Software Metrics for Package Remodularisation

(Des métriques logicielles pour la remodularisation de packages)

Deliverable: 1.1 - Cutter ANR 2010 BLAN 0219 02

Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora

30 November 2011

This deliverable is available as a free download.

First Edition, January, 2009. Final Edition, March, 2010.

Deliverable: 1.1

Title: Software Metrics for Package Remodularisation

Titre: Des métriques logicielles pour la remodularisation de packages

Version: 1.0

Authors: S. Ducasse, N. Anquetil, Usman Bhatti, A. Hora

Document identification

- First Public Version: 30 November 2011

Contents

Abstract

There is a plethora of software metrics [LK94, FP96, HS96a, HK00, LM06] and a large amount of research articles. Still there is a lack for a serious and practically-oriented evaluation of metrics. Often metrics lack the property that the software reengineer or quality expert can easily understand the situation summarized by the metrics. In particular, since the exact notion of coupling and cohesion is complex, a particular focus on such point is important. In the first chapter of the present document, we present a list of software metrics, that are commonly used to measure object-oriented programs. In the second chapter we present our proposition for package metrics that capture package aspects such as information hiding and change impact limits.

Chapter1. State of the Art on Software Metrics

In this chapter we present some of the software metrics that are the foundation of most software quality models. We sort them in the following groups: Primitive metrics and Design metrics. Design metrics deal with design principles. They quantify over source code entities to assess whether a source code entity is following a design principle. In particular, such metrics can be used to track down bad design, and correcting these could lead to an overall improvement.

- Primitive metrics: simple metrics capturing some structural aspects.
- Design Metrics: Class cohesion metrics are in particular relevant for remodularisation.
- Design Metrics: Class coupling metrics are in particular relevant for remodularisation.
- Design Metrics: Package architecture. They are the first metrics taking into account the fact that a package is not just a class at another level of granularity.

1.1 Primitive Metrics

Primitive metrics target some basic aspects of source code entities (DIT, NOM), or a simple combination of other primitives (WMC, SIX) to give an abstract, comparable view of such entities. While simple to understand, their interpretation depends highly on the context, including the program, its history, the programming language used, or the development process followed by the team.

The following metrics are known as the CK metrics because Chidamber and Kemerer grouped them to define a commonly used metric suite [CK94b]:

- WMC
- DIT
- NOC
- RFC

Some other primitive metrics have been defined by Lorenz and Kidd [LK94] :

- NOM
- NIM
- NRM
- SIX

Note that we do not list LCOM here since it was heavily criticized and revised. We discuss it in the cohesion part below. In addition it should be noted that many metrics and thresholds as defined by Lorentz are unclear or do not make real sense.

Names **Depth of Inheritance Tree, Hierarchy Nesting Level**

Acronyms **DIT, HNL**

References [LK94, CK94b, BMB96, GFS05, LH93a, HM95, TNM08]

Definition The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree, measured by the number of ancestor classes. The deeper a class within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but enhance the potential reuse of inherited methods.

Scope Class

Analysis There is a clear lack of context definition. DIT does not take into account the fact that the class can be a subclass in a framework, hence has a large DIT but in the context of the application a small DIT value. Therefore its interpretation may be misleading.

Since the main problem with DIT is that there is no distinction between the different kinds of inheritance, Tempero et al. [TNM08] have proposed an alternative for Java. They distinguish two kinds of inheritance in Java: extend and implement. They distinguish three domains: user-defined classes, standard library and third-party. They have introduced new metrics to provide on how much inheritance occurs in an application. Unfortunately, they do not propose metrics for indicator of "good-design" or fault predictor.

DIT measures the number of ancestor classes that affect the measured class. In case of multiple inheritance the definition of this metric is given as the longest path from class to the root class, this does not indicate the number of classes involved. Since excessive use of multiple inheritance is generally discouraged, the DIT does not measure this.

Briand et al. [BMB96] have made an empirical validation of DIT, concluding that the larger the DIT value, the greater the probability of fault detection.

Gyimothy et al. [GFS05] conclude that DIT is not as good predictor of fault than the other set of CKmetrics and they say that this metric needs more investigation to confirm their hypothesis : "A class located lower in a class inheritance lattice than its peers is more fault-prone than they are".

Moreover, DIT was really often studied but in most cases this was made with programs with few inheritance, and therefore this metric needs more empirical validation for programs with more inheritance. Li and Henry [LH93a] used the DIT metric as a measure of complexity, where the deeper the inheritance, the more complex the system is supposed to be. But as Hitz and Montazeri [HM95] notice, this means that inheritance increases the complexity of a system while it is considered a major advantage of the object-oriented paradigm.

This metric as well as other CKmetrics should be put in perspective: one measure is not really significant but the change of values between two measures should bring more information. Also, this metric should be applied at different scope because of its different interpretation depending on the context: counting only the user-defined classes or the standard library too.

Names **Number of Children**

Acronyms **NOC**

References [CK94b, GFS05]

Definition Number of children counts the immediate subclasses subordinated to a class in the class hierarchy

Scope Class

Analysis This metric shows the impact and code reuse in terms of subclasses. Because of change may impact all children, the more children have a class, the more changes require testing. Therefore NOC is a good indicator to evaluate testability but also impact of a class in its hierarchy. Because of counting only immediate subclass, this metric is not sufficient to assess the quality of a hierarchy.

Gyimothy et al. studied this metric and didn't state that it is a good fault detection predictor. Briand et al. [BMB96] found NOC to be significant and they observed that the larger the NOC, the lower the probability of fault detection, which seems at first contradictory.

Marinescu and Ratiu [MR04] characterize the inheritance with this 2 metrics : the Average Number of Derived Classes - the average of direct subclasses for all the classes defined in the measured system (NOC) - and the Average Hierarchy Height - the average of the Height of Inheritance tree (DIT). These 2 metrics indicate not only if the inheritance is used by the system but also if there are classes which use inheritance.

Names **Number of Methods**

Acronyms **NOM**

References [LK94]

Definition NOM represents the number of methods defined locally in a class, counting public as well as private methods. Overridden methods are taken account too.

Scope Class

Analysis NOM is a simple metric showing the complexity of a class in terms of responsibilities. However, it does not make the difference between simple and complex methods. WMC is better suited for that. NOM can be used to build ratio based on methods.

Names **Weighted Methods Per Class**

Acronyms **WMC**

References [CK94b]

Definition WMC is the sum of complexity of the methods which are defined in the class. The complexity was originally the cyclomatic complexity.

Scope Class

Analysis This metric is often limited when people uses as weighted function the function $fct = 1$. In such a case it corresponds to NOM. This metric is interesting because it gives an overall point of view of the class complexity.

Names **Cyclomatic Complexity Metric**

Acronyms **V(G)**

References [McC76]

Definition Cyclomatic complexity is the maximum number of linearly independent paths in a method. A path is linear if there is no branch in the execution flow of the corresponding code. This metric could be also called "Conditional Complexity", as it is easier to count conditions to calculate V(G) - which most tools actually do.

$$V(G) = e - n + p$$

where n is number of vertices, e the number of edges and p the number of connected components.

Scope Method, Class

Analysis This metric is an indicator of the psychological complexity of the code: the higher the V(G), the more difficult for a developer to understand the different pathways and the result of these pathways - which can lead to higher risk of introducing bugs. Therefore, one should pay attention to high V(G) methods.

Cyclomatic complexity is also directly linked to testing efforts: as V(G) increases, more tests need to be done to increase test coverage and then lower regression risks. Actually, V(G) can be linked to test coverage metrics:

- V(G) is the maximum amount of test cases needed to achieve a complete branch coverage
- V(G) is the minimum amount of test cases needed to achieve a complete path coverage

At class level, cyclomatic complexity is the sum of the cyclomatic complexity of every method defined in the class.

Names **Essential Cyclomatic Complexity Metric**

Acronyms **eV(G)**

References [McC76]

Definition Essential cyclomatic complexity is the cyclomatic complexity of the simplified graph - i.e. the graph where every well structured control structure has been replaced by a single statement. For instance, a simple "if-then-else" is well structured because it has the same entry and the same exit: it can be simplified into one statement. On the other hand, a "break" clause in a loop creates a new exit in the execution flow: the graph can not be simplified into a single statement in that case.

$$ev(G) = v(G) - m$$

where m is the number of subgraphs with a unique entry and a unique exit.

Scope Method, Class

Analysis This metric is an indicator of the degree of structuration of the code, which has effects on maintenance and modularization efforts. Code with lots of "break", "continue", "goto" or "return" clauses is more complex to understand and more difficult to simplify and divide into simpler routines.

As for V(G), one should pay attention to methods with high essential cyclomatic complexity.

Names **Number of Inherited Methods**

Acronyms **NIM**

References [LK94, BDPW98]

Definition NIM is a simple measure showing the amount of behavior that a given class can reuse. It counts the number of methods which a class can access in its superclasses.

Scope Class

Analysis The larger the number of inherited methods is, the larger class reuse happens through subclassing. It can be interesting to put this metric in perspective with the number of super sends and self send to methods not defined in the class, since it shows how much internal reuse happens between a class and its superclasses based on invocation (the same can be done for incoming calls to methods inherited, although this is harder to assess statically). Also, inheriting from large superclasses can be a problem since maybe only a part of the behavior is used/needed in the subclass. This is a limit of single inheritance based object-oriented programming languages.

Names **Number of overRiden Methods**

Acronyms **NRM**

References [LK94]

Definition NRM represents the number of methods that have been overridden i.e., defined in the superclass and redefined in the class. This metric includes methods doing super invocation to their parent method.

Scope Class

Analysis This metrics shows the customization made in a subclass over the behavior inherited. When the overridden methods are invoked by inherited methods, they represent often important hook methods. A large number of overridden methods indicates that the subclass really specializes its superclass behavior. However, classes with a lot of super invocation are quite rare (For the namespace VisualWorks.Core there are 1937 overridden methods for 229 classes: an average equals to 8.4 overridden methods per class) When compared with the number of added methods, this comparison offers a way to qualify the inheritance relationship: it can either be an inheritance relationship which mainly customizes its parent behavior or it adds behavior to its parent one. However, Briand et al. [BDPW98] conclude that the more overriding methods used, the more fault-prone software system becomes.

Names **Specialization IndeX**

Acronyms **SIX**

References [LK94, May99]

Definition

$$SIX = \frac{NRM \times DIT}{NOM + NIM}$$

The Specialization Index metric measures the extent to which subclasses override their ancestors classes. This index is the ratio between the number of overridden methods and total number of methods in a Class, weighted by the depth of inheritance for this class. Lorenz and Kidd precise : "Methods that invoke the superclass' method or override template are not included".

Scope Class

Analysis This metric was developed specifically to capture the point that classes are structured in hierarchy which reuse code and specialize code of their superclasses. It is well-defined, not ambiguous and easy to calculate. However, it is missing theoretical and empirical validation. It is commonly accepted that the more the Specialization Index is elevated, the more difficult is the class to maintain, but there is no validation to prove it.

Moreover, this index does not care about the scope of the class. And, because the SIX metric is based on the DIT metric, it has the same limits.

Rather than reading this index as a quality index, it should be read as an indicator requiring classes to be analyzed with more attention.

Lorenz and Kidd state that the anomaly threshold is at 15%. With $NRM = 3$, $DIT = 2$, $NOM = 20$, $NIM = 20$:

$$SIX = \frac{3 \times 2}{20 + 20} = 15\%$$

According to Mayer [May99], this measure seems reasonable and logical but in practice it is so coarsely grained and inconsistent that it is useless. He shows with two theoretical examples that this metric does not reflect the spontaneous understanding and says that it would be enough to simply multiply the number of overridden methods (NRM) by the DIT value : "dividing by the number of methods adds nothing to this measure; in fact, it greatly reduces its accuracy". Therefore we suggest not to use it.

Names **Response For a Class**

Acronyms **RFC**

References [CK94b]

Definition RFC is the size of the response set of a class. The response set of a class includes "all methods that can be invoked in response to a message to an object of the class". It includes local methods as well as methods in other classes.

Scope Class

Analysis This metric reflects the class complexity and the amount of communication with other classes. The larger the number of methods that may be invoked from a class through messages, the greater the complexity of the class is.

Three points in the definition are imprecise and need further explanations:

- Although it is not explicit in the definition, the set of called methods should include polymorphically invoked methods. Thus the response set does not simply include signatures of methods.
- Inherited methods, as well as methods called through them, should be include in the set, as they may be called on any object of the class.
- It is not clear whether methods *indirectly* called through local methods should be counted. If this is the case, the metric becomes computationally expensive. In practice, the authors limit their definition to the first level of nested calls, *i.e.,* only methods directly called by local methods are included in the set.

So many different implementations and interpretations make RFC unreliable for comparison, unless a more precise definition is agreed upon.

Names **Source Lines Of Code**

Acronyms **SLOC, LOC**

References [Kan02]

Definition SLOC is the number of effective source lines of code

Scope Method, Class, Package, Project

Analysis Comments and blank lines are often ignored. This metric provides a raw approximate of the complexity or amount of work. LOC does not convey the complexity due to the flow of an execution. Correlating it with McCabe complexity is important since we can get long a simple methods as well as complex ones.

For further information on the content of code, many tools calculate also the Number of Commented Lines (CLOC) - lines containing comments, including mixed lines where both comments and code are written - and also Percentage of Comments witch is used as an indicator of readability of code.

Names **Code Coverage**

Acronyms

References

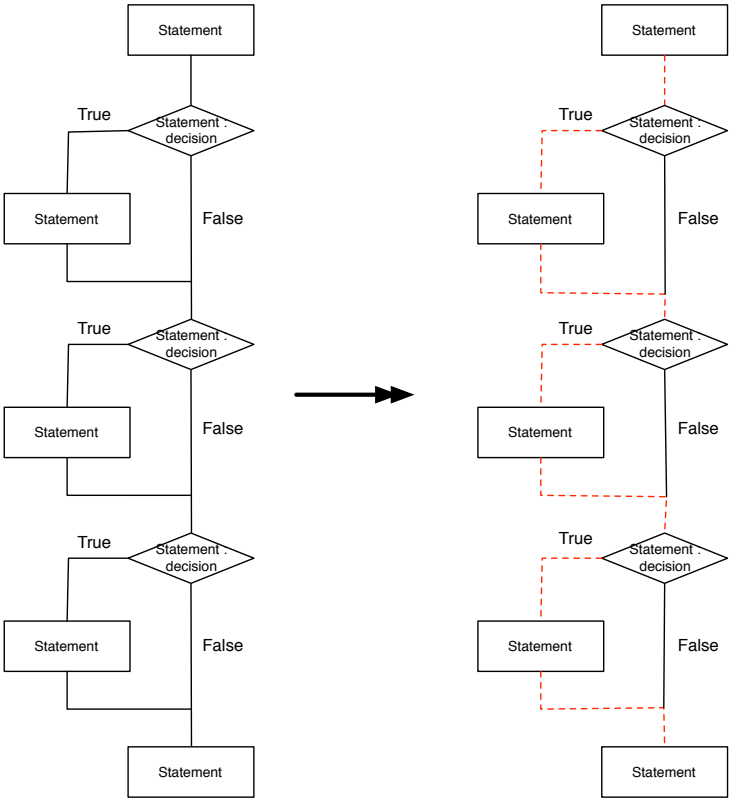


Figure 1.1: Paths considered by Statement Coverage at 100%.

Definition This suite of metrics assesses how testing covers different parts of the source code, which gives a measure of quality through confidence in code. Unit tests are performed to check code against computable specifications and to determine the reliability of the system. Different metrics evaluate different manners of covering the code.

Scope Project, Method

Analysis There are three principal kinds of coverage:

- Statement Coverage: This metric computes the number of lines of code covered by tests. But there is no guarantee of quality even if the coverage is 100 %. Covering all lines of code is useful to research broken code or useless code but it does not determine if the returns of the methods are complying with the expectations in all cases. This metric is easy to compute but does not take into account execution paths determined by control

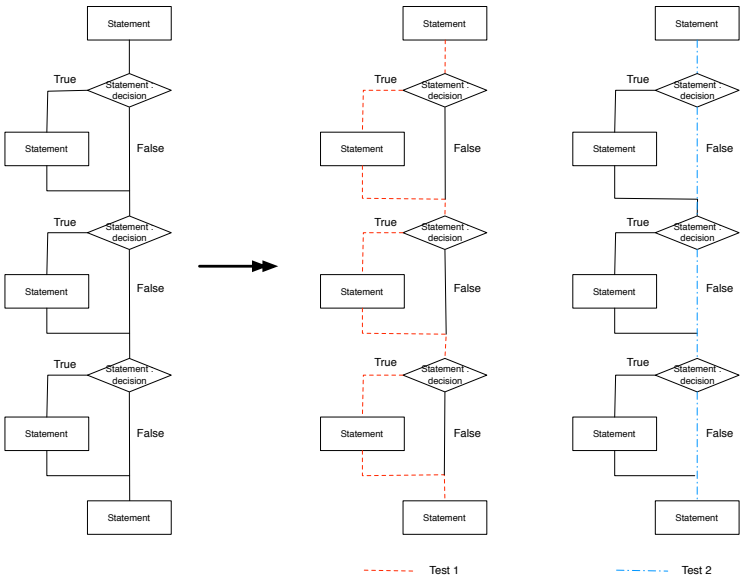


Figure 1.2: Paths considered by Branch Coverage at 100%.

structures. In Figure 1.1, Statement Coverage is 100 % because all statements are covered but there are still three paths which are not covered: the three False paths.

– Branch Coverage or Decision Coverage: This metric determines if tests cover the different branches introduced by the control structures in a system. For example, an 'if' introduces two branches and unit tests must include the two cases: True and False. In Figure 1.2, the three control structures introduce six branches and tests must include six cases: True and False for each control structure. So if tests cover the three True branches and then the three False branches, Branch Coverage is 100 % but does not take care of all possible paths in the system: what about the case True, True and False or False, True and True for example?

– Path Coverage: This metric determines if the tests cover all the possible paths in a system. Because of the number of possible paths in a system could be really high (for N conditions there is 2^N possible paths) or unbounded (if there is an infinite loop), this metric is coupled with the cyclomatic complexity. The number of paths to cover increases linearly with cyclomatic complexity, not exponentially. Generally, the cyclomatic complexity form's is $v(G) = d + 1$ where d is the number of binary decision node in G (G could be a method for example). In Figure 1.3 there are three conditions so there are $2^3 = 8$ possible paths, but only four paths to cover

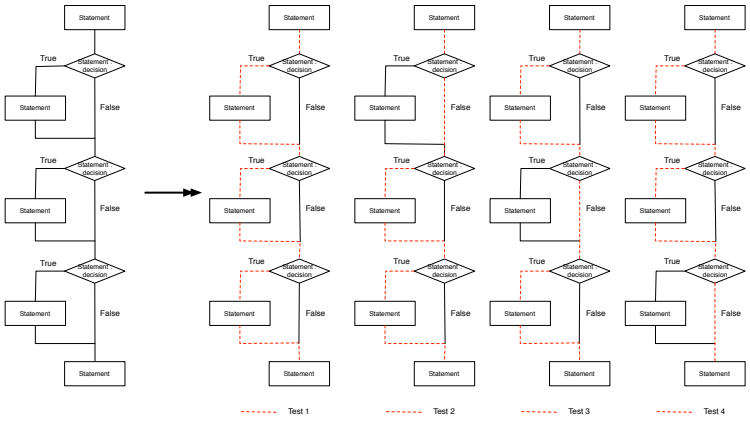


Figure 1.3: Paths considered by Path Coverage at 100%.

with cyclomatic complexity: $3 + 1$. A set of paths could be: True-True-True, False-True-True, True-False-True, True-True-False. The other paths are not independent paths so they are ignored.

These metrics indicate the level of tests but not the quality of them. They do not determine if the unit tests are well defined and a number value near 100% does not indicate that there are no more bugs in the source code.

1.2 Design Metrics: Class Coupling

High quality software design, among many other principles, should obey the principle of low coupling. Stevens *et al.* [SMC74], who first introduced coupling in the context of structured development techniques, define coupling as "the measure of the strength of association established by a connection from one module to another". Therefore, the stronger the coupling between modules, *i.e.,* the more interrelated they are, the more difficult these modules are to understand, change, and correct and thus the more complex the resulting software system.

Excessive coupling indicates the weakness of class encapsulation and may inhibit reuse. High coupling also indicates that more faults may be introduced due to inter-class activities. A classic example is the coupling between object classes (CBO), which considers the number of other classes "used" by this class. High CBO measure for a class means that it is highly coupled with other classes.

Names **Coupling Between Object classes**

Acronyms **CBO**

References [CK94b, FP96, Mar05a]

Definition Two classes are coupled together if one of them uses the other, *i.e.,* one class calls a method or accesses an attribute of the other class. Coupling involving inheritance and methods polymorphically called are taken into account. CBO for a class is the number of classes to which it is coupled.

Scope Class

Analysis Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class, the easier it is to reuse it in another application. Strong coupling complicates a system, since a module is harder to understand, change, and correct by itself if it is interrelated with other modules. CBO evaluates efficiency and reusability.

In a previous definition of CBO, coupling related to inheritance was explicitly excluded from the formula.

CBO only measures direct coupling. Let us consider three classes, A, B and C, with A coupled to B and B coupled to C. Depending on the case, it can happen that A depends on C through B. Yet, CBO does not account for the higher coupling of A in this case.

CBO is different from Efferent Coupling (which only counts outgoing dependencies) as well as Afferent Coupling (which only counts incoming dependencies).

Names **Coupling Factor**

Acronyms **COF**

References [BGE95],[BDW99a]

Definition Coupling Factor is a metric defined at program scope and not at class scope. Coupling Factor is a normalized ratio between the number of client relationships and the total number of possible client relationships. A client relationship exists whenever a class references a method or attribute of another class, *except* if the client class is a descendant of the target class. Thus, inheritance coupling is excluded but polymorphically invoked methods are still accounted for.

The formal definition of COF given by Briand *et al.* [BDW99a], for a program composed of a set TC of classes, is:

$$COF(TC) = \frac{\sum_{c \in TC} \mid \{d \mid d \in (TC - \{c\} \cup Ancestors(c)) \wedge uses(c,d)\} \mid}{\mid TC \mid^2 - \mid TC \mid -2\sum_{c \in TC} \mid Descendants(c) \mid}$$

uses(c,d) is a predicate which is true whenever class *c* references a method or attribute of class *d*, including polymorphically invoked methods.

Scope Program

Analysis Coupling Factor is a metric which is only defined at program scope and not at class scope. It makes it difficult to compare with metrics defined at class scope, which can only be summarized at program level.

The original metric was unclear whether polymorphic methods should be accounted for [BGE95].

Names **Message Passing Coupling**

Acronyms **MPC**

References [LH93a]

Definition MPC is defined as the "number of send statements defined in a class". The authors further refine the definition by indicating that calls to class own methods are excluded from the count, and that only calls from local methods are considered, excluding calls in inherited methods.

The formal definition of MPC is:

$$MPC(c) = \sum_{m \in M_I(c)} \sum_{m' \in SIM(m) - M_I(c)} NSI(m, m')$$

where m belongs to the set of local methods M_I and m' belongs to the set of methods statically invoked by m (*i.e.,* without taking polymorphism into account), and excluding local methods ($SIM(m) - M_I$). $NSI(m, m')$ is the number of static invocations from m to m'.

Scope Class

Analysis The authors give the following interpretation: "The number of send statements sent out from a class may indicate how dependent the implementation of the local methods is on the methods in other classes."

MPC does not consider polymorphism as only send statements are accounted for, not method definitions which could be polymorphically invoked.

1.3 Design Metrics: Class Cohesion

Cohesiveness of methods within a class is desirable since it promotes encapsulation and lack of cohesion implies that classes should probably be split into two or more subclasses. For example, although we can have more definitions for the lack of cohesion metric, they all show the cohesiveness of the class considering different relationships between the methods of the class. We distinguish class cohesion from package cohesion, since late-binding in object-oriented context may lead to good packages that have a low-cohesion as for example in the case of framework extensions.

The Lack of Cohesion in Methods (LCOM) metric was one of the first metric to evaluate cohesion in object-oriented code, based on a similar metric for procedural cohesion. Different versions of LCOM have been released across the years: some have been intended to correct and replace previous faulty versions, while others have taken a different approach to measure cohesion. As a consequence, numbering schemas for LCOM have diverged across references. Typically, LCOM1 was first called LCOM before being replaced by another LCOM, which was later renamed as LCOM2.

Most of the LCOM metrics are somehow naive in the intention that they want to capture: a cohesive class is not a class whose all methods access all the instance variables even indirectly. Most of the time, a class can have state that is central to the the domain it represents, then it may have peripheral state that may be used to share data between computation performed by the methods. This does not mean that the class should be split. Splitting a class is only possible when two groups of attributes and methods are not accessed simultaneously.

All LCOM metrics (LCOM1 to LCOM5) give inverse cohesion measures: a high cohesion is indicated by a low value, and a low cohesion is indicated by a high value. We followed the numbering of LCOM metrics given by [BDW98b], which seems the most widely accepted.

Original LCOM definitions only consider methods and attributes defined in the class. Thus inherited methods and attributes are excluded.

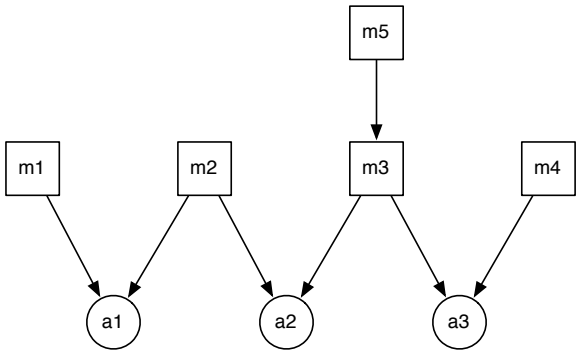


Figure 1.4: Sample graph for LCOM metrics with methods m_i accessing attributes a_j or calling other methods.

Names **Lack of Cohesion in Methods**

Acronyms **LCOM1**

References [CK94b][BDW98b]

Definition LCOM1 is the number of pairs of methods in a class which do not reference a common attribute.

Scope Class

Analysis In Figure 1.4, $LCOM1 = 7$. The pairs without a common attribute are (m1, m3), (m1, m4), (m2, m4), (m5, m1), (m5, m2), (m5, m3), (m5, m4).

The definition of this metric is naive and led to a lot of debate against it. In general it should be avoided as much as possible. In particular, it does not make sense that all methods of a class directly access all attributes of the class. It can give the same measure for classes with very different designs. This metric gives incorrect results when there are accessor methods.

This metric only considers methods implemented in the class and only references to attributes implemented in the class. Inherited methods and attributes are excluded.

Names **Lack of Cohesion in Methods**

Acronyms **LCOM2**

References [BDW98b]

Definition For each pair of methods in the class, if they access disjoint sets of instance variables, then P is increased by one, else Q is increased by one.

- $LCOM2 = P - Q$, if $P > Q$,
- $LCOM2 = 0$ otherwise,

$LCOM2 = 0$ indicates a cohesive class. $LCOM2 > 0$ indicates that the class can be split into two or more classes, since its instance variables belong to disjoint sets.

Scope Class

Analysis In the Figure 1.4, $LCOM2 = 4$. P is the sum of all pairs of methods which reference no common attribute thus $P = 7$ (pairs (m1, m3), (m1, m4), (m2, m4), (m5, m1), (m5, m2), (m5, m3), (m5, m4)). Q is calculated with other pairs ((m1, m2), (m2, m3), (m3, m4)), thus $Q = 3$. The result is $P - Q = 4$.

The definition of LCOM2 only considers methods implemented in the class and references to attributes implemented in the class. Inherited methods and attributes are excluded.

It can give the same measure for classes with very different design. LCOM2 may be equals to 0 for many different classes. This metric gives incorrect results when there are accessors methods. Moreover, LCOM2 is not monotonic because of "if $Q > P$, then $LCOM2 = 0$".

Names **Lack of Cohesion in Methods**

Acronyms **LCOM3**

References [BDW98b]

Definition LCOM3 is the number of connected components in a graph of methods. Methods are connected in the graph with methods accessing the same attribute.

Scope Class

Analysis In Figure 1.4, $LCOM3 = 2$. The first component is (m1, m2, m3, m4) because these methods are directly or indirectly connected together through some attributes. The second component is (m5) because the method does not access any attribute and thus is not connected.

This metric only considers methods implemented in the class and only references to attributes implemented in the class. Inherited methods and attributes are excluded.

This metric gives incorrect results when there are accessor methods because only methods directly connected with attributes are considered.

Constructors are a problem, because of indirect connections with attributes. They create indirect connections between methods which use different attributes, and increase cohesion, which is not real.

Names **Lack of Cohesion in Methods**

Acronyms **LCOM4**

References [BDW98b]

Definition LCOM4 is the number of connected components in a graph of methods. Methods are connected in the graph with methods accessing the same attribute or calling them. LCOM4 improves upon LCOM3 by taking into account the transitive call graph.

- $LCOM4 = 1$ indicates a cohesive class.
- $LCOM4 \geq 2$ indicates a problem. The class should be split into smaller classes.
- $LCOM4 = 0$ happens when there are no methods in a class.

Scope Class

Analysis In the Figure 1.4, LCOM4 = 1. The only component is (m1, m2, m3, m4, m5) because these methods are directly or indirectly connected to the same collection of attributes.

If there are 2 or more components, the class could be split into smaller classes, each one encapsulating a connected component.

Names **Lack of Cohesion in Methods**

Acronyms **LCOM5, LCOM***

References [BDW98b]

Definition

$$LCOM5 = \frac{NOM - \frac{\sum_{m \in M} NOAcc(m)}{NOA}}{NOM - 1}$$

where M is the set of methods of the class, NOM the number of methods, NOA the number of attributes, and $NOAcc(m)$ is the number of attributes of the class accessed by method m.

Scope Class

Analysis In Figure 1.4, LCOM5 = $\frac{3}{4}$, because $NOM = 5$, $NOA = 3$, $\sum NOAcc = 6$.

A common acronym for LCOM5 is LCOM*.

LCOM5 varies in the range [0,1]. It is normalized compared to others LCOM or TCC and LCC which have no upper limit of the values measured. But LCOM5 can return a measure up to two when there is for example only two methods and no attribute accesses.

This metric considers that each method should access all attributes in a completely cohesive class, which is not a good design.

Names **Tight Class Cohesion**

Acronyms **TCC**

References [BK95a, BDW98b]

Definition TCC is the normalized ratio between the number of methods directly connected with other methods through an instance variable and the total number of possible connections between methods.

A direct connection between two methods exists if both access the same instance variable directly or indirectly through a method call (see Figure 1.5).

- $NP = \frac{N \times (N-1)}{2}$: maximum number of possible connections where N is the number of visible methods
- NDC: number of direct connections
- $TCC = \frac{NDC}{NP}$

TCC takes its value in the range $[0, 1]$.

For TCC only visible methods are considered, ***i.e.,*** they are not private or implement an interface or handle an event. Constructors and destructors are ignored.

Scope Class

Analysis TCC measures a strict degree of connectivity between visible methods of a class. TCC satisfies all cohesion properties defined in [BDW98b].

The higher TCC is, the more cohesive the class is. According to the authors, $TCC < 0.5$ points to a non-cohesive class. $TCC = LCC = 1$ is a maximally cohesive class: all methods are connected.

Constructors are a problem, because of indirect connections with attributes. They create indirect connections between methods which use different attributes, and increase cohesion, which is not real.

Names **Loose Class Cohesion**

Acronyms **LCC**

References [BK95a, BDW98b]

Definition LCC is the normalized ratio between the number of methods directly or indirectly connected with other methods through an instance variable and the total number of possible connections between methods.

There is an indirect connection between two methods if there is a path of direct connections between them. It is defined using the transitive closure of the direct connection graph used for TCC (see Figure 1.5).

- $NP = \frac{N \times (N-1)}{2}$: maximum number of possible connections where N is the number of visible methods
- NIC: number of indirect connections

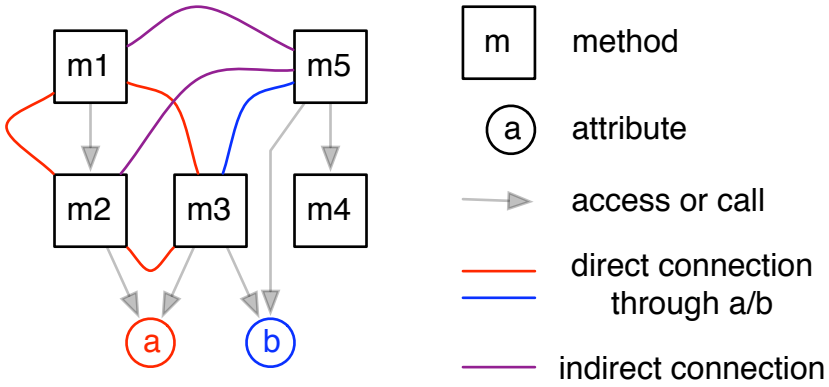


Figure 1.5: In this example depicting five methods and two attributes of a class, we have $TCC = \frac{4}{10}$ (counting red and blue lines) and $LCC = \frac{6}{10}$ (adding purple lines to the count).

- $LCC = \frac{NDC+NIC}{NP}$
- By definition, $LCC \geq TCC$

LCC takes its value in the range $[0, 1]$.

For LCC only visible methods are considered, *i.e.,* they are not private or implement an interface or handle an event. Constructors and destructors are ignored.

Scope Class

Analysis LCC measures an overall degree of connectivity between visible methods of a class. LCC satisfies all cohesion properties defined in [BDW98b].

The higher LCC is, the more cohesive the class is. According to the authors, $LCC < 0.5$ points to a non-cohesive class. $LCC = 0.8$ is considered "quite cohesive". $TCC = LCC$ indicates a class with only direct connections. $TCC = LCC = 1$ is a maximally cohesive class: all methods are connected.

Constructors are a problem, because of indirect connections with attributes. They create indirect connections between methods which use different attributes, and increase cohesion, which is not real.

1.4 Design Metrics: Package Architecture

Package Design Principles. R.C. Martin discussed principles of architecture and package design in [Mar97, Mar05a, Mar00]. He proposes several principles:

- Release / Reuse Equivalence principle (REP): The granule of reuse is the granule of release. A good package should contain classes that are reusable together.
- Common Reuse Principle (CRP): Classes that are not reused together should not be grouped together.
- Common Closure Principle (CCP): Classes that change together, belong together. To minimize the number of packages that are changed in a release cycle, a package should contain classes that change together.
- Acyclic Dependencies Principle (ADP): The dependencies between packages must not form cycles.
- Stable Dependencies Principle (SDP): Depend in the direction of stability. The stability is related to the amount of work required to make a change on it. Consequently, it is related to the package size and its complexity, but also to the number of packages which depend on it. So, a package with lots of incoming dependencies from others packages is stable (it is responsible to those packages); and a packages with not any incoming dependency is considered as independent and unstable.
- Stable Abstractions Principle (SAP): Stable packages should be abstract packages. To improve the flexibility of applications, unstable packages should be easy to change, and stable packages should be easy to extend, consequently they should be highly abstract.

Some metrics have been built to assess such principles: For example, the Abstractness and Instability metrics are used to check SAP.

Martin Package Metrics. The following metrics defined by Martin [Mar97] aim at characterizing good design in packages along the SDP and SAP principles. However, the measurements provided by the metrics are difficult to interpret.

Names **Efferent Coupling (module)**

Acronyms **Ce**

References [Mar05b]

Definition Efferent coupling for a module is the number of modules it depends upon (outgoing dependencies, fan-out, Figure 1.6).

Scope Class, Package

Analysis In [Mar00], efferent coupling for a package was defined as the number of classes outside the package that classes inside depend upon. In [Mar97], efferent coupling for a package was defined as the number of classes in the package which depend upon classes external to the package.

The current definition is *generalized* with respect to the concept of module, where a module is always a class or always a package.

Names **Afferent Coupling (module)**

Acronyms **Ca**

References [Mar05b]

Definition Afferent coupling for a module is the number of modules that depend upon this module (incoming dependencies, fan-in, Figure 1.6).

Scope Class, Package

Analysis In [Mar97], afferent coupling for a package was defined as the number of classes external to the package which depend upon classes in the package.

Names **Abstractness**

Acronyms **A**

References [Mar97]

Definition Abstractness is the ratio between the number of abstract classes and the total number of classes in a package, in the range $[0, 1]$. 0 means the package is fully concrete, 1 it is fully abstract.

Scope Package

Analysis This metric can not be analyzed in isolation. In any system, some packages should be abstract while other should be concrete.

Names **Instability**

Acronyms **I**

References [Mar97]

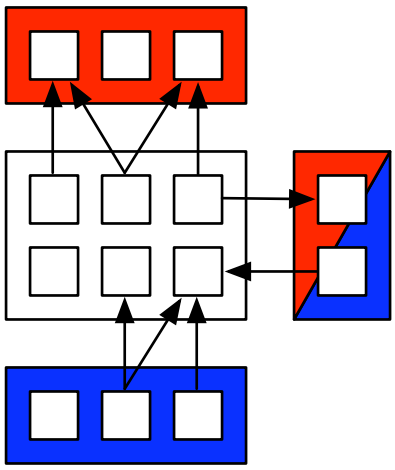


Figure 1.6: Each module is represented by a box with enclosed squares. It is either a class enclosing methods and attributes, or a package enclosing classes. Efferent coupling is the number of red modules ($Ca = 2$); afferent coupling is the number of blue modules ($Ce = 2$).

Definition $I = \frac{Ce(P)}{Ce(P)+Ca(P)}$, in the range $[0, 1]$. 0 means package is maximally stable (*i.e.,* no dependency to other packages and can not change without big consequences), 1 means it is unstable.

This metric is used to assess the Stable Dependencies Principle (SDP): according to Martin [Mar97], a package should only depends on packages which are more stable than itself, i.e. it should have a higher I value than any of its dependency. The metric only gives a raw value to be used while checking this rule.

Scope Package

Analysis Instability is not a measure of the possibility of internal changes in the package, but of the potential impact of a change related to the package. A maximally stable package can still change internally but should not as it will have an impact on dependent packages. An unstable package can change internally without consequences on other packages. Intermediate values, in the range $]0, 1[$, are difficult to interpret.

A better way to understand Instability is as a Responsible/Independent couple. A stable package ($I = 0$) is independent and should be responsible because of possible incoming dependencies. An unstable package is dependent of other packages and not responsible for other packages.

The stability concept, understood as sensitivity to change, should be transitively defined: a package can only be stable if its dependencies are themselves stable.

Names **Distance**

Acronyms **D**

References [Mar97]

Definition $D = \frac{A+I-1}{\sqrt{2}}$ or (normalized) $D' = A + I - 1$. A package should be balanced between abstractness and instability, *i.e.,* somewhere between abstract and stable or concrete and unstable. This rule defines the main sequence by the equation $A + I - 1 = 0$. D is the distance to the main sequence.

This metric is used to assess the Stable Abstractions Principle (SAP): stable packages should also be abstract packages ($A = 1$ and $I = 0$) while unstable packages should be concrete ($A = 0$ and $I = 1$).

Scope Package

Analysis This metric assumes a one-to-one inverse correlation between Abstractness and Instability to assert the good design of a package. However, such a correlation seems uneasy given the difference in data nature on which each metric is computed. To our knowledge, no large scale empirical study has been performed to confirm this assumption.

This metric is sensitive to extreme cases. For example, a package with only concrete classes ($A = 0$) but without outgoing dependencies ($I = 0$) would have a distance $D' = -1$. Yet this package does not necessarily exhibit a bad design.

Chapter2. New package metrics to support application remodularisation

The authors of the following work are Hani Abdeen and Stéphane Ducasse. A short version of these research results has been published as a short paper in WCRE (Working Conference on Reverse Engineering) with Houari Sahraoui as co-author and a long version as technical report on HAL http://hal.inria.fr.

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 [DK76].

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 [EGK+01, GN93]. 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 [CC92, BDW98a, BDW99b, BMB99, eAC94, LH93b, Li98, CK94a, GM00, DB10]. Few previous efforts measure the quality of some aspects of package organization and relationships [AK01, Fow01, Mar02, AG01, PN06]. 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[AG01, Mar02, DPS+07, ADSA09, ADPA10]. Fortunately, Santonu Sarkar *et al.* [SKR08, SRK07] 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 [Par72] and R. Martin [Mar02]. 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 2.1 underlines the modularity principles that we address with these metrics. In Section 2.2 we define the terminology and the notations we use to define our metrics. We define a set of coupling metrics in Section 2.3 and a set of complementary cohesion metrics in Section 2.4. In Section 2.5 we show how our metrics satisfy all the mathematical properties that are defined by Briand *et al.* [BDW98a, BDW99b]. We discuss our metrics in Section 2.6 and we compare them to previous works related to software metrics in Section 2.7.

2.1 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 [Par72] and R. Martin [Mar02], 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) [SSP07, SM08]: 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 with these metrics.

2.1.1 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*.

2.1.2 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*.

2.1.3 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. 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.

2.2 Terminology and Notation

We assume 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 report.

2.2.1 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.

2.2.2 Package Interfaces and Relationships

We assume 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.

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*).

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.

2.2.3 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.

2.2.4 Dependencies Notation

In this section we define the notations of different types of dependencies. 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.

2.3 Coupling Metrics:

Metrics related to Information-Hiding and Changeability Principles

2.3.1 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 2.1.1. 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 $Extends$.

Index of Inter-Package Usage

As defined in Section 2.2, 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 $\mathcal{P}$:

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

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

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

$$IIPU(\mathcal{M}) = 1 - \frac{UsesSum(\mathcal{P})}{UsesSum(\mathcal{C})} \tag{2.1}$$

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.

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.

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

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

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

$$IIPE(\mathcal{M}) = 1 - \frac{ExtSum(\mathcal{P})}{ExtSum(\mathcal{C})} \tag{2.2}$$

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 means 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.

2.3.2 Index of Package Changing Impact

In Section 2.3.1 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 2.3.1 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 characterizing, at package level, the maintainability of $\mathcal{M}$ (ref. Section 2.1.2). 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, Figure 2.1 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 2.2, 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:

$$IPCI(p) = 1 - \frac{Clients_p(p)}{1 - |\mathcal{P}|} \tag{2.3}$$

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

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.

2.3.3 Index of Package Communication Diversion

In this section we define metrics that measure the extent to which package communication (Section 2.3.1) is focused or diverted. Our vision of package communication diversion can be explained as follows: let p be a package that uses 5 classes packaged into 5 different packages, and let q be a package that 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).

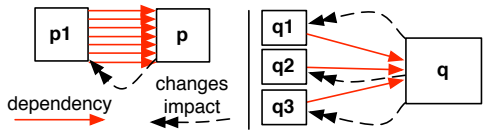


Figure 2.1: Explanation for Package Changing Impact

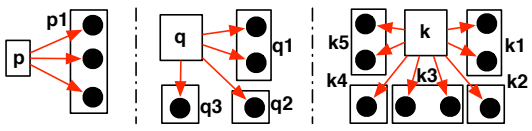


Figure 2.2: Explanation for Package Usage Diversion

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:

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

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

$$IIPUD(\mathcal{M}) = \frac{\sum_{p_i \in \mathcal{P}} IIPUD(p_i)}{|\mathcal{P}|} \tag{2.4}$$

Interpretation. IIPUD is the index of inter-package usage diversion and PUF is package usage factor. We used this factor to distinguish 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 Figure 2.2, p starts out with only one coupling path, where it uses only one package p_1. It means 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 have 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, Figure 2.2 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.

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 $Ext_c(p)$ denotes the classes that p extends; and let 1 be the minimal number of coupling paths that p may start with.

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

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

$$IIPED(\mathcal{M}) = \frac{\sum_{p_i \in \mathcal{P}} IIPED(p_i)}{|\mathcal{P}|} \tag{2.5}$$

Interpretation. The interpretation of IIPED(p) is similar to what we stated above for IIPUD(p) in Section 2.3.3. 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 implements interfaces that are completely not similar from the point of view of their providers. As summary, p is 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.

2.4 Cohesion Metrics:

Metrics related to Commonality-of-Goal Principle

2.4.1 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:

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

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

$$PF(p) = \frac{\sum_{p_i \in Clients_p(p)} Role(p,p_i)}{|Clients_p(p)|} \tag{2.6}$$

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

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 Figure 2.3, 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 Figure 2.3):

1. Figure 2.3 shows that the package q exposes 5 in-interfaces 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. Figure 2.3 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, Figure 2.3 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.

2.4.2 Index of Package Services Cohesion

Unlike what we stated above in Section 2.4.1, 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 $CS(p,k)$:

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

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.

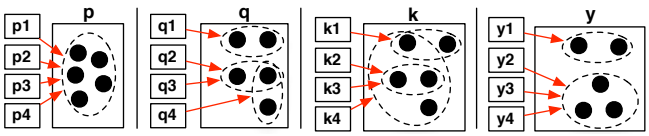
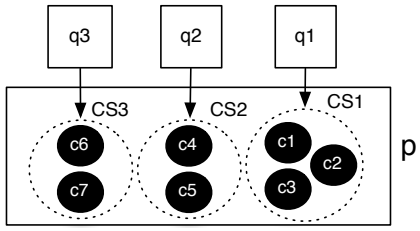
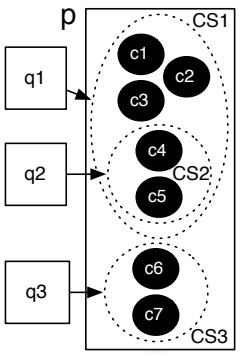


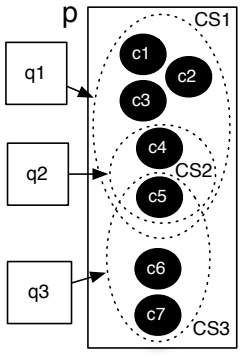
Figure 2.3: Explanation for Package Focus



(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): $CS_3 = \{c_5, c_6, c_7\}$.

Figure 2.4: Explanation for Package Services Cohesion.

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:

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

$$IPSC(p) = \frac{\sum_{q_i \in Clients_p(p)} CS_{cohesion}(p,q_i)}{|Clients_p(p)|} \tag{2.7}$$

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

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. Figure 2.4

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 Figure 2.4(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 Figure 2.4(b), the difference from Figure 2.4(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 Figure 2.4(c), the difference from Figure 2.4(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).

2.5 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.* [BDW98a, BDW99b].

2.5.1 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 2.3, 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 has 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.

2.5.2 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 2.4, 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.

2.6 Discussion

In this section, we discuss our metrics with regard to the modularity principles we underlined in Section 2.1.

2.6.1 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.* [CG05], 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.

2.6.2 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 $\mathcal{M}_1$.

2.6.3 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 2.1.3). $IPSC(p)$ 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 a non-identifiable (arbitrary) ways. In this last case, understanding the p services requires an understanding of each p in-interface aside from others.

2.7 Relevant Related Works vs. Our Metrics

To cope with software system complexity, Parnas *et al.* [Par72] 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 [CK94a, eAC94, LH93b, Li98, HS96b, BK95b, BK98, BMB99, BDW98a, BDW99b, DB10]. 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 [KKL01], or from the standpoint of their ability to predicate software maintability [BVT03, DJ03]. Other researchers argue that the measures resulted by the cohesion and coupling metrics of the previous works cited above are open to interpretation [KKL01, BDW99b]. 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.

2.7.1 Cohesion Metrics

Emerson presents a metric to compute cohesion applicable to modules in the sense of Pascal procedures [Eme84]. His metric is based on a graph theoretic property that quantifies the relationship between control flow paths and references to variables. Patel *et al.* [PCB92] 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 [AK01]. 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 [Mis01]. 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) [PN06]. 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 [SKR08]. 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.* [SKR08], 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.

2.7.2 Coupling Metrics

Martin [Mar02] 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 [Mar05], 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 [Hau02]. 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 [SKR08]. 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.

Chapter3. Conclusion

This document presents the state of the art on software metrics related to source code entities. We present different metrics related to classes, methods, and software packages. The document also provides a critique of the existing metrics.

Bibliography

[ADPA10] Hani Abdeen, Stéphane Ducasse, Damien Pollet, and Ilham Alloui. Package fingerprint: a visual summary of package interfaces and relationships. *Info. and Sof. Tech.*, 52:1312–1330, 2010.

[ADSA09] Hani Abdeen, Stéphane Ducasse, Houari Sahraoui, and Ilham Alloui. Automatic package coupling and cycle minimization. In *Int. Work. Conf. on Rev. Eng.*, pages 103–112. IEEE Computer Society Press, 2009.

[AG01] Fernando Britoe Abreu and Miguel 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.

[AK01] E. Allen and T. Khoshgoftaar. Measuring coupling and cohesion of software modules: An information theory approach. In *Seventh Int. Sof. Metrics Symposium*, 2001.

[BDPW98] L.C. Briand, J. Daly, V. Porter, and J. Wust. A comprehensive empirical validation of design measures for object-oriented systems. *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, pages 246–257, Nov 1998.

[BDW98a] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[BDW98b] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[BDW99a] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[BDW99b] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE TSE*, 25(1):91–121, 1999.

[BGE95] F. Brito e Abreu, M. Goulao, and R. Esteves. Toward the design quality evaluation of object-oriented software systems. In *Proc. 5th Int'l Conf. Software Quality*, pages 44–57, October 1995.

[BK95a] J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings ACM Symposium on Software Reusability*, April 1995.

[BK95b] J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. In *ACM Symposium on Software Reusability*, April 1995.

[BK98] J.M. Bieman and B.K. Kang. Measuring design-level cohesion. *IEEE TSE*, 24(2):111–124, February 1998.

[BMB96] Lionel C. Briand, Sandro Morasca, and Victor Basili. Property-based software engineering measurement. *Transactions on Software Engineering*, 22(1):68–86, 1996.

[BMB99] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Defining and validating measures for object-based high-level design. *IEEE TSE*, pages 722–743, 1999.

[BVT03] Rajendra K. Bandi, Vijay K. Vaishnavi, and Daniel E. Turk. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE TSE*, 29:77–87, 2003.

[CC92] J. Chris Coppick and Thomas J. Cheatham. Software metrics for object-oriented systems. In *ACM Conf. on Computer Science'92*, pages 317–322, 1992.

[CG05] Werner Callebaut and Diego Gutman. *Modularity: Understanding the Development and Evolution of Natural Complex Systems*. MIT press, 05.

[CK94a] Shyam R. Chidamber and Chris F. Kemerer. A metrics suit for object oriented design. *IEEE TSE*, 20:476–493, 1994.

[CK94b] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[DB10] Jehad Al Dallal and Lionel C. Briand. An object-oriented high-level design-based class cohesion metric. *Inf. and Sof. Tech.*, 52(12):1346–1361, 2010.

[DJ03] Melis Dagpinar and Jens 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.

[DK76] Frank DeRemer and Hans H. Kron. Programming in the large versus programming in the small. *IEEE TSE*, 2(2):80–86, 1976.

[DPS+07] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *IEEE Int. Conf. on Sof. Maint.*, pages 94–103, 07.

[eAC94] Fernando Brito e Abreu and Rogério Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Sys. Sof.*, 26:87–96, 1994.

[EGK+01] Stephen Eick, Todd Graves, Alan Karr, J. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE TSE*, 27(1):1–12, 2001.

[Eme84] Thomas Emerson. A discriminant metric for module cohesion. In *ICSE*, 1984.

[Fow01] Martin Fowler. Reducing coupling. *IEEE Software*, 2001.

[FP96] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996. 06-8147-I*, envoye a l'inria lille le 19 aout.

[GFS05] Tibor Gyimóthy, Rudolf Ferenc, and Istvá Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[GM00] Mohsen D. Ghassemi and Ronald 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.

[GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.

[Hau02] E. Hautus. Improving Java software through package structure analysis. In *Int. Conf. Sof. Eng. and App.*, 2002.

[HK00] Jiawei Han and Micheline Kamber. *Data Mining: Concept and Techniques*. Morgan Kaufmann, 2000.

[HM95] Martin Hitz and Behzad Montazeri. Measuring product attributes of object-oriented systems. In *Proc. ESEC '95 (5th European Software Engineering Conference*, pages 124–136. Springer Verlag, 1995.

[HS96a] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[HS96b] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2002.

[KKL01] Hind Kabaili, Rudolf K. Keller, and Frantçois 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.

[LH93a] W. Li and S. Henry. Object oriented metrics that predict maintainability. *Journal of System Software*, 23(2):111–122, 1993.

[LH93b] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Sys. Sof.*, 23:111–122, 1993.

[Li98] Wei Li. Another metric suite for object-oriented programming. *Journal of Sys. Sof.*, 44:155–162, December 1998.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[Mar05] Robert C. Martin. The tipping point: Stability and instability in oo design, 05. Software Development.

[Mar97] Robert C. Martin. Stability, 1997. www.objectmentor.com.

[Mar00] Robert C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.

[Mar02] Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[Mar05a] Robert C. Martin. The tipping point: Stability and instability in oo design, 2005. http://www.ddj.com/architect/184415285.

[Mar05b] Robert C. Martin. The tipping point: Stability and instability in oo design, 2005. Software Development.

[May99] Tobias Mayer. Only connect. an investigation into the relationship between object-oriented design metrics and the hacking culture, 1999.

[McC76] T.J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[Mis01] Vojislav B. Misic. Cohesion is structural, coherence is functional: Different views, different measures. In *Int. Sof. Metrics Symposium*. IEEE, 2001.

[MR04] Radu Marinescu and Daniel Raţiu. Quantifying the quality of object-oriented design: the factor-strategy model. In *Proceedings 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 192–201, Los Alamitos CA, 2004. IEEE Computer Society Press.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, December 1972.

[PCB92] Sukesh Patel, William Chu, and Rich Baxter. A measure for composite module cohesion. In *Int. Conf. on Sof. Eng.*, pages 38–48, 1992.

[PN06] Laura Ponisio and Oscar Nierstrasz. Using context information to re-architect a system. In *3rd Sof. Measur. Europ. Forum*, pages 91–103, 06.

[SKR08] Santonu Sarkar, Avinash C. Kark, and Girish Maskeri Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE TSE*, 34(5):700–720, 08.

[SM08] Inc. Sun Microsystems. Jsr-294: Improved modularity support in the java programming language. Technical report, Sun Microsystems, Inc., 08.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[SRK07] Santonu Sarkar, Girish Maskeri Rama, and Avinash C. Kark. Api-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE TSE*, 33(1):14–32, 07.

[SSP07] 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.

[TNM08] Ewan Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 667–691, Berlin, Heidelberg, 2008. Springer-Verlag.