Assessing the Quality of Architectural Design Quality Metrics

Nicolas Anquetil, Andre Hora

Assessing the Quality of Architectural Design Quality Metrics

Nicolas Anquetil*†, André Hora†
*CRIStAL, University of Lille-1, France
†Inria Lille—Nord Europe, France
nicolas.anquetil@inria.fr

May. 12, 2013

Abstract

As software ages, its quality degrades, leading to pathologies such as architectural decay, a problem that is hard to fight against. First, it is a slow drifting process that results from the natural evolution of software. Second, correcting it is a costly and dangerous operation that has the potential to affect the entire code base. A daring prospect on systems of multi-million lines of code. This spawned a lot of research, for example to measure the quality of a software architectures. Yet a fundamental issue remains, we have very little means of measuring the quality of a given architecture. One of the reasons for this seems to be that building a formal experiment to test these issues is difficult due to the fuzzy nature of what is a good architecture, and to the need to use large and real systems. In this paper we setup an experimental design with well know systems such as Eclipse or JHotDraw that would allow to test the relevance of architectural design metrics. We illustrate it on some well known metrics.

Keywords: Software Quality; Software Architecture Quality; Design Quality Metrics; Empirical experiment

1 Introduction

"As a software system evolves, its structure worsen, unless work is explicitly done to improve it" [1]. This law of software evolution spawned lot of work in automatic software restructuring with the idea of proposing a better architectural design for legacy systems.

Because we don't really know how to evaluate the quality of an architectural design, and because these approaches are based on their own measure of quality (the objective function), evaluation of the results have typically been difficult, sometimes relying on metrics similar to the one used to get the results (denounced in [7]); sometimes comparing against architectures of unknown value (*e.g.,* the actual structure of the systems used as test bed); and sometimes asking experts to rate these results. Evaluating objectively a software architecture is difficult because it embodies some of the most difficult aspects of software engineering: software systems are unique pieces of craftsmanship that need to answer to unique constraints, making it difficult to compare them one to another or generalize the results; a good architectural design is a fuzzy concept that accept several solutions to one given problem[1]; controlled experiments in laboratory are difficult to set up, if at all possible, because the real issues only appear on large, real world systems, with real world constraints; for the same reasons, manual evaluation is subjective and costly.

A fundamental issue seems to be able to build empirical evidence that any architectural design quality principle is relevant to the practice. This empirical evidence, however, is not easy to establish [7, 8] for the reasons listed above. For a given software system, there might be many different and equally valid possible architectural solutions. Architectural quality is a fuzzy problem that we do not fully understand.

In this paper, (i) we propose a formal experimental setup to assess the relevance of architectural design quality metrics; (ii) we implement it with a testbed of real world systems (among them Eclipse and JHotDraw); and (iii) we illustrate its application on some existing architectural quality metrics (cohesion/coupling metrics). The idea of the formal experiment is to consider real restructuring actions that were performed on subject systems and look at the values of the architectural quality metrics before and after

[1]Nice answers to the question "what is a good software architecture?" may be found on http://discuss.joelonsoftware.com/default.asp?design.4.398731.10 (*e.g.,* "It's a bit like asking *What is beautiful?*"). Last consulted on 12/10/2012.

this restructuring. Adequate architectural quality metrics should be able to measure the increase in quality after the restructuring.

This paper is organized as follows: In Section 2, we first review existing architectural quality metrics, mainly focusing on cohesion/coupling measurement and also illustrating the difficulties of assessing the architectural design of real world systems. We follow, in Section 3, with the presentation of our formal experiment to validate the relevance of these metrics. The next section (§4) details the choice of real systems as subjects of our experiments. We present and discuss the results of testing some known cohesion or coupling metrics on these systems in Section 5.

2 Architectural Quality Assessment

Although every software engineer will agree that defining a good software architecture for a software system is key, there is little agreement on what is a good software architecture[2] and general belief sees it as a subjective topic.

On the other hand, it is clear that having a reliable answer to this question would be of utmost utility to help people design their systems, redesign (restructure) them, or monitor their quality. One key precept on architectural design quality is that modules should be highly cohesive and loosely coupled. This was stated by Stevens *et al.* [2] in the context of structured development techniques. His objectives was to foster reusability by making it as easy as possible to extract a module from a system and plug it into another one.

Since then the ideas have been transposed to OO programming and continue to hold, be it at the level of classes (*e.g.,* [9]) or at the level of packages. In the OO paradigm, it spawned lot of research, particularly for quantifying the cohesiveness and coupling of classes as witnessed by Briand *et al.* [9].

In the following, we review existing cohesion/coupling metrics for packages. And discuss their relevance in practice.

2.1 Cohesion/Coupling metrics for packages

Many cohesion or coupling metrics may been found in the literature. They may apply to packages or classes. Because we consider system architecture, we are interested in the design quality of packages or groups of classes. But measuring cohesion and coupling of classes or packages are two well separated

[2]Again, see http://discuss.joelonsoftware.com/default.asp?design.4.398731.10

issues. Class metrics don't apply to architecture quality assessment, because a package containing highly cohesive classes could be non-cohesive if each classes deal with a specific topic, and a package containing highly coupled classes could be lowly coupled if its classes were all coupled together and not with other packages.

A review of different cohesion or coupling metrics for packages may be found in [10]. We chose five well known cohesion and coupling metrics, for demonstration purposes:

- Bunch, presented in [11], is a tool that remodularizes automatically software, based on the metrics of module cohesion and coupling defined by the approach. 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.

  The maximum number of dependencies within a package (cohesion) is the square of the number of classes within that package. The maximum number of dependencies between a package and the outside (*i.e.,* its client and provider packages) is the number of classes within the package multiplied by the number of classes outside the package.

- With Relational Cohesion in [12], Martin defines the cohesion of packages as the average number of internal relationships per class. Thus, it depends on the number of dependencies, as well as the number of classes within the package.

- Efferent coupling (Ce)[3], also proposed by Martin, looks at the number of classes outside the package on which classes inside it depend. Afferent coupling (Ca) is the number of classes external to the package which depend upon classes in the package. Both metrics are computed regardless of the number of dependencies between the classes.

2.2 Relevance of architectural quality metrics

Current evaluation of a software architecture's quality is limited to the high-cohesion/low-coupling dogma. Yet, some started to notice that we have little understanding of "how software engineers view and rate cohesion on

[3]Martin's Ce and Ca are not describe in Ebad's paper, but they make a useful match to Martin's cohesion metric.

an empirical basis" [8] (this was for classes); or that "it has been difficult to measure coupling and thus understand it empirically" [13]. The same holds at the level of packages [14].

Even worse, other critics were formulated, for example Brito de Abreu and Goulão state that "coupling and cohesion do not seem to be the dominant driving forces when it comes to modularization" [3]. A statement with which Bhatia and Singh agree [4].

Other researchers considered a more theoretical point of view: "we conclude that high coupling is *not* avoidable—and that this is in fact quite reasonable" [6]; or "we believe that additional investigations are required for assessing package modularity aspects" [15].

Yet, the critics are sometimes unclear, for example, Counsell *et al.* in the same publication [8] also states that the "concept of software coupling is relatively easy to both quantify and assess"; and Brito de Abreu and Goulão [3] or Bhatia and Singh [4] proposed their own cohesion/coupling metrics in response to their critics.

As explained in the introduction, it is difficult to formally assess the quality of a given architecture:

- The only known metrics are cohesion and coupling and we are arguing that none was ever validated;
- Asking the opinion of experts would be costly on a realistic scale because architecture should be evaluated on large systems, one must also note that, in industry, young programmers are not asked to design the architecture of complex systems and similarly, evaluating an architectural design would require experienced designers making it even harder and costlier to find some ;
- Comparing to some golden standard raises the issue of subjectivity of the solution: for one system, there are several possible, equally valid, architectures and the validation of any quality metric should take this into account.

To be of interest, evaluation of architectural design must be done on large, real, systems because architectural design presumably depends on many factors, other than cohesion and coupling [3], and the evaluation must consider these additional factors to bear any relevance. Evaluating a few packages out of context could cause an evaluator to base his opinion on too few parameters, thus leading to a possibly unrealistic evaluation.

3 Experimental Assessment of Architectural Quality Metrics

We wish to evaluate whether the existing architectural quality metrics hold to their promises in practice, that is to say whether the results they give correspond to the perception of architecture quality of actual software engineers on real world systems and in real conditions.

3.1 Experiment intuition

We present here the rational for the experiment setup we chose and start to discuss it's validity. The experimental setup itself is described in the following sections, and the present discussion is summarized in the threats to validity (Section 3.6).

What is a good system architecture and how to measure it? Cohesion/coupling mainly foster reusability, but this is not the sole goal of a good architecture. For example a Rational Software white paper lists as desired properties: "[to be] resilient [...], flexible, accommodate change, [be] intuitively understandable, and promote more effective software reuse" [16]. It is difficult therefore to decide how to measure a good architecture. We propose a practical approach, saying that a good architecture is one that is accepted as such by some expert software engineers (experts in the systems considered and in software architecture). Yet, as discussed in the previous section, we rule out the manual evaluation of results by experts because of the difficulties and costs involved in evaluating real system architectures in real contexts. We also explained that there are no other metrics to compare to. Our last solution is therefore to compare against actual architectures (designed by experts) of known quality and check whether the metrics accurately reflect this quality.

Ideally, one would like to set up a controlled experiment with architectures of known quality and apply the metrics to them to compare the results so that all confounding factors can be eliminated. However, we already explained that this is not possible because there are no scientifically proved, known quality of architecture. It is also important to remember that modularization is a subjective issue and that for a given problem, several equally valid architectures could be designed.

Cinnéide *et al.* [17] proposed a controlled experiment where they in-

troduce random modification in the code and measure the impact on the metrics. With this experiment, they were able to show that the metrics do not agree one with the other, and therefore some of these metrics are sure to be inadequate. However, they cannot tell us which metrics are relevant as they cannot automatically assess whether the random modification improved or hurted the architecture.

Because of the nature of software architecture, it seems difficult to set up a realistic controlled experiment. First the system must be of a reasonable size so that there is meaning in defining various packages and there can be enough interactions between them. Also, the difficulties of architectural design are linked to the many conflicting aspects that influence it. A realistic testbed should again be large enough so that these aspects may come into play. The same thing happens for restructuring. The conditions in real life (mainly the costs, but also the risks) make it difficult to restructure a system. Such attempts are only made with a well defined goal in mind, a goal that can only deeply influence the resulting architecture. It can be tricky to set a realistic goal for a laboratory restructuring effort of a large enough system that would not be biased by the experiments we are planning here.

We must thus perform a natural experiment (or quasi-experiment) and work with real architectures. The drawbacks of natural experiments are known as they include possibilities of the results being caused by undetected sources (confounding factor). On the other hand, software engineering is inherently an applied research field. It works with real data (*e.g.,* legacy systems of millions of lines of code in several programming language) and aims at impacting the practice of software development and evolution. From this point of view, controlled experiments have been criticized for being too artificial thus lacking relevance. Working with real architectures allows us to test the architectural quality metrics in the setting and conditions they are supposed to be designed for.

The main difficulty of the experiment is to find real architectures of known values (whether good or bad) for which we could check the results of the metrics. Absolute quality values are ruled out for lack of existing proven metrics, and because of the fuzziness of the domain. So we must turn to relative quality values with architectures whose quality is known relatively to other architectures. This implies working with systems with two different architectures and a known increaser or decrease of quality between them. The idea of systems having been remodularized springs to mind.

We hypothesize that the modular quality of a software system should

improve after an explicit remodularization effort. A similar hypothesis was informally used by Sarkar *et al.* in [18]. One of the validation of their metrics for measuring the quality of non-Object-Oriented software modularization was to apply them to "a pre- and post-modularized version of a large business application".

In the context of this experiment, this hypothesis needs to be considered carefully. First, how to identify the remodularization? We consider explicit remodularization effort, that is to say efforts clearly tagged as aiming to improve the architecture or modular structure of the system. Such explicit identification of the remodularization effort may appear in the documentation (News and Noteworthy), in official web sites, etc. Because we consider real restructuring efforts, one must expect that other changes will also occur between the two versions considered, typically bug fixes and feature addition. This is the kind of confounding factor that must be accepted in natural experiment. Bug fixes are typically localized and can therefore be ignored at the architectural level. New features may impact the architecture, but those would happen within the context of the remodularization effort and we expect that they be taken into account in the new architecture. We assume that possible new features do not adversely impact the new architecture but are rather part of it. If one bug fix should be so important as to impact the architecture, it should similarly be a planed fix already considered in the new architecture.

Second, did the architecture really improve? Considering the time and effort one must invest in a remodularization, typically without visible impact for the user, such task cannot be started lightly. As such we consider that an explicit remodularization will have a planned target architecture that must be the result of experienced software engineers' best efforts. It is hard to imagine that the new architecture would not be at least as good as the existing one. Additionally, we propose to focus on past remodularizations of systems that stood the proof of time. An unsuccessful explicit remodularization effort would be identified as such after some time and would similarly be documented in some way. Even an architecture not entirely successful (whatever the goal it was assigned), would allow us to get a glimpse on what the software engineers thing a good architecture is.

What if the two modularizations were different but equally valid? Again, we will rely on the best effort of the software engineers. Given the costs involved, one restructures a system to improve the situation. Two cases may occur, either the new architecture is very different from the previous

one (termed *global restructuring* in this paper), or they have some parts in common (termed *local restructuring* in this paper). In a global restructuring, we expect that the architectural quality, at least does not degrade, and possibly improves. In a local restructuring, we expect that the restructured parts improved their architectural quality. Additionally, if there are also new parts, we expect that they do not degrade the architectural quality of the old architecture, and possibly improves it. Finally for the possible dropped parts of the old architecture, we ignore them.

We will now formalize the experiment according to the experimental setup suggested in [19].

3.2 Experiment planning

Analyze architectural quality metrics

with the purpose of comparing

with respect to their results

from the point of view of researchers

in the context of real world, restructured, OO packages.

Note that the limitation to OO systems is not a strong one but the result of practical constrains. It should be trivial to use the same experiment setup for non OO systems.

3.3 Context and subjects selection

The *context* of the experiment will be packages from real OO systems, which have been explicitly restructured.

The ideal situation for us is that of a global system restructuring with the following restrictions:

- It must be an explicit and "pure" restructuring effort (as little bug fix or feature addition as possible) of limited duration. This will ensure that we can pinpoint versions just before and just after the restructuring effort and that the restructuring effects will not be diluted in other modifications such as enhancements. This is actually very difficult if at all possible to find in real life. Systems need to evolve, errors need to be corrected. This is a threat to validity that we must accept if we are to work with real systems and real restructurings.

- It is better if the restructuring is old enough to have sustained the "proof of time" that we assume is a guarantee of its success;
- The source code of the systems, before and after the restructuring, must be freely accessible, and in a programming language that we can easily parse (Java and Smalltalk for now), to allow computing the metrics.

Such systems are actually difficult to encounter. A search on Google CodeSearch[4] for keywords "restructure", "refactoring", and "remodularize" in files "readme", "news", "changelog", or "changes.html"[5] did not point to systems that fit our requirements.

We found two systems by indication: Eclipse when it migrated from an IDE to the Rich Client Platform (version 2.1 to 3.0), and Seaside between version 2.8 and 3.0. We also include another restructuring of Eclipse, from version 2.0.3 to 2.1, as a preparation of the RCP restructuring.

To increase the size of our test bed, we will also consider local restructurings, with the same restrictions as above. We therefore add the JHotDraw system at different moments of its history and the Vivo system between versions 1.4.1 and 1.5.

All the systems and their restructurings are described below in Section 4.

The *subjects* of the experiment are the packages of the chosen systems that were part of the restructuring(s).

3.4 Variable selection and Hypothesis formulation

The *independent variable* is the version of the system considered. Basically we are not interested in the specific version number or the release date, but whether the version is before (*base version*) or after (*restructured version*) the explicit restructuring effort. In some cases, two restructurings may have been performed consecutively. When it happens, a given version may be a restructured version (compared to its predecessor) and a base one (compared to its successor).

The *dependent variable* is the metric (cohesion or coupling) of the packages for a given version.

Considering that cohesion improves when it augments, we formalize the *null* and *alternative hypotheses* for cohesion metrics as follows:

[4] http://www.google.com/codesearch

[5] The exact searches were: `restructure file:(readme|news|changelog|changes.html)`, where "restructure" was also substituted by "remodularize" and "refactoring"

H_0^{coh}: The cohesion of restructured packages in a restructured version is less than the cohesion of packages in the corresponding base version.

H_a^{coh}: The cohesion of restructured packages in a restructured version is greater or equal to the cohesion of packages in the corresponding base version.

Opposite hypotheses may be formulated for coupling that decreases as it improves:

H_0^{coup}: The coupling of restructured packages in a restructured version is greater than the coupling of packages in the corresponding base version.

H_a^{coup}: The coupling of restructured packages in a restructured version is less or equal to the coupling of packages in the corresponding base version.

3.5 Experiment design and instrumentation

The test will compare one variable (a cohesion or a coupling metric) on two treatments (base and restructured versions).

On any given restructuring two experimental designs may be considered:

Paired: (within subjects) For packages present in both base and restructured versions, one measures their cohesion (or coupling) and compares the values. Paired design is usually considered best as it allows to derive conclusive results with fewer subjects.

Unpaired: (between subjects) Packages that are created in the restructured version don't exist in the base one, those removed in the restructured version, only exist in the base one. In this case one compares the average cohesion (or coupling) of the packages in each versions, packages existing in both base and restructured versions are considered different. Unpaired design still allows to draw conclusions but requires more data to get convincing results.

We perform two tests: Unpaired design for global restructurings and paired design for local and global restructurings.

Global restructurings are those affecting the entire system, or most of it. In this case we use an unpaired setting to consider all the packages of both versions. The idea is that a global restructuring should improve the quality of the system as a whole, which is measured by taking into account all the packages. The unpaired setting is not a problem because we have

enough packages to draw conclusions. There are two global restructurings in our testbed: Eclipse restructuring from version 2.1.3 to 3.0 and Seaside from version 2.8 to 3.0.

For the local restructurings, we cannot use the same setting because although the quality of the restructured packages should improve, the quality of the system as a whole may decrease (due to other modifications to the system). We therefore choose a paired setting where we compare only packages that exist in both versions and were restructured. There are less of these packages, but the higher sensitivity of the paired setting allows to draw conclusion. There are four purely local restructurings: Eclipse from v.2.0.2 to v. 2.1, JHotDraw from v. 7.3.1 to v. 7.4.1, v. 7.4.1 to 7.5.1, and v. 7.5.1 to 7.6, and Vivo from v.1.4.1 to v.1.5. We will also consider the packages present in both base and restructured version of the global restructurings.

Metrics will be computed with the Moose data analysis environment. It must be noted that one of the systems studied, namely Seaside, is written in Smalltalk, a language not statically typed. This implies that one can rarely infer the type of variables statically, and as a consequence that dependencies between classes cannot always be computed as accurately as with statically typed languages. We don't see this as an important issue: First, we also have Java systems to study so that the Smalltalk one is an additional contribution to the experiment; second, not all metrics are impacted by this fact; third, the problem is often mitigated by other factors, for example when the name of a method is unique in a system (not rare), one always knows to what class this method belongs; and fourth, any architecture quality metrics would also need to be computed for these languages, and therefore present the same difficulties.

3.6 Validity evaluation

We did not identify any *conclusion validity* threat. Validity of the results is tested using the appropriate statistical test at the 5% significance level which is customary. In software Engineering, data typically do not follow a Normal distribution (*e.g.,* [20, 21]), we test our hypotheses using a one-tail Wilcoxon test.

The fact that the cohesion/coupling metrics used might not actually measure what people mean by package cohesion and coupling is not a *construct validity threat* because we set to assess the validity of the metrics, not the validity of the cohesion/coupling principle.

One might consider that measuring accurately some of the metrics on the Smalltalk system is a *construct validity threat*. Because Smalltalk is a dynamically typed language, it is sometimes more difficult with statistic analysis to ascertain what method is invoked in the code. But even in this case, we are still measuring the validity of the metrics, *as they can be measured*, for this language.

We identified an *internal validity* threat to the experiment: Even for explicit restructuring efforts, in real situation, it must be expected that other modifications will be included such as bug correction or some enhancement. This was already discussed in Section 3.1. We monitor this threat by looking at the detailed description of all work done in the restructured versions.

We identified the following *external validity* threats: We had to rely on convenience selection of subjects systems and could not find many of them. This is a threat, but we have systems in different application domains, and in two programming languages which can only contribute to strengthen our conclusions. The experiments would need to be replicated with other systems that matches the requirements.

Another possible *external validity* threat is that the restructuring efforts might not have been as successful as we hope, and the resulting architecture quality have effectively decreased. To try to detect such cases, we looked at one more version after the restructuring to detect whether more work had been done on the architecture.

4 Subject Systems

As test bed for our experiments, we use four systems that underwent restructuring operations: Eclipse, JHotDraw, Vivo, and Seaside. In this section, we describe each system both qualitatively and quantitatively. Before that, we list some generic guidelines we use to define the testbed.

4.1 Guidelines to select source code

In these systems, it is not always clear what source code (packages) to include in the testbed. Eclipse for example is a very large project that went from 379 packages in version 2.0 to 751 in version 3.1. Not all of it is directly impacted by the restructurings that occurred whereas we want to consider as little packages as possible to avoid diluting the results in too much source code

not restructured. For JHotDraw, another difficulty is that the restructurings are sparingly documented (e.g. "Some changes in the package structure and renamings of classes have been made in order to improve the clarity of the frameworks") and we need to decide for ourselves what was impacted by the restructuring.

To select an apropriate body of code, we set some generic requirements, including those already stated in Section 3.3:

Explicit: Explicit restructuring effort to ensure as much as possible the quality of the new architecture;

Pure: "Pure" restructuring with as little extra changes as possible also to ensure the quality of the result.

Small: Just enough source code to cover the restructured part. The code need to be small not to dilute the result of the restructured code into other untouched parts.

Functional: Ideally we would like a body of code that can be successfully compiled so as to be meaningful. This is intended to ensure that the examples are realistic.

Consistent The source code should cover the same set of functionalities over all versions studied. It would be easier if the comparison could be on a somehow fixed set of classes and/or methods. This is, however, difficult because restructurings often include removing some class and introducing new ones. The same goes with functionalities which are harder to monitor.

To select which packages can be considered as restructured (well designed) or not, we defined simple rules:

Documented: the documentation may not always be as elusive as the example given above. When some package is explicitly mentioned, we, of course, consider it restructured.

New: If a package is introduced in a restructured version, we consider it is well designed just as we assume that new feature added during the restructuring would not damage the new architecture but be part of it (Section 3.1).

Note, on the other hand, that we cannot make a similar assumption for normal (not restructuring) versions. A new concept might be split up between already existing code, in various old packages, and new code, in the new package. In this case the new package could not be considered well designed.

Class transfer: If a class is transferred from one package to another between a normal version and a restructured one, we consider that both packages improved their design quality. The idea is that restructuring may be done by moving classes around. When this happens, we must hypothesize that it was not well placed, therefore the quality of the package loosing it improved. Similarly we must also hypothesize that the class is placed in the proper package after the restructuring, so the quality of this one also improved. Note that we only need to hypothesize that the packages' quality improved, we don't require for our experiments that they be the best possible packages.

We actually apply this rule only when two or more classes are transfered (possibly from/to two different packages) in fear, for example, that moving only one class out of 10 or 20 would not be meaningful enough.

For Smalltalk, class transfers are easy to find because class names are unique in the environment, for Java, when there are several classes with the same name (in different packages), we manually look at their code to identify which one was transfered.

4.2 Qualitative Description

We describe here the four subject systems we use and discuss the packages that were or not part of the restructurings.

4.2.1 Eclipse

Eclipse is the well known, open source, IDE. It is developed in Java. In 2004, Eclipse had two successive restructurings. The main one, from version 2.1.3 to 3.0 evolved Eclipse from the concept of an extensible IDE toward the Rich Client Platform (Eclipse RCP). The other one, from version 2.0.2 to 2.1, consisted in a preliminary restructuring in preparation for the former.

One interest of this restructuring is that it is well documented[6]. A graphical representation of the old and new structure of Eclipse[7] can be found in Figure 1.

One can see that this restructuring was mixed with enhancement, such as the introduction of OSGI in the Runtime layer. As discussed in 3.1, we do not see this as problem because the new architecture must have been designed with such new feature in mind, therefore the new Runtime layer should have been redesigned as best as the architects could make it.

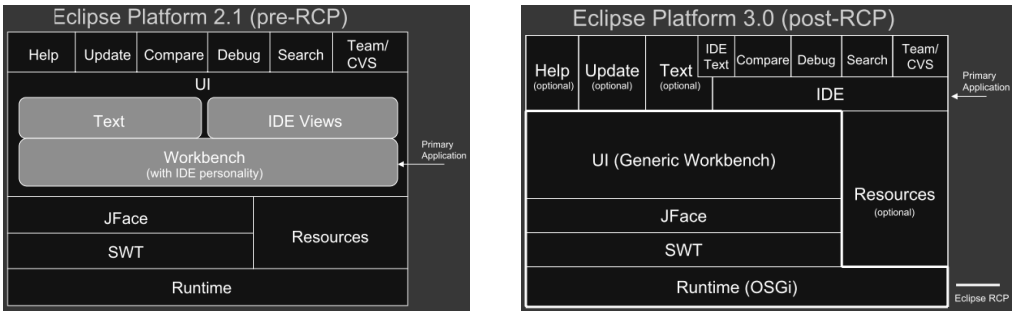

Figure 1: An illustration of the architecture of the Eclipse platform before RCP (v. 2.1) and after RCP (v. 3.0). From a presentation at EclipseCon 2004.

We looked at the base and restructured versions of the two restructurings (2.0.2, 2.1, 2.1.2 and 3.0) as well as all minor versions between 2.0 and 3.1 (2.0, 2.0.1, 2.0.2, 2.1.1, 2.1.2, 2.1.3, 3.0.1, and 3.0.2). The source code for these versions was downloaded from http://archive.eclipse.org/eclipse/downloads/, and we chose the archives "Source Build (Source in .zip)", files named "eclipse-sourceBuild-srcIncluded-*version*.zip".

Eclipse is a very large project, in accordance with the *Small* requirement, we consider only part of it. The platform is organized in plugins, a larger modularization structure than packages (a plugin typically contains several packages) that is actually perpendicular to the package decomposition (a

[6]For example see the "Original Design Documents" section of http://wiki.eclipse.org/index.php/Rich_Client_Platform, last consulted on 09/10/12

[7]From a presentation at EclipseCon 2004, http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/11_Edgar.pdf, last consulted on 09/10/12

package can be split over several plugins). We refined the requirement of section 4.1 into the following rules:

- Following the *Small* and *Functional* requirement, for the first version (2.0) we excluded plugins related to JDT (Java programming), PDE (Eclipse plugins development), and team (version control management) so as to keep only the core functionalities.
- SWT, the GUI framework of Eclipse, exists in various configurations (MacOS, Linux, Windows, ...). We choose the plugins related to Linux/GTK to avoid considering various possible implementations of the same classes (*Functional* requirement).
- If a plugin, included in the experiment, requires another plugin, then that other one is included too (*Functional* requirement), eventually recursively.
- If a plugin is included in one version, then include it for all preceding and following versions (i.e. all versions studied) where it can be found (*Consistent* requirement).
- If a plugin included in one version has packages that we could find in another plugin in another version (preceding or following), then include that other plugin (*Consistent* requirement).

As a result, we included 15 plugins in the first version (2.0) and 33 in the last (3.1). They are listed in Table 1.

4.2.2 JHotDraw

JHotDraw[8] is a framework for structured, two-dimensional, drawing editors. It is developed in Java. From its first version, HotDraw (the Smalltalk version) was developed as a "design exercise". For example, it relies heavily on well-known design patterns. Recently, several notes in the documentation[9] made explicit reference to restructurings of parts of the framework:

- v. 7.4 (2010-01-16) "The package org.jhotdraw.draw has been split up into sub-packages";

[8] http://www.jhotdraw.org/
[9] http://www.randelshofer.ch/oop/jhotdraw/Documentation/changes.html

Table 1: Plugins of Eclipse included in the experiment for the main versions considered

Packages	2.0	2.1	3.0	3.1
org.eclipse.ant.core	×	×	×	×
org.eclipse.compare	×	×	×	×
org.eclipse.core.commands				×
org.eclipse.core.boot	×	×		
org.eclipse.core.expressions			×	×
org.eclipse.core.filebuffers			×	×
org.eclipse.core.resources	×	×	×	×
org.eclipse.core.runtime	×	×	×	×
org.eclipse.core.runtime.compatibility			×	×
org.eclipse.core.variables			×	×
org.eclipse.help	×	×	×	×
org.eclipse.help.appserver		×	×	×
org.eclipse.help.base			×	×
org.eclipse.help.ide			×	
org.eclipse.help.ui	×	×	×	×
org.eclipse.jface		×	×	×
org.eclipse.jface.text		×	×	×
org.eclipse.osgi			×	×
org.eclipse.platform	×	×	×	×
org.eclipse.search	×	×	×	×
org.eclipse.swt	×	×	×	×
org.eclipse.text		×	×	×
org.eclipse.tomcat	×	×	×	×
org.eclipse.ui	×	×	×	×
org.eclipse.ui.cheatsheets			×	×
org.eclipse.ui.console			×	×
org.eclipse.ui.editors		×	×	×
org.eclipse.ui.forms			×	×
org.eclipse.ui.ide			×	×
org.eclipse.ui.views		×	×	×
org.eclipse.ui.workbench		×	×	×
org.eclipse.ui.workbench.compatibility		×	×	×
org.eclipse.ui.workbench.texteditor		×	×	×
org.eclipse.update.core	×	×	×	×
org.eclipse.update.ui	×	×	×	×
org.eclipse.update.ui.forms	×	×		

- 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."
- v. 7.6 (2011-01-06) "Drawing Framework – User interface classes which depend on the drawing framework have been moved from the org.jhotdraw.gui package into the new org.jhotdraw.draw.gui package."

For the first two, very shortly after the restructured version there is a minor version correcting some bugs. We consider these minor versions rather than the first restructured one (7.4.1, 7.5.1 and 7.6). We also add the last version before the first restructuring (7.3.1).

One can see that, contrary to Eclipse and Seaside (§4.2.4), these are localized restructurings. We follow the rules defined in Section 4.1 to identify which packages are restructured or not.

4.2.3 Vivo

Vivo[10] aims at enabling collaboration and facilitating discovery of researchers and collaborators across all disciplines. Vivo is an open source Java web application.

The system was restructured in version 1.5[11]: "The VIVO 1.5 development cycle has also included extensive design work on features anticipated for implementation beginning with version 1.6, *including increased modularity*, the introduction of a separate ontology for display and editing controls, and the addition of a graphical ontology class expression editor." (our emphasis).

The documentation makes clear references to enhancements to the system. We discussed this possibility in Section 3.1.

We looked at versions 1.4, 1.4.1 and 1.5.

4.2.4 Seaside

Seaside[12] is an open source framework for developing dynamic web applications. It is developed in Smalltalk. Between version 2.8 and 3.0, Seaside underwent a huge restructuring effort: "The monolithic Seaside package has been split into several independent modules"[13]. The architecture went from

[10] http://vivoweb.org/
[11] http://vivoweb.org/blog/2012/07/vivo-release-1-v15-announcement
[12] http://www.seaside.st
[13] http://www.seaside.st/community/development/seaside30

27 to 52 packages, with only 4 packages in common.

We consider the restructured version (3.0) as well as the one before (2.8) and the most recent one after (3.0.7).

4.3 Descriptive Statistics

We use different size metrics to give a synthetic view of the systems considered and a first impression on what happened between their successive versions (see Table 2). The metrics are: number of packages, average number of classes per package, average number of methods per class, average number of lines of code per class, and average cyclomatic complexity per method. Restructured versions (global or local) are marked with a (r).

Table 2: Descriptive statistics of the three subject systems. Versions where some restructuring occurred are marked with (r. For Eclipse, we do not detail all the minor revisions (2.0.1, 2.0.2, ...)

vers.	# packages	# classes /pckg	# methods /class	LOC /class	Cyclo.C. /meth.
Eclipse *(Java)*					
2.0	123	19.0	10.6	145.4	1.99
(r)2.1	133	19.7	10.9	152.0	2.05
(r)3.0	264	15.4	10.6	140.7	1.99
3.1	322	14.6	10.8	142.6	2.01
JHotDraw *(Java)*					
7.3.1	20	21.2	10.7	117.3	1.71
(r)7.4.1	38	11.3	10.8	118.6	1.70
(r)7.5.1	41	11.1	10.8	125.3	1.72
(r)7.6	41	11.2	11.0	126.6	1.71
Vivo *(Java)*					
1.4.?					
1.4.1	141	7.5	9.3	107.9	1.87
(r)1.5	157	7.3	9.4	106.8	1.81
Seaside *(Smalltalk)*					
2.8	23	11.0	10.1	41.3	1.30
(r)3.0	62	7.6	8.9	39.7	1.33
3.0.7	62	8.2	8.9	39.9	1.34

Almost all new versions imply in more packages. This is a known behaviour of software systems. This increase can be very large for some restructuring, for example Seaside 3.0 has three times more packages than its base version Seaside 2.8 (27 packages). In this case, we are facing a global restructuring with a common stating that: "The monolithic Seaside package has been split into several independent modules".

Restructurings seem to result in a diminution in the average size (number of classes) of the packages (although the total number of classes, not shown in the table, usually increases). The idea that a good design imposes restrictions on the size of packages is not new and was already noted in [3] or [4]. This is the only consistent phenomenon we can perceive in this preliminary analysis. This is something that we can test in our experiments.

We also give some descriptive statistics for the restructurings themselves (Table 3). We focus on the number of packages: how many before and after the restructuring, how many added, removed or restructured during the restructuring. The rules to define what packages were restructured are described in Section 4.1, in a nutshell a restructured package is one that exists in both versions and looses or gains more than two classes. "Impacted packages in base version" is the proportion of packages in the base version that were either removed or restructured. "Impacted packages in restructured version" is the proportion of packages in the restructured versions that were either added or restructured.

Table 3: Descriptive statistics of the restructurings considered. The two global restructurings are in bold face

	Eclipse		JHotDraw			Vivo	Seaside
	2.0.2 / 2.1	2.1.3 / 3.0	7.3.1 / 7.4.1	7.4.1 / 7.5.1	7.5.1 / 7.6	1.4.1 / 1.5	2.8 / 3.0
# packages in base version	124	**134**	20	38	41	141	**23**
# packages removed	5	**11**	0	0	1	5	**18**
# packages restructured	6	**14**	2	2	1	3	**3**
# packages added	14	**141**	18	3	1	21	**57**
# packages in revision	133	**264**	38	41	41	157	**62**
Impacted in base version	9%	**19%**	10%	5%	5%	6%	**91%**
Impacted in restruct. version	15%	**59%**	53%	12%	5%	15%	**97%**

The two global restructurings are highlighted in bold in the table. One can notice that these two restructurings impact a large number of packages, 95% of the packages in Seaside v. 3.0 were added or restructured from its base version 2.8; and 59% of the packages in Eclipse v. 3.0 were added or restructured from its base version 2.1.3. This is expected and justify calling these global restructurings. One local restructuring seems to come close to these with many new packages: JHotDraw 7.3.1/7.4.1 (53% of new or restructured packages). The decision whether we face a global or local restructuring is mainly rooted in the documentation of the systems, and may be we could consider JHotDraw 7.3.1/7.4.1 as a global restructuring. Note however, that this local restructuring differs from the global ones in that it removes or restructures few packages (10% of impacted packages in base version). We will look more closely at this local restructuring in our unpaired experiment.

In Table 3 on the third line, we indicate the number of individual packages that we identify as restructured from a base version to a restructured one. There are actually not many of these and it could harm the paired statistical test. Eclipse provides most (14 out of 36=39%) of all restructured packages. More restructurings would be needed to improve this aspect of the testbed. Since they don't need to be global restructurings, it should be possible to make progresses on this issue.

5 Experimental Results

We now present some experimental results on applying different metrics on our test bed. We test the size/complexity metrics used in Section 4.3, and five well known cohesion or coupling metrics (Section 2.1).

5.1 Global restructurings, unpaired setting

The results for the global restructurings (in bold) are given in Table 4. As discussed in the previous section, we also include results for one larger local restructuring (JHotDraw 7.3.1/7.4.1).

The size/complexity metrics decrease (negative variation from base to restructured versions) in the three cases. On average, there are less classes per packages, less methods and LOCs per class and the method have a smaller cyclomatic complexity. This agrees with intuition (see discussion in Section

Table 4: Difference in metric value (restructured version - base version) for three restructuring revisions. The two global restructurings are in bold face. Results significant at the 5% level are marked with (**), at the 10% level marked with (*). Wilcoxon test, unpaired setting.

	Eclipse 2.1.3/3.0	JHotDraw 7.3.1/7.4.1	**Seaside 2.8/3.0**
# class/package	**-2.0 ****	-2.0	**-4.0 ****
# method/class	**-0.9 ***	-1.7	**-2.8 ****
LOC/class	**-17.6 ****	-21.1	**-6.7**
Cyclo.C./method	**-4.2 ****	-4.2	**-3.2**
Marting Cohesion	**-0.101**	-0.039	**-0.737 ****
Bunch Cohesion	**-0.001**	0.035 *	**0.046**
Afferent coup. (Ca)	**-2.0 ****	-1.0	**-37.0 ****
Efferent coup. (Ce)	**-8.0 ****	-11.0	**-33.3 ***
Bunch coupling	**0.000 ****	0.001	**-0.043 ****

4.3 and [3] or [4]). But these results are statistically significant (when they are) only for the two global restructurings.

We conclude that in the two global there is a significant decrease in size and complexity of the packages.

For the cohesion metrics, The results are not what one could expect. Martin cohesion exhibits only negative differences, it decreased, being statistically significant at the 5% level for Seaside. Bunch cohesion has slightly better results we a positive difference for Seaside, but this result is not significant. It also increased for the JHotDraw local restructuring with a significance at the 10% level, however, the unpaired setting is probably not relevant for this structuring since it is not a global restructuring, suggesting the change in metric could be the result of some other activity not directly related to the restructuring.

Overall, these cohesion metrics do not seem able to reliably measure the increase in quality resulting from the global restructurings of our testbed. Cinnéide *et al.* [17] already shed some doubts on the relevance of these metrics.

For coupling the results appear better. Differences are mostly negative (particularly Martin's two coupling metrics) which indicates a lower cou-

pling after restructuring. This is the expected behavior. Bunch coupling is decreased for seaside and is stationary for the two other. Moreover, the results are statistically significant for the global restructurings (for Ce on Seaside 2.8/3.0, significance can only be found at the 10% level). Results are less clear for the local restructuring, but this is compatible with the idea that the overall design of the system could worsen while locally some packages are improved (Section 3.5).

These opposite results between cohesion and coupling are disturbing. For Bunch cohesion and coupling, the difference between the two formulas resides in considering dependencies within a package (cohesion) or dependencies between a package and the outside (coupling). For a constant set of dependencies, improving cohesion implies more dependencies within the packages, therefore, less dependencies crossing the boundaries of the packages, and therefore, an improved coupling too.

For Martin's metric things are more complex since cohesion is based on dependencies internal to a package (similar to Bunch cohesion), whereas the coupling metrics are based on external entities but not dependencies.

We already noted that, in our experiments, the number of packages and classes increased, sometime very much. We did not monitor the number of dependencies because it falls outside of the scope of this paper. But the relative variation in number of packages, classes and dependencies should have an impact on the metrics' values, independently of the quality of the modularization. More experiments will be needed to clarify this point.

5.2 Normal versions, unpaired setting

As a comparison base with the previous experiment, we applied the same tests to several "normal versions", i.e. versions that were not restructuring ones. We tried not to over represent any of the system in this experiment. These versions are, for Eclipse: 2.0/2.0.1 and 3.1/3.1.1; for JHotDraw: 7.2/7.3, and 7.3/7.3.1; for Vivo: 1.4/1.4.1; and for Seaside: 2.7/2.8, and 3.0/3.0.7. We do not give all the results here to save space, but we will comment them and highlight the most important ones.

In none of these minor, normal revisions, could we detect a statistically significant change in the value of any of the metrics. Moreover, many of the results are null (no variation). This seems natural as they can be small modifications (bug fix).

These results seem to indicate that there are real differences between

Table 5: Difference in metric value (minor revision - base version) for some "routine" revisions in the subject systems. None of the results is significant at the 10% level. Wilcoxon test, unpaired setting

	Eclipse		JHotDraw		Vivo	Seaside	
	2.0 / 2.0.1	3.1 / 3.1.1	7.2 / 7.3	7.3 / 7.3.1	1.4 / 1.4.1	2.7 / 2.8	3.0 / 3.0.7
# class/package	0.0	0.0	1.000	0.0	0.0	1.000	0.000
# method/class	0.0	0.0	0.488	0.0	0.0	-0.675	0.000
LOC/class	0.0	0.0	3.980	0.0	0.0	-7.334	-0.187
Cyclo.C./method	0.0	0.0	1.118	0.0	0.0	1.068	0.500
Marting Cohesion	0.0	0.0	0.000	0.0	0.0	0.357	0.000
Bunch Cohesion	0.0	0.0	-0.017	0.0	0.0	-0.016	0.000
Afferent coup. (Ca)	0.0	0.0	0.524	0.0	0.0	-11.000	-3.000
Efferent coup. (Ce)	0.0	0.0	4.000	0.0	0.0	-3.000	-4.000
Bunch coupling	0.0	0.0	0.000	0.0	0.0	-0.017	-0.005

normal versions and restructuration versions where the difference in metrics values very often statistically significant. The test with the local restructuring JHotDraw 7.3.1/7.4.1 (Table 4), already hinted at this.

5.3 Local restructurings (paired setting)

Finally, we also tested all restructurings in a paired setting, that is to say we compare the quality of individual packages before and after a restructuring. In this experiment, all the restructured packages from the three systems and all restructurings are mixed together (31 packages, see Table 3). This is not a problem since each package is only compared to itself.

As for the global restructurings experiment, the number of class per package and cyclomatic complexity metrics decrease after restructuring, and the results are statistically significant (at the 10% level for number of classes). This fits expectations.

Contrary to the global restructuring, the number of method per class and LOC per class increase. However, these results are not statistically significant, so we cannot deduce much from this. More data points (packages in other systems) could be necessary to obtain more significant results.

The cohesion metrics also give results similar to the previous experiment

Table 6: Difference in metric value (restructured version - base version) for all restructured packages (local + global restructurings) in the subject systems. Results significant at the 5% level are marked with (**), at the 10% level marked with (*). Wilcoxon test, paired setting

# class/package	-3.0
# method/class	-0.2
LOC/class	-1.7
Cyclo.C./method	-7.3 **
Marting Cohesion	-0.177 **
Bunch Cohesion	0.000
Afferent coupling (Ca)	0.0
Efferent coupling (Ce)	-4.0
Bunch coupling	0.000 *

with decreasing value for Martin's cohesion (statistically significant) and close to stationary value for Bunch cohesion (not significant). Again, this is contrary to expectations. One could argue that these results are biased by the over representation of Eclipse in the restructured packages (see Section 4.3). However, Eclipse is precisely a system for which Martin cohesion did not give statistically significant results in the global experiment. Similarly, the variation in this experiment is slightly positive (increase) for Bunch cohesion whereas it was slightly negative in the global restructuring experiment.

If the results of this experiments are coherent with the global restructuring experiment, this is less the case for coupling metrics. Afferent coupling (Ca) increased (not significant) whereas it decreased in the Section 5.1; Efferent coupling (Ce) decreases here as in the previous experiment, but the result is not statistically significant here; Bunch coupling continues stationary (statistically significant).

The lack of statistical significance of many results in this experiment could be due to the relative small number of subjects (31), or be a fundamental property of restructuring. More subject systems would be needed to clarify this point now that the experimental setup is defined.

6 Related work

6.1 Cohesion of Classes

We found, in the literature, various tentative evaluation of the practical pertinence of cohesion metrics for classes. Although class cohesion is an entirely different problem (see discussion in Section 2.1) they illustrate the different approaches used to validate such metrics.

In [9], Briand proposes some theoretical validation that a cohesion metric should respect, one called monotonicity, states that adding relationship to a class cannot decrease its cohesion; another states than merging two unrelated classes cannot increase its cohesion. Such requirements are strongly based on the high-cohesion/low-coupling principle, and not on the state of practice. Our experiments and previous research (see Section 2.2) suggest that this principle may not be as pertinent as usually believed.

Counsell *et al.*, in [8], assess whether some metrics correlate with the perception of developers on class cohesion. They asked 24 programmers to manually evaluated the cohesion of 10 C++ classes (randomly taken from real world application) and compared their answers to some metrics like class size, or two coupling metrics. The results suggest that class size does not impact the perception of cohesiveness, and that the two coupling metrics behave differently.

In [22], Alshayeb evaluates the effect of refactorings on five class cohesion metrics (LCOM1 to LCOM5). At the class level his approach is the same as our: take a real refactoring case and evaluate how well cohesion metrics report the expected increase in quality. The conclusion is that overall the cohesion metrics improved and are therefore validated by this very small experiment (eight classes involved).

Cinnéide *et al.* recently published the results of an effort to assess the validity of class cohesion metrics [17]. The experiment this time is on real world systems with a large number of classes. They setup a laboratory experiment where they apply randomly different refactorings and evaluate the impact they have on different class cohesion metrics. One conclusion is that the metrics do not agree among them, thus pointing to the probable lack of relevance of at least some of them (see also Section 3.1). This experiment however does not tell what metrics are more relevant in practice because the value of the refactored versions, generated randomly, is completely unknown.

Dallal and Briand [23] compare one new cohesion metric to eleven other

to show that it is not completely correlated and thus bring some new point of view on the problem. It does not tell whether one metric correlates better with the perceived cohesiveness of classes. They also correlate their new metric to fault proneness of classes. This is a practical validation, but it does not relate to the main purpose of the cohesion metric.

6.2 Cohesion of Packages

We could not find any evaluation of architectural design quality metrics. As already stated, working at the level of packages is much more difficult because the source code can be some order of magnitude larger.

In [7] the first author already alluded to the difficulty of evaluating the results of automatic remodularization techniques without any know proven instrument.

Other researchers perceived that problem [3, 4, 15] without proposing any solution to it.

7 Conclusion

Although the advantages of a good architecture are well heralded, there is no formal definition of how to measure this property. Existing architectural quality metrics mostly rely on the idea that packages should be highly cohesive and weakly coupled. But even this principle, and the metrics based on it, is untested and unproven.

A possible explanation for this lack of formal evidence may lie in the practical difficulty to test such metrics. Manual validation is virtually impossible due to the costs involved in validating the results of the metrics on large real systems. Validation against a golden solution is not acceptable either because several equally valid architectures may be proposed for a given system.

In this paper, we propose an experimental setup that allows to test architectural quality metrics. The basic idea is to use real cases of system restructurings and see whether a given metric is able to register the expected improve in quality between the previous version and the restructured version of the system.

We formally define our experimental setup and the constraints that subject systems (and their restructured versions) should respect. We then ex-

emplified the test-bed with four real systems (Eclipse, JHotDraw, Vivo, Seaside) in two languages (Java, Smalltalk). We finally applied the experiment on several metrics: size metrics and some well-known cohesion or coupling metrics.

From this example we could draw some initial conclusion on a "typical" restructuring:

- Decrease in the number of class per package;
- Decrease in the cyclomatic complexity of methods;
- possible decrease in the number of method per class and number of LOC per class (both results to be confirmed with more experiments);
- general lack of relevance of the five cohesion or coupling metrics tested, either because they did not give the expected results or because the results were not statistically significant. Martin's afferent and efferent coupling were the closest to be validated with significant and expected results on the global experiment (unpaired setting).

More experiments are required, first to try to validated existing cohesion/coupling metrics, second to obtain more definitive results. This second point would require more subject systems.

References

[1] M. Lehman, Laws of software evolution revisited, in: European Workshop on Software Process Technology, Springer, Berlin, 1996, pp. 108–124.

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

[3] F. B. Abreu, M. Goulão, Coupling and cohesion as modularization drivers: Are we being over-persuaded?, in: CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, 2001, pp. 47–57.

[4] P. Bhatia, Y. Singh, 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, 2006, pp. 972–979.

[5] 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.

[6] C. Taube-Schock, R. J. Walker, I. H. Witten, Can we avoid high coupling?, in: Proceedings of ECOOP 2011, 2011.

[7] N. Anquetil, T. Lethbridge, Comparative study of clustering algorithms and abstract representations for software remodularization, IEE Proceedings - Software 150 (3) (2003) 185–201. doi:10.1049/ip-sen:20030581.
URL http://rmod.lille.inria.fr/archives/papers/Anqu03a-IEESoft-ComparativeStudy.pdf

[8] S. Counsell, S. Swift, A. Tucker, 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, 2005, pp. 161–172.

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

[10] S. A. Ebad, M. Ahmed, An evaluation framework for package-level cohesion metrics, International Proceedings of Computer Science and Information Technology 13 (2011) 239–43.

[11] S. Mancoridis, B. S. Mitchell, Y. Chen, 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), IEEE Computer Society Press, Oxford, England, 1999.

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

[13] G. A. Hall, W. Tao, J. C. Munson, Measurement and validation of module coupling attributes, Software Quality Control 13 (3) (2005) 281–296. doi:10.1007/s11219-005-1753-8.

[14] N. Anquetil, J. Laval, Legacy software restructuring: Analyzing a concrete case, in: Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11), Oldenburg, Germany, 2011, pp. 279–286.
URL http://rmod.lille.inria.fr/archives/papers/Anqu11a-CSMR2011-Coupling.pdf

[15] H. Abdeen, S. Ducasse, H. A. Sahraoui, Modularization metrics: Assessing package organization in legacy large object-oriented software, in: Proceedings of the 18th IEEE International Working Conference on Reverse Engineering (WCRE'11), IEEE Computer Society Press, Washington, DC, USA, 2011.
URL http://hal.inria.fr/docs/00/61/45/83/PDF/ModularizationMetrics-INRIA.pdf

[16] n.n., Rational unified process: Best practices for software development teams (2000). doi:10.1.1.27.4399.
URL http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

[17] M. Cinnéide, L. Tratt, M. Harman, S. Counsell, I. Moghadam, Experimental assessment of software metrics using automated refactoring, in: Proc. Empirical Software Engineering and Management (ESEM), 2012, to appear.

[18] S. Sarkar, G. M. Rama, A. C. Kak, Api-based and information-theoretic metrics for measuring the quality of software modularization, IEEE Trans. Softw. Eng. 33 (1) (2007) 14–32. doi:10.1109/TSE.2007.4.

[19] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering: an introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[20] I. Turnu, G. Concas, M. Marchesi, S. Pinna, R. Tonelli, A modified Yule process to model the evolution of some object-oriented system properties, Inf. Sci. 181 (2011) 883–902.

[21] K. Mordal, J. Laval, S. Ducasse, Evolution et Rénovation des Systèmes Logiciels, Hermès, 2012, Ch. Modèles de mesure de la qualité des logiciels, à paraître.

[22] M. Alshayeb, Refactoring effect on cohesion metrics, in: Proceedings of the 2009 International Conference on Computing, Engineering and Information, ICC'09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 3–7. doi:10.1109/ICC.2009.12.
URL http://dx.doi.org/10.1109/ICC.2009.12

[23] J. Al-Dallal, L. C. Briand, A precise method-method interaction-based cohesion metric for object-oriented classes, ACM Trans. Softw. Eng. Methodol. 21 (2) (2012) 8. doi:10.1145/2089116.2089118.