \exists_{m \in M(c)}(UpCallby(m) \neq \emptyset)\}$. We will now define $BCFI$ as follows:

$$BCFI(p) = 1 - \frac{1}{|BCVSet(p)|}\sum_{c \in \mathcal{C}_p} BCVMax(c),$$
$$BCFI(\mathcal{S}) = \frac{1}{|\mathcal{P}'|}\sum_{p \in \mathcal{P}'} BCFI(p), \quad (3)$$
$$\text{where } \mathcal{P}' = \{p' \in \mathcal{P} | BCVSet(p) \neq \emptyset\} \\ = 1 \text{ when } \mathcal{P}' = \emptyset.$$

Observe that, for classes not in the set $BCVSet(p)$, $BCVMax(c)$ will always be 0. Hence, the value for $BCFI(p)$ will range between 0 and 1, with the worst case being 0 and the best case (no base-class violation) being 1.

Finally, the base-class violation for the entire system $BCFI(\mathcal{S})$ is the average of the $BCFI(p)$ for all of the modules in the system for which the violation is applicable. The value for $BCFI(\mathcal{S})$ ranges from 0 to 1, with 0 indicating a system that has the worst base-class violation and 1 indicating a system that is completely free of base-class violation.

6.3 Inheritance-Based Intermodule Coupling Metrics

These dependencies may be measured either at the module level or at the class level or at both. With $\mathcal{P}$ representing the set of modules and $C(p)$ denoting the set of all classes in the module p, the following metric measures such dependencies at the module level:

$$IC_1(p) = 1 - \frac{|\{p' \in \mathcal{P} | \exists_{d \in C(p')} \exists_{c \in C(p)}(Chld(c, d) \wedge p \neq p')\}|}{|\mathcal{P}| - 1} \\ = 1 \text{ when } |\mathcal{P}| = 1.$$

For a given module p, the numerator of the fraction is the number of other modules such that at least one of their classes is derived from at least one class of module p.

If the modules vary widely in the number of classes they contain and in the extent to which the classes in one module extend classes in other modules, the above metric may not tell the entire story about the inheritance-based intermodule dependencies. With $\mathcal{C}$ representing the set of all classes in the software system, the following metric measures such dependencies at the class level.

$$IC_2(p) = 1 - \frac{|\{d \in \mathcal{C} | \exists_{c \in C(p)}(Chld(c, d) \wedge Module(d) \neq p)\}|}{|\mathcal{C} - C(p)|} \\ = 1 \text{ when } |\mathcal{P}| = 1.$$

Here, the numerator of the fraction indicates the number of classes in other modules that extend any of classes in p. The above two metrics measure inheritance-based dependencies of the other modules on a given module p. The following metric measures such dependencies in the opposite direction. If all of the classes in a given module are derived from classes in other modules, the value of this metric would be zero for that module.

$$IC_3(p) = 1 - \frac{\left|\{c \in C(p) | \exists_{d \in Par(c)} (Module(d) \neq p)\}\right|}{|C(p)|}.$$

Here, the numerator indicates the number of classes in module p that are derived from any of the classes in any of the other modules. Given the three different ways to measure inheritance-based couplings between the modules, we now combine all of them in the form of a composite IC metric, first at the module level and then at the system level, as follows:

$$IC(p) = \min(IC_1, IC_2, IC_3), \tag{4}$$

$$IC(\mathcal{S}) = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} IC(p). \tag{5}$$

The maximum and minimum $IC(\mathcal{S})$ values are 1 and 0, respectively. An $IC(\mathcal{S})$ value of 1 is indicative of a system that does not have any inheritance-based couplings. Conversely, an $IC(\mathcal{S})$ value of 0 is indicative of a system that has maximum inheritance-based couplings.

7 A Metric for Not Programming to Interfaces

It is now accepted wisdom in object-oriented programming that a client of a class should only program to the interfaces that are at the roots of the class hierarchies and not to the concrete classes that implement those interfaces. This allows the implementation code to be altered without affecting the clients of a class hierarchy.

Our use of the word "interface" in this section is not to be confused with "interface" in a module's API. The interface as used in this section reflects the fact that each module will, in general, contain class hierarchies and each hierarchy will have a root interface. A recommended OO practice is to program to the interfaces at the roots of the class hierarchies as opposed to their implementations in the child classes.

In our context, what that means is that, when a module needs to act as a client of another module, the former should access the latter's functionality only through the root interfaces of the class hierarchies in that module. Obviously, in well-engineered code, module p_1 will access the functionality of module p_2 only through p_2's APIs and the methods listed in p_2's APIs will be a subset of the methods in the root interfaces of all of the class hierarchies in p_2. But, as the code ages, it is not unlikely that some of the methods in p_1 may call on some of the methods in p_2 that are only defined for the concrete classes of p_2 and that are NOT included in any of the APIs of p_2.

Our goal is to capture this aspect of code degradation through a metric we call *Not-Programming-to-Interfaces Index* (NPII) that we now present. With $Callby(m)$ denoting the set of methods called by m, let $AbsCallby(m) = \{m_{abs} | m_{abs} \in Callby(m)\}$ be the set of abstract methods called by m. Now, we consider calls made to those concrete methods whose defining classes belong to some inheritance hierarchy. We ignore calls to those methods which belong to isolated classes (that is, classes that do not belong to any inheritance hierarchies) from the $NPII$ perspective. Let $InhConcCallby(m)$ be those concrete methods called by m. It is defined by

$$\begin{aligned} InhConcCallby(m) = \{&m_{conc} | m_{conc} \in Callby(m) \\ &\wedge \exists_{c' \in \mathcal{C}} (m_{conc} \in M(c') \wedge Anc(c') \neq \emptyset)\}. \end{aligned}$$

Next, we introduce the notion of a bad call made to a concrete method from the $NPII$ perspective. Leaving aside the called methods that are defined/declared in the root interfaces, a call is considered to be bad when the called concrete method (defined in a class that belongs to some inheritance hierarchy) has overridden or implemented another method. This is defined by

$$\begin{aligned} BadConcCallby(m) = \{&m' | m' \in InhConcCallby(m) \\ &\wedge (DoesOvride(m') \vee DoesImpl(m'))\}, \end{aligned}$$

where $DoesOvride(m')$ is a predicate that is true when m' overrides a method and $DoesImpl(m')$ is a predicate that is true when m' is an implementation of some abstract method. Now, we define $NPII$ as

$$\begin{aligned} &AbsCallby(p) = \bigcup\nolimits_{m \in M(p)} AbsCallby(m), \\ &InhConcCallby(p) = \bigcup\nolimits_{m \in M(p)} InhConcCallby(m), \\ &BadConcCallby(p) = \bigcup\nolimits_{m \in M(p)} BadConcCallby(m), \\ &NPII(p) = \frac{|AbsCallby(p)|}{|AbsCallby(p)| + |InhConcCallby(p)|} + \\ &\frac{\psi |InhConcCallby(p) - BadConcCallby(p)|}{|AbsCallby(p)| + |InhConcCallby(p)|} \text{ where } 0 \leq \psi \leq 1 \\ &= \perp \text{ when } AbsCallby(p) \cup InhConcCallby(p) = \emptyset, \end{aligned}$$

where $\perp$ means undefined.

Let $\mathcal{P}' = \{p \in \mathcal{P} | AbsCallby(p) \cup InhConcCallby(p) \neq \emptyset\}$. We can now write

$$\begin{aligned} NPII(\mathcal{S}) &= \frac{1}{|\mathcal{P}'|} \sum_{p \in \mathcal{P}'} NPII(p) \\ &= 1 \text{ if } \mathcal{P}' = \emptyset. \end{aligned} \tag{6}$$

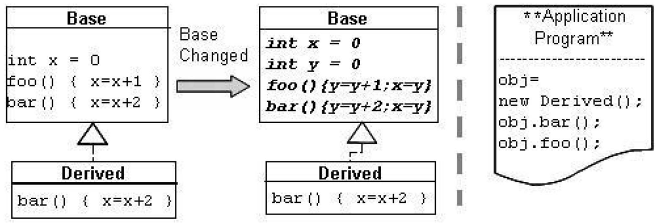
The value of NPII ranges between 0 and 1. The value of $NPII$ of a module is 1 when only the abstract methods in the root interfaces of class hierarchies in a given module are called by the implementation code in other modules.

8 Association-Induced Coupling Metrics

These measure the extent to which a class in a module uses another class in some other module either as an attribute or as a parameter in a method definition. As with the inheritance-based coupling, first we measure how the modules are coupled through association. This we accomplish with the help of the three metrics, $AC_1(p)$, $AC_2(p)$, and $AC_3(p)$, described here. We define

$$\begin{aligned} AC_1(p) &= 1 - \frac{\left|\{p' \in \mathcal{P} | \exists_{c \in C(p')} \exists_{d \in C(p)} (Uses(c,d) \wedge p \neq p')\}\right|}{|\mathcal{P}| - 1} \\ &= 1 \text{ when } |\mathcal{P}| = 1. \end{aligned}$$

Here, the numerator denotes the total number of other modules that contain at least one class that uses at least one



```
int x = 0
foo() { x=x+1 }
bar() { x=x+2 }
```

```
int x = 0
int y = 0
foo(){y=y+1;x=y}
bar(){y=y+2;x=y}
```

```
bar() { x=x+2 }
```

```
obj=
new Derived();
obj.bar();
obj.foo();
```

Fig. 3. Illustration of the problem that arises when the state of an instance is accessed directly.

class of module p as an attribute or as a parameter in a method definition. $AC_1(p)$ measures as to what extent other modules use p through associational dependencies. Next, we define

$$AC_2(p) = 1 - \frac{\left|\{c \in \mathcal{C} | \exists_{d \in C(p)} (Uses(c,d) \wedge Module(c) \neq p)\}\right|}{|\mathcal{C} - C(p)|}$$
$$= 1 \text{ when } |\mathcal{P}| = 1.$$

Here, the numerator indicates the total number of classes in other modules that use at least one class of module p as an attribute or as a parameter in a method definition. Finally, we define

$$AC_3(p) = 1 - \frac{\left|\{c \in C(p) | \exists_{d \in Uses(c)} (Module(d) \neq p)\}\right|}{|C(p)|}.$$

Here, the numerator indicates the total number of classes in module p that use at least one class from any of the other modules either as an attribute or as a parameter in a method definition. Given the three different forms of this metric, we can now combine them all into a composite AC metric, first at the module level and then at the system level, by

$$\begin{aligned} AC(p) &= \min(AC_1, AC_2, AC_3), \\ AC(\mathcal{S}) &= \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} AC(p). \end{aligned} \tag{7}$$

Like other metrics, the min value of $AC(\mathcal{S})$ is 0 and the max value is 1.

9 Coupling Induced by Directly Accessing the State

The commonly stated dictum that the state of an instance constructed from a class should only be accessed through appropriate accessor/mutator methods is honored more in the breach than the observance. That the effects of accessing the state directly can seriously compromise the integrity of the code is made clear by the following example by Mikhajlov and Sekerinski [47]. Let us say that a `Base/Derived` class pair started out as shown in Fig. 3 left, with `Derived` directly accessing the state variable x as shown. As shown in this figure, the state variable x is modified by both `foo()` and `bar()` methods. Assume that the writer of the original `Base` class decides to enlarge its functionality by providing it with another data member y. Also assume that the same programmer decides to reimplement `foo()` and `bar()` as shown. Further assume that this programmer does not alter the API description of $foo()$ and $bar()$ made available to the clients of this class since, in the base class itself, the behavior of $foo()$ and $bar()$ vis-a-vis the variable x has not changed. Unfortunately, for a client of `Derived`, this change in `Base`'s implementation code will alter the behavior of `Derived`. In the "Application Program" of Fig. 3 right, the value of x will be 3 before the base-class change and 1 afterward.

The problems created by directly accessing the state are perhaps more manageable when such accesses take place in the same module (for reasons that we stated earlier in Section 6.2). However, such accesses crossing module boundaries could be a source of serious code deterioration. The metric we present below measures to what extent the state of a given class has been accessed by classes in other modules.

In accordance with Section 4, suppose that $A(c)$ is the set of attributes of class c. $A(c)$ defines the state of the instances constructed from c. Some of these attributes will be private to c and the rest will be accessible to other classes. Also recall that $Uses(a)$ denotes the set of other classes that directly access the attribute $a \in A(c)$. In an ideal state, we obviously want that $\forall_{a \in A(c)} Uses(a) = \emptyset$.

Now, we measure the extent of intermodule and intramodule access of the state by first defining the following four sets. The first two sets, prefixed with "Inter," are for measuring the extent of intermodule access of state. The second two sets, prefixed with "Intra," are for measuring the extent of intramodule access of state:

$$\begin{aligned} InterStAcc1(c) &= \{a \in A(c) | \exists_{d \in Uses(a)} Module(c) \neq \\ &\qquad Module(d)\}, \\ InterStAcc2(c) &= \{p' \in \mathcal{P} | \exists_{a \in A(c)} \exists_{d \in C(p')} (Module(c) \neq \\ &\qquad p' \wedge used(a,d))\}, \\ IntraStAcc1(c) &= \{a \in A(c) | \exists_{d \in used(a)} Module(c) = \\ &\qquad Module(d)\}, \\ IntraStAcc2(c) &= \{d \in C(p) | \exists_{a \in A(c)} (Module(c) = \\ &\qquad Module(d) \wedge used(a,d))\}. \end{aligned}$$

Of the first two sets, $InterStAcc1(c)$ is the set of attributes of c that are accessed by classes residing in other modules, whereas $InterStAcc2(c)$ is the set of modules that directly access attributes of c. Similarly, $IntraStAcc1(c)$ is the set of attributes of c that are accessed by classes in the same module where c belongs. Finally, $IntraStAcc2(c)$ is the set of classes residing in the same module as c that directly access the attributes of c.

In terms of the above four sets, we now define a State Access Violation Index (SAVI) as follows: We first define it for a class, then for a module, and, finally, for the software system:

$$\begin{aligned} SAVI(c) = 1 - \Bigg(&\omega_1 \frac{|InterStAcc1(c)|}{|A(c)|} + \omega_2 \frac{|InterStAcc2(c)|}{|\mathcal{P}|} \\ &+ \omega_3 \frac{|IntraStAcc1(c)|}{|A(c)|} + \omega_4 \frac{|IntraStAcc2(c)|}{|C(p)|}\Bigg), \\ SAVI(p) = \frac{1}{|\mathcal{C}_p|} &\sum_{c \in \mathcal{C}_p} SAVI(c), \\ SAVI(\mathcal{S}) = \frac{1}{|\mathcal{P}|} &\sum_{p \in \mathcal{P}} SAVI(p). \end{aligned} \tag{8}$$

Observe that $SAVI(c)$ is a weighted average of the four different state access cases with the user-defined weights obeying the condition $\sum_{i=1}^{4} \omega_i = 1$. Also note that the value of $SAVI(\mathcal{S})$ is always between 0 and 1.

10 Size Uniformity Metrics

During the course of software development, as code evolves one class at a time and one module at a time, from an overall architectural standpoint it is useful to keep track of the size variations among the classes within the individual modules and the variations in the sizes of the modules themselves. Supervisory individuals in charge of the overall application development effort need access to these size-based characterizations of the software especially when the classes and the modules are developed by different teams of programmers.

Note that we are not advocating that all modules be roughly the same size and that this also be true of all of the classes. All we are saying is that it be possible to measure the variations in sizes of the modules and the classes. In some cases, if a class is disproportionately large in comparison with the other classes in the same module, it may indicate that the large class is not sufficiently focused in terms of the functionality it offers and that a superior software design would result if the large class were decomposed into smaller classes. Along the same lines, if a module is found to be disproportionately large in relation to the other modules, the module's design may call for reexamination to see if it should be decomposed into smaller modules.

Having provided a rationale for size-based metrics, the next question is: How does one measure the size of a class or of a module? Whereas the number-of-lines-of-code is a reasonable size metric for non-object-oriented software, it may not be as meaningful (or, for that matter, insightful) for object-oriented software. For object-oriented software, the size of a class may be measured in terms of the lines of code in a class and/or by the number of nontrivial (nonaccessor) methods defined for the class. By the same token, the size of a module may be defined either by the number of lines of code and/or by the number of classes in the module.

We propose three size-based metrics: 1) an index to measure the variability in the number of classes in the modules, MU; 2) an index to measure the class-size variability by counting the number of methods in the classes, CU_m; and 3) an index to measure the class-size variability by counting the number of lines of code in the classes, CU_l. These metrics are measured by the following formulas:

$$MU(\mathcal{S}) = \frac{\mu_p}{\mu_p + \sigma_p}, \tag{9}$$

$$CU_m(\mathcal{S}) = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \frac{\mu_m(p)}{\mu_m(p) + \sigma_m(p)}, \tag{10}$$

$$CU_l(\mathcal{S}) = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \frac{\mu_l(p)}{\mu_l(p) + \sigma_l(p)}, \tag{11}$$

where μ_p denotes the average size of a module in terms of the number of classes in each module and σ_p denotes the standard deviation of this quantity. Similarly, $\mu_m(p)$ denotes the average size of a class in a module p in terms of the number of the number of methods in the class and σ_m denotes the standard deviation of this quantity. Finally, $\mu_l(p)$ denotes the average size of a class in a module p in terms of the number of lines of code in the class and σ_l denotes the standard deviation of this quantity. The values of all three size metrics are always between 0 and 1.

11 A Metric Related to System Extensibility

As mentioned in Section 3 when we talked about the notion of "Extension API," it is not uncommon for software designers to create extensible systems whose functionality can be enhanced or modified by adding a new piece of code in the form of a plugin that does not impact the rest of the software.

As an example of the need for specialized features that one may wish to add to the software via a plugin module after the main body of code has already been programmed, consider a financial application that must support various types of withdrawal mechanisms for different types of customers. In general, it may not be possible (or even desirable) to initially conceive of all possible modes of cash withdrawal and how a given withdrawal should be recorded in the bookkeeping procedures of a bank. Depending on the nature of an account, a withdrawal may affect just the cash balance in the account or it may have to trigger certain other reporting actions (in case of very large withdrawals). A plugin-based approach would allow for different withdrawal modes to be incorporated as needed when the software is customized.

The key idea of the plugin-based approach is to define a set of extension APIs (E-APIs) for a module, as discussed in Section 3, and then have the plugin modules implement these E-APIs. Such plugin-based design has been incorporated in a number of commercially available "frameworks," examples being the Spring framework, the EJB container framework, the Eclipse plugin framework, and so on. In all of these cases, the interaction between the module and its plugins happens through what is usually referred to as the "inversion of control." We have inversion of control since the behavior of the plugin is invoked by the module instead of the plugin calling on the API methods of the module.

We believe that, since a plugin is meant for extending a specific functionality of a module, the plugin should be precise in the sense that all of its classes and methods should be there for extending the functionality of the module. However, it is not uncommon to see old plugin modules that have grown in an ad hoc manner with time. This ad hoc "code bloat" may result in what is now superfluous or extraneous code that is no longer needed by the latest operation of the plugin. Our next metric aims to measure this phenomenon in the code.

Let p be a plugin module that implements one or more E-APIs of a framework module $p1$. Let $I^E(p1)$ be the set of E-APIs that the plugin module p implements vis-a-vis the host module $p1$. Let $Abstract(m)$ be the abstract method of module $p1$ that is implemented by m of the plugin module p. With $M(p)$ denoting all the methods of the plugin module p, let

$$ImplExtn(p) = \{m \in M(p) | \exists_{\imath \in I^E(p1)} Abstract(p) \in \imath\}$$

be the set of all methods in p that implement one or more E-API of $p1$. We argue that all of the calls made by such methods m to other methods in the plugin module p and the transitive closure of such method calls with respect to the module p should be almost equal to the number of methods of p. When that happens, we can say that all the methods in p are playing some role in implementing the functionality assigned to the plugin. As defined in Section 4, let $Callby(m)$ be the set of methods that m calls and let $Callby^+(m)$ be the transitive closure of the method calls. Finally, let $ModuleClosure(m,p) = \{Callby^+(m) \cup \{m\}\} \cap M(p)$ be the set of methods that are in the module p that are in transitive closure of method calls from m. We now define a Plugin Pollution Index (PPI) for the plugin module p as follows:

$$PPI(p) = \frac{|\bigcup_{m \in ImplExtn(p)}\{ModuleClosure(m,p)\}|}{|M(p)|}, \qquad PPI(\mathcal{S}) = \frac{1}{|\mathcal{P}|}\sum_{p \in \mathcal{P}} PPI(p). \tag{12}$$

Observe that the value of PPI is always between 0 and 1, where 0 is the worst case.

12 Measurement of API Cohesiveness and Segregation

As mentioned earlier, the work we report here is based on the assumption that a module provides its functionality to the rest of the world through one or more S-APIs, as defined in Section 3. We would want each S-API to be well focused, serving a particular need vis-a-vis what is generally expected of the module. Even more particularly, for a clean and easy-to-understand relationship between a module and the rest of the world, it is best if the usages of the different S-APIs of the module are mutually exclusive to the maximum extent possible. In other words, we would want the S-APIs to possess the following characteristics: 1) Each S-API should be cohesive from a *Similarity of Purpose* perspective; and 2) each S-API should be maximally segregated from the other S-APIs from the standpoint of usage.

To illustrate the cohesiveness and segregation notions, consider the scenario where a module p possesses two S-APIs, $\imath_1 = \{foo1, foo2, foo3\}$ and $\imath_2 = \{bar1, bar2, bar3\}$. Further assume that there are four client modules, $p1$, $p2$, $p3$, and $p4$, that access these APIs. Suppose that $p1$ calls $foo1$ and $foo3$, $p2$ calls $foo1$, $foo2$, and $foo3$, $p3$ calls $bar1$, $bar2$, and $bar3$, and $p4$ calls $bar2$ and $bar3$. Here, $\imath_1$ would be considered cohesive from a usage standpoint since each of its client modules, $p1$ and $p2$, makes use of most of the methods of the S-API. The API $\imath_2$ would also be considered cohesive for the same reason. Besides being cohesive, $\imath_1$ and $\imath2$ are also well segregated since the clients of each interface are mutually exclusive. Now consider the case when $p1$ calls $foo1$, $p2$ calls $foo3$ and $bar1$, $p3$ calls $bar2$ and $foo2$, and $p4$ calls $bar3$. Now the APIs may not be considered to reflect a good design since none of the clients uses most of the methods of either API and due to the overlap between the clients of $\imath_1$ and $\imath_2$.

We now provide a metric called API Usage Index $APIU$ that measures the extent to which the properties of cohesiveness and mutual exclusiveness (or segregation) are obeyed by all of the S-APIs of a module. With $I^S(p)$ representing the set of all S-APIs of a module (as explained in Section 4), let $M(\imath)$ be the set of methods of an S-API $\imath \in I^S(p)$. Let

$$CallingMod(\imath) = \{p \in \mathcal{P} | \exists_{m_f \in M(p)} \exists_{m_t \in M(\imath)} Call(m_f, m_t) \wedge p \neq p'\}$$

be the set of other modules that call one or more methods of this API. Finally, let

$$M(\imath, p') = \{m \in M(\imath) | p' \in CallingMod(\imath) \wedge \exists_{m_f \in M(p')} Call(m_f, m)\}$$

be the subset of methods in $M(\imath)$ that are called by the module $p' \in CallingMod(\imath)$. Usage cohesiveness of an S-API is given by

$$APIUC(\imath) = \frac{\sum_{p' \in CallingMod(\imath)} |M(\imath, p')|}{|CallingMod(\imath)| \times |M(\imath)|}.$$

In reality, the number of methods in an API can vary widely. Since the cohesiveness of an API as defined here is a property of an aggregate, the APIUC values for different APIs of a module should be appropriately weighted to reflect their sizes. In other words, the APIUC for an API with, say, 100 methods should carry more weight than that of an API with just one or two methods. For a module p with $I^S(p)$ S-APIs, we define a simple weighing scheme as

$$\alpha(\imath) = \frac{|M(\imath)|}{\left|\bigcup_{\imath_j \in I^S(p)} M(\imath_j)\right|}$$

and use these weights to compute the overall cohesiveness of all S-APIs of a module p by

$$APIUC(p) = \frac{\sum_{\imath \in I^S(p)} (\alpha(\imath) \times APIUC(\imath))}{|I^S(p)|}.$$

To measure the segregation of all of the S-APIs for a given module p, we want to consider only the cohesive interfaces. If an S-API is not cohesive with respect to its usage in the first place, it may not be meaningful to analyze whether the interface also possesses the property that the clients of the module are mutually exclusive vis-a-vis this API. So that we can just focus on the cohesive APIs, we define $APIUCSet(p)$ to be the set of cohesive interfaces of p. This set is defined as

$$APIUCSet(p) = \{\imath \in I^S(p) | APIUC(\imath) \geq \lambda\},$$

where λ is a user-defined threshold for determining whether an S-API is cohesive. The segregation of the clients with respect to a cohesive S-API is given by

$$\begin{aligned} APIUS(p) = 1 - &\left(\frac{\left|\bigcup_{\imath_1, \imath_2 \in APIUCSet(p)} CallingMod(\imath_1) \cap CallingMod(\imath_2)\right|}{\left|\bigcup_{\imath \in APIUCSet(p)} CallingMod(\imath)\right|}\right) \\ = 0 \text{ if } &APIUCSet(p) = \emptyset. \end{aligned}$$

Combining the cohesiveness and segregation metrics, we can define an overall metric of these two related attributes by

$$APIU(p) = \frac{APIUS(p) + APIUC(p)}{2},$$
$$APIU(\mathcal{S}) = \frac{1}{|\mathcal{P}|}\sum_{p\in\mathcal{P}} APIU(p). \tag{13}$$

The system level $APIU$ value is always between 0 and 1, where 0 is the worst case.

13 Measurement of Common Use of Module Classes

It is not infrequently the case when the classes that get used together need to be packaged together [49]. If most of the classes in a module are used together, the impact of changing one or more classes in the module is likely to be localized with respect to the rest of the software. On the other hand, when a module contains classes that are not used together, even an innocuous change to one of the classes may require an impact analysis vis-a-vis all of the dependent modules, even when the dependent modules are not directly impacted by the change.[3] To what extent the classes that are packaged together in a module also get used together can be measured by using the notion of mutual information content between a set of classes grouped into a module and their usage co-occurrence.

Let $\mathcal{C}$ be the set of all classes in a software system and let $\mathcal{P}(\mathcal{C})$ be the power set that represents all potential usage co-occurrences of the classes. Thus, $s \in \mathcal{P}(\mathcal{C})$ represents a set of classes that may possibly be used together. Now, let $\varrho(s)$ be the probability of a set of classes s being used together in a module. Further, let $\mathbf{S}$ be a discrete random variable that takes a value from the power set $\mathcal{P}(\mathcal{C})$. Next, let $\varrho(p)$ be the probability with which a particular module is selected from the set of modules $\mathcal{P}$. Further, let $\mathbf{P}$ be the discrete random variable that takes any value from the set of modules $\mathcal{P}$.

Now, let us try to answer the question as to what a given set $\mathbf{S}$ chosen randomly from the power set $\mathcal{P}(\mathcal{C})$ can tell us about the class composition of a module $\mathbf{P}$? Said another way, to what extent can $\mathbf{S}$ predict the module $\mathbf{P}$? In a probabilistic framework, the answer to this question is provided by the conditional probability

$$\varrho(\mathbf{P}|\mathbf{S}) = \frac{\varrho(\mathbf{P}\cap\mathbf{S})}{\varrho(\mathbf{S})}.$$

What this definition implies is that, if all of the classes in a randomly chosen $\mathbf{S}$ are contained in the module $\mathbf{P}$, then $\mathbf{S}$ completely "predicts" the module $\mathbf{P}$. For another example, consider a hypothetical scenario where a specific element s from the power set $\mathcal{P}(\mathcal{C})$ consists of the 10 classes $\mathbf{S} = \{c_1 \cdots c_{10}\}$ which have been used together. Assume that, out of these classes used together, c_1, c_2 belong to module p_1, $c_3 \cdots c_5$ belong to p_2, and $c_6 \cdots c_{10}$ belong to p_3. In this case, the above definition tells us directly that $\varrho(\mathbf{P} = p_1|\mathbf{S} = s) = \frac{2}{10}$, $\varrho(\mathbf{P} = p_2|\mathbf{S} = s) = \frac{3}{10}$, and $\varrho(\mathbf{P} = p_3|\mathbf{S} = s) = \frac{5}{10}$.

The mutual information content between the sets of classes used together vis-a-vis modules can now be expressed as

$$I(\mathbf{P};\mathbf{S}) = H(\mathbf{P}) - H(\mathbf{P}|\mathbf{S}), \tag{14}$$

where $H(\mathbf{P})$ is the entropy associated with all of the modules in the system and $H(\mathbf{P}|\mathbf{S})$ denotes the conditional entropy of the modules vis-a-vis the different sets of classes that can be used together. The entropy $H(\mathbf{P})$ can be expressed as $H(\mathbf{P}) = -\sum_{p\in\mathcal{P}}\varrho(p)\log\varrho(p)$ and the conditional entropy is given by $H(\mathbf{P}|\mathbf{S}) = -\sum_{s\in\mathcal{P}(\mathcal{E})}\varrho(s)\sum_{p\in\mathcal{P}}\varrho(p|s)\log\varrho(p|s)$.

If the idea that the classes that are used together should be packaged together is a valid software design strategy, then the mutual information content between the set of classes used together and the modules should be as large as possible. This mutual information takes on a maximum value when one can predict with certainty a module based on the set of classes that are used together. When that happens, we have $H(\mathbf{P}|\mathbf{S}) = 0$.

The above discussion translates into the following metric, which measures the extent to which the classes used together can be the predictors of modules:

$$CReuM(\mathcal{S}) = \frac{I(\mathbf{P};\mathbf{S})}{H(\mathbf{P})}. \tag{15}$$

For this metric, the worst-case scenario ($CReuM(\mathcal{S}) = 0$) corresponds to the case when all of the classes are put in one monolithic module, i.e., when $|\mathcal{P}| = 1$. For such a case, we have

$$\begin{aligned} I(\mathbf{P};\mathbf{S}) &= H(\mathbf{P}) - H(\mathbf{P}|\mathbf{S}) \\ &= -\sum_{p\in\mathcal{P}}\varrho(p)\log\varrho(p) \\ &\quad + \sum_{s\in\mathcal{P}(\mathcal{E})}\varrho(s)\sum_{p\in\mathcal{P}}\varrho(p|s)\log(\varrho(p|s)) \\ &= 0 \; (since\ \varrho(p) = 1\ and\ \varrho(p|s) = 1). \end{aligned}$$

14 Summary of All OO Metrics

Table 1 summarizes the metrics presented in Sections 5 through 13 and their underlying rationale. To present the rationale succinctly for each metric, we have stated it in the form of a central question that the metric provides an answer to.

15 Experiments for Metrics Validation

This section presents a two-pronged approach for experimentally validating the new metrics. In the first approach, described in Section 15.4, we evaluate the metrics for some well-respected software systems and then justify the values thus obtained on the basis of manual examination of the software. In the second approach, presented in Sections 15.5 and 15.6, we first describe how we carry out code detuning to simulate how novice programmers may extend/modify a complex software system to meet changing requirements

3. We certainly do not wish to convey the impression that the classes that are used together must always be packaged together. This property, while desirable in some application domains, cannot be elevated to the status of a universal requirement. Therefore, the usefulness of the metric we formulate in this section depends entirely on the design intent incorporated in a software system. If the software architect intended for the classes that were used together to also be packaged together in the form of modules, then our metric can measure the deviation from this ideal. On the other hand, if such a design principle was irrelevant to an application, then the metric of this section ought not to be applied to the software.

TABLE 1
Summary Presentation of the Metrics

Metric	Rationale
MII	Is the intermodule method call traffic routed through APIs?
NC	To what extent are the non-API methods accessed by other modules?
$BCFI$	Does the fragile base class problem exist across the modules?
IC	To what extent are the modules coupled through inheritance?
$NPII$	To what extent does the implementation code in each class program to the public interfaces of the other classes?
AC	To what extent are modules coupled through association?
$SAVI$	To what extent do the classes directly access the state in other classes?
MU	To what extent are the modules different in size?
CU_m, CU_l	To what extent are the classes different in size?
PPI	How much superfluous code exists in a plugin module?
$APIU$	Are the S-APIs of a module cohesive from the standpoint of similarity of purpose and to what extent are the clients of an API segregated?
$CReuM$	To what extent are the classes that are used together also grouped together in the same module?

TABLE 2
Software Systems Used for Metrics Validation

Software System	Abbr. Name	Total classes	LOC	Total methods	Total call-dep	Total modules
JDT Eclipse Plugin	JDT	1383	397917	17975	62106	55
Open-Source Customer Relationship Management	Compiere	1261	317196	18128	39374	37
A J2EE Reference Implementation	Petstore	333	33282	1859	1444	46
Proprietary Customer Relationship Management	P-CRM	5063	2681573	66427	347287	35
Proprietary Financial Application	FinApp	551	153824	6755	9002	41
An Open-Source UML Modeling Tool	ArgoUML	1643	285284	12300	25603	56
Product Line Behavioral Synthesis Tool	PLiBS	308	70235	3701	6646	16
Open-Source Code Generator	Merlin	706	96863	4326	5675	42

under the pressure of deadlines. Section 15.6 then presents the metric values as the code is made to deteriorate with the help of the detuners.

This experimental validation process was applied to a mix of open source systems and proprietary business applications. These software systems are medium to large sized. In order to keep our experiments focused and to minimize language-specific nuances, we restricted ourselves to only the Java programming language. The open source software systems we chose are

1. Java Development Tool (JDT), which is a plugin to the Eclipse development framework,
2. Compiere, an open-source system for customer relationship management (CRM) and enterprise resource planning,
3. a proprietary CRM system,
4. a proprietary financial application,
5. Petstore, a best-practices reference application for J2EE technologies by Sun Microsystems, and
6. a popular open-source UML modeling tool, ArgoUML.

To these six software systems, we added the PLiBS tool from INRIA and the Merlin plugin (an open source code generator tool) for a reason that will be made clear in Section 15.4.

The code for both PLiBS, which stands for Product Line Behavioral Synthesis, and Merlin includes a significant portion created by automatic code generators based on Model Driven Design (MDD). Such generators can be trusted to produce code that does not violate the basic tenets of good OO design. More specifically, the auto-generated code in PLiBS and Merlin was produced by the Eclipse Modeling Framework (EMF). Table 2 shows the various statistics related to these software systems.

It is important to realize that the software systems listed in Table 2 do not come with the sort of module-level APIs that our metrics need for the characterization of software. The large object-oriented software systems of today often do group classes together into large modules, but the published APIs are still only at the class level in all of the cases we know of. The concept of information hiding at the package level (or at the superpackage level) cannot yet be found in the production code. But, obviously, that is bound to change as the need for more efficient organization of large software becomes an imperative.

Therefore, in the rest of this section, we first describe a module discovery step for systems listed in Table 2. We will show how, by manually analyzing the roles played by different classes and/or packages, we can bundle them together to form the modules in the sense in which we use that term in this paper. We will then describe an API discovery procedure for such modules in Section 3. We will finally present our two-pronged approach to metrics validation in the rest of this section.

15.1 Modules Formed for the Experimental Validation Study

The manual modularization was carried out on the basis of the functionality of the Java packages. In modern software, it is not uncommon that a given functionality would be provided by a group of packages working together, as opposed to by a single package. Even for a single functionality, the developers may choose a particular package-based organization for a variety of reasons: 1) As the development effort is divided among different teams around the globe, it may make sense for each team to create its own package; 2) security considerations may dictate that the vulnerable portion of the code needed for a given functionality be placed in a separate package; 3) the requirements of the communications back-end may dictate that, within a given functionality, the code related to the interaction with remote machines be placed in a separate package, and so on.

To create modules in the sense in which we have used that concept in this paper, it therefore makes sense to manually examine the different packages and to cluster them together into modules if the related packages are supposed to contribute to the same functionality. For an example of this manual creation of modules, the `Petstore` system comes with the com.sun.j2ee.blueprints.mailer package that has other finer-grained packages, such as `exceptions` and `util`, associated with it. These finer grained packages contribute solely to the functionality provided by the com.sun.j2ee.blueprints.mailer package. In other words, the finer grained packages are not meant to be exposed to that part of the `Petstore` software that is outside of the com.sun.j2ee.blueprints.mailer package. It therefore makes sense to group the `exception` and `util` packages along with the com.sun.j2ee.blueprints.mailer package into a larger module.

In summary, we used the following manual procedure for partitioning a given software system into modules for the purpose of our validation experiments:

1. We extracted the package hierarchy of the system.
2. We analyzed the functional architecture of these systems through documentation wherever possible. For instance, the open-source *Compiere* system comes with elaborate documentation that describes in adequate detail how the functionality of the system is distributed across the packages. Similarly, the *Petstore* application comes with an architectural overview of the system that can be used to manually combine several fine-grained packages to form modules in the sense used in this paper.
3. Wherever possible, we verified the final modular structure with the human experts involved in managing the systems.

15.2 API Identification

In order to define an S-API for a module created in the manner described above, we first identified the set of public abstract methods implemented in each module. Such public abstract methods were considered to constitute the superset of the S-API methods of a module. A cardinal tenet of object-oriented programming is that one must always program to the interface, meaning to the abstract methods declared at the root of a class hierarchy, as opposed to their implementations in specific concrete classes that descend from the root. Assuming that the writers of a software system adhered to this tenet, it makes sense to think of such abstract methods in a module as constituting one or more S-APIs for that module.

Obviously, not all of the abstract methods in a module would belong to its S-APIs since some of them may actually belong to its E-APIs, as we will explain shortly. So, from all of the abstract methods discovered in a module, we isolated those that are actually implemented in the module itself. From the pool of abstract methods thus found, we removed those that are never called by any of the other modules—since these would be intended solely for the internal workings of the module. To the remaining abstract methods, we added those concrete methods (assuming they were not already in the pool of the remaining abstract methods) that are frequently called by the other modules.

Many popular object-oriented systems such as Eclipse IDE, Mozilla browser, the open-source Wiki application xWiki, etc., utilize the notion of plugins. So, one would expect the main modules of these applications to possess E-APIs as defined in Section 3. Unfortunately, none of these applications follows a uniform approach for formally declaring extension interfaces. This made it somewhat challenging for us to easily identify extension interfaces by analyzing the code. Therefore, for the purpose of our experiments, we used the following heuristics to identify the E-APIs:

- We identified the set of abstract methods in each module whose implementations existed in the other modules.
- As abstract method identified as above was retained for an E-API only if it was called in the same module in which the abstract method was first declared. In other words, in order to be retained for an E-API, an abstract method needed to have its implementation code outside the module in which it was declared and it needed to receive only internal calls from that module.

15.3 Identification of Classes Used Together

For the metric presented in Section 13, we need to determine the classes that are frequently used together. In order to find such sets of classes, we used the well-known frequent-itemset mining algorithm called Apriori [50]. The frequent-itemset mining approach, as represented by Apriori [50], aims at finding frequently occurring sets of items in a large database of items. An itemset is a set of items, as the reader would expect.

Recently, researchers (see, for example, Sartipi and Kontogiannis [51] and Li et al. [52]) have used the notions of item, itemset, and basket (a transaction involving items) in the context of software systems to extract implicit programming rules from software. By implicit programming rules we mean commonly used coding patterns in the source code. In our case, we treat each class as an item and apply the frequent-itemset data mining method to the database of method-call dependency relations. For a given class c, we first identify the set of all other classes whose methods are called by some method of c and treat this set as a basket. After creating a set of all possible baskets, we apply the frequent-itemset mining algorithm implemented by Borgelt [53] to get a set of frequent itemsets. Each frequent itemset is essentially the set of classes whose methods are frequently called together. In other words, a frequent itemset is a set of classes that are frequently used together.

15.4 Metrics Validation—Part 1 (Analysis of the Metric Values for As-Is Open-Source Software)

This section contains the first of a two-pronged approach we have used for validating the proposed metrics. In this section, we present the values of the metrics for the software systems listed in Table 2 and then justify these values on the basis of our understanding of the software packages.

The metric values we obtain for the different software systems of Table 2 are shown in Fig. 4. In the rest of this section, we will present our insights as to why the values presented in Fig. 4 are reasonable.

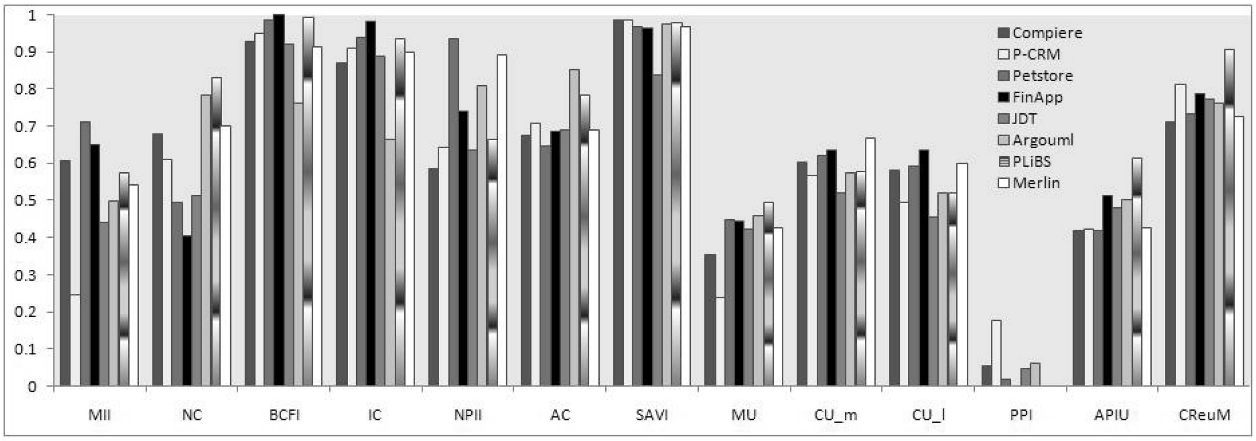


Fig. 4. Metric values calculated for the software systems of Table 2.

Starting with the values of MII at the extreme left in Fig. 4, we see a rather wide variation in values between the systems. We manually peered into the software systems to find an explanation for these values. As one would expect, with respect to the S-APIs formed as discussed in Section 15.2, we found significant intermodule method call traffic that was not routed through the identified S-APIs. In other words, the modules made direct calls to the methods of the concrete classes in the other modules, as opposed to going through the calls included in the S-APIs. That the MII metric behaved as expected was further confirmed by its values for PLiBS and Merlin. As mentioned previously, a significant portion of these systems is autogenerated with the EMF model-driven code generator. Since the autogenerated portion of each system does not violate the basic tenets of good OO design, it should contribute to enhancing the value of MII. As shown in Fig. 4, MII values for Merlin and PLiBS are above the average of the MII values for all of the systems that we considered. Our manual inspection revealed that EMF generates a Java interface and an implementation class for each model element and ensures that the interaction among these model elements takes place only through the interfaces and abstract methods. It therefore implies that, without the portion of the code that is manually added after the autogenerated code comes into existence, the MII values would be even closer to 1. For Merlin, 32 percent of all its methods are manually created, whereas the same for PLiBS is 57 percent. It is also instructive to compare the MII values of PLiBS and Merlin with the worst MII values shown in Fig. 4. These are for the P-CRM system. We observed that much of the intermodule method call traffic in P-CRM does not pass through the identified S-APIs for the modules. So it is not surprising that, when the MII is calculated on a per-module basis, we have more than 60 percent of the P-CRM modules with MII values of less than 0.3. As a specific case in point, for one of the modules of P-CRM, only 3,983 out of a total of 20,099 incoming method calls were routed through the module's S-API.

The second cluster of bars in Fig. 4 is for the NC metric of Section 5.2. As the reader will recall, when a large software system is in a state of semimodularization, some of the non-API methods of a module may participate in intermodule method calls. The second column of Table 3, referred to as "%Bad method calls," shows the percentage of non-API methods calls. Observe that Merlin and PLiBS performed well with respect to NC, indicating that a clear distinction was made between the API methods and the non-API methods in these systems. The NC value of FinApp is intriguing. Even though the extent of intermodule call traffic through non-API methods is smaller, as reflected in its respectable MII value, the extent of non-API methods getting called by other modules is high. For more than 55 percent of modules in FinApp, the value of NC was less than 0.5.

The third cluster of bars in Fig. 4 is for the $BCFI$ metric of Section 6. A manual examination of the software for Petstore and FinApp revealed the absence of the Fragile Base-Class Problem in these two systems. For the financial application FinApp, not a single class made calls to its inherited methods defined in other modules. So it is not surprising that the $BCFI$ index is high for both of these systems in Fig. 4. It is interesting to note that ArgoUML, a popular open-source UML design tool, has a low value for $BCFI$. That is to be expected for such software since it critically depends on the polymorphic behavior of the functions associated with signal emitting GUI widgets. For ArgoUML, a total of 531 classes contain method calls that invoke other class methods that are overridden in derived classes in other modules.

That brings us to the fourth bar-cluster in Fig. 4 for the IC metric of Section 6.3. Recall that, for a given module, the composite IC metric measures the extent to which the

TABLE 3
Statistics Related to Bad Methods, Direct Access to State, and Percent of Classes in Inheritance Hierarchy

System Name	%Bad method calls	Number of direct state accesses		%Classes in inheritance tree
		Intramodule	Intermodule	
JDT	48.870%	12068	11783	59.219%
Compiere	31.996%	838	1712	61.14%
Petstore	50.423%	124	48	16.51%
P-CRM	38.933%	5426	19776	68.24%
FinApp	59.650%	420	1361	3.63%
ArgoUML	21.539%	295	1242	69.63%
PLiBS	16.641%	290	210	31.81%
Merlin	30.034%	14	220	30.87%

software is free of direct inheritance-induced intermodule couplings. The best result for the IC metric are for FinApp and Petstore and the worst for ArgoUML. For Petstore, 9 out of 46 modules have inheritance-based linkages with the other modules. FinApp has only 4 such modules out of a total of 41. ArgoUML, on the other hand, has 51 modules (out of a total of 56) that are related to other modules through inheritance relationships. The $IC3$ component of the composite IC value for several of the ArgoUML modules was 0 since all classes in such modules were derived from classes in other modules.

The fifth cluster of bars in Fig. 4 is for the $NPII$ metric of Section 7. Recall that this metric measures the extent to which software is free of the discouraged practice of not programming to interfaces. The following observation about the software systems has a special bearing on this metric: Some of the systems contained individual classes that did not belong to any class hierarchies. So, naturally, these classes did not descend from any root interfaces. Therefore, the public methods of such classes were not represented in the S-APIs of the modules. The last column of Table 3 shows the ratio of classes that do belong to class hierarchies characterized with root interfaces to the total number of classes in each of the software systems of Table 2. All classes that existed only as individual classes were excluded from the $NPII$ calculations. Comparing the last column of Table 3 with the $NPII$ values shown in Fig. 4, it is interesting to see that Compiere and P-CRM, with a majority of their classes operating on a stand-alone basis, also score low as far as the $NPII$ metric is concerned. At the opposite end, systems such as Petstore, Merlin, and ArgoUML have properly defined public interfaces at the root of most of all their class hierarchies. So, the $NPII$ metric scores high for these software systems.

The sixth set of bars in Fig. 4 shows the composite AC metric values introduced in Section 8. This metric tells us to what extent the software is free of associational couplings between the modules. As the AC bars in Fig. 4 show, ArgoUML scores the highest on this metric with a value of 0.9, implying that this software system has low associational coupling between the modules. On the other hand, Petstore has the lowest score of below 0.7, implying that this software has relatively high associational coupling.

It is interesting to compare the IC and the AC bars in Fig. 4. Software systems that score high on IC score low on AC and vice versa. This should not be surprising. A system like ArgoUML uses inheritance and polymorphism as primary mechanisms to couple the high-level functionality encoded in the classes that sit close to the roots of their respective hierarchies with the operating-system specific details in the derived classes that are farther removed from the root. So, ArgoUML scores low on IC (because a low value for IC means high inheritance-based coupling). On the other hand, a software system such as Petstore uses associations as a primary mechanism to couple classes and modules. So, it scores low on AC. As mentioned earlier, low AC values mean high associational coupling between the modules.

That brings us to the bar cluster for the $SAVI$ metric of Section 9 in Fig. 4. Most of the systems considered adhered to the principle of not directly accessing the state of a class. The small number of violations we did observe were for JDT and P-CRM. Surprisingly, the number of direct state access violations was much higher in JDT than in P-CRM. This is due to the fact that a large number of classes in JDT directly access the state of other classes. We have tabulated such state accesses for the different software systems in Table 3. The table also shows a breakdown of the number of such accesses on intermodule and intramodule bases.

We next consider the bar clusters of Fig. 4 for the size-based metrics MU, CU_l, and CU_m of Section 10. FinApp exhibits the lowest size variability for all three metrics. The highest variability with respect to the MU metric is exhibited by P-CRM and with respect to both CU_m and CU_l by JDT. With regard to the MU metric, the number of classes packed into the individual modules of FinApp range from just 1 to 80. On the other hand, the number of classes packed into the individual modules of P-CRM range from 2,506 to 1. With regard to the CU_m metric, the largest class of FinApp has 125 methods, while there exist classes with just one method. On the other hand, the largest class in JDT has 417 methods, while there exist classes with just one method. With regard to the CU_l metric, the largest class contains 6,000 lines of code and the smallest just 10 lines. On the other hand, the largest class of JDT contains more than 8,000 lines of code and there exist classes with just 10 lines of code.

With regard to the bars shown in Fig. 4 for the PPI metric of Section 11, note that not all of the systems come with plugin modules. Additionally, even in systems that do come with plugins, the number of E-APIs discovered by our prototype metric validation tool is small. The PPI metric values shown for the systems that had E-APIs reflect the code quality in the extension modules.

That brings us to the bars shown for the $APIU$ metric of Section 12 in Fig. 4. Recall that this metric measures S-API segregation and cohesiveness. What confounded the measurement of this metric was that, for many of the systems, an S-API services only a single client module. When this is the case, it is obviously not possible to determine whether the S-API of a service provider module is cohesive. Therefore, we excluded such cases from the metric calculation. This metric has low values for Compiere. As it turns out, Compiere contains several modules with S-APIs that provide service to highly overlapping clients. For instance, for the `org.compiere.model` module, all of the S-APIs used for calculating $APIUCSet$ had completely overlapping client modules. As a result, the $APIUS$ of this module was 0. System level $APIU$ was found to be good for ArgoUML, JDT, and PLiBs. In PLiBS, we have seen several highly cohesive S-APIs (APIUC value between 0.5 and 1).

The last set of bars in Fig. 4 is for the *CReuM* metric of Section 13. Except for PLiBS, the software systems score more or less the same on this metric. Recall that this metric measures the extent to which the classes that are defined together in a single module are also used together in other modules. The result shows that, in PLiBS, most of the classes that are frequently called together have also been grouped into the individual modules.

15.5 Code Detuning for Simulating Software Decay

An important step in our experimental validation study is code detuning to inject randomized modularization deficiencies into the software. Obviously, it is important that this detuning not be specific to any of the metrics. We want the detuners to capture, to the extent it is feasible to do so without getting into a complex simulation of human behavior, the scenario of a team of novice programmers trying to extend a large OO software system. The goal of this section is to list the detuning operators we used for that purpose.

Using the notation introduced in Section 4, each detuning operator DT essentially modifies a software system $\mathcal{S}$. This modification may be represented by

$$DT : \mathcal{S} =< Ent, Rel > \quad \rightarrow \quad \mathcal{S}' =< Ent', Rel' > .$$

The modifications consist of 1) adding a new entity to Ent; or 2) adding a new relation into Rel; or both. The system degradation is achieved by a repeated application of these detuners.

Detuner for injecting non-API method calls (DT_1). When novice programmers extend object-oriented software for large business applications, it is not unlikely for them to have one module call another module's non-API method. The non-API method may be exactly the right method to get the job done quickly, but, of course, its usage would be at the cost of injecting a non-API method call into the software. We simulate this form of code decay with the DT_1 code detuner. This operator introduces a new call dependency, $< m, m' >$, across modules where a randomly chosen method m makes a call to another randomly chosen public method m' in another randomly selected module.

Detuner for not programming to interfaces (DT_2). Novice programmers are also likely to directly invoke a noninterface concrete method of a concrete class for quickly extending the software, instead of invoking the methods through the public interfaces of the abstract classes at the roots of the inheritance trees. This sort of code decay is simulated by the DT_2 code detuner. This operator introduces a new call dependency, $< m, m' >$, where a randomly chosen method m calls another randomly chosen concrete method m' of a randomly selected child class.

Detuner for injecting abuse of inheritance (DT_3). A programmer may introduce couplings between modules when a class in one module is extended as a derived class in another module. The detuning operator for injecting such inheritance-based couplings introduces a new class, c_{bad}, in a randomly selected module that is subclassed from another randomly selected class c that already exists in the system. The detuner also introduces a new override relationship, $< m, m_{bad} >$, into the system, where m is a randomly chosen public method of c and m_{bad} is a new method of c_{bad} that overrides m.

Detuner for injecting direct access to state (DT_4). As mentioned previously, generally allowing public access to the attributes defined for a class is considered to be a poor practice. The detuning operation that enhances such direct access adds a new $Uses(c, d)$ relationship into the software by choosing a randomly selected class c and class d that has an attribute with public access.

Detuner that makes API usage more diffuse (DT_5). When complex software is extended under the pressure of deadlines, it is particularly difficult to adhere to the more subtle of the good programming practices. For example, even when a module is used only through its API, some other module may use that API in a piecemeal fashion in order to meet the immediate requirements on the extended software. Such piecemeal extension would cause the API usage to become more diffuse. In other words, with piecemeal extensions, the APIs could lose their cohesiveness. We simulate this aspect of code deterioration with the following detuner: The detuning operator introduces into the software a set of calls $\{< m, m' >, < m_1, m' >, \cdots\}$ to a randomly selected API method m' from a number of other randomly selected methods $\{m, m_1, \cdots\}$ in other modules.

15.6 Metrics Validation—Part 2 (Validation with Code Detuning)

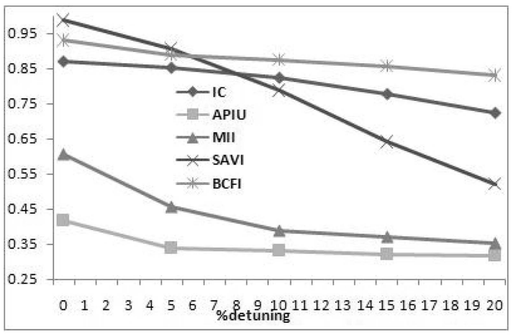
As mentioned earlier, the code detuners we presented are for the purpose of validating the metrics. That is, we want to show that, if the code undergoes a certain extent of deterioration, the metrics would be able to record that deterioration by changes in their values.

With regard to validating the metrics with code detuners, note that this approach is not appropriate for all of the metrics listed in Table 1. Obviously, some of the metrics, like the size-based metrics MU, CU_m, and CU_l, need no validation anyway since they stand on their own. The task assigned to them is to measure size variations and they do so in a straightforward manner.

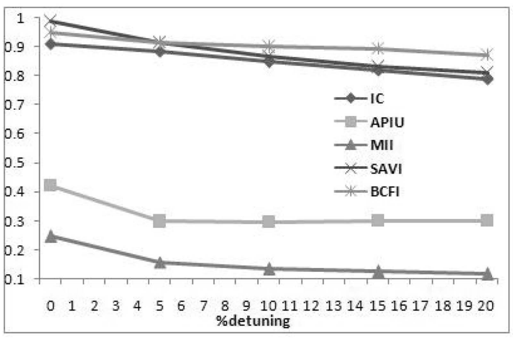
A code detuning-based validation would also seem inappropriate for metrics like PPI, $CReuM$, and $NPII$. The sort of code deterioration represented by PPI falls in a category by itself. Recall that this metric tells us to what extent there may exist superfluous code in the plugin modules of a software system. We could, of course, inject a certain amount of code bloat into the plugin modules in order to then measure it by the PPI metric. But, that strategy would not constitute a fair test of the metric. The metric would, of course, measure the presence of the deliberately added superfluous code, just as a size-based metric would record the change in the size of a module if you added random code to it. Causing code deterioration in this manner would be too specific to a particular metric and would not necessarily represent the actions of a novice programmer. No novice programmer is going to gratuitously add code to a plugin module. One could make similar comments about why a detuning-based approach would be inappropriate for validating the $CReuM$ and $NPII$ metrics.

Therefore, in this section, we will use the detuners to focus on the validation of the MII, $BCFI$, IC, $SAVI$, and $APIU$ metrics. Fig. 5 shows plots of how these metrics change when a software system undergoes deterioration by a certain percentage amount. When we say that the software has deteriorated by x percent, we mean we injected x percent deterioration into the system with equal contributions from each of the five detuners listed in the previous section.

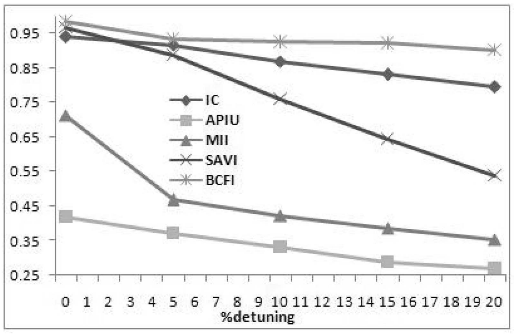
The important thing to note in Fig. 5 is that all of the metrics shown there exhibit monotonically decreasingly values as each software system is subject to increasing deterioration through the application of all five code detuning operators. It is more than likely that each metric



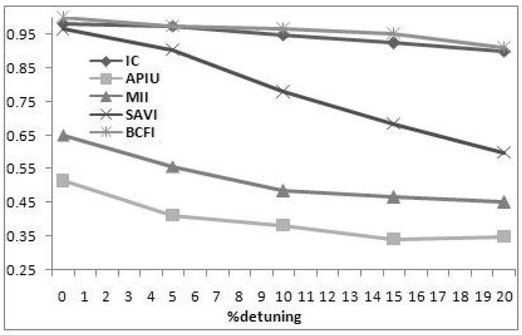
(a)



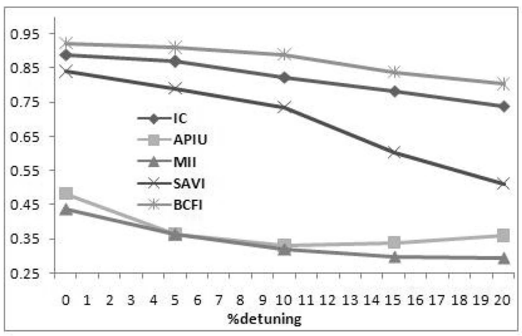
(b)



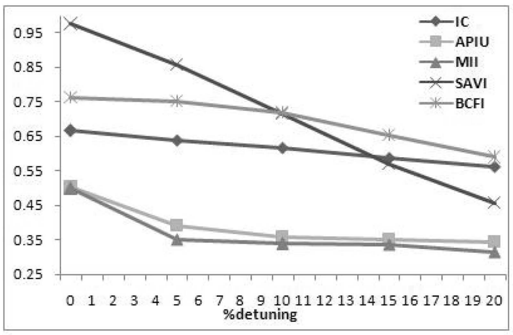
(c)



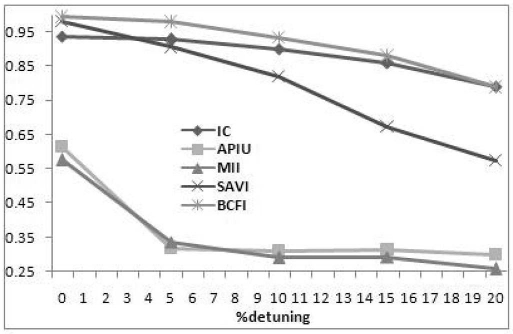
(d)



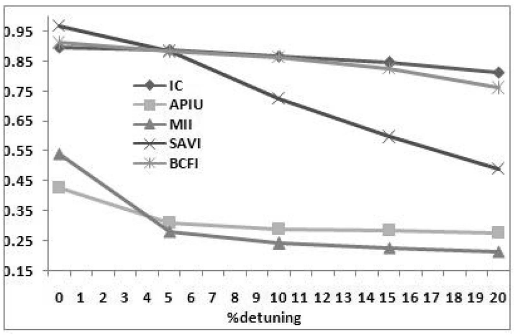
(e)



(f)



(g)



(h)

Fig. 5. Validation of metrics with code detuning. The plots show the metric values versus code detuning for (a) Compiere, (b) P-CRM, (c) Petstore, (d) FinApp, (e) JDT, (f) ArgoUML, (g) PLiBS and (h) Merlin.

plotted in the figure has different degrees of sensitivity to the different detuning operators. One would, for example, expect that MII was affected mostly by the decay introduced by the DT_1 operator since it is this operator's job to inject non-API calls into a system. By the same token, we can say that $BCFI$ was likely to be most affected by the DT_3 operator because it injects random inheritance-based dependencies between the modules. For obvious reasons, we can also consider DT_3 to be the main culprit for the changes in the values of the IC metric in the different plots of Fig. 5. As to which detuning operator the $SAVI$ metric might be most sensitive to, we suspect it is DT_4 since this operator randomly injects new $Uses(c,d)$ relations into the software to create random accesses to the state. Finally, with regard to the $APIU$ metric, we believe that the intermodule couplings injected by the DT_5 operator are what the metric is measuring with increasing decay of the software.

15.7 Threats to Validity

In general, before one can place confidence in experimental outcomes, one must understand various forms of potential threats to the validity of those outcomes. These threats are different for the different forms of validity related to the experiments. The major categories of the validity are the construct validity, the internal validity, and the external validity [54], [55].

The construct validity questions whether the proposed approach is really measuring what it is supposed to measure. Construct validity in our case boils down to the trust we can place in our metrics. By using two totally independent experimental procedures for metrics validation, we can say with confidence that our metrics actually measure what they claim to measure—the quality of modularization of large-scale OO software.

The internal validity questions the design of the experiment itself. The main experimental step that has a bearing on internal validity is the identification of the APIs of the modules. Here, we followed what is probably the best known tenet of object-oriented programming: *One must always program to interfaces*. By seeking only those functions that were declared in the root interfaces of the class hierarchies, we made API identification less arbitrary. As matters stand today, this approach to making the discovered APIs less arbitrary definitely applies to the S-APIs. But, as mentioned in Section 15.2, with regard to E-APIs, there is currently a significant departure from the practice of programming to interfaces. So, our conclusions regarding the PPI metric can be trusted only to the extent to which the heuristics we used for identifying the E-APIs are a reflection of the reality. Another experimental design factor that relates to the internal validity is the correctness of the module formation. We believe that the steps we have used for software partitioning, as described in Section 15.1, protect against this being a potential threat to the validity. Yet another internal validity issue that affects the $CReUM$ metric is the procedure we use to identify the classes that are used together. For the software systems we have analyzed, we have observed that the frequent itemset mining based approach is reasonably accurate for identifying such classes.

As for the external validity, the question here is to what extent our metrics can be used for all large object-oriented software systems, considering especially that all of our validation studies were carried out on Java-based platforms. Considering that the fundamental notions of object-oriented programming—inheritance, polymorphism, encapsulation, etc.—are very similarly implemented in all major OO languages, we are confident that our metrics would also apply to software systems in other languages.

16 Conclusion

We have proposed a new set of metrics to analyze the quality of modularization of large object-oriented software in which each module may consist of a large number of classes, with the modules exposing only a small number of the public methods of some of their classes in their APIs.

If there is one theme that could be considered central to the new metrics reported here, it is the notion of API. Fundamentally, an API is a portal through which all traffic directed at the module must pass. To the extent to which every module respects the APIs of all other modules, we can consider modularization to be perfect from at least the perspective of there being no hidden intermodule linkages created by non-API calls. Unfortunately, as software ages, it rarely stays in that sort of a pristine condition. And, as non-API method call linkages start to proliferate, the software becomes increasingly more difficult to maintain and extend. Our MII and NC metrics are meant to record the extent to which the modules are coupled through different forms of intermodule method call traffic. Since the notion of API plays a large role in our work, our metrics must include measures of the quality of APIs themselves. That is the reason for the formulation of the $APIU$ metric.

Obviously, and especially in object-oriented software, there are other mechanisms besides the intermodule call traffic that can create intermodule couplings. Modules can, for example, become coupled because of inheritance and associational relationships between the classes in different modules. Inheritance and the related mechanism of polymorphism create a particularly insidious form of coupling between the modules; this we referred to as the Fragile Base-Class Problem. Our $BCFI$, IC, and AC metrics assess software with respect to the inheritance-induced couplings between the modules. While it is true that the $BCFI$, IC, and AC metrics are not aware of the APIs of the modules, nonetheless they define critical ancillary software properties without which any measurements made by MII and NC would not be useful.

In addition to the aforementioned mechanisms, another not-so-uncommon practice that creates intermodule linkages is that of the implementation code in one class directly accessing the state of objects constructed from classes that reside in other modules. Our $SAVI$ metric measures the extent to which software is free of such practices. The rationale for including the PPI metric is that object-oriented software for large applications increasingly relies on third-party plugins for extending the functionality of the original software; it is not uncommon to see the occurrence of code-bloat in these third-party packages for the plugins. The metric $NPII$ tries to measure the extent to which the object-oriented software does not follow the recommended practice of programming to interfaces. The $CReuM$ metric that could prove useful in some applications measures the extent to which classes that are defined together also get used together. It is generally true that, when classes that are defined together largely get used together, it becomes easier to carry out an impact analysis of any changes to those classes since the impact tends to be relatively localized in the rest of the software. Our other metrics—MU, CU_m, and CU_l—can be thought of as support metrics. As software is being developed, it is good to keep an eye on the various size parameters associated with the software.

We used a two-pronged approach to validate the proposed metrics. In our first approach, we computed the metrics for eight different software systems, including some very well-known open-source software systems. We then argued that the values of the metrics conformed to the various attributes gleaned from manual examination of the software. In our second approach to metrics validation, we presented the idea of code detuners to simulate how novice programmers can be expected to extend a body of object-oriented code. We applied these code detuning operators to create different degrees of decay in the software systems we had analyzed earlier. By computing the metric values for the software that had been subjected to this simulated decay, we showed that the metrics behaved as one would want them to. As we pointed out, this second approach is not appropriate for all of the metrics. Nonetheless, it provided us with a powerful tool to validate a core set of metrics to which the approach could be applied.

With regard to future extensions of our work, one aspect that is readily open for improvement is the automatic discovery of APIs (both S-APIs and E-APIs). The heuristics we have used for API discovery are generic and do not make assumptions regarding underlying frameworks or application domains. For instance, the EJB framework has notions such as the "Bean interface" that can be exploited to better identify the APIs. Customizing our API discovery procedures to specific frameworks could be a potentially useful extension of the research reported here. Another topic for future study would be to carry out a multivariate analysis of the metrics presented here to see as to what extent the metrics are independent and consistent. In order to be statistically meaningful, such a study would require a large number of software systems. Finally, one of our motivations for developing the new set of modularity metrics is to assess the quality of the modularization obtained when legacy code is reorganized to conform to modern software engineering practice. It will be interesting to see to what extent the metrics proposed here succeed in accomplishing that difficult task.

REFERENCES

[1] "JSR-277: JavaModule System," Technical Report 2006, Sun Microsystems, Inc., http://jcp.org/en/jsr/detail?id=277, 2008.

[2] "JSR-294: Improved Modularity Support in the Java Programming Language," Technical Report 2007, Sun Microsystems, Inc., http://jcp.org/en/jsr/detail?id=294, 2008.

[3] R. Strnisa, P. Sewell, and M. Parkinson, "The Java Module System: Core Design and Semantic Definition," *Proc. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages and Applications,* vol. 42, no. 10, pp. 499-514, 2007.

[4] S. Sarkar, G.M. Rama, and A.C. Kak, "API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization," *IEEE Trans. Software Eng.,* vol. 33, no. 1, pp. 14-32, Jan. 2007.

[5] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, pp. 476-493, 1994.

[6] N. Churcher and M. Shepperd, "Towards a Conceptual Framework for Object-Oriented Software Metrics," *ACM SIGSOFT Software Eng. Notes,* vol. 20, no. 2, pp. 69-75, 1995.

[7] C.J. Coppick and T.J. Cheatham, "Software Metrics for Object-Oriented Systems," *Proc. ACM Ann. Computer Science Conf.,* pp. 317-322, 1992.

[8] M.H. Halstead, *Elements of Software Science.* Elsevier, 1977.

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

[10] F. Brito e Abreu and R. Carapuca, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework," *J. Systems and Software,* vol. 26, pp. 87-96, 1994.

[11] J.-Y. Chen and J.-F. Lum, "A New Metric for Object-Oriented Design," *Information of Software Technology,* vol. 35, pp. 232-240, 1993.

[12] Y.-S. Lee, B.-S. Liang, and F.-J. Wang, "Some Complexity Metrics for Object-Oriented Programs Based on Information Flow," *Proc. Sixth IEEE Int'l Conf. Computer Systems and Software Eng.,* pp. 302-310, 1993.

[13] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Approach.* Prentice Hall, 1994.

[14] W. Li and S. Henry, "Object Oriented Metrics that Predict Maintainability," *J. Systems and Software,* vol. 23, pp. 111-122, 1993.

[15] W. Li, "Another Metric Suite for Object-Oriented Programming," *J. Software and Systems,* vol. 44, pp. 155-162, 1998.

[16] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity.* Prentice Hall, 1996.

[17] L.C. Briand, S. Morasca, and V.R. Basili, "Defining and Validating Measures for Object-Based High-Level Design," *IEEE Trans. Software Eng.,* vol. 25, no. 5, pp. 722-743, Sept./Oct. 1999.

[18] F. Brito e Abreu, "The MOOD Metrics Set," *Proc. Ninth European Conf. Object-Oriented Programming Workshop Metrics,* 1995.

[19] F.B. e Abreu, M. Goulao, and R. Estevers, "Towards the Design Quality Evaluation of OO Software Systems," *Proc. Fifth Int'l Conf. Software Quality,* 1995.

[20] R. Harrison, S.J. Counsell, and R.V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 491-496, June 1998.

[21] S. Counsell, S. Swift, and J. Crampton, "The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design," *ACM Trans. Software Eng. and Methodology,* vol. 15, no. 2, pp. 123-149, 2006.

[22] J. Bansiya, L. Etzkorn, C. Davis, and W. Li, "A Class Cohesion Metric for Object-Oriented Designs," *J. Object Oriented Program,* vol. 11, no. 8, pp. 47-52, 1999.

[23] S. Counsell, E. Mendes, S. Swift, and A. Tucker, "Evaluation of an Object-Oriented Cohesion Metric through Hamming Distances," Technical Report BBKCS-02-10, Birkbeck College, Univ. of London, 2002.

[24] M.D. Ghassemi and R.R. Mourant, "Evaluation of Coupling in the Context of Java Interfaces," *Proc. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages and Applications,* pp. 47-48, 2000.

[25] M. Hitz and B. Montazeri, "Chidamber and Kemerers Metrics Suite: A Measurement Theory Perspective," *IEEE Trans. Software Eng.,* vol. 22, pp. 267-271, 1996.

[26] 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.

[27] R.K. Bandi, V.K. Vaishnavi, and D.E. Turk, "Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics," *IEEE Trans. Software Eng.,* vol. 29, no. 1, pp. 77-86, Jan. 2003.

[28] L. Etzkorn, C. Davis, and W. Li, "A Practical Look at the Lack of Cohesion in Methods Metrics," *J. Object Oriented Programming,* vol. 11, no. 5, pp. 27-34, 1998.

[29] 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, Jan./Feb. 1999.

[30] H. Kabaili, R.K. Keller, and F. Lustman, "Cohesion as Changeability Indicator in Object-Oriented Systems," *Proc. Fifth European Conf. Software Maintenance and Reengineering,* pp. 39-46, 2001.

[31] 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.

[32] M. Dagpinar and J.H. Jahnke, "Predicting Maintainability with Object-Oriented Metrics—An Empirical Comparison," *Proc. 10th Working Conf. Reverse Eng.,* p. 155, 2003.

[33] R. Marinescu, "Detecting Design Flaws via Metrics in Object Oriented Systems," *Proc. 39th Int'l Conf. and Exhibition on Technology of Object-Oriented Languages and Systems,* pp. 173-182, 2001.

[34] B. Lague, D. Proulx, E.M. Merlo, J. Mayrand, and J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," *Proc. Int'l Conf. Software Maintenance,* 1997.

[35] K. Kontogiannis, "Evaluating Experiments on the Detection of Programming Patterns Using Software Metrics," *Proc. Working Conf. Reverse Eng.*, pp. 44-54, 1997.

[36] V.R. Basili, L.C. Briand, and W. Melo, "A Validation of Object Oriented Design Metrics as Quality Indicators," *IEEE Trans. Software Eng.*, vol. 22, pp. 751-761, 1996.

[37] L.C. Briand, J.K. Wust, J.W. Daly, and D.V. Porter, "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems," *J. Systems and Software*, vol. 51, pp. 245-273, 2000.

[38] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 897-910, Oct. 2005.

[39] R. Subramanyam and M. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Trans. Software Eng.*, vol. 29, 2003.

[40] H.M. Olague, L.H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes," *IEEE Trans. Software Eng.*, vol. 33, no. 6, pp. 402-419, June 2007.

[41] R. Barker and E. Tempero, "A Large-Scale Empirical Comparison of Object-Oriented Cohesion Metrics," *Proc. 14th Asia-Pacific Software Eng. Conf.*, pp. 414-421, 2007.

[42] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics Based Refactoring," *Proc. Fifth European Conf. Software Maintenance and Reengineering*, pp. 30-38, 2001.

[43] M. Alshayeb and W. Li, "An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes," *IEEE Trans. Software Eng.*, vol. 29, no. 11, pp. 1043-1049, Nov. 2003.

[44] M.M. Carey and G.C. Gannod, "Recovering Concepts from Source Code with Automated Concept Identification," *Proc. 15th IEEE Int'l Conf. Program Comprehension*, 2007.

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

[46] R.E. Johnson and B. Foote, "Designing Reusable Classes," *J. Object-Oriented Programming*, vol. 1, no. 2, pp. 22-35, 1988.

[47] L. Mikhajlov and E. Sekerinski, "A Study of the Fragile Base Class Problem," *Proc. 12th European Conf. Object Oriented Programming*, pp. 355-383, 1998.

[48] J. Aldrich and K. Donnelly, "Selective Open Recursion: Modular Reasoning about Components and Inheritance," *Proc. Third Workshop Specification and Verification of Component-Based Systems*, 2004.

[49] R. Martin, "Design Principles and Design Patterns," www.objectmentor.com, 2000.

[50] R. Agarwal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 20th Int'l Conf. Very Large Data Bases*, pp. 487-499, 1994.

[51] K. Sartipi and K. Kontogiannis, "Component Clustering Based on Maximal Association," *Proc. Eighth Working Conf. Reverse Eng.*, p. 103, 2001.

[52] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 176-192, Mar. 2006.

[53] C. Borgelt, "Efficient Implementations of Apriori and Eclat," *Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations*, 2003.

[54] A.S. Lee and R.L. Baskerville, "Generalizing Generalizability in Information Systems Research," *Information Systems Research*, vol. 14, no. 3, pp. 221-243, 2003.

[55] T.D. Cook, W. Shadish, and D. Campbell, *Experimental and Quasi-Experimental Designs for Generalized Causal Inference.* Houghton Mifflin, 2002.

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 vice president of R&D of Accenture Technology Labs, India. He was with Infosys Technologies Ltd. during the course of the work reported here. He has nearly 16 years of experience in applied research, industrial software, and product development. His current research interests include architecture modeling, metrics, program comprehension, and reengineering techniques. He is a member of the IEEE and the IEEE Computer Society.

Avinash C. Kak is a professor of electrical and computer engineering at Purdue University and a consultant to Infosys. His book *Programming with Objects* (John Wiley & Sons, 2003) is used by a number of leading universities as a text on object-oriented programming. His most recent book *Scripting with Objects* (John Wiley & Sons, 2008) 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 2009, will be titled *Designing with Objects*. In addition to computer languages and software engineering, his research interests 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.

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