

Using Contextual Information to Assess Package Cohesion

Laura Ponisio, Oscar Nierstrasz
Software Composition Group
University of Bern
Switzerland
{ponisio,oscar}@iam.unibe.ch

IAM-06-002

May 31, 2006

Abstract

Complex systems are decomposed into cohesive packages with the goal of limiting the scope of changes: if our packages are cohesive, we hope that changes will be limited to the packages responsible for the features we are changing, or at worst the packages that are immediate clients of those features. But how should we measure cohesion? Traditional cohesion metrics focus on the explicit dependencies and interactions between the classes within the package under study. A package, however, may be conceptually cohesive even though its classes exhibit no explicit dependencies.

We propose a group of contextual metrics that assess the cohesion of a package based on the degree to which its classes are used together by common clients.

We apply these metrics to various case studies, and contrast the degree of cohesion detected with that of traditional cohesion metrics. In particular, we note that object-oriented frameworks may appear not to be cohesive with traditional metrics, whereas our contextual metrics expose the implicit cohesion that results from the framework's clients.

Keywords: Packages, cohesion, software measurement, metrics, program understanding, reverse engineering

1 Introduction

Programmers usually organize large object-oriented systems by partitioning them into disjoint sets, or *packages*, where classes within the same package are conceptually interrelated. This is done to reduce complexity and to facilitate maintenance. Ideally, packages perform only one kind of task, and all their classes are related to the accomplishment of this task [21].

To achieve this, developers need guidelines for evaluating the relationship among the classes in the package. One of the most useful guidelines for evaluating this relationship in development and maintenance is cohesion, described by Fenton and Pfleeger [13] as “*the extent to which its individual components are needed to perform the same task.*”

The problem is to measure the extent to which classes put together in a package belong together. Traditionally, cohesion is regarded as an internal property [11], but cohesion can be viewed also as an external property [2][16], *i.e.*, the “purpose” of the package [19]. Such a property does not have to be represented in explicit relations between the classes of the package, an often-used basis for cohesion measurements.

A package whose classes have few class-to-class relations can still be highly cohesive. An example of this is the relationship between frameworks and client applications: the classes of a package that has as its responsibility the interaction with a framework are not directly related, but the package is still cohesive.

To capture the degree of accomplishment of the Common Reuse Principle in a package, we take as cohesion indicators the usage relationships originating from entities *outside* of the package. If two classes of the package help to fulfil the responsibility of a common client they are conceptually related, regardless of the explicit relationships that exist between them.

By applying this idea we propose Common-Use (CU), which measures package cohesion by taking into account the way a package’s classes are accessed by other packages.

Structure of the paper. In Section 2 we review the problems of defining cohesion measures useful in the context of packages in object-oriented systems. In Section 3 we present our approach in a nutshell. In Section 4 we present the model and the interactions between classes that are the bases of the measure. In Section 5 we present the new contextual metrics and give examples of their use. In Section 6 we show examples of cohesive packages related to the interaction between client and framework. In Section 7 we show the results of applying CU to the case studies. In Section 8 we elaborate with some discussion and future work. In Section 9 we refer to related work before concluding in Section 10.

2 Problems Measuring Cohesion

In 1974, Stevens, Myers and Constantine introduced the concept of cohesion in the context of structured design [23], together with the notion that increasing cohesion and minimizing coupling helps to cope with complexity.

Many metrics have been defined to compute the cohesion of a module [1][3][4][5][7][10][11][12][16][17][20][21]. The different approaches may consider a module as a set of processing elements, a class, or a cluster, depending on programming paradigm. In this paper we focus on *packages* (collections of classes) being the module concept for object-oriented languages.

Our research is motivated by the observation that traditional cohesion metrics sometimes produce low cohesion values for packages that nevertheless upon manual inspection appear to be conceptually cohesive.

A typical example is that of frameworks with packages whose purpose is to bundle tailorable functionality provided to clients. Although the classes of such packages may reveal little or no explicit dependencies, they are commonly used, referenced or subclassed together by the framework’s clients.

We take our cue from Fenton and Pfleeger’s definition of cohesion [13] (Section 1) since individual elements can also contribute to the “same task” even if they are only coincidentally associated.

3 Common-Use in a Nutshell

We address the problem of failing to detect cohesiveness in packages by analyzing the *usage* of the package’s elements.

We consider two classes to be related if there exist dependencies between them or if they are both used by the same client package. Our approach consists in identifying the classes that are not used together, *i.e.*, that do not have common clients. We take advantage of the way a package is used by others to derive implicit relations between its classes. This allows us to widen the focus and consider cohesion of a module in the context of the system in which it is used.

Example Suppose that a package P1 has two classes that are used by classes of other packages as in Figure 1. The usage dependencies in Figure 1(a) show that the two classes have different clients and since there is no direct dependency between them we predict that they don’t have much in common, making package P1 in Figure 1(a) less cohesive than P2 in Figure 1(b).

Our approach has the benefit that the proposed measures present to the developer the concrete pairs of classes which

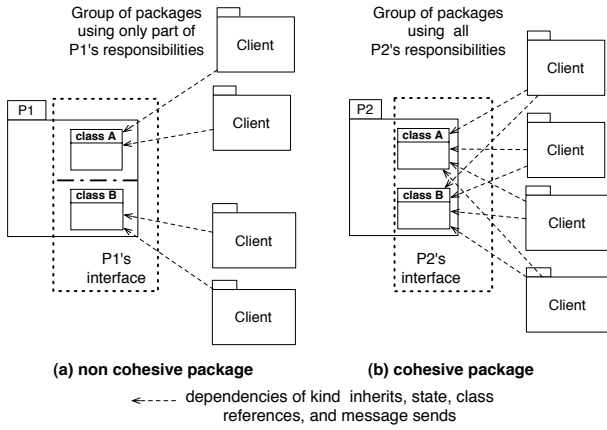


Figure 1. Deducing cohesion from the interface.

either contribute to the cohesion or lowers the cohesion of the package. This transparency can be helpful for subsequent maintenance efforts.

4 Package Interactions

Packages reflect semantic and design decisions. For example one package may contain only the classes that are open for extension by clients, another all the classes related to the user interface, and yet another all the tests. This organization is useful to understand the entire system.

Of all the kinds of interactions that we can find in object-oriented code and that could be used to analyze coupling and cohesion [7] [8] we concentrate on an essential subset, representing the strongest ties between elements of a system. Establishment of such a relation over package boundaries signifies the increase of complexity and the loss of package independence and reusability. Therefore, these four kinds of class interactions form the basis of our measures:

1. *Inheritance*: a class is a subclass of another. A subclass inherits behaviour and state from its parent. (inherits dependencies).
2. *State*: a class may directly access instance variables inherited from its ancestors. (accesses dependencies).
3. *Class Reference*: a class makes an explicit (*i.e.*, static) reference to another *e.g.*, by instantiating the class (references dependencies). Here we only consider static relationships and not run-time interactions.
4. *Message sends*: messages sent within a method of a given class cause methods of other classes to be invoked. Due to polymorphism, in dynamically typed

languages we cannot statically determine the run-time class of the target object. Our approach consists in creating a dependency between classes for each pair of candidates.

Since these different kinds of interaction may indicate different relationships between a package and its clients, we consider them both separately and in combination, thus yielding a group of closely related contextual metrics. In the case studies we will see that these metrics may yield different contextual cohesion results.

4.1 The Model

Our source model consists of classes, packages, and dependencies. To express the cohesion measures unambiguously we provide the following set-theoretic formalism.

An object-oriented system consists of a set of classes, \mathcal{C} , where A, B, C range over classes.

$$A, B, C \in \mathcal{C}$$

Let \mathcal{P} be some partitioning of \mathcal{C} , where P, Q, R range over partitions.

$$P, Q, R \in \mathcal{P}$$

There are dependencies between classes. Each dependency is of kind inherits, accesses, references, or sends.

$$\text{inherits, accesses, references, sends} \subseteq \mathcal{C} \times \mathcal{C}$$

Each dependency determines a client and provider

$$\text{depends} \subseteq \mathcal{C} \times \mathcal{C}$$

$$\text{depends} = \text{inherits} \cup \text{accesses} \cup \text{references} \cup \text{sends}$$

The *clients* of a class are the classes that depend on it:

$$\text{clients}(C) = \{A \in \mathcal{C} \mid A \text{ depends } C\}$$

A partition P may contain classes that have clients in other partitions. These classes constitute the *interface* of P .

$$\text{interface}(P) = \{C \in P \mid \text{clients}(C) - P \neq \emptyset\}$$

The classes of P that do not belong to the interface of P are *internals*.

$$\text{internals}(P) = P - \text{interface}(P)$$

5 Common-Use (CU): Inferring Cohesion from Reuse

Common-Use (CU) measures cohesion in P by taking into account the way client packages use the responsibilities of P .

The intuition behind CU is that if all the clients use the same set of classes in P , these classes contribute to the purpose of P , and therefore P is cohesive. But if some clients use a subset of classes in P and other clients use a disjoint subset, then P apparently fulfils different, possibly unrelated responsibilities, which makes it not cohesive.

Figure 1 depicts both situations: in (a) some client packages access only class A and others access only class B , indicating that $P1$ could be split, but in (b) every client accesses class A and class B , indicating that $P2$ should not be split.

5.1 Distinguishing Packages: the Need for a Weight

Not every client contributes to P 's cohesion to the same degree. For example, a package P_{client} (see Figure 2(a)) that accesses every class in the system, including the classes of P , does not tell us very much about P 's cohesion!

We find therefore the need to differentiate client packages that indicate P 's cohesion from those that don't. To achieve this we introduce the notion of *weight*.

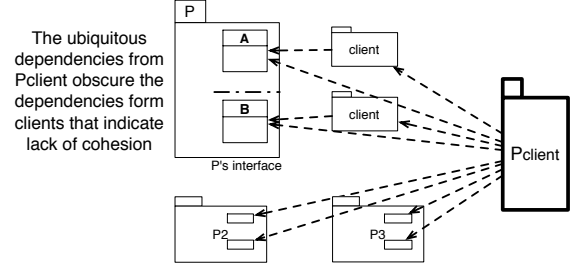
The weight contributes to lowering the cohesion of P described by CU if the *clients* of P exhibit poor procedural abstraction.

Definition 1 We define the weight of a (client) package P_{client} as the inverse of the number of connections that P_{client} has with other packages.

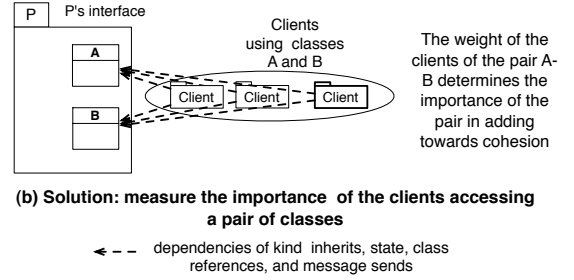
$$w(P_{client}) = \frac{1}{fan\ in(P_{client}) + fan\ out(P_{client})}$$

The weight is intended to reduce the contribution to the cohesion of P from clients that are very promiscuous in their connections to packages of the system. In particular, we do not want poorly-structured applications to "accidentally" indicate that their packages are highly cohesive simply because everything accesses everything else!

If a client package P invokes common methods which are implemented by classes everywhere in the system (e.g. 'printOn:'), then the number of fan in and fan out dependencies of this package will be high, which in turn diminishes its weight and when P acts as client pointing out cohesion of a provider Q , it reduces the CU value of Q .



(a) Problem: P_{client} indicating P cohesive when P is not



(b) Solution: measure the importance of the clients accessing a pair of classes

Figure 2. Example of the effect of ubiquitous clients in measuring cohesion (a) and the weight of clients as a solution (b)

5.2 Defining CU

Definition 2 We define Common-Use (CU) as the sum of weighted pairs of classes from the interface of a package having a common client package (f), divided by the number of pairs that can be formed with all classes in the interface.

$$CU = \sum_{a,b \in I} \frac{f(a,b) * weight(a,b)}{\#Pairs}$$

Where

$$\begin{aligned} I &= \text{interface}(P) \\ \#Pairs &= \frac{|I| \times (|I| - 1)}{2} \\ C &= \text{clients}(a) \cap \text{clients}(b) \\ f(a,b) &= \begin{cases} 1, & \text{if } C \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \\ weight(a,b) &= \sum_{c \in C} \frac{w(c)}{|C|} \end{aligned}$$

Note that if $\#Pairs = 0$ (i.e., if $|I| = 0$ or 1), then CU is undefined, since a package without at least two interface classes can neither be cohesive nor not cohesive from the point of view of its clients.

CU results in a number between 0 and 1, where 0 means that the classes of the interface have disassociated (disjoint) responsibilities, and a number close to 1 indicates that all

the classes in the interface of the package are used together by client packages.

With this definition, CU can be computed using inheritance-related interactions or non-inheritance related interactions. We therefore distinguish:

- **CUinh** (Common-Use-Inheritance) when we compute CU using inheritance-related interactions (*i.e.*, inherits and accesses),
- **CUref** (Common-Use-Reference) when we use only references, and
- **CUinv** (Common-Use-Invocation) when we use only sends.

We differentiate references and sends because references could be seen as denoting instability and because some of the invocations registered by sends are not real due to polymorphism and the static nature of our approach.

For the sake of comparison, we will also consider a coalesced CU which considers all three kinds of interactions together, and CU_{noweight} which elides the weighting function.

5.3 Case Studies

We contrasted the way that traditional cohesion metrics rated the following case studies with the results obtained using the CU group of contextual cohesion metrics.

CODECRAWLER is a code visualization tool¹, MOOSE² is an extensible language-independent reengineering environment, and SMALLWIKI is a collaborative web authoring tool³.

Each of the case studies is a mature tool developed over 3 to 6 years. They are all implemented in VisualWorks, a Smalltalk development environment. CODECRAWLER has 420 classes distributed over 10 packages, MOOSE has 480 classes in 42 packages and SMALLWIKI has 1414 classes in 42 packages. We selected these systems because they are being developed within our research group (but not by us), so knowledge about the code in general and specifically about the reasons for the organization of classes into packages is available to us. We also recognized that they reflect different programming styles.

5.3.1 Traditional Cohesion Measures

ILCO (Internal Lack of Cohesion): We name ILCO Henderson-Seller's LCOM* [16] adapted to packages by replacing class attributes and methods in LCOM* with internal classes and interface classes respectively. This measure

ranges from 0 to $(\frac{n}{n-1})$. But it is undefined when there is only one interface class and when there are no internal classes.

$$ILCO(P) = \frac{(\frac{1}{m} \sum_{j=1}^m \mu(A^j)) - n}{1 - n}$$

$$\begin{aligned} m &= | \text{internals}(P) | \\ n &= | \text{interface}(P) | \\ \mu(A) &= | \{B \mid B \in \text{interface}(P) \wedge B \in \text{clients}(A)\} | \end{aligned}$$

CR (Cohesion Ratio): If we take modules to be classes we can use Fenton's inter-modular measure of cohesion, Cohesion Ratio (CR) [13], defined as:

$$CR = \frac{\text{number of modules having functional cohesion}}{\text{total number of modules}}$$

Cohesion Ratio requires us to know if a module (class) is functionally cohesive. This is the case if its Tight Class Cohesion (TCC), defined as the relative number of directly connected methods, where two methods are directly connected if they access a common instance variable of the class, is greater than 0.7. The range of CR is 0 to 1.

IDR (Internal Dependencies Ratio): The number of direct relationships between classes divided by the maximum possible number of direct relationships. Its range is 0 to 1.

$$IDR = \frac{\text{number of direct dependencies}}{\text{number possible dependencies}}$$

Where $\frac{\text{number possible dependencies}}{\text{classes}(|\text{classes}|-1)} * 4$ is

DCO (Degree of Cohesion of Objects): This is the total fan-in (the number of client packages) for every class in the package divided by the total number of classes in the package [20]. DCO can take values from 0 to greater than 1.

$$DCO = \frac{\text{total fan-in for objects}}{\text{total number of objects}}$$

The Need for Ranking Packages to Compare Them

The values of the *different* measures cannot be compared directly. There is no reason why 0.5 in DCO should express the same level of cohesion as 0.5 in CU. In order to compare the results of the different metrics, we therefore needed to rank the packages of our case studies. We ranked the packages from most to least cohesive according to the value of the measure, rank 1 being attributed to the most cohesive package.

¹www.iam.unibe.ch/~scg/Research/CodeCrawler

²www.iam.unibe.ch/~scg/Research/Moose/

³smallwiki.unibe.ch/smallwiki

5.3.2 Comparing traditional and contextual cohesion metrics

CodeCrawler CCUI is a heterogeneous package that contains classes related to the user interface of the application, e.g., Dialogs, Viewers and an Editor.

CUref, CUinh and CUinv describe the package as *not* cohesive. It is for them one of the least cohesive packages of CODECRAWLER. There is no pair of interface classes that is actually accessed by the same client package. Unfortunately, traditional metrics give disparate values. Because they acknowledge class-to-class interactions in the package, they in some cases wrongly indicate CCUI cohesive. The following list shows of position of CCUI for the different metrics in the ranking of most cohesive packages: ILCO: 3, DCO: 3, IDR: 4, CR: 6, ICU: 8, CUref: 9, CUinh: 9, CUinv: 9

Figure 3 show a simplified schema that explains the interactions of the classes defined in CCUI.

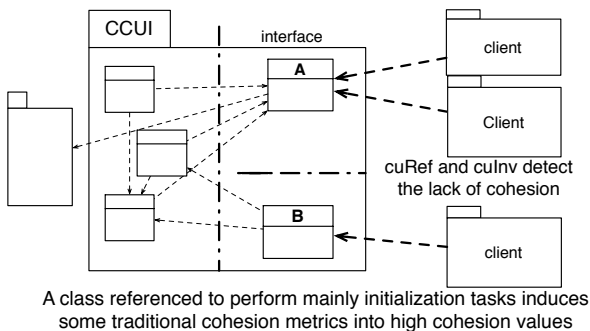


Figure 3. Example of non-cohesive package

There is a class *A* that is the central point of CCUI. *A* references or is referenced explicitly several times inside CCUI. However, these interactions are mainly initialization tasks, not the intention of the developer of communicating structure of the code, neither design decisions.

The code revealed as well that the interface of CCUI is responsible for different tasks: dialogs to let the user choose among options, a visual launcher, and a tool icon library; each of them interacting closely with a different client.

CodeCrawler CCMoose is devoted to the interaction with a framework named MOOSE, whose entities CODECRAWLER wraps in this package and exposes to its clients in the form of loosely-coupled plug-ins (classes defined in CCMoose). These classes need intensive testing because the client applications of CODECRAWLER heavily rely on them.

Our metrics involving the use of the package indicate cohesion regarding *sends* interaction and less cohesion regarding *references*.

This shows that CUinv detects the uniformity of the plug-in classes present in the interface of CCMoose because most of them are invoked uniformly by two low-coupled packages.

Lowering cohesion, CUref detects that the classes defined in CCMoose *referenced* from client packages form two groups. CUref is not high, but slightly higher than the average, meaning that CCMoose is cohesive regarding the application since many packages have two responsibilities. Analysis of the code confirmed that by revealing that most classes were targeted only by the test package and some others by a package of CODECRAWLER developed later.

A combined CU that considers all the interaction types indicates that it is cohesive due to the difference between the amount of invocations (1255) and references (35).

However, most of traditional metrics indicate lower values of cohesion than the average, a consequence of relying on internal dependencies almost non-existing in CCMoose.

Moose MoosePropertyOperators contains classes that follow one of two patterns: either they are operators, or they are operator factories. This is the place where developers of applications client of MOOSE look when they are working with operators.

Traditional metrics such as ILCO inherits, ILCO references, ILCO sends, and IDR do not detect this cohesion, because of the few explicit internal dependencies between the classes.

Nevertheless, CUref and CUinv mark MoosePropertyOperators as the most cohesive package of MOOSE. The reason why is that the methods here are accessed in a uniform way, a design decision enforced by the fact that the operator classes have methods with the same selector. Figure 4 depicts this situation.

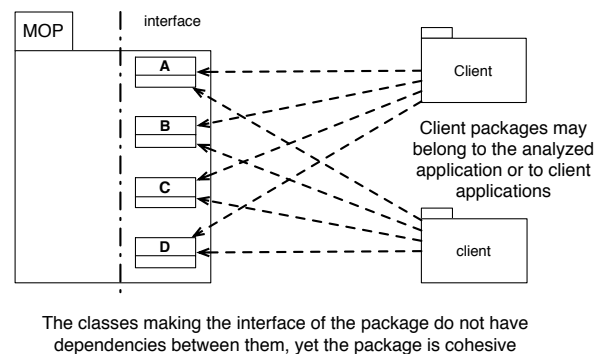


Figure 4. Example of cohesive package.

SmallWiki Visitor This package contains 9 classes which implement visitors that traverse the page tree of the

Metric	CCUI	CCMoose	HOTDRAW		Moose P. Op.	CC avg.	Server Swazoo	Visitor	SW avg.
			isolated	in context					
Manual	not cohesive	cohesive	not cohesive	cohesive	cohesive	-	not cohesive	cohesive	-
CUinh	undef.	undef.	1	0.0257601	undef.	0.023	undef.	undef.	0
CUref	0	0.022	undef.	0.0260135	0.0322581	0.021	undef.	0.105014	0.107759
CUinv	0.0283938	0.0245893	0.0262269	1	undef.	0.02802	undef.	0.0700096	0.103575
CU _{noweight}	1.0	0.858696	1.0	0.919328	1	0.89	1	0.466667	0.68695
ILCO ref.	0	0	0	0	0	0.238	undef.	0	0.42
ILCO inh	0	undef.	0	0	0	0.0734	0	undef.	0.18
ILCO inv	undef.	0	0	0	0	0.404	0	0	0
CR	0.166	0	0.578947	0.578947	0	0.163	0.5	0.111111	0.208783
IDR	0.155	0.1175	0.447368	0.447368	0.2147	0.271	0.75	0.284722	0.151286
DCO	0.48	0.68	0.5	0.78	0.5	0.399	0.0625	0.375	0.225
num. inc. inh.	0	1	6	6	0	3.47826	0	1	0.230769
num. inc. acc.	0	0	0	160	0	9.0	0	0	0
num. inc. ref.	10	35	0	36	3	81.5333	1	15	81.4615
num. inc. inv.	255	1255	0	3181	1470	1536.37	0	0	0

Table 1. Package Measures.

website. This crucial functionality is used throughout the system. Most traditional metrics signal the package as non cohesive, but CU and DCO rightfully indicate the opposite because its classes are accessed by packages where the visited resources are. The visitor pattern [14] followed here, with its visiting classes calling back, is acknowledged by CU. In general, where the visitor package is implemented, CUinv indicates high cohesion.

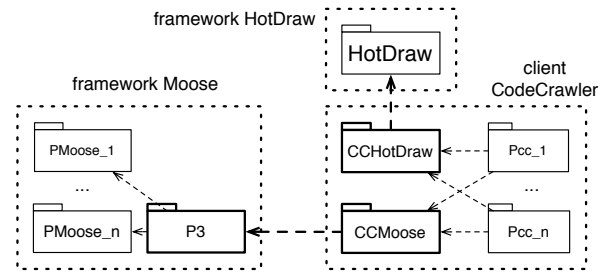
6 Package Cohesion in the Context of the Client and Framework Interaction

When the designers organize a system by grouping classes into packages, those packages appear that contain classes with no interaction between them, but that do intuitively belong to the package.

In this section we show examples of such packages from a case study involving two frameworks HOTDRAW⁴ [6] and MOOSE, and a client application CODECRAWLER.

As we concentrated our attention on the cohesion and coupling of the packages, we observed that packages that seemed to be *not* cohesive when looked at in isolation revealed themselves to be cohesive when observed in the context of being used by their clients.

We analyzed the interaction and the cohesion of the packages of CODECRAWLER, HOTDRAW and MOOSE in two contexts: firstly observing only the client (CODECRAWLER), *i.e.*, not counting the interactions with MOOSE and HOTDRAW, and secondly including the interactions between CODECRAWLER, MOOSE, and HOTDRAW. Figure 5 represents this application.



Client CodeCrawler connecting to both frameworks through dedicated packages CCHotDraw and CCMoose

Figure 5. Client application with packages devoted to interact with its frameworks.

6.1 Implicit Framework Cohesion

We suspected that a framework package P that appeared not to be cohesive when examine in isolation, could turn out to be cohesive from the context of the framework client.

Consequently, we searched for an example in the framework HOTDRAW, consisting in one single package that is not particularly cohesive according to traditional metrics. The column "isolated" below HOTDRAW shows the values of the metrics in Table 1. We observed that HOTDRAW is a package most actively used in the interaction with the client CCHotDraw (*e.g.*, 5 of its 16 classes inherit from HOTDRAW) without having a particularly high number of internal class-to-class interactions compared to the other packages (IDR = 0.44).

When we measured HOTDRAW while it was being used by a client, including CODECRAWLER and MOOSE, it turned out to be the 9th *most* cohesive package, out of 49. It

⁴st-www.cs.uiuc.edu/users/brant/HotDraw/HotDraw.html

did make sense to put all the classes of the framework that would be used by clients in the same package. HOTDRAW was cohesive, confirming our suspicion.

6.2 The Non-Cohesive Package With a Reason

CCHotDraw, has lower CUref cohesion than the average, considering CODECRAWLER in isolation. It is, in fact, among the packages with the lowest cohesion.

What happened here is that the developers intended this package to be the container of all the interactions with the framework HOTDRAW, that being its *role* and only purpose. With its classes having only that property in common, CCHotDraw provides different functionalities consumed by 8 of the 10 packages in CODECRAWLER, and then has low cohesion.

6.3 The Expected Cohesive Package

CCMoose was created to be the package extending the framework. Its role is to contain the plugins for the classes of MOOSE. But besides that, and the fact that have many internal class-to-class dependencies, CCMoose is cohesive for the average. The reason behind is that it has “invisible” clients in the clients of CODECRAWLER and explicit clients in two packages of CODECRAWLER, one of which is a test package which reproduces what the clients *use* of this package.

7 Validation: Package Usage Hits

To validate a metric that captures if the classes in a package belong together using traditional Fenton’s [13] or Zuse’s validation [25] is very difficult because the concept of cohesion is loosely understood [5] and dependent on the context.

Instead, we analyzed on the one hand the usefulness of CU to describe cohesion in packages and on the other hand how many times the cohesion values obtained with our metrics agreed with the opinion of a person analyzing the code. We discussed usefulness of CU to extract insights about the code in Section 5 and in Section 6.

In this section we assessed the capability of CU to measure cohesion compared against traditional metrics. After presenting the experimental setup, we explain the results obtained.

We considered four case studies: CODECRAWLER, DUPLOC and ALCHEMIST. All of the systems are written in VisualWorks Smalltalk [24]. ALCHEMIST has 207 classes in 8 packages, and DUPLOC has 1059 classes distributed in 23 packages.

Tool	Man.	CU	ILCO	CR	IDR	DCO
Code Crawler	100	80	20	20	40	50
DUPLOC	100	78	39.5	50.5	30.5	55
ALCHEMIST	100	75	37.5	62.5	25	25

Table 2. Success rates of CU and traditional metrics for each system.

We analyzed and classified forty-one packages of four systems. We classified each package as cohesive or not cohesive, and supported the classification with a brief description of the responsibilities of each package, as we understood from the code. We were unaware of the values of the measures, but had a clear definition of cohesion that coincided with the one presented in this paper.

Separately, we classified the forty-one packages according to the values obtained from applying the proposed and traditional measures to the case studies and different thresholds.

We define a *hit* as an instance that the classification of the package into cohesive and not cohesive done with a measure coincides with the manual classification.

We chose as thresholds the average of the values obtained from applying the measures.

If our proposed measures indicate functional package cohesion better than traditional approaches, then the number of hits obtained with CU would be larger than the number of hits obtained by using the traditional measures.

The experiments showed that the packages were most effectively classified by CU in the proposed case studies, though no statistical tests was applied to the results. Table 2 shows the percentages of hits obtained for each tool and for each measure.

For instance, the first row shows that CU’s classification of CODECRAWLER’s packages coincided 80 % with the manual classification, meaning that 8 out of 10 packages were counted as a hit.

Two main threats to the validity of the results of this experiment that we identified are: a) low researcher’s variability (only two researchers performed the manual classification), and b) restricted notion of cohesion to the one used in this paper.

8 Discussion

In this section we discuss some matters for consideration that the characteristics of our approach arise.

Measuring the Relevance of a Client We have chosen the method of weighing client packages using their number of external connections to compute the weight of client

packages. For reasons of space we concentrate in this work only on this method. However, the weight assigned to a client package could be set to any suitable heuristic, i.e., a cohesion, complexity or coupling measure, or even determined manually.

To assign manually a weight to all the packages P could only work in small or medium sized systems, but in big applications, to assign weight manually to every package is not realistic. Future work includes, to weight manually *certain* client packages that are structural or by special design, and to develop measurements to detect outliers.

The Weight of Client Packages A heavily accessed cohesive package could be signaled as having low cohesion. Consider the case of a package with two classes where the clients access both classes. It is intuitively cohesive. Promiscuous clients may distort the apparent cohesion of a package, and may need to be filtered out.

Different Measures for Different Class-interaction Types Inheritance based coupling, which refers to coupling between a class and its ancestors, and non inheritance-based coupling, coupling between a class and a class that is not related to via inheritance (neither ancestor neither descendant) [8] do *not* contribute in the same way to cohesion. Inheritance cohesion provides a view of the hierarchical structure of the system, but it is purely structural and it doesn't mean that it is used. Besides, if `CUinv` marks the package as cohesive, we do not know (with a static approach) if the interactions are actually used.

Properties of the Code that influence Common-Use `CU` can produce low cohesion values in highly cohesive packages. Consider the case where an abstract class is subclassed by concrete classes. Clients seem to access two groups of classes when in fact the package was designed as cohesive.

A similar problem appears under the presence of the pattern facade, and yet another when the analyzed package has only one class accessed by clients and this class is a God class. The package with only one class could represent multiple responsibilities or not, but this escapes the granularity level of our approach. For reasons of space, we leave the analysis of the class usage relationship for future work. Finally, experiments showed that the visitor pattern gives accurate results, though.

9 Related Work

There exist many cohesion measures for modules in structured programming and classes in object-oriented systems which can eventually be extended to packages. However, there are few cohesion measures devoted to packages

as sets of classes [20] [13] [1]. Emerson presents a measure to compute cohesion applicable to modules in the sense of Pascal procedures [12]. His measure is based on a graph theoretic property that quantifies the relationship between control flow paths and references to variables. Bieman and Ott compute cohesion using a slice abstraction of a program based on data slices [5]. Patel et al. [21] compute the cohesion of Ada packages based on the similarity of its members (programs).

In object-oriented programming, Chidamber and Kemerer propose a measure for class cohesion named LCOM [10] [11], criticized and improved by Henderson-Sellers's LCOM* [16]. Hautus [15] provides `Pasta`, an indicator of the quality of the package architecture in Java. Allen et al. define information theory-based (as opposed to counting) coupling and cohesion measures for subsystems [1].

Bieman and Kang's TCC [3] measure cohesion for classes. It assesses cohesion using the number of pairs of methods in a class that access common instance variables. They provide an intramodule cohesion measure for cohesion based on the design level information [4].

Misic adopts a different perspective and measures the cohesion of a package as an external property of a module [19]. Following an approach closer to ours, he claims that the internal organization of a module isn't enough to determine its cohesion. Morris follows this line by computing module cohesion considering the fan-in of the contained objects [20].

In the software clustering area, Anquetil proposes Modularization Quality (MQ). Together with the tool Bunch[18], MQ uses the dependencies between modules of two distinct subsystems and the ones between the modules of the same subsystem (in that differs from ours) to determine the cohesion of clusters and to reward the creation of highly cohesive ones.

Schwanke's information sharing heuristic [22] "*If two procedures use several of the same unit-names, they are likely to be sharing significant design information*", used to compute a similarity metric between two procedures, is analogous to the package use heuristic that we use to compute `CU`. The innovation of our approach regarding Schwanke's similarity metric is that `CU` captures the extent to which the functionality *exposed* by a package is related, while Schwanke uses the interconnections where a package is client as well as the ones where it is provider. Packages devoted to test classes or that contain root and constant classes have low cohesion values considering Schwanke's metric, but the developers find them cohesive as well as `CU`.

Apart from measurement procedures that require the knowledge of the programmer, none of the measures, to the best of our understanding, can detect cohesion when the relations that bind the module's elements together are not explicit, which causes traditional measures to falsely rate

specific packages as not cohesive.

10 Conclusion

We have presented an approach to measure package cohesion based on client usage rather than explicit dependencies within a package. We abstract from the implementation of the package to measure cohesion from the functionality exposed by the package. The more related this functionality, the more cohesive the package. To see how related its functionality is, we measure the classes that are reused together. The more its classes are reused together, the more cohesive the package.

We argue that the role of the package gives its sense of cohesion. Furthermore, the role of the package should indicate which measure to use.

Our measure, Common-Use (CU), considers cohesion to be an external property based on the interface that the package presents to the world and the way the rest of the packages use that interface.

To the best of our understanding, this is a novel approach based on the idea that similarity of the object can be derived from the intersection of its sets of properties [9] and that external attributes can also indicate cohesion [13].

CU showed in our experiments to be consistently better than selected traditional metrics in capturing the cohesion of packages when compared with the criteria of the evaluator.

We have showed with examples how CU captures the intuitive idea of cohesion. Metrics focused only in class-to-class interactions failed to mark as cohesive packages designed to abstract clients from their framework. We showed in this way a leap of faith in the design of applications. Namely, the commonly accepted belief that good design implies low coupling and high class-to-class measured cohesion [23]. Indeed, in the case of client-framework interaction, packages in well designed applications can still be cohesive even though not complying with the aforementioned belief.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “RECAST: Evolution of Object-Oriented Applications” (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

References

- [1] E. Allen and T. Khoshgoftaar. Measuring coupling and cohesion of software modules: An information theory approach. In *Seventh International Software Metrics Symposium*, 2001.
- [2] E. V. Berard. *Essays On Object-Oriented Software Engineering*, volume 1. Prentice-Hall, 1993.
- [3] J. Bieman and B. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings ACM Symposium on Software Reusability*, Apr. 1995.
- [4] J. Bieman and B.-K. Kang. Measuring design-level cohesion. *IEEE Transactions on Software Engineering*, 24(2):111–124, Feb. 1998.
- [5] J. Bieman and L.M.Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–658, Aug. 1994.
- [6] J. Brant. Hotdraw. Master’s thesis, University of Illinois at Urbana-Champaign, 1995.
- [7] L. C. Briand, J. W. Daly, and J. Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.
- [8] L. C. Briand, J. W. Daly, and J. K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.
- [9] M. Bunge. *Treatise on Basic Philosophy: Ontology I: The Furniture of the World*. Riedel, 1977.
- [10] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 197–211, Nov. 1991.
- [11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
- [12] T. Emerson. A discriminant metric for module cohesion. In *Proceedings of the 7th International Conference on Software Engineering (ICSE)*, 1984.
- [13] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.
- [14] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.
- [15] E. Hautus. Improving Java software through package structure analysis. In *International Conference Software Engineering and Applications*, 2002.
- [16] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

- [17] A. Lakhota. Rule-based approach to computing module cohesion. In *Proceedings 15th ICSE*, pages 35–44, 1993.
- [18] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.
- [19] V. B. Mišić. Cohesion is structural, coherence is functional: Different views, different measures. In *Proceedings of the Seventh International Software Metrics Symposium (METRICS-01)*. IEEE, 2001.
- [20] K. Morris. Metrics for object-oriented software development environments. Master's thesis, Sloan School of Management. MIT, 1989.
- [21] R. B. S. Patel, W. Chu. A measure for composite module cohesion. In *Proceedings of the 14th International Conference on Software Engineering*, pages 38–48, 1992.
- [22] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.
- [23] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.
- [24] Cincom Smalltalk, Sept. 2003. <http://www.cincom.com/scripts/smalltalk.dll/>.
- [25] H. Zuse. *Software Complexity, Measures and Methods*. Walter De Gruyter, 1990.