# Refactoring - Improving Coupling and Cohesion of Existing Code

Bart Du Bois and Serge Demeyer
Lab On ReEngineering
Universiteit Antwerpen
{bart.dubois, serge.demeyer}@ua.ac.be

Jan Verelst
Lab on Evolvable Information Systems Architectures
Universiteit Antwerpen
jan.verelst@ua.ac.be

## Abstract

*Refactorings are widely recognised as ways to improve the internal structure of object-oriented software while maintaining its external behaviour. Unfortunately, refactorings concentrate on the treatment of symptoms (the so called code-smells), thus improvements depend a lot on the skills of the maintainer. Coupling and cohesion on the other hand are quality attributes which are generally recognized as being among the most likely quantifiable indicators for software maintainability. Therefore, this paper analyzes how refactorings manipulate coupling/cohesion characteristics, and how to identify refactoring opportunities that improve these characteristics. As such we provide practical guidelines for the optimal usage of refactoring in a software maintenance process.*

Refactorings — behavior-preserving source-code transformations — allow the automated redistribution of pieces of source code over the class hierarchy. The underlying objective is to improve the quality of the software system, with regard to future maintenance and development activities. Unfortunately, while it is clear that we *can* use refactorings to restructure software systems, it is unclear *how* to use them in order to improve specific quality attributes that are indicators for a good design.

We start from the assumption that coupling and cohesion characteristics may serve as indicators for the optimal distribution of responsiblities over the class hierarchies. Thus, rather than saying that refactoring will improve the design, we aim for a less ambitious goal of improving the coupling and cohesion. *Cohesion* then corresponds to the degree to which elements of a class belong together, and *coupling* is the strength of association established by a connection from one class to another [11].

Therefore, in this paper we analyze under which conditions applications of refactorings improve specific coupling and cohesion dimensions. Guidelines for applying the refactorings under these conditions are composed, and validated on an open source software system regarding both improvement and applicability.

The paper is organized as follows. Section 1 discusses which guidelines optimize the improvement of coupling and cohesion for specific refactorings. Section 2 elaborates on a validation of these guidelines regarding their improvement and applicability. Section 3 discusses lessons learned during the detection of the refactoring opportunities and their resolution. Section 4 outlines future research and related work. Finally, Section 5 summarizes the contributions of this work.

## 1. Mapping refactorings to coupling/cohesion

Refactoring opportunities are locations in the source where a) there is a need for improvement regarding a quality attribute; b) a refactoring can be applied that will reorganize the code while preserving the behavior of the software system; and c) the application of the refactoring will indeed improve the quality attribute.

To be able to recognize opportunities for improving coupling and cohesion, we analyze how refactorings affect these quality attributes. First, we will describe the concrete dimensions in which these two abstract quality attributes can be divided. Second, we introduce the refactorings we set out to investigate. Finally, we explain how we can map these refactoring postconditions on coupling and cohesion metrics.

### 1.1. Coupling and cohesion dimensions

Coupling and cohesion are very wide concepts, which consist of many different dimensions. We use the categorization of [3], which identified 2 cohesion and 5 coupling dimensions using principal component analysis. For each of these categories, a set of metrics can be associated as key indicators [2, 1].

**Cohesion** when methods share common attribute usages, they are similar regarding internal data usages, and therefore belong together. They can also be dependent upon each

other by method invocation. We differentiate between **non-normalized** and **normalized** cohesion, in which the latter is independent of the number of methods of the class.

**Import coupling** method invocations and attribute usage bind classes together. By calling a method, you import its implementation in your scenario. Such dependencies are the main transport routes for ripple effects during local alterations. The value of this import coupling depends upon the number of distinct methods called, and the calling frequency.

**General coupling** in references to other classes, implicit assumptions can turn invalid over time. The dependency caused by these references can be as weak as simple class-name dropping, or as strong as manipulating internal resources of the class.

**Export coupling** when a class publishes services or is simply instantiated, other classes can refer to these resources. This causes a form of dependency where local changes can ripple to clients. Moreover, not only does a class publish its own implementation, it also indirectly publishes the implementation upon which it depends itself, being the methods called.

**Aggregated import coupling** when a class manipulates data, it depends upon the class of which the data is an instance. Among others, this import coupling can be caused by attributes or parameters.

We omitted one dimension: export coupling to descendant classes. This is a special case dimension which is specific to the context of inheritance. We feel that this would require deeper investigation into other aspects of inheritance, which is not the focus of our work.

### 1.2. Refactorings under study

The objective of our selection was to investigate those refactorings which redistribute responsibilities either within the class or in between classes. This redistribution is the main principle to address coupling and cohesion problems.

Fowler's catalogue describes 72 refactorings over 6 categories [6]. As an initial set of refactorings under study, we selected one refactoring which breaks up a method (*Extract Method*), one refactoring that redistributes a method over the class hierarchy (*Move Method*), two refactorings that compose new responsibilities (*Replace Method with Method Object, Replace Data Value with Object*), and one refactoring that divides responsibilities (*Extract Class*). These refactorings are among the most fundamental and most widely used.

Once we have identified how these refactorings can change coupling and cohesion characteristics, we set out guidelines to optimize their usage for improvement. We will specify profiles that describe how opportunities for the application of each guideline can be found. This heuristic for reverse engineering refactoring opportunities filters out those opportunities of which the resolution will not lead to an improvement in coupling/cohesion.

### 1.3. Composing refactoring guidelines

In this section, we illustrate how refactoring guidelines can be composed by exploiting coupling and cohesion impacts. Our hypothesis is that adherence to these guidelines will provide better results regarding coupling and cohesion.

The analysis of this impact is based on a uniform metamodel in which both refactoring postconditions and metrics were expressed. This makes it possible to rewrite the refactoring postconditions as conditions on model instance cardinalities, and subsequently on the metrics using these calculations. For further technical details, we direct the interested reader to [5].

It is clear that the impact of a refactoring on a coupling or cohesion dimension is dependent upon the specific source context. Therefore, it actually makes more sense to talk about an impact spectrum, in which the specific source context - the refactoring opportunity - dictates the position within the spectrum. The boundaries of this impact-spectrum can be an improvement (+), deterioration (-) or no change at all (0) of a specific coupling or cohesion dimension for specific applications of the refactoring.

Table 1 illustrates this spectrum by specifying the best and worst case impact for each of the refactorings under study. Respectively, this is the impact of the most optimal and the least optimal application of a specific refactoring.

**Extract Method** extracts a set of statements appearing in one or more methods into a new method.

Extracting statements which reference attributes can transform attribute-dependency into method-interaction. When those statements are extracted which a) reference a reasonable proportion of attributes; and b) appear in duplicates over a reasonable amount of methods, the implicit similarity among the methods (attribute-cohesion) is made explicit through invocation of the extracted method (method cohesion).

Multiple extraction - extracting a group of statements from more than one method - can decrease import coupling as it removes dependencies which are duplicated across methods. Aggregated import coupling and export coupling

**Table 1. Best and worst case impact of a refactoring on a cohesion or coupling dimension. The table indicates to which extent guidelines on the application of a refactoring can imply a shift in the spectrum of impact on coupling or cohesion. Rather than describing the increase/decrease of the associated metrics, the cells indicate an improvement (+), deterioration (-) or neutral impact (0).**

| | Extract Method | Move Method | Replace Method w Method Object | Replace Data Value w Object | Extract Class |
|---|---|---|---|---|---|
| Normalized cohesion | + — - | + — - | 0 | + — 0 | + — - |
| Non-normalized cohesion | + — - | + — 0 | 0 | + — 0 | + — - |
| Import Coupling | + — 0 | + — - | + — - | - | + — - |
| General Coupling | 0 | + — - | + — - | - | + — - |
| Export Coupling | - | + — - | + — - | - | + |
| Aggregated import coupling | 0 — - | + — 0 | 0 | + — - | + — - |

are made explicit in the signature of the new method, and can therefore increase.

To exploit these impacts, we compose the following guideline:

G1 *Localize dependencies* Extract those groups of statements which have a lot of dependencies in common with many other methods of the class.

**Move Method** moves a method from one class to another, possibly adding a parameter when resources of the original class are used, and removing that parameter which is an instance of the target class.

Moving a method that does not refer to local attributes or methods, or is called upon by only few local methods will increase cohesion.

Moving a method that calls external methods more frequently than it is called itself will decrease import coupling. Moving a method that encapsulated coupling to other classes to a known class decreases both import and general coupling. Export coupling can be decreased by moving a methods that do not refer to neither attributes nor methods nor the name of the class.

G2 *Localize dependencies* Move those methods that do not use local resources, are called upon seldom and themselves refer mostly to a single external class.

G3 *Separate concerns* Break up a method that depends on many different external classes into pieces which mostly refer to only a single external class. Apply G2 on each of these extracted methods. Thereafter, the original method will then act as a coordinator which directs the collaborations between the responsible classes, and can be moved itself to a class which fits this coordination responsibility.

**Replace Method with Method Object** creates a new class which has only one method and a constructor. The local variables and parameters of the method are promoted to attributes of the newly created class.

Replacing a method that does not references local attributes or methods will increase cohesion.

Replacing a method that encapsulates coupling to other classes decreases import and general coupling. Export coupling can be decreased when the method manipulates instances of other classes, yet has no local variables or parameters that are instances of the original class.

G4 *Localize dependencies* Replace those methods which refer mostly to *many different* external resources, and are called upon frequently by local methods of the class. The original method will be transformed in a client of the coordination role which the newly created class will play.

**Replace Data Value with Object** encapsulates a set of attributes in a new class. This set of attributes in the original class is replaced by an instance of that new class.

When a group of attributes is frequently used together by many local methods, cohesion will increase by replacing them with an object.

Replacing attributes with a single object will introduce both import and general coupling to the newly created class. Export coupling will remain constant when the attributes are not instances of the local class. When there were no other attributes of the same external type as the selected ones, aggregated import coupling will decrease.

G5 *Localize dependencies* Group those attributes which are used as a unity throughout the local methods of the class and which are mostly instances of external classes.

**Extract Class** creates a new class which contains a selected group of methods and attributes from the original class.

Cohesion will increase when a connected set of attributes and methods are extracted. Such connection is bidirectional: a method is connected to another method when it either calls it, or is called by it.

Import coupling (including aggregated) will decrease when the extracted set of attributes and methods encapsulated coupling to external classes. Export and general coupling will decrease when the extracted methods and attributes do not refer to the original class (minus the extracted part).

G6 *Separate concerns* Extract those groups of methods and attributes that are neither referenced by, nor refer themselves to other methods or attributes by respectively method invocation or attribute usage.

## 2. Validation

In order to support the claim that these guidelines help in improving coupling and cohesion of existing code, we must demonstrate that a) their usage improves key coupling and cohesion metrics; and b) potential targets for their application are not hard to find. These form the two major subgoals for a realistic validation of these guidelines.

To challenge the usefulness of these guidelines, we applied each of the guidelines several times and evaluated their impact. We set out to detect the associated refactoring targets in those classes of the open source software system Apache Tomcat which exhibit deteriorated cohesion and coupling characteristics. These were located in the org.apache.tomcat.{core,startup} packages. The resulting improvement regarding coupling and cohesion characteristics are presented in SubSection 2.2.

First, we provide our resulting impression of the applicability of these guidelines. We feel this aspect is crucial, as it allows to focus future investments in tool support for those guidelines which improve coupling and cohesion and can be applied most frequently.

### 2.1. Applicability of the guidelines

Each of the guidelines specifies a profile for refactoring opportunities based on their (common) attribute and method usage. During the validation, we searched for method profiles which matched the specified guideline profiles interactively. This is our heuristic for finding refactoring opportunties.

To do this, we implemented a tool that analyzed the attribute and method usage of each method of a class, and performed primitive calculations as required for each of the metrics associated with the coupling/cohesion dimensions.

However, as our tool is still immature, some computations such as finding duplicate groups of statements containing attribute references were done by reading the code. Therefore, our application of the heuristic for finding refactoring opportunities is imperfect. Moreover, these frequencies are specific to (the selected parts of) the chosen software system. Yet, we feel that these numbers can help for a first impression.

We found 5 opportunities for Extract Method$_{G1}$, 4 opportunities for Move Method$_{G2}$ and 4 for Move Method$_{G3}$, 1 for Replace Method with Method Object$_{G4}$, 3 for Replace Data Value with Object$_{G5}$ and 3 for Extract Class$_{G6}$. These opportunities were detected in 3 packages, consisting of a total of 12 classes, 167 methods and 3797 lines of code. This limited number of opportunities is caused by their strict profiles:

- Extract Method$_{G1}$ can only be applied when multiple methods access the same set of attributes in the same way. Together with Move Method$_{G3}$, opportunities for this guideline can be found in classes consisting of few large methods.

- Move Method$_{G2}$ and Replace Method with Method Object$_{G4}$ can only be applied on methods which do not use local resources. These are methods which are either explicitly or implicitly (can be made) static. To differentiate between these profiles, opportunities for Move Method$_{G2}$ are mostly small methods as they manipulate only a single class. Opportunities for Move Method$_{G3}$ and Replace Method with Method Object$_{G4}$ are larger methods which refer to numerous external classes. Specifically, Replace Method with Method Object$_{G4}$ opportunities are large, data oriented methods, which have a lot of parameters and local variables.

- Replace Data Value with Object$_{G5}$ can only be applied on mainly data-oriented classes, in which many methods share a common set of referenced attributes. Opportunities for Extract Class$_{G6}$ are different in that the attribute usage is different over the various methods, and is therefore different from simple reading and writing of the attribute value.

These informally described profiles allow to identify refactoring targets by analyzing attribute and method usages. This identification filters out potential targets for the application of a refactoring which would not result in the improvement of coupling or cohesion. By using this filtering principle, we can build up a heuristic for identifying valuable refactoring opportunities.

**Table 2. Results of the validation, in which each of the guidelines were applied numerously. The cell values indicate wether the average result could be classified as a (B)est case, (W)orst case, (E)xpected or (S)uboptimal impact.**

| | Extract Method$_{G1}$ | Move Method$_{G2}$ | MM$_{G3}$ | Replace Method w Method Object$_{G4}$ | Replace Data Value w Object$_{G5}$ | Extract Class$_{G6}$ |
|---|---|---|---|---|---|---|
| Normalized Cohesion | W | B | W | E | B | B |
| Non-normalized Cohesion | W | B | B | E | B | B |
| General coupling | E | B | B | S | N | S |
| Export Coupling | E | B | B | B | E | E |
| Aggregated import coupling | B | W | E | S | W | W |

### 2.2. Results from applying the guidelines

Table 2 summarizes the results of finding and resolving refactoring opportunities for the proposed guidelines in those classes of the Open Source software system Apache Tomcat which exhibit deteriorated coupling and cohesion characteristics.

These opportunities were located in the org.apache.tomcat.{core,startup} packages. After resolving the opportunity, changes in the values of the set of metrics associated with each coupling/cohesion dimension were calculated. Finally, we calculated the "average impact" of these resolutions on the various dimensions. No cases were found where the impact of the application of a particular refactoring on a particular metric was an improvement for the resolution of one opportunity and a deterioration for the resolution of another. Therefore, this average impact indicates how a coupling/cohesion dimension was affected in the average resolution of a refactoring opportunity.

Each cell value categorizes the average impact of the guideline as conforming either to the best case, worst case, expected case or suboptimal impact on the specific coupling/cohesion dimension. For the import coupling dimensions, we were unable to compute associated metrics due to lack of tool support.

We chose for this representation as it compares what has been experienced in practice with the spectrum resulting from the impact analysis which was illustrated in Table 1. This way, we can evaluate whether the adherence to the guideline truly had the best possible impact for that particular refactoring (as bounded by the impact spectrum). As the table is self-describing, we will not run through it's content in the text.

### 2.3. Interpretation

Excellent results were provided by the guidelines regarding Move Method (G2 and G3). Both guidelines are an excellent help in improving coupling and cohesion of existing code.

Good results were provided by the guidelines on Replace Method with Method Object, Replace Data Value with Object and Extract Class. These guidelines are a good help in improving cohesion, yet provide only limited help in resolving coupling issues. More specifically, they all trade strong coupling with a number of external classes with strong coupling to only a single class.

Disappointing results were provided by the guideline on Extract Method. This guideline does not help in improving neither cohesion nor coupling. We stongly believe that strict boundaries should be specified on the minimal number of attribute or method usages in the extracted statements.

Summarizing, we are convinced that the guidelines on Move Method, Replace Method with Method Object, Replace Data Value with Object and Extract Class can be used to identify good refactoring opportunities. Their application will improve coupling and cohesion. While we assume this improvement will lead to improved maintainability, this still requires rigorous empirical validation of the reduction in maintenance effort for representative maintenance tasks on representative software systems.

## 3. Lessons Learned

**Opportunities for improvements in coupling and cohesion are scarce** Over the domain of all possible applications of a refactoring on a software system, those opportunities which can truly improve coupling and cohesion are hard to find.

Most of the time, the resolution of a refactoring opportunity leads to the advent of new opportunities. This is because the resolution of a refactoring opportunity is triggered by specific coupling/cohesion characteristics, and subsequently changes these characteristics. This can cause thresholds for other guidelines to be surpassed, thereby introducing a new refactoring opportunity.

In other words, the refactoring process is a dynamic pro-

cess which requires continuous re-evaluation of the targeted quality attributes over the detection-resolution cycle. Therefore, the number of refactoring opportunities will increase and decrease over the application of refactorings. This corresponds to the changing needs for improvement regarding coupling/cohesion during the evolution of a software system.

**Evaluation of refactoring series requires automation** In order to optimize the efficiency of a refactoring process, both analysis and resolution can be automated. To automate the analysis, information on both internal and external attribute and method usage, including the frequency, must be retrievable.

Using this information, a profile for each method can be computed, which can be compared with the profile of each guideline. For efficiency, this comparison can be ordered according to the improvement regarding coupling and cohesion, as described in the previous sections.

Once a match between a method (or set of methods) and a guideline has been identified, the target for redistribution can be determined using the information gathered in the analysis.

When tool support for such an automated refactoring process is available, a series of specific applications of guidelines can be considered as paths in a search space. Therefore, it is possible to compute such an optimal path by trial-and-error, and only commit the result of that series of refactorings which performed optimal regarding the improvement of coupling cohesion.

This way, refactoring strategies can be statistically identified as being a) beneficial for coupling and cohesion; and b) frequently applicable. Such efforts could make reuse of refactoring know-how at the level of (series of) composite refactorings feasible.

**Guidelines should be customizable** We specified the proposed guidelines using qualitative descriptions such as: a lot of common attribute usage, references to many different external classes, etc. To be able to adhere to a guideline, these descriptions must be quantified, by searching for specific thresholds for each of the analysis elements. Research on the customization of such metric thresholds to improve design quality has been performed by [9]. Such a research approach for the customization of guidelines can also be applied on the level of refactorings we adhere to in this paper. This would be very interesting, as this is the level of refactoring for which (limited) tool support exists in current popular Integrated Development Environments.

Moreover, a generic strategy for calculating these thresholds for separate software projects can provide even further customization. Such a strategy would dictate how to customize the detection of refactoring opportunities for specific software systems as maintained by specific teams. This would actually be no luxury, as different people respond differently on specific coupling and cohesion characteristics.

Our main argument for stimulating customization in the refactoring process is that maintainability should be instantiated towards the people maintaining the software. When such a maintenance team would change, different maintainability criteria can be set out, and the refactoring process can be customized to achieve these specific maintainability criteria.

**Refactoring opportunities can be prioritized** During the analysis and, using the guidelines, its subsequent resolution, it became evident that there exists a natural order between the different guidelines. The specific coupling and cohesion characteristics of the class in question can dictate which guidelines should be applied first. For example, Replace Method with Method Object$_{G4}$, Replace Data Value with Object$_{G5}$ and Extract Class$_{G6}$ can be considered as the most coarse grained and efficient improvement in cohesion, as they literally split off responsibilities and compose collaborations. Once cohesive classes have been established, coupling can be targeted at a more fine grained level. To do so, responsibilities can be distributed between classes with guidelines Move Method$_{G2}$ and Move Method$_{G3}$ to improve coupling.

Such feedback allows to formulate suggestions for a specialized refactoring process to improve coupling and cohesion characteristics under the form of a decision tree. When alternative refactoring opportunities are found, we can use this prioritization to identify the most efficient.

However, before such a decision tree can be composed, more refactorings must be analysed. Moreover, more quantitative data on the application of these guidelines over a larger set of software systems is required. Therefore, more, and more advanced tool support is needed, to both automatically detect the associated refactoring opportunities, and also, to automatically apply the associated guidelines.

## 4. Future work

Our work currently disregards other recognized indicators for maintainability such as size and inheritance complexity. As a result, it is imaginable that the improvement in coupling/cohesion is traded for a deterioration regarding these other quality attributes. Therefore, we were very cautious not to over-generalize the usefulness of the proposed guidelines. We explicitly deferred from claims which were unrelated with coupling/cohesion.

In the future work, we plan to apply the approach presented in this paper on these other quality attributes. We expect that trade offs between the improvement of one, and

the deterioration of another quality attribute will be identified.

Ultimately, these trade offs can be evaluated in manipulation experiments in which hypothesis on the effort reduction for specific maintenance tasks caused by the application of the guidelines can be tested.

Our enthusiasm about the results of this work stimulates us in analyzing more refactorings. Again, tool support is a limiting factor in the validation of such analysis. However, during the analysis, the profiles for refactoring opportunities associated with the specific guidelines specify how automated support for refactoring can be implemented.

The main criteria to select other refactorings for future study are: a) intuitive assumption that the refactoring can affect the particular quality attributes; and b) intuitive assumption about the applicability of the refactoring. In example, regarding coupling/cohesion, we are also interested in the replacement of type code (with Class, Subclass or State/Strategy), Replace Subclass with Fields and Replace Parameter with Explicit Methods. Regarding size and inheritance complexity, our wish list includes Pull Up/Push Down Field/Method, Replace Inheritance with Delegation, Replace Conditional with Polymorphism, Replace Type Code with Subclasses.

### 4.1. Related work

We'd like to remark the difference between this work and [4]. While they composed heuristics to find applications of refactorings over the history of a software system, we composed heuristics to apply refactorings.

Our approach for finding refactoring opportunities is extremely close to the work of [9], which described the automatic detection of transformations by selecting candidates based on rules defined in terms of metric thresholds. These rules can also be interpreted as code smells, and be expressed in languages such as the Object Constraint Language [13].

Sahraoui also analyzed the impact spectrum of a refactoring on specific metric values. The difference between our work lies in that we a) translate the analyzed impact on specific metrics to the associated coupling and cohesion dimensions; and b) exploit the results of this analysis by providing guidelines to optimize the improvement of coupling/cohesion.

Another work which is quite related is provided by [12]. Tahvildari analyzed the impact of meta-pattern transformations on an object-oriented metrics suite consisting of metrics for various quality attributes. Our focus is on lower level refactorings and their impact on the specific quality attributes coupling and cohesion.

A quantitative evaluation method to measure the maintainability enhancement effect of program refactoring is presented in [7]. They analysed three phases in the process of program refactoring, of which their contribution is towards the phase of validation of the refactoring effect. They analyse the effect of a number of refactorings on coupling metrics by pre- and post-refactoring measurements.

Lastly, another approach for finding refactoring opportunities is by using visualization techniques [10]. Excellent work on software visualization has resulted in a tool called CodeCrawler, which allows the visualization of software elements to be arranged according to metric calculations [8].

## 5. Conclusion

Finding refactoring opportunities is a non-trivial activity which should be based on insight in ways to improve particular quality attributes. In this work, we focused on reverse engineering these refactoring opportunities that can lead to an improvement in coupling/cohesion of the code.

We identified specific applications of Move Method, Replace Method with Method Object, Replace Data Value with Object and Extract Class to be beneficial. However, we also experienced that guidelines can be insufficiently specific. This was the case for a specific application of Extract Method, which was harmful for cohesion.

Guidelines were composed which describe these specific applications. Profiles were specified for each of the targeted refactoring opportunities, which stipulate the associated detection analysis.

Concluding, we demonstrated that by exploiting the results from coupling/cohesion impact analysis, it is possible to achieve quality improvements with restricted refactoring efforts. This effort is restricted to the analysis and resolution of a limited set of refactoring opportunities which are known to improve the associated quality attributes.

## 6. Acknowledgments

This work has been sponsored by the Belgian National Fund for Scientific Research (FWO) under grants 'Foundations of Software Evolution' and 'A Formal Foundation for Software Refactoring'. Other sponsoring was provided by the European Science Foundation by means of the project 'Research Links to Explore and Advance Software Evolution (RELEASE)'.

## References

[1] L. C. Briand, J. Daly, and al. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Engineering*, 25(1):91–121, 1999.

[2] L. C. Briand, J. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[3] L. C. Briand and J. Wüst. The impact of design properties on development cost in object-oriented system. *IEEE Trans. Software Engineering*, 27(11):963–986, 2001.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. Int. Conf. OOPSLA 2000*. ACM Press, 2000.

[5] B. Du Bois. Opportunities and challenges in deriving metric impacts from refactoring postconditions. to be published in International Workshop on Object Oriented Reengineering (WOOR), ECOOP-workshop, 2004.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[7] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proc. Int'l Conf. Software Maintenance*, pages 576–585. IEEE Computer Society Press, 2002.

[8] M. Lanza and S. Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003.

[9] H. A. Sahraoui, R. Godin, and T. Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *Proc. International Conference on Software Maintenance*, pages 154–162, october 2000.

[10] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proc. European Conf. Software Maintenance and Reengineering*, pages 30–38. IEEE Computer Society Press, 2001.

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

[12] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *J. Syst. Softw.*, 66(3):225–239, 2003.

[13] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of UML 2003 – The Unified Modeling Language*. Springer-Verlag, 2003.