The Case for Non-Cohesive Packages

Nicolas Anquetil, Muhammad Bhatti, Stéphane Ducasse, André Hora, Jannik Laval

The Case for Non-Cohesive Packages

NICOLAS ANQUETIL, MUHAMMAD USSMAN BHATTI, STÉPHANE DUCASSE, ANDRE HORA and JANNIK LAVAL, Inria - Lille Nord Europe – Cristal - University of Lille – CNRS

While the lack of cohesiveness of modules in procedural languages is a good way to identify modules with potential quality problems, we doubt that it is an adequate measure for packages in object-oriented systems. Indeed, mapping procedural metrics to object-oriented systems should take into account the building principles of object-oriented programming: inheritance and late binding. Inheritance offers the possibility to create packages by just extending classes with the necessary increment of behavior. Late binding coupled to the "Hollywood Principle" are a key to build frameworks and let the users branch their extensions in the framework. Therefore, a package extending a framework does not have to be cohesive, since it inherits the framework logic, which is encapsulated in framework packages. In such a case, the correct modularization of an extender application may imply low cohesion for some of the packages. In this paper we confirm these conjectures on various real systems (JHotdraw, Eclipse, JEdit, JFace) using or extending OO frameworks. We carry out a dependency analysis of packages to measure their relation with their framework. The results show that framework dependencies form a considerable portion of the overall package dependencies. This means that non-cohesive packages should not be considered systematically as packages of low quality.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**] Metrics; D.2.9 [**Software Engineering**] Software quality assurance

1. INTRODUCTION

Cohesion and coupling principles have been first defined for procedural languages with a black box model of control flow in mind [Stevens et al. 1974; Yourdon and Constantine 1979]. These principles state that a module should have high (internal) cohesion and low (external) coupling so that it implements a well defined functionality and can be easily reused [Briand et al. 1998; Briand et al. 1999].

We argue that when porting the principles to the object-oriented paradigm, some intrinsic properties of the object-oriented paradigm got ignored. We claim that packages with low cohesion are not necessarily packages with low quality. This is due in particular to the presence of inheritance and late-binding, which are the cornerstones of incremental definitions of classes. Classes in one package can extend the behavior of classes defined in other packages and just define a small increment in functionality. Late-binding is also the key mechanism to build white-box frameworks [Pree 1995] using the *Hollywood Principle* ("don't call us, we'll call you"). The application of this principle leads to *the situation where an extending package may exhibit low cohesion and still be a well designed package*.

In this paper, an experiment is performed on various software systems to understand package cohesion and coupling when they are developed as extensions of frameworks. The analysis is performed by studying package dependencies of extender applications with their frameworks.

The contributions of the paper are the following: (1) identification of a common cohesion/coupling misunderstanding for object-oriented programming; (2) a validation of the hypothesis on several real cases; and (3) a simple model to represent cohesion/coupling in a context of frameworks where no

email: nicolas.anquetil@inria.fr, stephane.ducasse@inria.fr

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, 11-13.9.2017, Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

universally accepted metrics are available (e.g. [Allen and Khoshgoftaar 2001; Bieman and Kang 1995; Ponisio 2006]).

The rest of the paper is structured as follows: Section 2 describes the building principles of OO frameworks and Section 3 presents limitations of existing coupling and cohesion metrics. Section 4 describes our experimental settings. The results are presented in sections 5 and 6. Section 7 discusses the experiment and the results of the paper and Section 8 presents the work related to this study. Finally, Section 9 concludes the paper.

2. LATE-BINDING AND FRAMEWORKS

Late-binding is the fact that the receiver of a message is used to determine the method that will be executed in response to this message. It is the cornerstone of object-oriented programming. Late-binding is the key mechanism behind the Hook and Template Design Patterns [Gamma et al. 1995] and the building of white box frameworks [Fayad et al. 1999; Pree 1995; Roberts and Johnson 1997].

A framework is a set of interacting classes (possibly abstract) defining the context and the logic of behavior. Frameworks are reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate [Roberts and Johnson 1997].

Frameworks are often built around a core package with interfaces and abstract classes. Such core classes interact among themselves to define the logic of the domain and they exhibit hook methods that extender applications will be able to specialize (see Figure 1). These classes can be subclassed/specialized in other framework packages but they are also extended by concrete extender applications.

The idea of cohesive modules is based on the notion of blackbox reuse (Figure 1, left). Inheritance is a white box reuse mechanism, which means a module can be plugged into another module (from which it inherits) without specifying its whole behavior [Fayad et al. 1999; Pree 1995; Roberts and Johnson 1997]. In particular OO frameworks are based on the *Hollywood Principle* ("don't call us, we'll call you"), which supposes that the extender application can "inject" code into the underlying framework through inheritance (Figure 1, right). In this case, the extender application (class D) may call a method offered by an internal class (class C) but actually provided by a framework class (class A). Conversely, the framework (class B) may send a message to an instance that ends up being implemented in the extender application (an instance of class C). Bieman and Kang [Bieman and Kang 1995] recognize two ways to reuse a class: by instantiation and by inheritance. Similarly, we consider that a framework class can be used in these two ways. Defining what is a cohesive module in this case is more complex than for black box reuse.

We will distinguish two layers in the systems studied:

—**Provider Framework**: (abbreviated f/w) any software application that is *extended* to create another software system. In this sense, the provider framework could be a complete software application.

—**Extender Application**: the application that is being analyzed, this is the subject of the experiment. We consider that the extender application is developed using framework classes.

Working hypothesis. There exists in the common perception of package cohesion a rampant misunderstanding: *In an object-oriented language, a well designed package does not have to be necessarily cohesive*.

Our claim is that in the presence of late-binding, weakly cohesive packages are not necessarily of bad quality. Over this article we will refine this claim and show evidence of this fact.

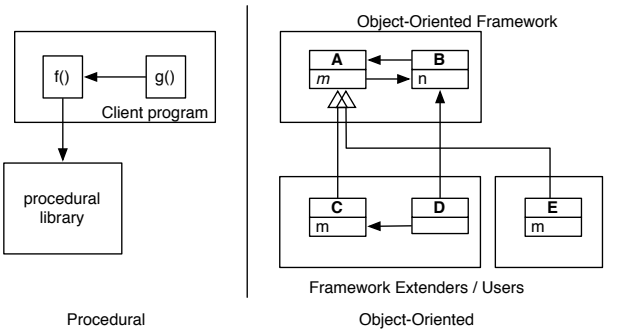


Fig. 1. Differences in accessing an underlying library (or framework). In the procedural world (left) the accesses are well known and controlled; in the OO world (right), inheritance and late binding may branch extender classes into the framework, creating new interactions between them.

3. COHESION AND COUPLING OF PACKAGES

Stevens *et al.* [Stevens et al. 1974], 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". From then on, the mantra of good design has been that a module (package, class, function) must exhibit high cohesion and low coupling. The importance of this belief in the software engineering culture might be measured by the quantity of work on the topic of measuring cohesion and coupling. In OOP (Object-Oriented Programming) for example, this is attested by the study of Briand *et al.* [Briand et al. 1999].

In this section, we review existing metrics at the package level and show that they are based on dependencies between the classes of the packages. We will further show that many metrics at the class level are also based on dependencies and those that are not, cannot be made to measure cohesion and coupling at the package level. We will conclude that we are lead to use a metric based on dependencies. We will further examine the existing package metrics and highlight some shortcomings that drove us to dismiss them. These conclusions will serve as bases to choose cohesion/coupling metrics in Section 4.

3.1 Cohesion and coupling at package level

At the level of packages (or groups of classes) the following publications were found to propose cohesion/coupling metrics:

—COF (Coupling Factor) [Brito e Abreu et al. 1995] is a metric defined at program scope, it is a normalized ratio between the number of client relationships and the total number of possible client relationships of the classes. A client relationship exists whenever a class references a method or attribute of another class.

—Bunch [Mitchell and Mancoridis 2006] is a tool that remodularizes automatically software, based on the metrics of module cohesion and coupling defined by the approach. Similarly to COF, cohesion and coupling are defined as a normalized ratio between the existing dependencies of a package's classes and the maximum number of such dependencies, but Bunch considers incoming and outgoing dependencies.

—Ponisio *et al.* [Ponisio and Nierstrasz 2006] propose a technique to measure package cohesion by analyzing how client packages access the classes of a given provider package.

—A similar approach is presented in [Mišić 2001] that calculates package cohesion based on the fan-in of the contained objects.

—Abdeen [Abdeen 2009] presents a set of metrics to capture the quality of a package in the context of an existing structure (set of packages).

—Bavota *et al.* propose to analyze the structural and semantic relationships between classes in a package to define new packages with higher cohesion [Bavota et al. 2010].

—With Relational Cohesion [Martin 2002], Martin defines the cohesion of packages as the average number of internal relationships per class, and efferent coupling looks at the number of classes outside the package that classes inside depend upon (Afferent coupling is the number of classes external to the package which depend upon classes in the package).

We conclude that, at the package level, all metrics are based on connectivity between classes inside the package with classes in the same package (cohesion) or in other packages (coupling). This connectivity is computed from dependencies between the classes.

3.2 Cohesion and coupling at class level

There has been lot of work on cohesion/coupling at the class level too:

—With CBO (Coupling Between Object) [Chidamber and Kemerer 1994], 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.

—MPC (Message Passing Coupling) [Li and Henry 1993] is defined as the "number of send statements defined in a class". The authors further refine the definition by indicating that calls to the class's own methods are excluded from the count, and that only calls from local methods are considered, excluding calls in inherited methods.

—The highly debated LCOM (Lack of cohesion of Method) and followers LCOM* [Briand et al. 1998; Chidamber and Kemerer 1994] try to capture class cohesion by measuring how well methods access the class state.

—TCC (Tight Class Cohesion) [Bieman and Kang 1995], 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.

—C3 (Conceptual Cohesion of Classes) captures the semantic information in code for cohesion computation *i.e.,* textual similarity amongst methods [Marcus et al. 2008].

From this, we conclude that, again at the class level, cohesion/coupling metrics are based on some count of dependencies, internal for cohesion, external for coupling: CBO: two classes are coupled if one uses the other; MPC: number of send statements (*i.e.,* method calls).

Some diverging metrics, consider shared dependencies (called sibling dependencies in [Anquetil and Lethbridge 2003]) instead of direct dependencies: TCC: number of methods connected with other methods through an instance variable; and LCOM*: how well methods access the class state And a final one is based on textual similarity amongst methods: C3.

However, class metrics cannot easily be aggregated at the level of package: a package containing highly cohesive classes could be non-cohesive if each classes deal with a specific topic. A package containing highly coupled classes could be lowly coupled if its classes were all coupled together and not with other packages.

3.3 Cohesion and coupling from a higher point of view

We showed that cohesion/coupling is typically measured from the dependencies that software components have within themselves (cohesion) or with the outside (coupling). Even the research taking a higher point of view on the problem and trying to understand what is meant by cohesion/coupling or design quality (*e.g.,* [Abreu and Goulão 2001; Bhatia and Singh 2006; Counsell et al. 2005; Taube-Schock et al. 2011]) fall back to the same basics: they measure the connectivity between members of the packages.

—Brito e Abreu and Goulão [Abreu and Goulão 2001] and Bhatia and Singh [Bhatia and Singh 2006] do criticize the hegemony of the "high cohesion, low coupling" mantra, but they still define module cohesion as intra-modular class coupling and module coupling [. . .] as inter-modular class coupling, with 12 possible class coupling measures (direct inheritance, class parameter, attribute type, message recipient, . . .) that can all be summarized as: one member of a class needs to access another class or another class' member.

—Counsell *et al.* tried to correlate some software metrics with human perception of class cohesion (for a small set of classes) [Counsell et al. 2005]. The metric that gave best result for expert software engineers was NAS: number of associations. "This metric includes coupling due to inheritance and [. . .] aggregation, the return type of a method or the parameter of a method."

—Finally, Taube-Schock *et al.* argue that high coupling cannot be avoided due to the power-law distribution of the connectivity metric [Taube-Schock et al. 2011]. "Connections between source code entities are modelled as links between nodes; these include parent/child relationships, method invocations, superclass/subclass relationships, super-interfaces, type usage, variable usage, and polymorphic relationships." Again, they use some sort of dependency between classes to compute coupling at a higher level.

4. EXPERIMENT SETTING

Having reviewed existing possibilities to compute cohesion and coupling of packages, we now plan the details of our experiment according to the experimental setup suggested in [Wohlin et al. 2000].

4.1 Experiment definition

Analyze packages
with the purpose of comparing
with respect to their internal and external dependencies
from the point of view of researchers
in the context of real world, framework based, OO applications.

4.2 Context and subjects selection

The *context* of the experiment will be packages from real, well-designed, OO systems, which are based on a framework and for which the source code is available. The restriction to OO systems based on framework is dictated by our hypothesis. We need real systems to ensure that our experiment is meaningful.

To avoid unexpected bias in the results, it is considered best if subjects are selected randomly inside the entire population. It would be, however, difficult to identify exhaustively all systems that fit the context of our experiment. We will, therefore, rely on convenience sampling.

A difficulty of this research is to better define good and bad modularization. We cannot rely on traditional design metrics ("high cohesion, low coupling") because our goal is precisely to show that good

modularization may result in low cohesion. So, we will resort to qualitative analysis of the applications and frameworks studied based on their developers opinion to consider they are well modularized although they might exhibit low cohesion. We will then assume in our formal hypothesis that the quality of the modularization is established, and we will explain how we establish it for each subject system.

We selected the following systems that are known to be based on some framework and for which we found some arguments as to their adequate architectural design: JHotDraw, Eclipse, and JEdit.

JHotDraw[1] is itself a framework for structured, two-dimensional, drawing editors. It is developed in Java and is based on the Swing+AWT framework. We analyzed a recent version (*i.e.,* 7.6; 2011-01-09). In the experiment, we will exclude the `samples` package because it consists of small example applications developed to demonstrate the usage of the JHotDraw framework. As such they do not fit the experiment context. This system is considered well structured for two reasons:

—From its first version, HotDraw (the Smalltalk version) was developed as a "design exercise". For example, it relies heavily on well-known design patterns. As a consequence, particular attention has been paid on its good design.

—Several notes in the documentation[2] explicitly mention restructurings of the application: v. 7.0.4 (2006-06-25) "Reorganized package structure"; v. 7.0.7 (2006-11-12) "Reorganized file structure"; v. 7.5 (2010-07-28) "Some changes in the package structure and renamings of classes have been made in order to improve the clarity of the frameworks."

Eclipse[3] is an open source platform that is mainly known as an IDE for various languages. We will restrict ourselves to the user interface part (org.eclipse.ui) based on two toolkits: JFace and SWT. To ensure that the application has a good design, we selected version 3.0 of Eclipse (June 2004). This version was the first of the Eclipse Rich Client Platform and was the result of a large restructuring effort [4].

JEdit[5] is a famous text editor for programmers. It is well known in research and served as a test bed for many experiments. Again, we will concentrate only on two specific packages of JEdit that compose the GUI part, `org.gjt.sp.jedit.gui` and `org.gjt.sp.jedit.gui.statusbar`. Because they are only two packages with a clear goal (documentation for "gui" states: "Various GUI controls and dialogue boxes"), we will suppose they are correctly designed in the sense that they don't contain code not related to GUI features.

We give in Table I (page 10) some indication on the size of the three subject systems.

4.3 Cohesion and coupling metric selection

We want to study whether, under particular conditions, packages could be well designed although exhibiting poor cohesiveness. But we must first understand that there is no absolute value of a high or low cohesion. There is little meaning in a fully cohesive or fully non-cohesive package. All real, well designed, OO packages will most probably present some level of internal cohesion as well as some coupling with the rest of the system in order to achieve something of significance. How can we measure poor cohesiveness in such situation?

We can define no absolute threshold and there is no known per-system threshold either. We propose to work at the package level, considering that a package does not exhibit "high cohesion and low

[1] http://www.jhotdraw.org/
[2] http://www.randelshofer.ch/oop/jhotdraw/Documentation/changes.html
[3] http://www.eclipse.org/
[4] http://eclipse.org/rcp/generic_workbench_structure.html
[5] http://www.jedit.org/

coupling" if its classes, collectively, are more coupled to the outside than to its own components. Conceptually, it is debatable whether this requirement is meaningful at all. One can argue that the two notions are independent. But we showed (Section 3) that the state of the practice usually bases the two metrics on some measure of dependencies between the classes. This makes them very close and even interdependent. One can argue that, given a constant amount of dependencies in a system, if the cohesion of a module increases, its coupling must decrease. In this practical view of cohesion and coupling, it makes sense to compare the two metrics, providing their definitions are based on the same dependencies and they are expressed in the same unit.

As for the metrics themselves, we already explained why existing cohesion/coupling metrics at the class level could not be used at the level of package (Section 3.2). We will also have to reject most of the existing metrics at the level of package:

—The metrics proposed by Ponisio *et al.* [Ponisio and Nierstrasz 2006] or Mišić *et al.* [Mišić 2001] are cohesion metrics without an associated coupling metric. So they do not satisfy our requirement of two comparable metrics.

—Martin's afferent coupling, Abdeen metrics and Bunch metrics use incoming dependencies. This would bias the results against coupling since extender application classes cannot have incoming static dependencies from provider framework classes (see in Section 4.5 why we use static analysis).

—Bunch cohesion and coupling metrics were also found to depend too much on the size of the packages [Anquetil and Laval 2011]. Because coupling normalizes the number of external dependencies found by the maximum number of possible dependencies (ratio of number of external dependencies on number of classes inside times number of classes outside the package), it is impacted by the size of the system: the larger the system, the smaller the coupling.

We are left we two candidates: Martin's Relational cohesion and Efferent coupling. Unfortunately, they are not expressed in the same unit, Efferent coupling counts the number of external dependencies, while Relational cohesion is the average number of dependencies of the package's classes. We propose to use Efferent coupling as defined by Martin and Relational cohesion multiplied by the number of classes of the package, *i.e.,* raw number of dependencies between classes within the package. Because we will be comparing between themselves cohesion and coupling of each package, we could as well have used the other solution (pure Relational cohesion and averaged Efferent coupling), the result of the comparison between the two metrics is independent of the package size in both cases. Note that this is very different of the Bunch metrics for which the coupling also depends on the size of the entire system, whereas the cohesion does not (reason why we rejected these two metrics).

The class dependencies we will consider are: inheritance between classes, invocation between methods, access to class attributes, or explicit references to classes (*e.g.,* in the case of a "new"). We will consider dependencies between classes as a boolean property, without considering the possible strength of that dependency. That is to say, one class could inherit from another, call several of its method, access its attributes, we would still count only one dependency.

4.4 Variable selection and Hypothesis formulation

We will use as *dependent variable* the raw number of dependencies from classes within the studied packages.

The *independent variable* will be the destination of the dependencies (see Figure 2):

—**Local Dependency:** The dependency targets a class that resides in the same package as the referencing class. This corresponds to cohesion.

—**Framework Dependency:** The dependency targets a class that resides in the framework. This corresponds to coupling with the provider framework.

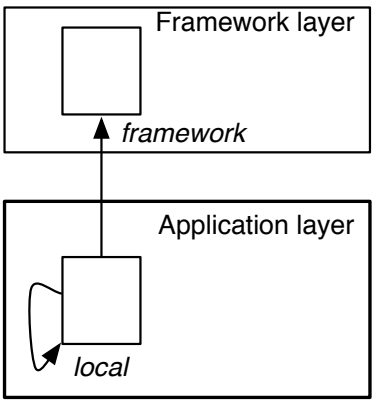


Fig. 2. The two kind of dependencies we consider in our experiment: small squares are packages and large boxes represent "layers" (see also Figure 1). Here we only consider local dependencies in the context of the application layer (see text).

We already introduced informally the experiment hypothesis, saying that we want to test whether package cohesion is inferior to package coupling. This is a strong hypothesis because one could assume that even if the cohesion of a package were only a little superior to its coupling, it would not qualify as "high cohesion, low coupling". We will actually need to make it even stronger, because we are only interested in the coupling with the provider framework.

From this, we can now formalize the following *null* and *alternative hypotheses*:

H_0: package-local-dependencies $\geq$ package-framework-dependencies

H_a: package-local-dependencies $<$ package-framework-dependencies

4.5 Experiment design and instrumentation

The test will compare one variable (number of dependencies) on two treatments (local or framework dependencies). Both treatments apply to all the subjects (packages in the systems studied): for each package, we will look at its local dependencies and dependencies to the framework. This is a paired design.

—We will use static analysis to obtain the data related to the dependencies;

—We will use the Moose environment and its accompanying tools to gather the metrics for Java programs [Ducasse et al. 2005];

—In the analysis of results, we will distinguish inheritance from the other dependencies.

About static analysis, in Section 2, we introduced our working hypothesis in relation to late-binding used in OO framework. It is the goal of late-binding to identify the called method only at execution time according to the instance that receives the message. Therefore in the presence of late-binding, static analysis can have problems to identify the exact method invoked. For Java, the class implementing an invoked method is computed from the declared type of variables. This may give wrong results if the variable actually contains an instance of a sub-class of its declared type.

Yet we chose to use static analysis because the alternative (dynamic analysis) presented other problems. Dynamic analysis requires to define:

—Realistic usage scenarios: In our experiment this was not always easy, for example, when considering JHotDraw, which is not an application, but a framework itself and therefore cannot run on its own. We also experimented with just a selection of packages of the systems. In such cases, it is difficult to elaborate realistic scenarios that will use the framework and the part of the system we are interested in.

—Usage scenarios covering the whole application: We need usage scenarios in sufficient number to trigger all the features of the target application, so as to execute all the code and not miss any dependency. With complex applications and/or frameworks as our subjects, it is difficult to ensure that completeness.

About the Moose environment, we chose it because it provides tools to extract code information for different languages (including Java) and offers the needed infrastructure. Moose tools rely on the Famix meta-model to represent source code. It includes packages, classes, attributes, methods and their associated relationships [Demeyer et al. 2001]. We will therefore consider the following dependencies: method invocation, field access, class reference (e.g. "Collections.sort(. . .)"), and inheritance.

Finally, we distinguish inheritance from the other dependencies because inheritance is a strongly structuring relationship in OOP. We will term these other relationships (field access, method invocation, class references) "accesses". For dependency counting, when inheritance between two classes exists, we ignore the other dependencies (accesses) between the two classes. Therefore, the total dependencies between two classes is not the sum of inheritance links and accesses.

4.6 Validity evaluation

We did not identify any *conclusion validity* threats. Validity of the results will be tested using the appropriate statistical test. Because the data do not follow a Normal Distribution, we will test our hypothesis using a one-tail Wilcoxon test.

We identified the following *construct validity* threats: our definition of low cohesion (*local dependency* is less than *framework dependency*) is a limited, syntactical, view on cohesion that does not fully measure what people mean by "high cohesion", however this simplification is an accepted trade-off in architectural design measurement (see also Section 4.3).

We did not identify any *internal validity* threats to the experiment.

We identified the following *external validity* threats: We had to rely on convenience selection of subjects. As such, all our subjects were identified as being Java systems and based on UI frameworks (Swing, AWT, SWT). These two characteristics may be partly responsible for our results, for some unknown reason. We, however, do not see any reason why this could be the case. Another problem with our subject systems is that Eclipse represents about two thirds of the data point (see Section 5). We tried to compensate for this possible bias by presenting individual experimental results for each system studied.

Another possible *external validity* threat is that we had to accept an informal, qualitative, definition of "good architectural design" of the subject systems, based on their developers opinion. This was necessary because our goal was precisely to show that good modularization may result in low measured cohesion.

5. EXPERIMENT RESULTS AND ANALYSIS

Table I lists some size metric results for the three systems considered. One can observe a dissimilarity in importance of each systems, with Eclipse weighting 65% of all the packages and 71% of local+framework dependencies.

Table I. Size metrics for Eclipse (user interface), JEdit (GUI) and JHotDraw.

	JHotDraw	Eclipse (UI)	JEdit (GUI)
Packages	41	80	2
Classes	430	1,540	97
Lines Of Code	88,452	311,744	24,791
Dependencies	2,952	10,626	1,371
Local dep.	700	3,024	276
Framework dep.	2,252	7,602	1,095

"Dependencies" include the local and framework dependencies.

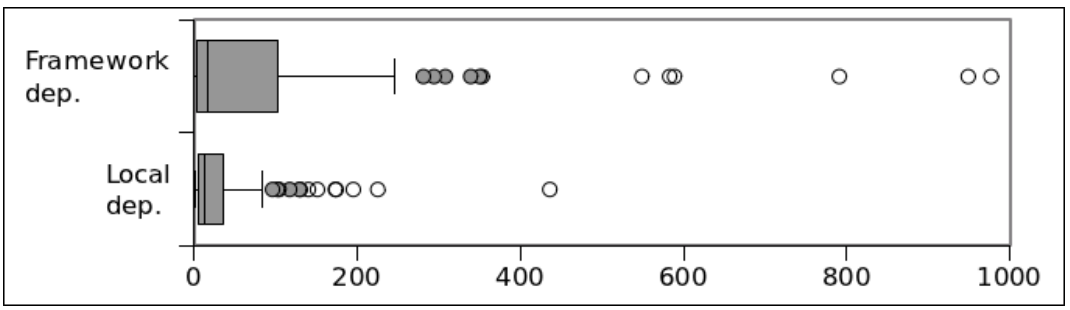


Fig. 3. Number of package dependencies for our experiment with Eclipse (user interface), JEdit (GUI) and JHotDraw

Figure 3 gives the box plot for the two tested variables. A box plots shows the shape of the distribution through five number summary of the data under analysis. The box is drawn from 25th percentile (on the left side) to 75th percentile (on the right side) values. The band in the middle depicts the median of the data. The whiskers on both sides of the box contain the non-outlier points *i.e.,* the data points within the distance of 1.5 times the box size. Beyond non-outliers, data points are outliers: the values that are regarded as "too far" from the central value. In the figure, the 75th percentile value for framework dependencies is higher than those for local dependencies, thus supporting our hypothesis.

Table II gives a summary of some descriptive statistics for our experiment. There are 123 packages in the three systems considered. The distribution for the two dependent variables is not normal, it is skewed to the right (i.e. more data on the left, long tail on the right). For example the median is less than the mean and skewness is high. This is also confirmed by the boxplots (Figure 3). For information, Spearman rank correlation coefficient shows that the two variables are positively correlated ($r = 0.80$): packages that have more dependencies within themselves also tend to have more dependencies with the underlying framework.

We also see that mean and median number of local dependencies ($mean = 32.5$, $median = 13$) are lower than for framework dependencies ($mean = 89$, $median = 19$), which again fits our hypothesis (packages have more dependencies toward their respective framework than inside themselves). This will need to be formally tested.

Table II. Descriptive statistics for our experiment with packages from Eclipse (user interface), JEdit (GUI) and JHotDraw

	local dependencies	f/w dependencies
Mean	32.5	89.0
Median	13	19
Standard Deviation	55.5	172.4
Skewness	4.2	3.4
Minimum	1	0
Maximum	436	977
Correlation	0.80	

Total packages: 123

Because data are not normally distributed, we use the Wilcoxon test to test our hypothesis. As this test is known to be resistant to outliers, we will not take any special precaution for them and include all data in the test. The result of the test ($W = 3109.5$, $p = 1.3E - 10$) allow us to reject the null hypothesis with high confidence. We may therefore accept the alternative hypothesis that packages have more dependencies to the underlying framework than within themselves.

This experiment confirms that, for these systems, that are well designed, packages are not cohesive according to a definition of cohesiveness that is compatible with the ones used in literature.

6. DETAILED EXPERIMENTS

In this section, we present the results of additional experiments. One goal of these more detailed experiments is to evaluate the influence of Eclipse in the results of the previous section (as described in the threats to validity). We will therefore study each system independently and even parts of them. For each experiment, we define the "extender application" considered and its provider framework.

These experiments will also serve to gain a more in-depth understanding. For this we consider two additional dependent variables (inheritance relationships, and access dependencies = attribute access + method invocations + direct class references). The dependent variable used in the main experiment will be re-termed "total dependencies".

6.1 JHotDraw (f/w=Swing+AWT)

Table III presents the results of the experiment. Note that Total dependencies is not the result of Inheritance+Access because we count multiple dependencies from one class to another as just one (see Section 4.5).

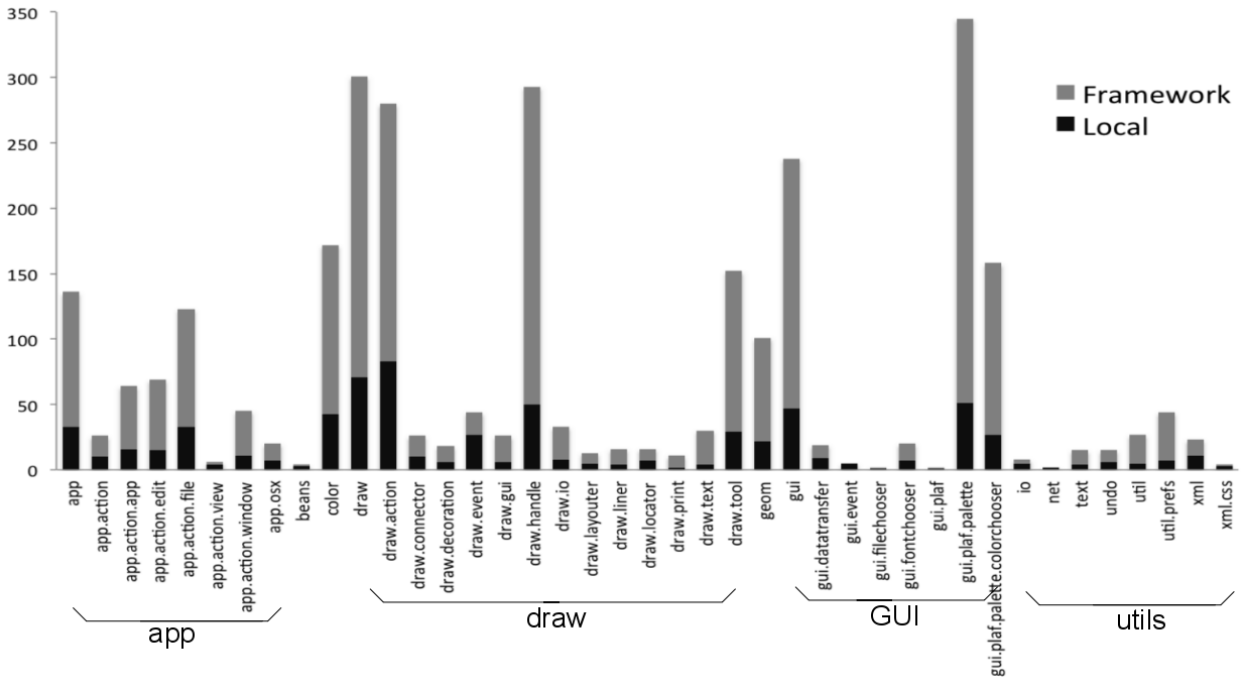
For JHotDraw, total number of local dependencies (700) is inferior to the dependencies to the framework (2,252) and the Wilcoxon test ($W = 58.5$, $p = 2E - 06$) allows us to reject the null hypothesis. We conclude that results for the main experiment were not solely caused by the preponderance of the Eclipse system since JHotDraw exhibits the same property.

Table III. Total Local and Framework Dependencies for JHotDraw (f/w Swing+AWT)

	Local	Framework
Inheritance	267 / 39.5%	290 / 42.9%
Access	700 / 17.6%	2,091 / 52.5%
Total Dep.	700 / 16.5%	2,252 / 53.2%

Because JHotDraw is well organized, we will single out some of its main parts. Figure 4 illustrates the individual dependencies of the JHotDraw packages. Most of the packages can be combined into

Fig. 4. Total Local and Framework Dependencies for each JHotDraw's package (f/w Swing+AWT)



four groups based on their name as shown in the figure: app, GUI, draw and utils. Informally, one can see that in all the four groups, the total count of framework dependencies (gray) outnumber the count of local dependencies (black).

We now study more in depth the org.jhotdraw.gui package and its sub-packages to better illustrate how packages that are well designed may end up depending a lot on an underlying framework. For the GUI group of packages, the description of the main package org.jhotdraw.gui says: "provides general purpose graphical user interface classes leveraging the javax.swing package". It seems only natural that it depends more on the framework (Swing+AWT).

Table IV presents the results of the experiment restricted to this set of packages. There are 8 pacakges in this experiment. Again, the number of local dependencies (148) is largely inferior to the dependencies to the framework (641) and the Wilcoxon test ($W = 2$, $p = 0.047$) allows us to reject the null hypothesis.

Table IV. Total Local and Framework Dependencies for org.jhotdraw.gui packages (f/w Swing+AWT)

	Local	Framework
Inheritance	14 / 9.4%	131 / 87.9%
Access	148 / 19.7%	547 / 72.6%
Total Dep.	148 / 17.4%	641 / 75.4%

Looking deeper into this group of packages (again Figure 4), we see *org.jhotdraw.gui.plaf.palette* standing out as very dependent on the framework. We show the inheritance hierarchy of the classes

for this package in Figure 5. In the figure, black nodes represent *framework* classes and grey nodes represent *local* classes; white nodes represent classes in the other packages of JHotDraw. The class at the top is java.lang.Object and is included for illustration purposes. It should not be counted as a framework class. This figure illustrates well our working hypothesis by showing that many classes, local to this package (in grey), inherit from a black class (in the framework). Considering only inheritance dependencies, 19 local classes inherit directly from framework ones, 2 others inherit from application classes (in white) that themselves inherit from framework classes, and only six do not have framework classes in their ancestors.

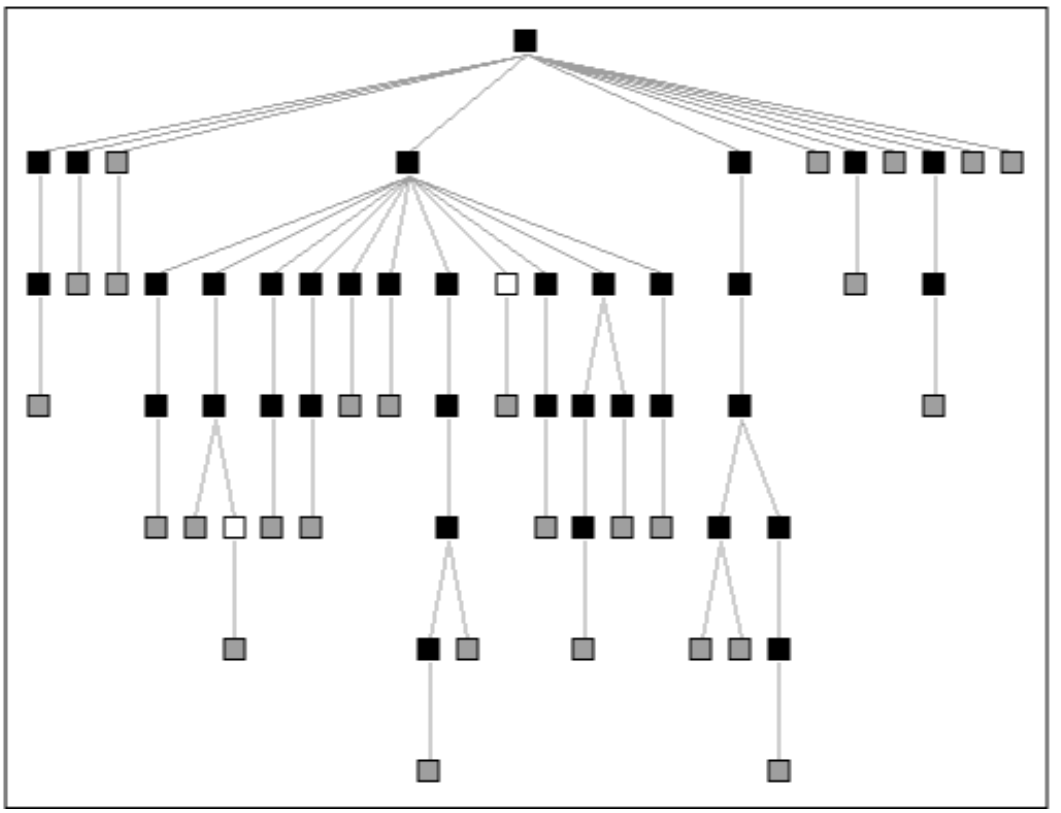


Fig. 5. Inheritance Hierarchy for the package `gui.plaf.palette`

6.2 JHotDraw Samples (f/w: JHotDraw +Swing+AWT)

JHotDraw is itself developed as a framework so that applications can be developed by extending its classes and interfaces. For this purpose, some samples have been developed to demonstrate its usage. We analyze here these samples (grouped in the org.jhotdraw.samples package), where JHotDraw+Swing+AWT serve as provider framework. We hypothesize that the quality of these samples is equal or superior to that of JHotDraw because the samples are smaller and receive much less maintenance (these are not actual applications used by someone).

Table V summarizes the results of this analysis. Again, the total number of local dependencies (323) is largely inferior to the dependencies to the framework (2076) which supports our formal hypothesis. The Wilcoxon test gives a p-value=0.047 (under the 5% acceptance level) confirming the statistical validity of this result.

Table V. Total Local and Framework Dependencies for JHotDraw Samples (f/w: JHotDraw +Swing+AWT)

	Local	Framework
Inheritance	37 / 16.2%	192 / 83.8%
Access	323 / 13.6%	1949 / 82.3%
Total Dep.	323 / 12.9%	2076 / 83.2%

As expected, org.jhotdraw.samples demonstrates strong dependency relationship with the provider framework.

6.3 JEdit (f/w: Swing+AWT)

Table VI. Total Local and Framework Dependencies for JEdit (f/w: Swing+AWT)

	Local	Framework
Inheritance	240 / 29.4%	451 / 55.3%
Access	2300 / 33.0%	2390 / 34.3%
Total Dep.	1149 / 18.6%	2708 / 43.8%

Table VI presents the results of the experiment with JEdit. Total number of local dependencies (1149) is well below the dependencies to the framework (2708), yet the Wilcoxon test gives a p-value=0.432 which does not allow us to reject the null hypothesis that the actual average number of local dependencies is more than the average number of framework dependencies.

Considering the large difference in numbers this seems surprising, but looking closer to the results, we see that two packages: org.gjt.sp.jedit.gui and org.gjt.sp.jedit.gui.statusbar cumulate 1095 dependencies to the framework, more than 45% of all the framework dependencies. If we look at these two packages, we obtain a total of 276 local dependencies and four times as much (1095) framework dependencies. Because they are only two packages, with a clear goal (documentation for "gui" states: "Various GUI controls and dialogue boxes"), we will suppose they are correctly designed in the sense that they don't contain code not related to GUI features. We claim these two packages support our working hypothesis. Because there are only two packages, it would be meaningless to test the statistical validity of this conclusion.

6.4 Eclipse-UI (f/w: JFace+SWT)

Table VII presents the results of the experiment conducted on Eclipse-UI. Total number of local dependencies (3024) is well below the dependencies to the framework (7602), and the Wilcoxon test confirms the statistical significance of this result (p-value=4E-06), showing that Eclipse-UI supports our hypothesis.

Table VII. Total Local and Framework Dependencies for Eclipse (f/w: JFace+SWT)

	Local	Framework
Inheritance	627 / 26.1%	1261 / 52.5%
Access	5480 / 33.4%	6649 / 40.5%
Total Dep.	3024 / 19.8%	7602 / 49.7%

6.5 JFace (f/w: SWT)

As part of the Eclipse experiment, we also analyzed JFace with regard to its underlying framework: SWT. The JFace UI toolkit is an extension of SWT as implicitly indicated by the documentation[6]: "the only prerequisites for JFace [were] reduced to SWT".

Table VIII presents the results of the experiment. Total number of local dependencies (914) is below the dependencies to the framework (1539), and the Wilcoxon test confirms the statistical significance of this result (p-value=0.017).

Table VIII. Total Local and Framework Dependencies for JFace (f/w: SWT)

	Local	Framework
Inheritance	305 / 44.3%	213 / 30.9%
Access	1593 / 43.4%	1331 / 36.3%
Total Dep.	914 / 27.5%	1539 / 46.3%

6.6 Swing (f/w: AWT)

Swing[7] is the GUI toolkit of java. Java has another, prior, GUI toolkit: AWT. Swing inherits and use several of the classes of AWT, however AWT may not be seen as an underlying framework for Swing[8]. For example, AWT uses the native, operating system-supplied, widgets (menus, windows, buttons, ...) whereas Swing elements are entirely written in Java with no native code. We thought it would be interesting to do this experiment to see whether the good-design/low-cohesion property also holds in more general cases. We have no indication of the design quality of Swing, but this is not an issue in this case (see below).

The results of the analysis are presented in Table IX. Local dependencies (3492) are superior to framework dependencies (1815). There is no need to test the significance of this result since it does not support our formal hypothesis.

Table IX. Total Local and Framework Dependencies for Swing (f/w: AWT)

	Local	Framework
Inheritance	595 / 38.9%	257 / 16.8%
Access	2897 / 43.1%	1558 / 23.2%
Total Dep.	3492 / 42.3%	1815 / 22.0%

[6] http://wiki.eclipse.org/index.php/JFace
[7] http://java.sun.com/javase/technologies/desktop/
[8] http://en.wikipedia.org/wiki/Swing_(Java)#Relationship_to_AWT

7. HYPOTHESIS TESTING AND DISCUSSION

We now summarize the results of all our experiments with regard to the initial working hypothesis.

In the systems studied, which are based on an OO framework, experimental data support our hypothesis that a well designed system (based on an OO framework) may have higher coupling than cohesion. This translated to packages having more dependencies towards classes in their underlying framework than their own classes.

When the numbers do not agree with our hypothesis as in the case of JEdit, we can argue that:

—We do not pretend that all well-designed packages will have this property, only that it is a possibility that one must take into account. In this case, JEdit has some packages, possibly well-designed, that do not depend on Swing. This should not be a surprise.

—We did find that the two GUI packages of JEdit have lower internal cohesion than coupling toward the underlying Swing framework. Again, this seems only natural, and concur with our hypothesis.

For Swing and AWT, again cohesion was higher than coupling towards the external library. We performed this experiment because AWT is not a framework on which Swing is based. This might be an indication that the scope of our good-design/low-cohesion rule is restricted to the case where a framework is used.

As already mentioned, our experiments targeted GUI related frameworks (Swing, SWT) which can be a bias of the study. GUI framework are among the more widely used and known because so many applications need to use one of them. It would be interesting to perform the same experiment with frameworks in more restricted domains: web application frameworks (e.g. Struts); ORB frameworks (providing implementation of Corba or RMI), etc. These examples are still generic and may apply to any business domain, other application domain frameworks may also be studied. Some systems use annotations to inject dependencies and the impact of the annotation dependencies needs to be studied through further experimentation.

8. RELATED WORK

Related work to the paper in the domain of program metrics is already presented in Section 3.

Dong and Godfrey [Dong and Godfrey 2007] propose a visualization approach to understand the usage dependencies between aggregated components formed through static object descriptions. The study does not take into account library (framework) classes and primarily serves as an approach for program understanding. Package blueprint takes the focus of a package and shows how such packages use and are used by other packages [Ducasse et al. 2007].

Abreu and Goulão demonstrate that the criteria used by practitioners to modularize object-oriented systems falls short to achieve the objective of minimal coupling and maximal cohesion [Abreu and Goulão 2001]. Anquetil and Laval present a study that shows that package cohesion and coupling metrics do not demonstrate the inverse correlation for successive restructured versions of eclipse platform [Anquetil and Laval 2011]. Furthermore, they describe that cohesion and coupling degrades with restructuring that aim to improve system structure. Recently Taube-Schock *et al.* conclude that high coupling is not avoidable and some high coupling is necessary for good design [Taube-Schock et al. 2011].

9. CONCLUSION

Cohesion and coupling is described for modules in the procedural paradigm to gauge the design quality of modules. A program is deemed to have a good design if it exhibits low (external) coupling and high (internal) cohesion. These concepts are then ported to the object-oriented paradigm. However, the notion of cohesion and coupling in object-oriented programs is not similar to procedural programs because

of the hollywood principle and late binding, which allow to leverage the benefits of the object-oriented paradigm. Object-oriented programs are developed on top of frameworks and this relationship results in tight coupling between an *extender* application and its underlying framework. In this paper, we present a study of package dependencies in object-oriented programs, which serves as a quantitative evidence that application is coupled to its underlying framework. Strong dependency of the extender application on its framework is perfectly acceptable. We believe that while calculating the package metrics for cohesion and coupling, these factors should be taken into account. Moreover, while deducing the results of these metrics, high coupling and low cohesion should not be considered harmful so that the measures reflect the reality of the object-oriented paradigm. We plan to continue investigating this phenomenon further so that adequate measure for object-oriented application can be developed to ascertain their design quality appropriately.

REFERENCES

Hani Abdeen. 2009. *Visualizing, Assessing and Re-Modularizing Object-Oriented Architectural Elements*. Ph.D. Dissertation. Université de Lille. http://rmod.lille.inria.fr/archives/phd/PhD-2009-Abdeen.pdf

Fernando Brito Abreu and Miguel Goulão. 2001. Coupling and Cohesion as Modularization Drivers: Are We Being Over-Persuaded?. In *CSMR '01*. IEEE Computer Society, 47–57.

E. Allen and T. Khoshgoftaar. 2001. Measuring Coupling and Cohesion of Software Modules: An Information Theory Approach. In *Seventh International Software Metrics Symposium*.

Nicolas Anquetil and Jannik Laval. 2011. Legacy Software Restructuring: Analyzing a Concrete Case. In *CSMR 2011*. Oldenburg, Germany. http://rmod.lille.inria.fr/archives/phd/Anqu11a-CSMR2011-Coupling.pdf

Nicolas Anquetil and Timothy Lethbridge. 2003. Comparative study of clustering algorithms and abstract representations for software remodularization. *IEE Proceedings - Software* 150, 3 (2003), 185–201. DOI:http://dx.doi.org/10.1049/ip-sen:20030581

Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2010. Software Re-Modularization Based on Structural and Semantic Metrics. In *Reverse Engineering, Working Conference on*. 195–204.

Pradeep Bhatia and Yogesh Singh. 2006. Quantification Criteria for Optimization of Modules in OO Design. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006*, Vol. 2. CSREA Press, 972–979.

J.M. Bieman and B.K. Kang. 1995. Cohesion and Reuse in an Object-Oriented System. In *Proceedings ACM Symposium on Software Reusability*.

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

Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering* 25, 1 (1999), 91–121. DOI:http://dx.doi.org/10.1109/32.748920

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

Shyam R. Chidamber and Chris F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (June 1994), 476–493.

Steve Counsell, Stephen Swift, and Allan Tucker. 2005. Object-oriented cohesion as a surrogate of software comprehension: an empirical study, In Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation. *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation* (2005), 161–172.

Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. 2001. *FAMIX 2.1 — The FAMOOS Information Exchange Model*. Technical Report. University of Bern.

Xinyi Dong and M.W. Godfrey. 2007. System-level Usage Dependency Analysis of Object-Oriented Systems. In *ICSM 2007*. IEEE Comp. Society. DOI:http://dx.doi.org/10.1109/ICSM.2007.4362650

Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. 2005. Moose: an Agile Reengineering Environment. In *Proceedings of ESEC/FSE 2005*. 99–102. DOI:http://dx.doi.org/10.1145/1081706.1081723 Tool demo.

Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. 2007. Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships. In *ICSM '07*. 94–103. http://scg.unibe.ch/archive/papers/Duca07cPackageBlueprintICSM2007.pdf

Mohamed Fayad, Douglas Schmidt, and Ralph Johnson. 1999. *Building Application Frameworks: Object Oriented Foundations of Framework Design*. Wiley and Sons.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

W. Li and S. Henry. 1993. Object Oriented Metrics that predict maintainability. *Journal of System Software* 23, 2 (1993), 111–122.

Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. 2008. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Transactions on Software Engineering* 34, 2 (2008), 287–300. DOI:http://dx.doi.org/10.1109/TSE.2007.70768

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

Brian S. Mitchell and Spiros Mancoridis. 2006. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Transactions on Software Engineering* 32, 3 (2006), 193–208.

Vojislav B. Mišić. 2001. Cohesion is Structural, Coherence is Functional: Different Views, Different Measures. In *Proceedings of the Seventh International Software Metrics Symposium (METRICS-01)*. IEEE.

Laura Ponisio and Oscar Nierstrasz. 2006. *Using Contextual Information to Assess Package Cohesion*. Technical Report IAM-06-002. University of Bern, Institute of Applied Mathematics and Computer Sciences. http://scg.unibe.ch/archive/papers/Poni06bAlchemistPackageCohesion.pdf

María Laura Ponisio. 2006. *Exploiting Client Usage to Manage Program Modularity*. Ph.D. Dissertation. University of Bern, Bern. http://scg.unibe.ch/archive/phd/ponisio-phd.pdf

Wolfgang Pree. 1995. Framework Development and Reuse Support. In *Visual Object-Oriented Programming*, Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis (Eds.). Manning Publishing Co., 253–268.

Don Roberts and Ralph E. Johnson. 1997. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley.

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

Craig Taube-Schock, Robert J. Walker, and Ian H Witten. 2011. Can we avoid high coupling?. In *Proceedings of ECOOP 2011*.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA.

E. Yourdon and L. Constantine. 1979. *Structured Design: Fundamentals of a Discipline of Computer Programs and System Design*. Yourdon Press/Prentice Hall.