Table of Contents

Workshop on Object-Oriented Reengineering

Serge Demeyer[1], Stéphane Ducasse[2], Kim Mens[3], Adrian Trifu[4], Rajesh Vasa[5], and
Filip Van Rysselberghe[1]

[1] Department of Mathematics and Computer Science, University of Antwerp — Belgium
[2] Software Composition Group, University of Berne — Switzerland
[3] Département d'Ingnierie Informatique, Université catholique de Louvain — Belgium
[4] Programmstrukturen, FZI Forschungszentrum Informatik, Karlsruhe — Germany
[5] Department of Information Technology, Swinburne University of Technology — Australia

1 Introduction

The ability to reengineer object-oriented legacy systems has become a vital matter in today's software industry. Early adopters of the object-oriented programming paradigm are now facing the problems of transforming their object-oriented "legacy" systems into full-fledged frameworks.

To address this issue, a series of workshops have been organized to set up a forum for exchanging experiences, discussing solutions, and exploring new ideas. Typically, these workshops are organized as satellite events for major software engineering conferences, such as ECOOP'97 [5], ESEC/FSE'97 [10, 11], ECOOP'98 [17, 16], ECOOP'99 [14, 13], ESEC/FSE'99 [12]. The last of this series so far has been organized in conjunction with ECOOP'03, its proceedings were published as a technical report from the University of Antwerp [8], and this report summarizes the key discussions and outcome of the workshop[6].

For the workshop itself we chose a format which balanced presentation of position papers against time for discussion, using the morning for presentation of position papers and the afternoon for discussion in working groups. Due to time restrictions we could not allow for every author to present. Instead, we invited three authors to not only present their own work, but also to summarize two related position papers. This format resulted in quite vivid discussions during the presentations, because authors felt more involved and because the three persons presenting (Roland Bertuli, Ragnhild Van Der Straeten, and Adrian Trifu) did such a splendid job in identifying key points in the papers. Various participants reported that it was illuminating to hear other researchers present their work.

During the discussion we maintained a list of "points of interest" on the blackboard, which later served as a guidance for identifying common issues. Based on this list, we broke up in two working groups, one on *Visualisation of Software Evolution*, the other on *Reengineering Patterns*. The workshop itself was concluded with a plenary session where the results of the two working groups were ventilated in the larger group. Finally,

---

we discussed some practical issues, the most important one being the idea to organize a similar workshop next year.

2 Summary of Position Papers

In preparation of the workshop, we received 12 promising position papers (none of them has been rejected) which naturally fitted into three categories: (a) Dynamic Analysis, (b) Design Consistency, (c) Methods and techniques in Support of Object-Oriented Software Evolution. After a reviewing phase, the authors were allowed to revise their initial submission and the resulting position papers were collected in the WOOR'03 proceedings [8]. These proceedings were sent out to all participants beforehand in order to allow them to prepare for the workshop.

For each of the categories, we asked one author to summarize the position papers; you will find these summaries below.

2.1 Dynamic Analysis

Understanding how a program under maintenance is structured, is an important step in re-engineering that application. Such program comprehension techniques either use static, i.e. source code and other documents, or dynamic, i.e. runtime information like method invocations, information. However the different nature of object-oriented languages with its late binding and polymorphism, makes it harder to rely solely on static information. This makes run-time analysis an important research topic within the field of object-oriented re-engineering.

The four papers on this subject of run-time analysis focussed on two different problems. Coping with the huge amount of information generated by run-time traces, was the concern of the first two papers ([31] and [4]). Where the third [22] and fourth [23] paper presented their solution to allow an optimal instrumentation of the code.

Using a Variant of Sliding Window to Reduce Event Trace Data. In [31], Zaidman and Demeyer present a technique based on the ideas of sliding window and frequency spectrum analysis. Using the execution frequency of methods, they partition the methods of a program in 3 groups: (1) those that are executed frequently (those methods point to low-level functionality, (2) the midrange and (3) those that are executed infrequently (very high-level, e.g. the main() of en program). According to the authors the main interest is in the midrange sector because these methods can give clues about a programs architecture. A variant of the sliding-window mechanism, well-known in the world of telecommunications, is used to pass over the execution trace, identifying regions containing a high degree of methods catalogued in the midrange category.

Run-time Information for Understanding Object-Oriented Systems. In [4], Bertuli, Ducasse and Lanza measure aspects of a running software system, such as the number of created instances or the number of method invocations. these measurements are then visualized using so-called 'polymetric views' [7, 21]. The paper identifies four configuration of views and metrics that offer important information about the running system and the role of identified classes.

A Mechanism for Instrumentation Based on the Reflection Principle. / A New Strategy for Selecting Locations of Instrumentation. In [22], Li and Chen separate the instrumentation code from the actual code by using meta-level objects. of course, this approach has been shown to work for other languages such as Smalltalk and Java. Yet, these authors show that it is feasible to apply this approach for C++ systems as well. Since C++ lacks meta-objects, the authors actually rely on an *open compiler* to mix the meta-level code with the base objects.

Li and Chen submitted a second paper [23], reporting on a possible application of their open compiler approach for instrumenting code. The idea is to identify good places for instrumenting code by analyzing the call-graph. The approach is validated in a tool called XDRE.

2.2 Design Consistency

Four position papers addressed the problem of *design consistency*, i.e. the recurring problem of ensuring that the design documentation remains synchronized with all other project artifacts (i.e. requirements specifications, other design documents, the implementation). In principal, there are two different approaches to tackle design inconsistencies. The first one is the *horizontal approach*, attacking inconsistencies between different design documents; the other one is the *vertical approach*, tackling inconsistencies between models at different levels of abstraction. As a representative for the vertical approach, we had the paper [24], [18]. As a representative for the horizontal approach we had the papers [26], [20].

Intentional Source-Code Views. Mens and Poll [24] propose the lightweight abstraction of intentional source-code views as a way to codify high-level information about the architecture, design and implementation of a software system, that an engineer may need to better understand and maintain the system. They report on some experiments that investigate the usefulness of intentional source-code views in a variety of software maintenance, evolution and reengineering tasks, and present the results of these experiments in a pattern-like style.[7]

Maintaining Consistency among UML Models. Mens, Van Der Straeten and Simmonds [26] address the problem of preserving consistency among the various UML models of which a software design typically consists, in particular after some of the models have evolved. To achieve the detection and resolution of consistency conflicts, the use of description logics and their associated tools is proposed. The authors argue how this approach allows them to partially automate the detection and resolution of design inconsistencies, thus increasing the maintainability of the software.[8]

Extending UML for Enabling Refactoring. In their position paper, Van Gorp et al [18] address the gap between existing UML tools on the one hand, and refactoring tools

---

[7] An extended version of this paper has been published at the ICSM2003 conference [25].

[8] An extended version of this paper has been published at the UML2003 conference [29].

on the other. Whereas the former are designed to produce analysis and design models, the latter are designed to manipulate program code. Current tool vendors are trying to bridge this gap by regenerating program code from evolving UML models and vice versa (Model Driven Architecture is a typical example of this kind of approach). Including support for program code refactorings into the infrastructure of these novel UML tools is not trivial however. The authors describe some of the problems and propose a solution which they also implemented in a running tool prototype. [9]

Tracing OCL Constraints on Evolving UML Diagrams. Whereas the formal foundations of refinement and refactoring of program code have been widely studied, much less attention has been paid to formal methods for specification redesign and requirements tracing. A specification is usually spread through several documents and changes frequently during software development. As such it is very hard to trace the requirements and to validate compliance of a software system to its requirements. In his position paper, Kosiuczenko [20] studies the problem of tracing requirements in UML class diagrams with OCL constraints. He proposes a term rewriting approach to automatically derive traces which allow one to navigate through several specifications having different levels of abstraction as well as to trace requirements in forward and backward direction in distributed and changing specifications.

2.3 Methods and Techniques in Support of OO Software Evolution

A software system evolves as a consequence of the alternating phases of reengineering and functionality extensions. The process of reengineering [6, 2] is typically composed of three main phases: a reverse engineering phase, a restructuring phase and a forward engineering phase, also referred to as change propagation.

Of the four submitted position papers that address the topic of object–oriented evolution, [1] and [30] deal with the reverse engineering and restructuring phases of reengineering respectively, [3] deals with aspects of software metrics formalization, and [28] presents a method for investigating the evolution process of concrete systems, as a whole. A short summary of each paper is given below.

A Class Understanding Technique Based on Concept Analysis. In [1], Arévalo presents a novel method called *X-ray views*, which allows gaining insight into how a class operates internally, as well as how it interacts with other classes. This is useful when trying to understand a class in the context of reverse engineering a software system.

Understanding how a class works translates into identifying and evaluating different types of dependencies between the entities of the class: its instance variables, representing the state, and its methods, representing the behavior. By evaluating these dependencies and grouping them together, we can obtain a high level view of several aspects of the analyzed class, such as: (a) how clusters of methods interact to form together a precise behavior of the class; (b) how instance variables are related (i.e. used together);

---

[9] An extended version of this paper has been published at the UML2003 conference [19].

(c) what is the public interface of the class; (d) which methods are the so called "entry points" (methods that are part of the interface and that call other methods inside the class); (e) which methods use all of the class' state and which ones use only parts of it.

Dependencies are defined as sets of directly or indirectly related class entities, and are determined using Concept Analysis. For the unambiguous specification of dependencies, the author introduces a simple formalism, and using this formalism, defines a number of seven different dependencies. Some examples include *direct accessors*, *exclusive direct accessors*, *collaborating instance variables* and *interface methods*.

Based on the concept of dependencies, the author defines the higher level concept of *view*, as a combination of a set of dependencies. She then exemplifies the technique by defining two views *core attributes* and *public interface*. For example, the view called *public interface* is defined in terms of the two dependencies *interface methods* and *externally used state*. The defined views are applied on a concrete Smalltalk class, and results are discussed.

As future research, the author intends to define and evaluate more views, as well as investigate inheritance relations from the standpoint of the approach.

Strategy Based Restructuring of Object–Oriented Systems. In [30], Trifu and Dragoş present a method for object–oriented design improvement. The motivation behind their work is the conceptual gap that exists between state of the art design flaw detection techniques and current source code transformation technology. More precisely, the problem addressed is how to start from a given design problem, and (automatically) derive a sequence of source code transformations that eliminate the problem, while at the same time improve the system with regard to a predefined set of quality criteria. By design flaw, the authors refer to structural flaws in the code, caused by violations or misuse of principles and heuristics of good design (e.g. *god class*, *feature envy*, *data class*, etc. see [27]).

At the heart of the approach lies the novel concept of *correction strategy*, defined as a "structured description of the mapping between a specified design flaw and all possible ways to correct it". Selecting between alternative paths through a strategy is supported by the quality related information contained in the strategy itself, as well as a suitable quality model. The quality model, as specified by the methodology, is a prediction tool, used to estimate the quality impact of choosing one path from the several possible ones, thus enabling quality-aware decisions to be taken either automatically or manually.

Based on *correction strategies*, a quality-driven, highly automatable methodology for design improvement is presented. The methodology has the advantage that it allows for a great deal of flexibility by allowing the software engineer to configure the desired level of automation. This way, a tool that implements the methodology is able to function in a large number of configurations, ranging from a fully automatic to a fully assisted mode, in which the software engineer has complete control over all decisions taken in the restructuring process.

A set of correction strategies, together with the methodology and the quality model provides the missing link, and therefore a complete solution to the problem of design flaw removal from object–oriented systems.

The paper provides a critical overview of the state of the art in the field of restructuring software systems in general and design improvement in particular. As future work, the authors intend to continually improve on all aspects of the approach, provide a comprehensive catalogue of correction strategies, as well as perform a thorough evaluation of the approach on real-life case studies.

A Formal Library for Aiding Metrics Extraction. The approach presented in [3] does not directly address any of the phases of software evolution, but instead can be considered as a fundamental contribution to all reengineering approaches that are based on software measurement. In the paper, Baroni and Brito e Abreu present a library called FLAME (a Formal Library for Aiding Metrics Extraction), that can be used to formalize object–oriented design metrics definitions.

The declared purpose of the library is to encourage the use of software metrics among software designers by providing them with formalized design metrics, that are based on the UML model rather than source code. This way, the benefits of software measurements with increased tool support are made possible in early phases of software development, before source code is even available. Moreover, such definitions are at the same time formal and executable, thus making them appropriate for experiments replication (a recurring problem in the area).

The library contains a number of about 90 formally defined functions, at different levels of abstraction, such as *classifier*, *package*, *attribute* and *operation*. As formal language, the OCL (Object Constraint Language) is used on the core UML meta–model. FLAME functions are further classified as general, set, percentage and counting functions. Examples of FLAME functions include *new features*, *all attributes* and *defined operations*, defined at classifier context, and *classes number*, *internal base classes* and *supplier classes*, defined at package context.

In the end, the authors present formal definitions for a few well known metrics from the literature and draw conclusions.

As future work, the authors intend to extend the scope of the library to the complete UML model, especially the behavioral parts. The authors also consider formalizing the same metrics but using different meta–models. As the goal of their research, the authors propose to provide precise definitions for most accepted sets of metrics, as well as the creation of a metrics framework which will allow the creation and comparison of metrics.

Method for Investigating the Evolution of Concrete Software Systems. Although it is a widely accepted fact that any software system must continue to evolve in order to remain successful, little is known today about how successful systems have evolved. Therefore little has been learnt from past experience and mistakes. This is the motivation behind the technique described by Van Rysselberghe and Demeyer in [28].

The approach presented in the paper tries to shed some light upon the nature of the phenomenon of evolution, by applying already established methods for software visualization and detection of duplication in large amounts of data, to successive versions of real-world systems. The lessons learnt in this way should improve our methods and understanding of software evolution.

The idea behind the approach is the same as the one used to reconstruct the past evolution processes of early life on earth: comparing successive releases of the software systems (the fossil remainders) and analyzing the differences. Differences in the implementation of two consecutive versions are highlighted using the technique presented in [15], where a two dimensional matrix, called a *dot plot diagram*, shows duplicate lines of code as dots and mismatched lines as empty space. A dot plot diagram corresponding to the comparison between a system and itself would be a square, and would only show a diagonal (a perfect diagonal means that every line of code is identical to itself). However, when comparing different versions of the same system, the original diagonal is broken up into segments that are shifted or cut out. By studying these changes and identifying patterns that correspond to known refactorings, one can reconstruct the evolution process of the system.

The authors exemplify the technique on the refactoring "pull-up-method", which is a typical refactoring used for eliminating duplicated code. It moves duplicated methods higher up the class hierarchy and replaces them by a single method to be reused by all subclasses. The corresponding patterns of change in the dot plot diagram are presented and analyzed in detail.

In order to increase the scalability of the approach, the authors propose the use of various data-reduction as well as automatic pattern recognition techniques.

3   Visualization of Software Evolution

The goal of the working group on *evolution visualization* was to discus different approaches to visualize the evolution of a software system. To achieve this goal two visualization techniques were presented and discussed by the participants. Although no general conclusions on how evolving programs should be visualized in order to understand their evolution were formulated, it did help both ideas presented to mature. Through the discussion possible improvements or applications were presented.

The focus of the first discussion was mainly on the visualization of object oriented software systems. Rajesh Vasa presented an initial idea of using hierarchical layering of classes in a given software system based on dependency analysis. Though details of the algorithm have not been worked out, the key concept is the use of Fan-In and Fan-Out metrics to layer the classes in a system. Once this hierarchical layering has been completed, one could see a visual representation of the software system where each layer had a number of classes. A number of these layer diagrams would have to be generated, one for each version of the software system under observation. Using animating algorithms we can then visualize the software system as it is changing. In the open discussion various participants put forward suggestions and ideas on how to get a better picture of these evolutions. For fine granularity one could focus on how a single class in the overall system and observe its evolution. Observing a single class or a set of classes can be very useful if the development team wanted to get an overview of how a set of classes evolved in the past. This type of observation would be very useful in determining quickly the set of classes that changed most (comparatively). Further, this approach can help identifying classes where the dependencies are changing frequently. Visual representation can be easily achieved as a simple chart where only the classes

reaching a certain pre-defined threshold would be shown. The alternative approach discussed focused on how the overall layer diagram would change as the system evolved. This type of visualization will require a simple animation and provide a quick feedback of how the system was developed. The animation can provide an indication of the development methodology used, as in top-down or bottom-up. In a scenario where a development team built the library classes first and then focused on building the user-interface layers, we should see a bottom-up build up of the classes in the hierarchical layer diagram animation. This animation should mirror the development methodology used. The discussion did not identify specific uses for this, but it can be valuable for managers as it can provide insight into how the engineers are building the system and to ensure that the planned development approach is being undertaken.

The second discussion was focussed on the visualization of changes. Van Rysselberghe shortly presented the idea for using a combination of clone detection tools and dotplots to visualize the changes made from one version of a program to another [28]. For the visualization of changes, such technique is indeed a good solution certainly because it allows to spot changes rapidly. However as some noticed, some problems go together with this visualization. A first problem is that of order. Reordering the contents of a source file, causes many mismatches which aren't interesting for its evolution. However a solution found for this problem is to use a pretty printer to format and order the attributes, methods, etc within a file. A harder problem is that of the visualizations scalability. Both the author as some of the participants didn't expect any help from filtering techniques which would be applied. However making the whole process a two step process is probably solving this problem. The idea is to use metrics first, just to find the possible changes which might have happened. The author himself, argued that using a some kind of similarity metric between files might succeed in solving the problem. After locating a file and its evolved counterpart, it would compare both using the prescribed technique, increasing the technique's and visualisation's scalability (applicability).

4   Reengineering Patterns

The goal of this workgroup was to 'mine' for new reengineering patterns, similar but complementary to those that can be found in the Reengineering Patterns' book [9]. The approach taken was to come up with a list of known reengineering problems first, and distill potential patterns out of each of these[10].

Each reengineering pattern description contains a specific *problem* related to software reengineering, an *example* of the problem, a proposed *solution* to that problem as well as some *trade-offs* regarding that particular solution. Given the limited time we had for discussion, the patterns we 'identified' should only be considered as 'potential' patterns. For example, we did not discuss the other parts of which a reengineering pattern consists: a *rationale* motivating the importance of the pattern, *known uses* of the pattern and *what next* to do after having applied the pattern. All reengineering patterns we identified are listed below.

---

[10] if the Reengineering Pattern book [9] did not already contain a pattern that addressed that particular problem

4.1 Changing libraries

Problem. A software application uses a certain library (or component) which we want to replace by a new one.

Example 1. We have a Java application that works with an Oracle database and want to modify it to work with a Sybase (or some other) database.

Example 2. We have an application of which the user interface is based on AWT and want to change it so that it uses SWING.

Solution. Rather then just modifying the existing application to make it work with the new library (or component), first add an intermediate layer in between the application and the library. Like this the application becomes much less coupled to the library it depends on. Given a sufficiently abstract layer in between, accommodating similar libraries in the future will become much easier.

Trade-offs. The solution should not be used when writing a new intermediate layer would be too expensive. This is for example the case when the problem domain is not well-established yet, i.e. when it is difficult to find a possible abstraction for all libraries of a certain kind. For example, adding an intermediate layer that works with different database management systems is a quite standard and frequently occurring solution. Writing an intermediate layer that is independent of the particular user interface being used, on the other hand, is a task that should not be taken on lightly.

4.2 Predicting change impact

Problem. How can we predict the impact of changes to a software system?

Example. Changing, for example, the default value of a variable may have far reaching consequences throughout the entire implementation, wherever this value is explicitly or implicitly, directly or indirectly relied on.

Solution. Many tools and techniques exist to help in predicting the impact of changes to a program: program analysis, simulation on system model (dependency graph), testing after change is implemented. However, all these techniques have their limitations and there might always be cases (i.e. possible impacts) that we missed.

Trade-offs. Do not rely too much on the tools and techniques that exist. Sure, they will help, but be aware that they are not 'all powerful'. Also, the stronger the technique (i.e., the more exact it is) the less efficient it probably is.

4.3 Missed abstractions

Problem. In 'bad' object-oriented programs, a lot of the problems are often due to a lack of abstraction, or having chosen the wrong abstractions.

Examples. Code duplication, wrong use of object-oriented concepts, overuse of global variables, long parameter lists, ...

Solution. Refactorings can often do a quite good job in rectifying some of these situations.

Trade-offs. Make sure that the 'bad code' is really 'bad code' and not optimized (or deliberately written in that style for some other important reason).

4.4 Eliminating dead code

Problem. How can we eliminate dead program code from an application?

Example. A method that is not called internally (from within the same class) neither from any other class. Why keep this method if it is not used anyway?

Solution. Although there are some tools and techniques that may help in detecting 'dead code', the problem is that one never really knows whether the code is really dead. Maybe it does not seem to be used in the application itself but it is called from other sources that you don't have; or maybe instead of being dead it is just 'not born yet' in the sense that someone already put in place a piece of code with the idea of using it in some near future. Therefore, we suggest not to remove the dead code entirely but to replace it with assertions and keep the old code for a time (say one or two years or so), to play safe.

Trade-offs. It is difficult to decide how long to keep the code and who will take the final decision to remove the code in the end.

4.5 Enforcing coding conventions

Problem. How can we make sure that a given set of coding standards and conventions will be consistently used throughout an application?

Example 1. We want to ensure that the instance variables of a class are never called directly, but always through an accessing method.

Example 2. We want to ensure that all mutator methods of instance variables have a consistent name. For example, the naming convention adopted in Java is to name it after the name of the variable preceded by 'get'. The naming convention in Smalltalk is to name it after the name of the variable concatenated with a colon ':'.

Solution. Several research tools are available today that allow us to detect breaches of such naming conventions, as well as to enforce their consistent usage.

Trade-offs. However, care should be taken with enforcing these conventions if the development team is not entirely comfortable with them. For example, maybe the team already understands its code sufficiently well so that it is not really needed to 'clean it up' to make it more readable (by enforcing the use of certain conventions throughout the entire system).

4.6 Other potential reengineering patterns

Some other reengineering problems we discussed but from which we did not have the time to extract a potential reengineering pattern are the following: (a) How to detect and solve usage of *data classes* in object-oriented programs. (b) How to detect and correct (run-time) *dangling references* in programs. (c) How to *ensure class cohesion* in object-oriented programs. (d) *Preserving behavioral contracts in inheritance relations*, i.e. making sure that inherited methods in subclasses respect the intended behavior of their parent methods. (e) *Controling external and internal quality specifications* of a system during reengineering.

5 Conclusion

In this report, we have listed the main ideas that were generated during the workshop on object-oriented reengineering. Based on a full day of fruitful work, we can make the following recommendations.

- *Viable Research Area.* Object-Oriented Reengineering remains an interesting research field with lots of problems to be solved and with plenty of possibilities to interact with other research communities (dynamic analysis, modeling, UML, metrics, visualization to name those that were touched upon during the workshop).
- *Establish a research Community.* All participants agreed that it would be wise to organize a similar workshop at next year's ECOOP.
- *Workshop Format.* The workshop format, where some authors were invited to summarize position papers of others worked especially well.

References

1. G. Arevalo. X-ray views on a class using concept analysis. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 76–xxx, 2003.
2. H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A.-M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. The FAMOOS object–oriented reengineering handbook, 1999.
3. A. L. Baroni and F. B. e Abreu. A formal library for aiding metrics extraction. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 62–70, 2003.
4. R. Bertuli, S. Ducasse, and M. Lanza. Run-time information for understanding object-oriented systems. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 10–20, 2003.
5. E. Casais, A. Jaasksi, and T. Lindner. FAMOOS workshop on object-oriented software evolution and re-engineering. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology (ECOOP'97 Workshop Reader)*, volume 1357 of *Lecture Notes in Computer Science*, pages 256–288. Springer-Verlag, Dec. 1997.
6. E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
7. S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE '99 (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.
8. S. Demeyer, S. Ducasse, and K. Mens, editors. *Proceedings of the ECOOP'03 Workshop on Object-Oriented Re-engineering (WOOR'03)*, Technical Report. University of Antwerp - Department of Mathematics and Computer Science, June 2003. http://..../.
9. S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.
10. S. Demeyer and H. Gall, editors. *Proceedings of the ESEC/FSE Workshop on Object-Oriented Re-engineering*, TUV-1841-97-10. Technical University of Vienna - Information Systems Institute - Distributed Systems Group, Sept. 1997.
11. S. Demeyer and H. Gall. Report: Workshop on object-oriented re-engineering (WOOR'97). *ACM SIGSOFT Software Engineering Notes*, 23(1):28–29, Jan. 1998.

12. S. Demeyer and H. Gall, editors. *Proceedings of the ESEC/FSE'99 Workshop on Object-Oriented Re-engineering (WOOR'99)*, TUV-1841-99-13. Technical University of Vienna - Information Systems Institute - Distributed Systems Group, Sept. 1999.

13. S. Ducasse and O. Ciupke. Experiences in object-oriented re-engineering. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology (ECOOP'99 Workshop Reader)*, volume 1743 of *Lecture Notes in Computer Science*, pages 164–183. Springer-Verlag, Dec. 1999.

14. S. Ducasse and O. Ciupke, editors. *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-engineering*, FZI report 2-6-6/99. FZI Forschungszentrum Informatik, June 1999.

15. S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, Sept. 1999.

16. S. Ducasse and J. Weisbrod. Experiences in object-oriented reengineering. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, volume 1543 of *Lecture Notes in Computer Science*, pages 72–98. Springer-Verlag, Dec. 1998.

17. S. Ducasse and J. Weisbrod, editors. *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-engineering*, FZI report 6/7/98. FZI Forschungszentrum Informatik, July 1998.

18. P. V. Gorp, H. Stenten, T. Mens, and S. Demeyer. Enabling and using the uml for model driven refactoring. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 37–40, 2003.

19. P. V. Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent uml refactorings. In *Proc. 6th International Conference on the Unified Modeling Language*. Springer Verlag, 2003.

20. P. Kosiuczenko. Tracing requirements during redesign of uml class diagrams. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 41–47, 2003.

21. M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, sep 2003.

22. Q. Li and P. Chen. A mechanism for instrumentation based on reflection principle. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 21–25, 2003.

23. Q. Li and P. Chen. A new strategy for selecting locations of instrumentation. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 26–31, 2003.

24. K. Mens and B. Poll. Supporting software maintenance and reengineering with intentional source-code views. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 32–36, 2003.

25. K. Mens, B. Poll, and S. González. Using intentional source-code views to aid software maintenance. In *Proceedings of ICSM2003*, 2003.

26. T. Mens, R. V. D. Straeten, and J. Simmonds. Maintaining consistency between uml models with description logic tools. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 48–54, 2003.

27. A. J. Riel. *Object–Oriented Design Heuristics*. Addison–Wesley, first edition, 1996.

28. F. V. Rysselberghe and S. Demeyer. Studying software evolution using clone detection. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 71–75, 2003.

29. R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *Proc. 6th International Conference on the Unified Modeling Language*. Springer Verlag, 2003.

30. A. Trifu and I. Dragos. Strategy based elimination of design flaws in object-oriented systems. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 55–61, 2003.

31. A. Zaidman and S. Demeyer. Using a variant of sliding window to reduce event trace data. In S. Demeyer, S. Ducasse, and K. Mens, editors, *WOOR'03 Proceedings*, pages 4–9, 2003.