Modularization Metrics: Assessing Package Organization in Legacy Large Object-Oriented Software

Hani Abdeen, Stéphane Ducasse, Houari Sahraoui

Modularization Metrics: Assessing Package Organization in Legacy Large Object-Oriented Software

Hani Abdeen
RMoD team, INRIA - Lille Nord Europe
USTL - CNRS UMR 8022, Lille, France
Email: hani.abdeen@gmail.com

Stéphane Ducasse
RMoD team, INRIA - Lille Nord Europe
USTL - CNRS UMR 8022, Lille, France
Email: stephane.ducasse@inria.fr

Houari Sahraoui
DIRO, Université de Montréal
Montréal(QC), Canada
Email:sahraoui@iro.umontreal.ca

***Abstract*—There exist many large object-oriented software systems consisting of several thousands of classes that are organized into several hundreds of packages. In such software systems, classes cannot be considered as units for software modularization. In such context, packages are not simply classes containers, but they also play the role of modules: a package should focus to provide well identified services to the rest of the software system. Therefore, understanding and assessing package organization is primordial for software maintenance tasks. Although there exist a lot of works proposing metrics for the quality of a single class and/or the quality of inter-class relationships, there exist few works dealing with some aspects for the quality of package organization and relationship. We believe that additional investigations are required for assessing package modularity aspects. The goal of this paper is to provide a complementary set of metrics that assess some modularity principles for packages in large legacy object-oriented software: Information-Hiding, Changeability and Reusability principles. Our metrics are defined with respect to object-oriented inter-package and intra-package dependencies. The dependencies that are caused by different types of inter-class dependencies, such as inheritance and method call. We validate our metrics theoretically through a careful study of the mathematical properties of each metric.**

I. Introduction

Since some decades now, there exist many legacy large object-oriented software systems consisting of a large number of inter-dependent classes. In such systems, classes are at a low level of granularity to serve as a unit of software modularization. In object-oriented languages such as Java, Smalltalk and C++, package structure allows people to organize their programs into subsystems. A well modularized system enables its evolution by supporting the replacement of its parts without impacting the complete system. A good organization of classes into identifiable and collaborating packages eases the understanding, maintenance, test and evolution of software [16].

However, even for well modularized software systems, code decays: as software evolves over time with the modification, addition and removal of classes and inter-class dependencies. As consequence, the modularization gradually drifts and looses quality, where some classes may not be placed in suitable packages and some packages need to be re-structured [19], [23]. To improve the quality of software modularization, assessing the package organization and relationships is required.

Although there exist a lot of works in the literature proposing metrics for object-oriented software, the majority of these previous works focused mostly on characterizing a single class [13], [8], [9], [10], [18], [28], [27], [12], [22], [15]. Few previous efforts measure the quality of some aspects of package organization and relationships [4], [21], [30], [3], [34]. Much of these efforts are focused on package cohesion and coupling from the point of view of maximizing intra-package dependencies. But although this point of view is important for assessing an aspect of package structure, it is definitely not enough for assessing software modularization [3], [30], [17], [2], [1]. Fortunately, Santonu Sarkar *et al.* [35], [36] have recently proposed a set of metrics that characterize several aspects of the quality of modularization. Their metrics are defined with respect to the APIs of the modules (*i.e.*, packages), and with respect to object-oriented intermodule dependencies that are caused by inheritance, associations, state access violations, fragile base-class design, etc. Unfortunately, their metrics are mainly based APIs. They assume that each package explicitly declares its APIs, which is not the case for most legacy object-oriented software systems. Therefore, in the absence of declared APIs at package level, their metrics could not be applied without additional interpretations and heuristics. Although their metrics are valuable and they characterize many aspects of software modularization, we believe that some important aspects are still not characterized.

Our goal is to provide a complementary set of metrics that follow the principles of good software modularity as explained by Parnas [32] and R. Martin [30]. Since we address large legacy software systems, consisting of a very large number of classes and packages, we consider that packages are the units of software modularization: a package should provide well identified services to the rest of the software system, where the role of classes inside a package is implementing the package services. But unfortunately, in legacy object-oriented systems, APIs/Services are often, if any, not pre-defined explicitly at package level (*i.e.*, systems are not API-based). Therefore, the metrics we propose in this paper are *are not API-based*.

In this paper, we consider *package* and *module* to be synonymous concepts. On another hand, we consider the interfaces

of a given package to be the package classes interacting with classes of other packages. For a given modularization, we consider classes to be at the lowest level of granularity.

In the following, Section II underlines the modularity principles that we address in this paper. In Section III we define the terminology and the notations we use to define our metrics. We define a set of coupling metrics in Section IV and a set of complementary cohesion metrics in Section V. In Section VI we show how our metrics satisfy all the mathematical properties that are defined by Briand *et al.* [8], [9]. Before we conclude in Section IX, we discuss our metrics in Section VII and we compare them to previous works related to software metrics in Section VIII.

II. Modularity Principles

In software engineering practices, a module is a group of programs and data structures that collaborate to provide one or more expected services to the rest of the software. According to Parnas [32] and R. Martin [30], a module should hide its design decisions and should provide its services only through its interfaces. The main goal of modules is *information hiding*: only the module interfaces are accessible by other modules. Some object-oriented programming technologies support the definition of module interfaces through the declaration of APIs (Application Programming Interface) [37], [38]: an API may be a set of methods that are implemented in the module classes. Unfortunately, almost all legacy object-oriented software systems are not API-based.

In addition, it is not enough to group some programs inside a module, then declaring the module interfaces. Since a module should provide well identified services to other modules, the module programs should have a common goal: capturing the module design decisions and implementing the module services. It is widely known that a good modularization of a software supports the software changeability, maintainability and analyzability. In the rest of this section we underline the principles that a software modularization should follow, and that we address in this paper.

A. Hiding Information and Encapsulation.

As mentioned above, modules should encapsulate their implementations and provide their services via well identified interfaces.

For a legacy object-oriented systems, where the APIs are not pre-defined, the identifiable package interfaces are the package classes that interact with classes of other packages. In such a case, we assume that the principle of hiding information and of encapsulation requires the following (**Principle I**):

- We consider that hiding information is similar to hiding information exchange or communications. In this way, *the communications (interactions) between packages should be as little as possible*.
- We consider that every package should encapsulate its design decisions and hide its implementation from other packages as much as possible. Thus, *the number of methods/implementations/classes that a package exposes to other packages should be relatively small*.

B. Changeability, Maintainability and Reusability.

It's well known that *Maintenance* activities represent a large and important part of the software life-cycle. One of the primary expected goals from modules is facilitating the software maintenance. Such a goal is usually supported by the localization of module changes impact on other modules. In other words, when changing a given module, the propagation of changing impact on other modules should be minimal (as little as possible). In the same context, modules should be reusable pieces of software. As stated earlier, a module should interact with other modules via well identified interfaces and requires identifiable services from other modules. Therefore, to support software changeability, maintainability and reusability at package level, we assume that (**Principle II**):

- *Inter-package connectivity should be as little as possible*.

C. Commonality-of-Goal vs. Similarity-of-Purpose.

As explained above, a module should provide particular (*i.e.*, well identified) services to other modules. Therefore, the programs inside a module should have a common goal, which should be: capturing the module design decisions and implementing the module services. We refer to this principle as *commonality-of-goal*. In another side, if a module is expected to provide more than one service, the module services then should be as segregated as possible: each service should have a well identified purpose, and the service purpose should be different than the purpose of other provided services. To fulfill these objectives, we assume the following (**Principle III**):

- *Ideally, a package should be as a provider of only one service to the rest of the software*.
- *If not, the goal of each package interface should be as consistent as possible*.

In other words, for an interface participates with other interfaces to provide a service, it is an ideal state if that interface does not participate, aside from those other interfaces, to provide different service(s). In such a case, all the interfaces that participate to provide a service will have only one common goal, which is: implementing and providing that service.

Our Contribution. In this paper, we propose a set of metrics that measure *"to which extent a given OO software modularization is well-organized"*, with regard to the principles we have underlined in this section. We organize our metrics into two subsets: 1) metrics characterize packages *coupling* with regard to the principles I and II; 2) and metrics characterize packages *cohesion* with regard to the principle III.

III. Terminology and Notation

We assume in this paper that the dependencies of a package to other packages are due to the dependencies of the classes inside the considered package to classes outside it. Those dependencies are either method calls or inheritance relationships. In this section we define the terminology and the notation we use in this paper.

A. Dependency Types

By definition, we say that a class c_1 *extends* another class c_2 if c_1 is a direct subclass of c_2. We say that a class a is a *subclass of* another class b if a belongs to the subclass hierarchy of b. By definition, we say that a class x *uses* a class y if x is not a subclass of y, and there is a method directly implemented in x either calls a method directly implemented in y, or refers to an attribute directly defined in y.

For packages, whatever the number of classes inside a package p that have dependencies pointing to classes inside another package q, we say that p *depends on* q. This is regardless the number of concerned classes inside q and the number of inter-class dependencies. In this way, we say that p is *client* to q and q is *provider* to p. By definition, we say that package p *extends* another package q if there is a class in p that extends a class in q. Similarly, we say that p $uses$ q if there is a class in p that uses a class in q.

B. Package Interfaces and Relationships

We assume in this paper that packages in large object-oriented software are the units of the software modularization. The role of classes inside a package is to implement and fulfill the package services. But due to the complexity of the services to provide, a package may contain a large number of classes, and it may require services from other packages.

In absence of pre-defined APIs at package level, we assume that the relationships of a package p to other packages form two sets of p classes. Those classes of p that play the role of p *interfaces* to the rest of the software system: p classes that have either incoming dependencies from classes outside p or outgoing dependencies pointing to classes outside p.

1) In-Interfaces: for a package p, the *in-interfaces* are the p classes that have incoming use or/and extend dependencies from p *clients* (*i.e.*, from classes belonging to other packages). The p in-interfaces via *use* dependencies represent the services that p provides to the rest of the software system. The p in-interfaces via *extend* dependencies represent the p's services (*abstract* services) that other packages extend (*implement*).

2) Out-Interfaces: for a package p, the *out-interfaces* are the p classes that have dependencies pointing to p *providers* (*i.e.*, pointing to classes belonging to other packages). The p out-interfaces via *use* dependencies represent the p classes that require services from the rest of the software system. They represent the requirements that p needs to fulfill its services. The p out-interfaces via *extend* dependencies represent the p's implementations of abstract services declared in other packages.

C. Modularization, Packages, Classes and Interfaces

We define a Modularization of an object-oriented software system by $\mathcal{M} =< \mathcal{P}, \mathcal{D} >$. $\mathcal{P}$ is the set of all packages and $\mathcal{D}$ is the set of pairwise dependencies among the packages: $\mathcal{D} \subseteq \mathcal{P} \times \mathcal{P}$. Packages are the containers of classes: each package $p \in \mathcal{P}$ involves a set of classes $C(p) \subseteq \mathcal{C}$: $\mathcal{C}$ is the set of all classes. Every class c belongs to only one package $p(c)$.

The classes of a package p that have dependencies to classes outside p represent the interfaces of p $Int(p) \subseteq C(p)$. Formally, $\mathcal{I} \subseteq \mathcal{C}$: $\mathcal{I}$ is the set of all interfaces. The interfaces of a package p are either in-interfaces $InInt(p)$ relating p to its client packages $Clients_p(p) \subset \mathcal{P}$, or out-interfaces $OutInt(p)$ relating p to its provider packages $Providers_p(p) \subset \mathcal{P}$: $Int(p) = InInt(p) \cup OutInt(p)$. Taking liberties with the notation $Clients_p(p)$, we use $Clients_p(c)$ to denote the set of all packages containing classes that depend upon c. Similarly, we use the notation $Providers_p(c)$ to denote the set of all packages containing classes that c depends upon them. We also use the notation $Clients_c(p)$ to denote the set of all classes outside p that depend upon classes inside p. Similarly, we use the notation $Providers_c(p)$ to denote the set of all classes outside p that c depends upon them.

D. Dependencies Notation

In this section we define the notations of different types of dependencies that we use in this paper. We define the set of dependencies by $\mathcal{D} = \{Uses \cup Extends\}$. According to this, we define the notations of dependencies as follows:

Extend dependencies. For two classes c_1 and c_2 we define the predicate $Ext(c_1, c_2)$ that is true if c_1 extends c_2. For convenience, we use the same predicate at package level: $Ext(p_1, p_2)$ is true if p_1 extends p_2. We also use the one-argument version of the Ext predicate, as in $Ext(c)$, to denote the set of all classes directly extended by the class c. Similarly we use this version at package level, as in $Ext(p)$, to denote the set of all packages extended by the package p.

Use dependencies. For two classes c_1 and c_2 we define the predicate $Uses(c_1, c_2)$ that is true if c_1 uses c_2. For convenience, we use the same predicate at package level: $Uses(p_1, p_2)$ is true if p_1 uses p_2. We also use the one-argument version of the $Uses$ predicate, as in $Uses(c)$, to denote the set of all classes used by the class c. Similarly, we use this version at package level, as in in $Uses(p)$, to denote the set of all packages used by the package p.

IV. Coupling Metrics: Metrics related to Information-Hiding and Changeability Principles

A. Index of Inter-Package Interaction

In this section we want to measure to which extent packages hide inter-class communication. This is by answering the following questions: to which extent classes belonging to different packages are *not* dependent on each other?

The goal of this section is to provide metrics that address the Hiding-Information principle explained in Section II-A. We define 2 similar metrics, $IIPU$ (Index of Inter-Package Usage) and $IIPE$ (Index of Inter-Package Extending): one dealing with $Uses$ and the other with $Extends$.

1) Index of Inter-Package Usage: As defined in Section III, let $\mathcal{C}$ and $\mathcal{P}$ denote respectively the set of all classes and the set of all packages. Let $UsesSum(\mathcal{C})$ be the sum of all use dependencies among the classes $\mathcal{C}$, and let $UsesSum(\mathcal{P})$ be the sum of all the use dependencies among the packages $\mathcal{P}$:

$$UsesSum(\mathcal{C}) = \sum_{c_i \in \mathcal{C}} |Uses(c_i)|$$

$$ExternalUses(c) = \{x \in \mathcal{C} | Uses(c, x) \& p(x) \neq p(c)\}$$

$$UsesSum(\mathcal{P}) = \sum_{p_j \in \mathcal{P}} \sum_{c_i \in C(p_j)} |ExternalUses(c_i)|$$

$$IIPU(\mathcal{M}) = 1 - \frac{UsesSum(\mathcal{P})}{UsesSum(\mathcal{C})} \quad (1)$$

Interpretation. IIPU is the index of inter-package usage within a modularization $\mathcal{M}$. It takes its value in the range [0,1] where 1 is the optimal value and 0 is the worst value: the greater value IIPU has, the smallest inter-package usage the modularization has. IIPU provides an index about the extent to which packages hide the actual inter-class usage. It is an indicator to the degree of collaboration among classes belonging to same packages.

2) Index of Inter-Package Extending: We define a similar metric to $IIPU$ but from the extending interactions standpoint. Let $ExtSum(\mathcal{C})$ be the sum of all inheritance dependencies among the classes $\mathcal{C}$, and let $ExtSum(\mathcal{P})$ be the sum of all inheritance dependencies among the packages $\mathcal{P}$: *i.e.*, the sum of all the inheritance dependencies among classes belonging to different packages.

$$ExtSum(\mathcal{C}) = \sum_{c_i \in \mathcal{C}} |Ext(c_i)|$$

$$ExternalExt(c) = \{x \in \mathcal{C} | Ext(c, x) \& p(x) \neq p(c)\}$$

$$ExtSum(\mathcal{P}) = \sum_{p_j \in \mathcal{P}} \sum_{c_i \in C(p_j)} |ExternalExt(c_i)|$$

$$IIPE(\mathcal{M}) = 1 - \frac{ExtSum(\mathcal{P})}{ExtSum(\mathcal{C})} \quad (2)$$

Interpretation. IIPE is the index of inter-package extending within a modularization $\mathcal{M}$. It takes its value in the range [0,1] where 1 is the optimal value and 0 is the worst value: the greater value IIPE has, the smallest number of inter-package inheritance dependencies the modularization has. IIPE provides an index about the extent to which class hierarchies are well organized into packages. It is an indicator to the degree of concreteness of packages (*i.e.*, concreteness of services that packages provide). As example, let the value of IIPE for a given modularization be 0.5, which mean that 50% of inheritance dependencies are among classes belonging to different packages. This could be interpreted as follows:

- The software system is mainly plugins-based, where the core software (some packages) declares abstract services that are implemented by other packages (plugins).
- The software system contains some packages that play as the root of large number of other packages. In this case some packages declare abstract services that are implemented by a large number of other packages. In this way, we can say that a large number of packages are similar from the point of view of interfaces they implement.
- Finally, it may simply mean that the class hierarchies are not well organized into packages. In this case the modularization need a revision.

B. Index of Package Changing Impact

In Section IV-A we defined measurements that characterize inter-package coupling/interactions based on inter-package dependencies. In this section we want to provide other measurements that complement those defined in Section IV-A by answering the following: in the context of a given modularization $\mathcal{M}$, to which extent $\mathcal{M}$ packages are inter-connected? to which extent modifying a package within $\mathcal{M}$ may impact other packages?

We want to define metrics characterize, at package level, the maintainability of $\mathcal{M}$ (ref. Section II-B). We believe that reducing inter-package dependencies, if it does not take into account the number of inter-dependent packages, may negatively affect the modularization maintainability. By example, Fig. 1 shows the package p has 7 dependencies coming from classes belonging to one package; while the package q has 3 incoming dependencies coming from classes belonging to 3 distinct packages. In such a case, and at package granularity, maintaining/modefying p may require an impact analysis to one package (p_1), while maintaining q may require an impact analysis to 3 packages (q_1, q_2 and q_3). Therefore, from the point of view of impact localization, q is harder to be maintained than p.

As stated in Section III, let $Clients_p(p)$ denotes the set of all packages that depend on p; let $\mathcal{P}$ denotes the set of all packages in the modularization $\mathcal{M}$. According to what we stated above, we define the index of package changing impact as follows:

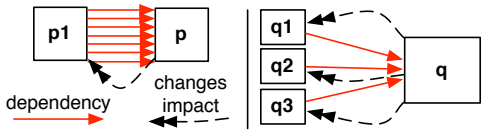


Fig. 1. Explanation for Package Changing Impact

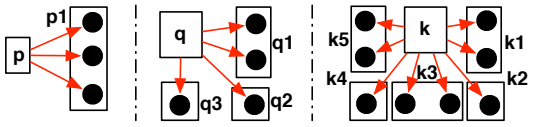


Fig. 2. Explanation for Package Usage Diversion

$$IPCI(p) = 1 - \frac{Clients_p(p)}{1 - |\mathcal{P}|} \quad (3)$$

$$IPCI(\mathcal{M}) = \frac{\sum_{p_i \in \mathcal{P}} IPCI(p_i)}{|\mathcal{P}|}$$

Interpretation. IPCI takes its value between 0 and 1, where 1 is the optimal value and 0 is the worst value. For a package p in a modularization $\mathcal{M}$, a $IPCI(p)$ value of 0 indicates that all packages in $\mathcal{M}$ are dependent on p. As a consequence, any changes on p may impact the whole modularization. In the context of the whole modularization, the $IPCI(\mathcal{M})$ value indicates the extent to which $\mathcal{M}$ is free for changes: *i.e.*, the index to which $\mathcal{M}$ packages are not inter-dependent.

C. Index of Package Communication Diversion

In this section we define metrics that measure the extent to which package communication (Section IV-A) is focused or diverted. Our vision of package communication diversion can be explained as follows: let p be a package uses 5 classes packaged into 5 different packages, and let q be a package uses 5 classes packaged in one package. In such a case we say that p communication is completely diverted, while q communication is completely focused. This is because p communication starts out with maximal number of coupling paths (5 provider classes cause 5 different coupling paths via 5 different provider packages), while q starts out with minimal coupling paths (one coupling path via one provider package).

1) Index of Inter-Package Usage Diversion: we define inter-package communication diversion on the usage dependencies. Let $Uses(p)$ denotes the packages that p uses; let $Uses_c(p)$ denotes the classes that p uses; and let 1 be the minimal number of coupling paths that p may start with. Then we define index of inter-package usage diversion as follows:

$$PUF(p) = \frac{1}{|Uses(p)|}$$

$$IIPUD(p) = \begin{cases} PUF(p) \times (1 - \frac{1-|Uses(p)|}{|Uses_c(p)|}) \\ \\ 1 \quad : \quad |Uses_c(p)| = 0 \end{cases}$$

$$IIPUD(\mathcal{M}) = \frac{\sum_{p_i \in \mathcal{P}} IIPUD(p_i)}{|\mathcal{P}|} \quad (4)$$

Interpretation. IIPUD is the index of inter-package usage diversion and PUF is package usage factor. We used this factor to distinct packages that use a large number of packages from packages that use a small number.

The IIPUD(p) value ranges from 0 to 1. A IIPUD(p) value of 1 indicates that p communication diversion is minimal: as shown in Fig. 2, p starts out with only one coupling path, where it uses only one package p_1. It mean that p requires services from only one package, thus it requires particular, non-dispersed, functionalities. Otherwise, the smallest value the IIPUD(p) has, the largest diversion of usage communication p has. We assume by our definition of PUF(p) and IIPUD(p) the following: if a given package uses a large number of packages it will has a worst IIPUD(p) value than another package that uses a smaller number of packages, this is even if the term $\frac{1-|Uses(p)|}{Uses_c(p)}$ has the same value for both packages. As example, Fig. 2 shows that q uses 4 classes distributed over 3 packages, and shows that k uses 8 classes distributed over 5 packages. In such a case, $\frac{1-|Uses(q)|}{Uses_c(q)} = \frac{1-|Uses(k)|}{Uses_c(k)} = 0.5$. But the IIPUD($q$) value ($\frac{0.5}{3}$) is better than the IIPUD(k) value ($\frac{0.5}{5}$) –Since q uses a smaller number of packages than k.

For a given modularization $\mathcal{M}$, a max IIPUD($\mathcal{M}$) value of 1 indicates an ideal focusing of package usage communication: each package in $\mathcal{M}$ uses, at maximum, only one package. Otherwise, as the value of IIPUD($\mathcal{M}$) decreases, the diversion of usage communication between packages increases. This can be an indicator to the following:

- A large number of packages require services that are dispersed over distinct packages. In such a case, the schema of usage communication paths is characterized as complex. Thus, a revision to $\mathcal{M}$ is required.
- Some packages are characterized by very small value of IIPUD(p). In such a case, to minimize the schema's complexity of usage communication paths, the maintainers may start by focusing on those packages.

2) Index of Inter-Package Extending Diversion: we define the index of package extending diversion (IIPED) similarly to IIPUD defined above, but with regard to extending dependencies. Let $Ext(p)$ denotes the packages that p extends; let $Ext_c(p)$ denote the classes that p extends; and let 1 be the minimal number of coupling paths that p may start with.

$$PEF(p) = \frac{1}{|Ext(p)|}$$

$$IIPED(p) = \begin{cases} PEF(p) \times (1 - \frac{1-|Ext(p)|}{|Ext_c(p)|}) \\ \\ 1 \quad : \quad |Ext_c(p)| = 0 \end{cases}$$

$$IIPED(\mathcal{M}) = \frac{\sum_{p_i \in \mathcal{P}} IIPED(p_i)}{|\mathcal{P}|} \quad (5)$$

Interpretation. The interpretation of IIPED(p) is similar to what we stated above for IIPUD(p) in Section IV-C1. Note that IIPED is defined with regards to extending dependencies rather than usage dependencies. IIPED also takes its value between 1 and 0, where 1 is the optimal value and 0 is the worst value. When the value of IIPED(p) goes closer to 0 is an indicator that p extends a relatively big number of classes that are distributed over distinct packages. This could mean that p plays the role of a plugin of a big number of packages. It also indicates that p implement interfaces that are completely not similar from the point of view of their providers. As summary, p is may expected to provide complex service(s). Take as example a package p that extends 10 classes belonging to 10 different packages, thus IIPED(p) = 0.01. In such a case, p is a plugin for 10 packages and it requires a particular attention.

V. Cohesion Metrics: Metrics related to Commonality-of-Goal Principle

A. Index of Package Goal Focus

In this section we assume that a package, in its ideal state, should focus on providing *one* well identified service to the rest of the software system. What do we mean by focused service? and how to characterize such an aspect?

From the point of view of the package role, we say that a package provides a focused service if it plays the same role with all its client packages. In other words, for a package p, we say that p services are focused if they are always used together by every client package to p. In such a case, for an ideal situation, the p in-interfaces are always used together, so they represent a single composite service provided by p to the rest of the system. In this way, we say that p is focused (*i.e.*, the p goal is focused). Otherwise, where the p in-interfaces are used via relatively small portion per client package, p is then not focused: p plays different roles with its clients.

Let $Req(x, c)$ be true if x uses or extends c; let $InInt(p,q)$ denotes the set of p in-interfaces required by q; and let $Role(p, q)$ denotes the role that p plays with its client q. We define then the Focus of a package p as the average of p roles with respect to all p clients:

$$InInt(p, q) = \{c \in InInt(p) | \exists x \in q : Req(x, c)\}$$

$$Role(p, q) = \frac{InInt(p, q)}{InInt(p)} : q \in Clients(p)$$

$$PF(p) = \frac{\sum_{p_i \in Clients_p(p)} Role(p, p_i)}{|Clients_p(p)|} \quad (6)$$

$$PF(\mathcal{M}) = \frac{\sum_{p_i \in \mathcal{P}} PF(p_i)}{|\mathcal{P}|}$$

Interpretation. PF(p) always takes its value between 0 and 1, where 1 is the optimal value and 0 is the worst value. The largest value PF(p) has, the highest frequency of requiring largest portion of p in-interfaces. Ideally, when the package in-interfaces are always used together by every client package to that package, as for p in Fig. 3, PF(p) takes then a max value of 1. The goal of PF(p) is to provide one answer for both following questions: (1) to which extent p services are required together? (2) to which frequency p clients require all the p services?

To better understand the behavior of PF(p) we take 3 examples (the cases of q, k and y in Fig. 3):

1) Fig. 3 shows that the package q exposes 5 in-inerfaces to 4 clients q_1, q_2, q_3 and q_4; where each of them uses only 2 classes of q in-interfaces. In this case, $PF(q) = \frac{2}{5}$
2) Fig. 3 shows that the package k also exposes 5 in-interfaces to 4 clients k_1, k_2, k_3 and k_4; where k_4 uses 4 classes of k in-interfaces, while each of other clients uses only 2 classes. In this case, $PF(q) < PF(k) = \frac{1}{2}$
3) Finally, Fig. 3 shows that the package y also exposes 5 in-interfaces to 4 clients: y_1, y_2, y_3 and y_4; where each of y_2, k_3, k_4 uses 3 classes of y in-interfaces, while y_1 uses only 2 classes. In this case, $PF(k) < PF(y) = \frac{11}{20}$

As summary, when the value of PF(p) decreases, we expect that p clients *frequently* require relatively-small sets of p services. For a given modularization $\mathcal{M}$, PF($\mathcal{M}$) also takes its value from 0 to 1, where 1 is the optimal value and 0 is the worst value. When PF($\mathcal{M}$) decreases, we say that the definition of package roles within $\mathcal{M}$ gets worse.

B. Index of Package Services Cohesion

Unlike what we stated above in Section V-A, in this section we assume that a package may be expected to provide several services. Therefore, we want to address the following questions: what if the purpose of a package p is to play distinct roles with regard to its clients? in this case, to which extent p services are cohesive with regard to their *common use*?

In absence of pre-defined APIs at package level that declare explicitly which services a package provides to the rest of the software, we assume the following: since each client package q to a package p represents a requirement to a subset of p in-interfaces, we define such a subset as a *composite service* $CS(p, q)$ provided by p to q:

$$CS(p, q) = \{int \in InInt(p) | int \in Provider_c(q)\}$$

According to this definition, we say that two composite services of p, $CS(p, q)$ and $CS(p, k)$: $q, k \in Clients_p(p)$, are identical if both represent the same group of classes. In this way, we measure the cohesiveness for a composite service by measuring the similarity of purpose of the service classes: to which extent the service classes are required together?

For example, let α be a composite service presented by 3 classes $\{c_1, c_2, c_3\}$, and suppose that either these classes are always required together or there is no subset of these classes used apart. In this case we say that α is fully cohesive from similarity of purpose perspective. Another example, let β be also a composite service of p to q, $CS(p, q)$. Suppose that each class of β classes is always required, by other client packages than q, aside from other ones. In this case we say that β is fully segregated from similarity of purpose perspective.

Let $\lambda_{q,k}$ denotes the set of classes results from the intersection of 2 composite services of a package p: $\lambda_{q,k} = |CS(p, q) \cap CS(p, k)|$. Let $SP_k(p, q)$ a measurement of the similarity of purpose for a composite service $CS(p, q)$ with regard to another composite service $CS(p, k)$:

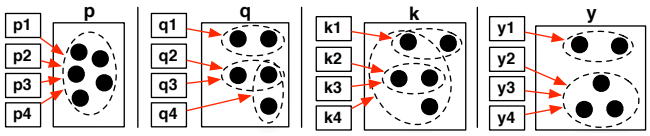
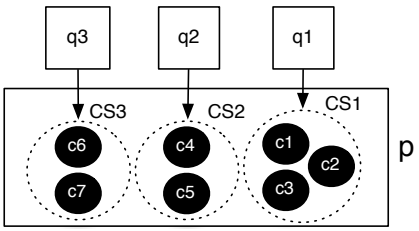
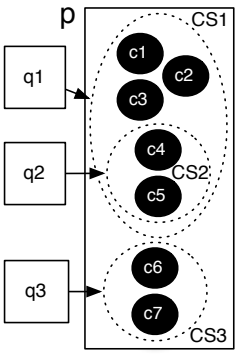


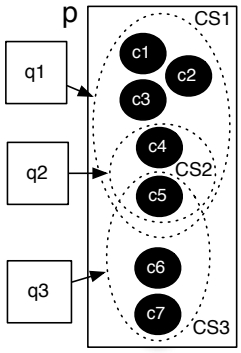
Fig. 3. Explanation for Package Focus



(a) p provides 3 composite services: $CS_1 = \{c_1, c_2, c_3\}, CS_2 = \{c_4, c_5\}$ and $CS_3 = \{c_6, c_7\}$.



(b) CS_1 is changed from (a): $CS_1 = \{c_1, c_2, c_3, c_4, c_5\}$.



(c) CS_3 is changed from (b): $CS_3 = \{c_5, c_6, c_7\}$.

Fig. 4. Explanation for Package Services Cohesion.

$$SP_k(p,q) = \begin{cases} \frac{|\lambda_{q,k}|}{|CS(p,q)|} & |\lambda_{q,k}| \neq 0 \\ 1 & Else \end{cases}$$

The similarity of purpose for a composite service α with respect to another one β is given by: the relative size of the subset of in-interfaces that are shared in both services with respect to the size of the α classes set. The largest set of α classes is involved in β, the highest value of similarity that α has with regard to β. $SP_k(p,q)$ always takes its value between 0 and 1, where 1 is the optimal value and 0 is the worst value. If there is no classes shared between α and β, then we say that β does not affect the similarity of purpose of α.

According to what we stated above, we define the cohesion of a composite service by the average of its similarity of purpose with regard to all p's clients. We define then the index of package services cohesion, for a package p, by the average of cohesion for all the composite services that p provides:

$$CS_{cohesion}(p,q) = \frac{\sum_{k_i \in Clients_p(p)} SP_{k_i}(p,q)}{|Clients_p(p)|}$$

$$IPSC(p) = \frac{\sum_{q_i \in Clients_p(p)} CS_{cohesion}(p,q_i)}{|Clients_p(p)|} \quad (7)$$

$$IPSC(\mathcal{M}) = \frac{\sum_{p_i \in \mathcal{P}} IPSC(p_i)}{|\mathcal{P}|}$$

Interpretation. $CS_{cohesion}(p,q)$ takes its value between 0 and 1, where a value of 1 indicates that $CS(p,q)$ is completely cohesive, while a value of 0 indicates that $CS(p,q)$ is completely segregated. IPSC(p) and IPSC($\mathcal{M}$) both take their value between 0 and 1, where 1 is the optimal value and 0 is the worst value. Fig. 4 shows a package p in 3 different cases from the perspective of *common usage* of p services. In the 3 cases, p provides 7 classes $\{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$ to 3 client packages $\{q_1, q_2, q_3\}$. The figure also shows that p provides 3 different composite services: CS_1, CS_2 and CS_3.

- In Fig. 4(a), the classes of any composite service CS of p are always required together: none of the CS classes is used aside from the other classes of CS. Therefore, we say that similarity of purpose for the CS classes is very well defined: all the classes that are in a CS have the same purpose, which is providing services to the same group of client packages. Thus, $CS_{i_{cohesion}} = 1$. In this case, the value of IPSC for p is maximal: $IPSC(p) = 1$.
- In Fig. 4(b), the difference from Fig. 4(a) is that a small subset $\{c_4, c_5\}$ of CS_1 classes has also another purpose, which is providing services to q_2. Therefore, the similarity of purpose for the CS_1 classes is not well defined: $CS_{1_{cohesion}} = \frac{4}{5}$. In this case, the $IPSC(p)$ value is smaller than in the previous case (a): $IPSC(p) = \frac{14}{15}$.
- In Fig. 4(c), the difference from Fig. 4(b) is that the subset $\{c_5\}$ of CS_2 classes has also another purpose, which is providing services to q_3. This negatively affects the similarity of purpose for both CS_3 and CS_2, since $\{c_5\}$ is currently a subset of CS_3 also: $CS_{2cohesion} = \frac{2}{3}$ and $CS_{3_{cohesion}} = \frac{2}{3}$. In this case, the $IPSC(p)$ value $\frac{32}{45}$ is smaller than in the previous case (b).

VI. Validation

In this section we provide a theoretical validation of our coupling and cohesion metrics. This is by showing that our metrics satisfy all the mathematical properties that are defined by Briand *et al.* [8], [9].

A. Coupling Metrics Validation

The widely known properties to be obeyed by a coupling metric are: *Non Negativity*, *Monotonicity* and *Merging of Modules*. The following of this section shows how our coupling metrics ($IIPU$, $IIPE$, $IPCI$, $IIPUD$ and $IIPED$) satisfy these properties.

Non Negativity property: according to this property, for any given software modularization $\mathcal{M}$, the coupling metric value for $\mathcal{M}$ should be greater than 0. According to what we discussed in Section IV, all our coupling metrics take their value between 0 and 1, where 0 is the worst value.

Monotonicity property: this property assumes that adding additional interactions to a module cannot decrease its coupling. To check this property: let p be a package in a given modularization $\mathcal{M}$, that has d dependencies pointing to or/and coming from n packages. Now let p' in $\mathcal{M}'$ be the same package than p but with one additional dependency $(d+1)$ pointing to one additional client/provider package

$(n+1)$. In this case, all the following conditions are true: $IIPU(\mathcal{M}) \geqslant IIPU(\mathcal{M}')$ ($IIPE(\mathcal{M}) \geqslant IIPE(\mathcal{M}')$) ; $IPCI(\mathcal{M}) \geqslant IIPCI(\mathcal{M}')$; $IIPUD(\mathcal{M}) \geqslant IIPUD(\mathcal{M}')$ ($IIPED(\mathcal{M}) \geqslant IIPED(\mathcal{M}')$). This means that all our coupling metrics satisfy the monotonicity property.

Merging-of-Modules property: this property assumes that the sum of the couplings of two modules is not less than the coupling of the module which is composed of the data declarations of the two modules. To check this property, let p and q be two packages in $\mathcal{M}$, that have respectively n and m dependencies pointing to or/and coming from x and y packages. Now, let k be the merging of p and q (*i.e.*, k contains only the classes of both packages), and let $\mathcal{M}'$ be the resulting modularization after the merging. In this case, the sum of the dependencies that k have with other packages N cannot be greater than $n+m$ ($N \leqslant n+m$). Similarly, the number of the k client and provider packages R cannot be greater than $x+y$ ($R \leqslant x+y$). In this case, any of our coupling metrics will indicate that the coupling in $\mathcal{M}'$ is less than (or equal to) the coupling in $\mathcal{M}$. As consequence, all our coupling metrics satisfy the merging-of-modules property.

B. Cohesion Metrics Validation

The widely known properties to be obeyed by a cohesion metric are: *Normalization*, *Monotonicity* and *Cohesive Modules*. The following of this section shows how our cohesion metrics (PF and $IPSC$) satisfy these properties.

Normalization property: this property assumes that the value of a cohesion metric should belongs to a specified interval $[0, Max]$. As explained in Section V, our cohesion metrics are normalized and take their value in the interval $[0,1]$. Therefore, our cohesion metrics satisfy this property.

Monotonicity property: this property assumes that adding cohesive interactions to a module/modularization cannot decrease its cohesion. To check this property, let p be a package in a given modularization $\mathcal{M}$. Supposing that we add to p a new class c, where c is always used by other packages in $\mathcal{M}'$ together with a non-empty set of p' in-interfaces, and it is never used aside from that set: p' and $\mathcal{M}'$ are respectively the resulting package and modularization after adding c to p. In this case, the value of both $PF(p')$ and $IPSC(p')$ metrics cannot be smaller than their values for p. In this way, our cohesion metrics satisfy the monotonicity property.

Cohesive-Modules property: this property assumes the following: if p_1 and p_2 are cohesive packages in $\mathcal{M}$, but there is no cohesive relationships between p_1 classes and p_2 classes, then merging p_1 and p_2 into one package q in $\mathcal{M}'$ should not increase the modularization cohesion. To check these property, we suppose that none of the p_1 in-interfaces is required by packages require p_2 in-interfaces. In this case, the value of $PF(q)$ cannot be greater than $PF(p_1)$ value nor than $PF(p_2)$ value. Thus, $PF(\mathcal{M}')$ value cannot be greater than $PF(\mathcal{M})$. As a consequence, PF satisfies the cohesive-modules property.
In the same context, since none of the p_1 in-interfaces is required by packages require p_2 in-interfaces, the composite services CSs of q are exactly those of p_1 and p_2 and their cohesion values still the same. On another hand, the number of the q's client packages is equal to the sum of the p_1 client packages and the p_2 client packages. Thus the $IPSC(q)$ value cannot be greater than the $IPSC(p_1)$ value nor the $IPSC(p_2)$ value. In this way, $IPSC(\mathcal{M}')$ value cannot be greater than $IPSC(\mathcal{M})$. As a consequence, $IPSC$ also satisfies the cohesive-modules property.

VII. Discussion

In this section, we discuss our metrics with regard to the modularity principles we underlined in Section II.

A. Assessing Package Encapsulation

The goal of the $IIPU$ and $IIPE$ metrics is measuring the extent to which packages hide inter-class communication. They measure the extent to which packages encapsulate system complexity at class granularity, where this last is given by the frequency of inter-class interactions. According to Callebaut *et al.* [11], where they suppose that: *"the frequencies of interaction among elements in any particular subsystem of a system should be two times greater than the frequencies of interaction between the subsystems"*. From this point of view, we defined our metrics to assess packages encapsulation within a given modularization by the ratio of inter-package interactions to all interactions at class granularity.
From $IIPU$ perspective, for a given modularization $\mathcal{M}$, if all the method call interactions are among classes belonging to different packages, thus they all represent inter-package interactions. In this case, packages encapsulation of inter-class usage is at the worst level, where $IIPU(\mathcal{M}) = 0$.

As complementary metrics to $IIPU$ and $IIPE$, we defined $IIPUD$ and $IIPUE$ that measure to which extent the interactions of a package p are spread over other packages. It is worth to note that other aspects can also participate in assessing package encapsulation, such as: the relative number of in-interfaces that a package exposes (*i.e.*, $\frac{InInt(p)}{C(p)}$). At method granularity, the relative number of methods used outside their classes' package can also be an indicator for package encapsulation quality.

B. Assessing Package Changeability

The goal of the $IPCI$ metric is assessing package changeability from the standpoint of the localization of changes impact. Our standpoint is that changing a package may directly impacts other packages depending on the changed package. According to this, the $IPCI(p)$ metric assesses p changeability with regards to the p clients packages. We defined $IPCI(p)$ as a ratio to the number of all packages within the modularization to give a measurement relative to the context of the given software: *i.e.*, let p and q be two packages within the modularizations $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively; where $\mathcal{M}_1$ consists of 1000 packages and $\mathcal{M}_2$ consists of 20 packages; suppose that p and q have the same number (*e.g.*, 10) of client packages; in such a context, the impact of changing q on $\mathcal{M}_2$ is greatly larger than the impact of changing p on $\mathcal{M}_1$.

C. Assessing Package Role and Reusability

To characterize the role of a given package p with within its modularization $\mathcal{M}$ and to assess p reusability, we defined the metrics PF and $IPSC$. On the first hand, the goal of $PF(p)$ is to provide us with answers to the following questions: (1) does p provide one service to the rest of the rest of the software? (2) to which extent p classes are used together by the rest of the software?
On the other hand, if p provides multiple services, the goal of $IPSC(p)$ is to measure the cohesiveness of p services from the *commonality-of-goal vs. similarity-of-purposes* perspectives Section II-C. $IPSC(p)$ provides us with answers to the following questions: (3) to which extent p is a provider of well-identified (particular) services to the software system? (4) to which extent p is a provider of utility (general) services to the software?
Our standpoint is that if p services are used together in an identifiable way, then it is easier to understand the goal, the scope and the purpose of p services than if p services are used together in an non-identifiable (arbitrary) ways. In this last case, understanding the p services requires an understanding of each p in-interface aside from others.

VIII. Relevant Related Works vs. Our Metrics

To cope with software system complexity, Parnas *et al.* [32] have introduced the idea of decomposing software systems with the intention of increasing module cohesion and minimizing inter-module coupling. Since then, many metrics have been defined to compute the cohesion and coupling of a module, where module concept is usually used to represent a composite software entity (*e.g.*, a class or a package).

A large body of previous works on Object-Oriented software metrics is mainly focused on the issue of characterizing the class design, either looking at class internal complexity or relationships between a given class and other classes [12], [18], [28], [27], [25], [6], [7], [10], [8], [9], [15]. Some of these works characterize a class by counting its internal components, such as counting the number of methods and the number of attributes. Others characterize a class by looking at its relationships with other classes, as for the coupling between objects (CBO), or characterize the class cohesion with regard to the similarity between pairs of methods and pairs of attribute types in the given class. Few number of these previous works provide metrics that do not characterize a single class, such as metrics measure the depth of the inheritance tree in a software.

In the literature, there is also a body of work that focus on object-oriented metrics from the standpoint of their correlation with software changeability [26], or from the standpoint of their ability to predicate software maintability [5], [14]. Other researchers argue that the measures resulted by the cohesion and coupling metrics of the previous works cited above are open to interpretation [26], [9]. This is due to polymorphic method calls, where it is difficult to capture through static analysis which method is actually being called for execution.

In general, there are few metrics in the literature devoted to packages. In the following we present those metrics according to their perspectives: either *Cohesion* or *Coupling* perspective.

A. Cohesion Metrics

Emerson presents a metric to compute cohesion applicable to modules in the sense of Pascal procedures [20]. His metric is based on a graph theoretic property that quantifies the relationship between control flow paths and references to variables. Patel *et al.* [33] compute the cohesion of Ada packages based on the similarity of their members (programs). The idea is to measure cohesion based on subprograms similarity. They use the keywords shared between subprograms. They consider only the specification of the package, not the keywords present in the body, which are invisible from outside the package. Similarly, Allen and Khoshgoftaar define information theory-based (as opposed to counting) coupling and cohesion measures for subsystems [4]. Their measures are applied to modules, which are represented as graphs. They define cohesion in terms of the similarity between the objects of the concerned modules. However these approaches do not take into account classes and the relationships they cause inter-packages and/or intra-package.

Misic adopts a different perspective and measures the cohesion of a package as an external property [31]. He claims that the internal organization of a package is not enough to determine its cohesion. Similarly, Ponisio *et al.* introduce the notion of use cohesion (or conceptual cohesion) [34]. They measure the cohesion of a package considering the usage of the package classes from the client packages. Their cohesion metric does not take into account the explicit dependencies among the package classes (*e.g.*, *method call*).

Recently, Martin proposed the *Rational Cohesion* metric. It is defined as the average number of package internal dependencies per class. Martin's cohesion metric measures the connectivity among the internal classes of a given package, regardless the amount of dependencies that the package classes have with external classes.

Finally, Sarkar *et al.* proposes an API-based cohesion metric [35]. They define the APIU metric that measures the extent to which a *service*-API is cohesive, and the extent to which it is segregated from other service-APIs. This is from the common usage point of view. However, their metric is API based and apply that each module (*i.e.*, package) explicitly declares its service-APIs. Otherwise, the metric is not applicable.

Our Cohesion metrics. The $IPSC$ cohesion metric we provide is similar to certain extent to the $APIU$ metric provided by Sarkar *et al.* [35], but it is not API-based. In addition, we provide a new cohesion metric (PF) with the aim to measure the extent to which a package plays a consistent role with regard to its usage by its client packages. The standpoint is that, ideally, a package should focus to provide one service for other packages. Otherwise, where a package provides more than one service, we provide the $IPSC$ metric that measures the cohesiveness of package services from the *similarity-of-purpose* perspective.

B. Coupling Metrics

Martin [30] defines two kinds of package coupling: *efferent coupling* (C_e) and *afferent coupling* (C_a). The C_e is to assess the coupling degree between a package p and its *provider* packages. While the C_a is to assess the coupling degree between p and its *client* packages.. He defines the C_e metric for a package p as the number of p's provider classes, and defines the C_a metric as the number of p's client classes. Recently, in 2005 [29], he redefines these metrics: p's C_e is the number of p's provider packages, while p's C_e is the number of p's client packages. However, these coupling metrics do not take in consideration the context of the package modularization. Hautus addresses cyclic package coupling [24]. He proposes a tool to analyze the structure of Java programs and a metric that indicates the percentage of changes to make a package structure acyclic.

Finally, Sarkar *et al.* propose coupling metrics [35]. First of all, they propose API-based coupling metric (MII) that calculates how frequently the methods listed in a module's APIs are called by the other modules. Then they assume that modules may also interact with each other by calling methods that are not listed in the APIs of the modules. Therefore, they provide another metric (NC) that measures, for a given module, the disparity between the declared API methods and the methods that are actually participating in intermodule call traffic. However, both metrics are not applicable when modules are not API-based. In the same paper, Sarkar *et al.* propose also the following coupling metrics: (1) The IC metric, to measure *inheritance-based intermodule coupling*; (2) The AC metric, to measure *intermodule association-induced coupling*. IC and AC, are defined in the same way, but with regard to $Uses$ and $Extends$ dependencies, respectively. For a package p, the value of AC (IC) is given by the smaller value among the following: the number of p's client classes, the number of p's client packages, or the number of p's out-interfaces. In this way, they do not take care about the evidence indicates that the number of p's client packages is surely not bigger than the number of p's client classes. Also, they also do not provide us with the rationale beyond their definition, nor with an interpretation of their metrics.

Our Coupling metrics. They are not API-based and characterize three different aspects of inter-package coupling within a given modularization. First of all, we provided metrics ($IIPU$ and $IIPE$) that measure the extent to which packages follow the hiding-information principle, with regard to inter-package communication. Then, we provided other metrics ($IIPUD$ and $IIPUE$) which measure the extent to which the package communication is focused or dispersed. Finally, we provided a metric ($IPCI$) measures the package changing impact: it measures the extent to which a package modification impacts the whole software modularization.

IX. Conclusion and Future Works

In this paper, we tackled the problem of assessing modularizations for not API-based object-oriented software systems. We defined a complementary set of coupling and cohesion metrics that assess packages organization in large legacy object-oriented software. While designing our metrics, we addressed some modularity principles related to packages *encapsulation*, *changeability* and *reusability*. In addition, we defined metrics characterizing packages role within a given modularization. We defined our metrics with regard to two different types of object-oriented inter-class dependencies: *method call* and *inheritance* relationships. We also provided our metrics with exhaustive interpretations for both types of dependencies.

We successfully showed that all our metrics satisfy the mathematical properties that cohesion and coupling metrics should follow.

We plan to investigate our metrics on real large software systems and validate their utility with independent software maintainers.

References

[1] H. Abdeen, S. Ducasse, D. Pollet, and I. Alloui. Package fingerprint: a visual summary of package interfaces and relationships. *Info. and Sof. Tech.*, 52:1312–1330, 2010.

[2] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui. Automatic package coupling and cycle minimization. In *Int. Work. Conf. on Rev. Eng.*, pages 103–112. IEEE Computer Society Press, 2009.

[3] F. B. Abreu and M. Goulao. Coupling and cohesion as modularization drivers: are we being over-persuaded? In *Fifth Europ. Conf. on Sof. Maintenance and Reengineering*, pages 47–57, 2001.

[4] E. Allen and T. Khoshgoftaar. Measuring coupling and cohesion of software modules: An information theory approach. In *Seventh Int. Sof. Metrics Symposium*, 2001.

[5] R. K. Bandi, V. K. Vaishnavi, and D. E. Turk. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE TSE*, 29:77–87, 2003.

[6] J. Bieman and B. Kang. Cohesion and reuse in an object-oriented system. In *ACM Symposium on Software Reusability*, Apr. 1995.

[7] J. Bieman and B. Kang. Measuring design-level cohesion. *IEEE TSE*, 24(2):111–124, Feb. 1998.

[8] L. C. Briand, J. W. Daly, and J. Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[9] L. C. Briand, J. W. Daly, and J. K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE TSE*, 25(1):91–121, 1999.

[10] L. C. Briand, S. Morasca, and V. R. Basili. Defining and validating measures for object-based high-level design. *IEEE TSE*, pages 722–743, 1999.

[11] W. Callebaut and D. Gutman. *Modularity: Understanding the Development and Evolution of Natural Complex Systems*. MIT press, 05.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suit for object oriented design. *IEEE TSE*, 20:476–493, 1994.

[13] J. C. Coppick and T. J. Cheatham. Software metrics for object-oriented systems. In *ACM Conf. on Computer Science'92*, pages 317–322, 1992.

[14] M. Dagpinar and J. H. Jahnke. Predicting maintainability with object-oriented metrics - an empirical comparison. In *10th Work. Conf. on Rev. Eng.*, WCRE '03, pages 155–164. IEEE Computer Society, 2003.

[15] J. A. Dallal and L. C. Briand. An object-oriented high-level design-based class cohesion metric. *Inf. and Sof. Tech.*, 52(12):1346–1361, 2010.

[16] F. DeRemer and H. H. Kron. Programming in the large versus programming in the small. *IEEE TSE*, 2(2):80–86, 1976.

[17] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *IEEE Int. Conf. on Sof. Maint.*, pages 94–103, 07.

[18] F. B. e Abreu and R. Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Sys. Sof.*, 26:87–96, 1994.

[19] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE TSE*, 27(1):1–12, 2001.

[20] T. Emerson. A discriminant metric for module cohesion. In *ICSE*, 1984.

[21] M. Fowler. Reducing coupling. *IEEE Software*, 2001.

[22] M. D. Ghassemi and R. R. Mourant. Evaluation of coupling in the context of java interfaces. In *the conf. on OO prog., sys., lang., and app. (Addendum)*, OOPSLA '00, pages 47–48. ACM, 2000.

[23] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.

[24] E. Hautus. Improving Java software through package structure analysis. In *Int. Conf. Sof. Eng. and App.*, 2002.

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

[26] H. Kabaili, R. K. Keller, and F. Lustman. Cohesion as changeability indicator in object-oriented systems. In *Fifth Europ. Conf. on Sof. Maintenance and Reengineering*, CSMR '01, pages 39–46, Washington, DC, USA, 2001. IEEE Computer Society.

[27] W. Li. Another metric suite for object-oriented programming. *Journal of Sys. Sof.*, 44:155–162, December 1998.

[28] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Sys. Sof.*, 23:111–122, 1993.

[29] R. C. Martin. The tipping point: Stability and instability in oo design, 05. Software Development.

[30] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[31] V. B. Mišić. Cohesion is structural, coherence is functional: Different views, different measures. In *Int. Sof. Metrics Symposium*. IEEE, 2001.

[32] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, Dec. 1972.

[33] S. Patel, W. Chu, and R. Baxter. A measure for composite module cohesion. In *Int. Conf. on Sof. Eng.*, pages 38–48, 1992.

[34] L. Ponisio and O. Nierstrasz. Using context information to re-architect a system. In *3rd Sof. Measur. Europ. Forum*, pages 91–103, 06.

[35] S. Sarkar, A. C. Kark, and G. M. Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE TSE*, 34(5):700–720, 08.

[36] S. Sarkar, G. M. Rama, and A. C. Kark. Api-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE TSE*, 33(1):14–32, 07.

[37] R. Strnisa, P. Sewell, and M. Parkinson. The java module system: Core design and semantic definition. *OO Prog. Sys., Lang. and App.*, 42(10):499–514, 07.

[38] I. Sun Microsystems. Jsr-294: Improved modularity support in the java programming language. Technical report, Sun Microsystems, Inc., 08.