Workshop on Object-Oriented Reengineering

Serge Demeyer[1], Kim Mens[2], Roel Wuyts[3], Yann-Gaël Guéhéneuc[4], Andy Zaidman[1], Neil Walkinshaw[5], Ademar Aguiar[6], and Stéphane Ducasse[7]

[1] Department of Mathematics and Computer Science, University of Antwerp — Belgium
[2] Département d'Ingénierie Informatique, Université catholique de Louvain — Belgium
[3] Département d'Informatique, Université Libre de Bruxelles — Belgium
[4] Group of Open and Distributed Systems, Université de Montréal — Canada
[5] Computer and Information Sciences, University of Strathclyde, Glasgow — UK
[6] INESC Porto, Universidade do Porto — Portugal
[7] LISTIC Laboratory, University of Savoie — France

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 problem, a series of workshops have been organised to set up a forum for exchanging experiences, discussing solutions, and exploring new ideas. Typically, these workshops are organised as satellite events of major software engineering conferences, such as ECOOP'97 [1], ESEC/FSE'97 [3], ECOOP'98 [7], ECOOP'99 [6], ESEC/FSE'99 [4], ECOOP'03 [2], ECOOP'04 [16]. The last of this series so far has been organised in conjunction with ECOOP'05 and this report summarises the key discussions and outcome of that workshop.[8] As preparation to the workshop, participants were asked to submit a position paper which would help in steering the workshop discussions. Moreover, researchers working on dynamic analysis were invited to compare the results of their approaches by applying their tools on a common case (ArgoUML). As a result, we received 10 position papers, of which 9 authors were present during the workshop. Together with 3 organisers and 5 participants without position paper, the workshop numbered 17 participants. The position papers, the list of participants, and other information about the workshop are available on the web-site of the workshop at http://smallwiki.unibe.ch/WOOR.

For the workshop itself, we chose a format that balanced presentation of position papers and time for discussions, using the morning for presentations of position papers and the afternoon for discussions in working groups. Due to time restrictions, we could not allow every author to present. Instead, we invited two authors to summarise the position papers. This format resulted in quite vivid discussions during the presentations, because authors felt more involved and because the two presenting persons (Andy

[8] The workshop was sponsored by the European Science Foundation Research Network "Research Links to Explore and Advance Software Evolution (RELEASE)" and a Research Network on "Foundations of Software Evolution" of the Fund for Scientific Research – Flanders (Belgium).

Zaidman and Yann-Gaël Guéhéneuc) did a splendid job in identifying key points in the papers. Various participants reported that it was illuminating to hear other researchers present their own work.

Before the workshop, the workshop organisers (Serge Demeyer, Kim Mens, Roel Wuyts, and Stéphane Ducasse) classified the position papers in two groups, one group on *Dynamic Analysis* and one group on *Design Recovery*. Consequently, in the afternoon, the workshop participants separated in two working sessions, during which they could discuss and advance their ideas. The workshop was concluded with a plenary session where the results of the two working groups were exposed and discussed in the larger group. Finally, we discussed practical issues, the most important one being the idea to organise a similar workshop next year.

2 Summary of Position Papers

In preparation to the workshop, we received 10 position papers (none of them was rejected), which naturally fitted into two categories: (a) Dynamic Analysis and (b) Design Recovery.

- Dynamic Analysis
  1. Marc Roper, Murray Wood and Neil Walkinshaw, "Extracting User-Level Functions from Object-Oriented Code".
  2. Michael Pacione, "VANESSA: Visualisation Abstraction NEtwork for Software Systems Analysis".
  3. Andy Zaidman and Serge Demeyer, "Mining ArgoUML with Dynamic Analysis to Establish a Set of Key Classes for Program Comprehension".
  4. Orla Greevy and Stéphane Ducasse, "Characterizing the Functional Roles of Classes and Methods by Analyzing Feature Traces"
  5. Tewfik Ziadi and Yann-Gaël Guéhéneuc, "Automated Reverse-engineering of UML v2.0 Dynamic Models".
- Design Recovery
  1. Danny Dig, Can Comertoglu, Darko Marinov and Ralph Johnson, "Automatic Detection of Refactorings for Libraries and Frameworks".
  2. Carlos López, Yania Crespo and Raúl Marticorena, "Parallel Inheritance Hierarchy: Detection from a Static View of the System".
  3. Naouel Moha and Yann-Gaël Guéhéneuc, "On the Automatic Detection and Correction of Software Architectural Defects in Object-Oriented Designs".
  4. Nuno Flores and Ademar Aguiar, "JFREEDOM: a Reverse Engineering Tool to Recover Framework Design".
  5. Jalagam Rajesh and D. Janakiram, "Design Patterns Induction from Multiple Versions of Software".

For each of these categories, we asked one reporter to summarise the position papers; their summaries are presented in the next two sections.

2.1 Position Papers on Dynamic Analysis

This section briefly discusses each of the position papers on *Dynamic Analysis*. We continue with an overview of each of the five techniques and highlight similarities and differences between the techniques.

Extracting User-Level Functions from Object-Oriented Code. Neil Walkinshaw is working on call graph analysis, particularly as a way to focus on those edges in the call graph that are relevant to user-level functions exercised in some scenarios. A very important concept for this technique is the concept of *landmark method*, a method that *must be* executed in a specific use case. The success of this technique hinges on the selection of these landmark methods, so these must be carefully selected.

Once landmark methods are identified, the call graph is transformed into a Hammock graph. A Hammock graph contains a single entry and a single exit node, and all of the paths in the graph lead from the entry node to the exit node. The entry and exit nodes of this graph correspond to two landmark methods. This transformation into a Hammock graph, however, is insufficient as nodes (methods) outside the Hammock graph can still influence the execution of nodes in the Hammock graph.

This problem is solved with slicing techniques. The relevant paths are identified using call sites in the Hammock graphs as slicing criteria to identify call sites for relevant indirect calls. A backwards slice on a slicing criterion (a statement and a set of variables in that statement) returns the set of statements that may influence the execution of the slicing criterion.

More information on this technique can be found in [15].

VANESSA: Visualisation Abstraction NEtwork for Software Systems Analysis. Michael Pacione's work belongs to the field of software visualisation. His research provides visualisation techniques for large-scale software understanding. These visualisation techniques are based on a unified model that allows to view a piece of software at different levels of abstraction, either statically or dynamically. The different levels of abstraction and the two facets of the unified model can be seen in Table 1:

		Facets	
		Static	**Dynamic**
Levels	5	BusinessStructure	BusinessBehaviour
	4	SystemStructureDeployment	SystemBehaviourDistribution
	3	SystemArchitecture	ComponentInteraction
	2	InterClassStructure	InterObjectInteraction
	1	IntraClassStructure	IntraObjectInteraction
	0	**Program code / Event trace**	

Table 1. Unified model

the combination of views is of particular interest:

– Combining different levels within the same hierarchy allows the *traceability* of low-level artifacts in high-level system properties.
– Combining same level views of different hierarchies results in a unified visualisation of structural and behavioural characteristics.

More information can be found in [14].

Mining ArgoUML with Dynamic Analysis to Establish a Set of Key Classes for Program Comprehension. The technique presented by Andy Zaidman aims at providing a set of *key classes*. He argues that these key classes are candidate classes for early program comprehension, i.e., these key classes should be studied first when trying to become familiar with an unknown system in a short period of time. The technique is based on a measure of runtime coupling and on the idea that strongly coupled classes can heavily influence the control flow of the entire system.

Coupling measures corresponds to binary relationships between classes. However, a weakly coupled class can still be important for program comprehension purposes. As such, Andy Zaidman defines a transitive measure for coupling, which allows a weakly coupled class (that is, a class directly connected in the control flow graph with a tightly coupled class) to also become marked as important.

Web-mining principles are used to compute the transitive measure of coupling. When studying web-mining techniques, Andy Zaidman found that *hubs* are classes with a high-level of export coupling, while *authorities* often have a significant degree of import coupling. He uses hubs, which have a high level of export coupling, as the basis for important classes.

More detailed information about this technique can be found in [17].

Characterizing the Functional Roles of Classes and Methods by Analyzing Feature Traces. Orla Greevy's work is based on the idea that correlating end-user features to classes and–or methods can help in making the program comprehension process more efficient. The technique the authors propose is called "*feature characterisation*", which can best be described as the mapping between classes and–or methods, on the one hand, and traces from different scenarios, on the other hand. With this mapping, they are able to answer three essential questions:

– How do features relate to classes/methods?
– How do classes/methods relate to features?
– How are features related to each other?

As a result, classes (or methods) are classified according to the classification schema of Table 2. This classification (and its visual representation) can help the software engineer during the program comprehension process.

More information can be found in [9].

Characterisation	Presence
Not Covered (NC)	Class/method not present in any trace
Single Feature (SF)	Class/method present in 1 trace
Group Feature (GF)	Class/method present in $<$50% of traces
Infrastructural Feature (IF)	Class/method present in $>$50% of traces

Table 2. Feature characterisation

Automated Reverse-engineering of UML v2.0 Dynamic Models. Yann-Gaël Guéhéneuc's position paper is about using the composition operators that are available in UML v2.0 to combine sequence diagrams from different traces. This composition of traces leads to a general overview of the behaviour of the program.

Using the composition mechanism allows conformance checking and pattern identification. Pattern identification is important because patterns can help in improving understandability and maintainability of a program. However, to detect patterns, both structural and behavioural data are needed, e.g., the Command pattern can only be distinguished from other patterns when both static and dynamic data are available. Furthermore, adding behavioural data to structural data allows to reduce the search space when mining for patterns.

More information can be found in [10].

Summary. In Figure 1, we applied Orla Greevy's feature characterisation technique on the five papers that were presented in the *Dynamic Analysis* session. On the left hand side are the authors of the five papers; on the right hand side are the most important properties of each technique. Between parentheses, behind each technique, is the number of incoming edges, i.e., the number of times a technique adheres to or uses the property.

Some interesting facts from Figure 1 are that, although we are in the *Dynamic Analysis* session, most techniques also rely on static analyses. Furthermore, two techniques work at the *method-level*, while two other work at the *class-level*. Two techniques also explicitly mention the importance of user-level features and use cases.

2.2 Position Papers on Design Recovery

The *Design Recovery* session was composed of five papers. We summarise the papers according to their contexts, their themes (global research subjects of the papers), their subjects (specific subject of each paper), and we attempt to abstract a global picture from the papers. The following section 2.2 summarises each paper along three main lines: theme of the paper, subject, and miscellaneous introduced ideas.

Summary. Table 3 is a summary of the presented papers.

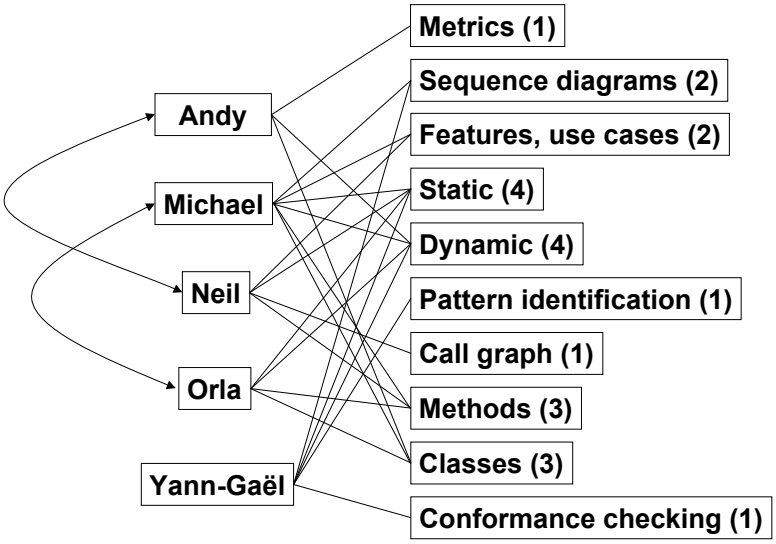

Fig. 1. Feature characterisation of the 5 presented dynamic analysis techniques

Putting It All Together. The papers presented in the *Design Recovery* session described studies realised in different contexts. Two papers were clearly related to software improvement, while the other papers were related to software development and customisation, reuse of software components, and the maintenance of object-oriented software.

However, despite the seemingly different contexts, all papers made similar assumptions: software is imperfect (wrt. developers, users, documentation...) and software changes. Thus, the papers clearly show that software development and maintenance are not strictly separated activities anymore and they highlight the need for an integrated software development life-cycle (in the broad sense of the term, including maintenance, retirement...).

The themes of the papers showed a wide range of interests related to design recovery. Papers focused on design patterns and refactorings, requirements and frameworks, bad smell detection, cost of reuse during maintenance, cost of maintaining OO architectures. These themes, although different by definitions, are different levels and views on improving the tasks of software engineers and, thus, can be cast in a more general framework along three lines: level of abstraction, views, and tasks. The level of abstraction can be requirements, architecture, design, or code. The views relate to documentation/comprehension of software or refactoring of software. The tasks can be improvements, understanding, or reuse.

The subjects of the papers differed widely, ranging from design pattern inference to recovery of framework design and use, through parallel inheritance hierarchy detection, renaming (clone) detection, and software architectural defects. These wide differences highlight clearly the need for a classification of design recovery tools. Indeed, despite

Theme	Subject	Misc.
D. Janakiram and Jalagam Rajesh: "Design Patterns Induction from Multiple Versions of Software" [11]		
Design patterns and refactorings	Design pattern inference	Use of multiple versions. Prolog unification. Semantics of extensibility. Accuracy of inference. Correspondence between versions.
Nuno Flores and Ademar Aguiar: "JFREEDOM: a Reverse Engineering Tool to Recover Framework Design" [8]		
Requirements and frameworks	Framework design and use recovery	Learning and understanding. Documentation (or lack thereof). Level of abstractions. Design decisions (flexibility, clarity, visualisation). Hooks, templates, meta- and design patterns.
Raúl Marticorena, Carlos López, and Yania Crespo: "Parallel Inheritance Hierarchy: Detection from a Static View of the System" [12]		
Bad smell detection	Detection of parallel inheritance hierarchies	Human detection. Metrics and name conventions. Language independence. Static and dynamic data. Data mining.
Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson: "Automatic Detection of Refactorings for Libraries and Frameworks" [5]		
Cost of reuse during maintenance	Renaming (clone) detection	Detection of refactorings. Closed- vs. open-world assumptions. Deprecation and coexistence. Feedback and threshold. Syntactic vs. semantic data.
Naouel Moha and Yann-Gaël Guéhéneuc: "On the Automatic Detection and Correction of Software Architectural Defects in Object-Oriented Designs" [13]		
Cost of maintaining OO architecture	Software architectural defects	Anti-patterns. Design defects. Code smells. Taxonomy. Structural, numerical tools.

Table 3. Summary of the Design Recovery Papers

the related contexts and themes, there still remains much latitude for design recovery tools and, thus, it is difficult to compare tools with one another, which impedes their evaluation and the focus of research on important issues.

Finally, the papers suggested many interesting criteria that could serve as discriminating factors between design recovery tools when making a classification :

- Assumptions made by the tools, e.g., closed vs. open-world assumption.
- Versions, e.g., use of one or multiple versions of a program for the recovery.
- Goals, e.g., design recovery (hooks, templates, meta-patterns, and design patterns), refactoring opportunities, clones, bad smells, anti-patterns, defects.
- Kind of analysed data, e.g., static or dynamic, and semantic level of the data (lexical, structural or semantic).
- Qualities, e.g., flexibility, clarity, accuracy, extensibility.
- Recovery techniques, e.g., manual, Prolog unification, constraint programming, data mining, metrics (including hash values), visualisation.
- Identified problems, e.g., language independence, need for a taxonomy, use of semantic data, correspondence between versions.

Discussion. The papers suggested several issues that the design recovery community must strive to address. First of all, every design recovery tool should make clear its objectives: What is it trying to achieve? What are the problems addressed? Also, the context in which a tool operates should be formalised, if possible, and if not, specified informally to allow distinguishing among tools targeting different contexts of uses. Finally, we need to define more formally, if possible, the software engineers' goals and tasks, to put in perspective tools with respect to their users' goals and tasks.

The papers also emphasised the lack of agreed-upon definitions for seemingly well-known concepts and the need for a unified classification of design recovery tools.

At last, in contrary to the *Dynamic Analysis* session, in which participants had a common piece of software on which to exercise their tools to compare results, there is no consensus on typical programs on which design recovery tools should operate, no consensual results for certain classes of design recovery tools, and not even a consensual definition of what is a typical piece of software to analyse. This lack of "standard" examples impedes advancement in research because many tools do the same things on different types of software while other things are not studied for lack of interest or of concrete examples and of known expected results.

3 Wrap Up

Given the above position papers, we decided to split up in two working groups. As announced before the workshop, the group on *Dynamic Analysis* would compare the results of the different approaches and tools on a common case. For the group on *Design Recovery*, it was observed that the bibliographic references were rather disjunct, while the papers overlap in topic; hence the idea to work on a common taxonomy, which could be used to compare the different approaches.

3.1 Dynamic Analysis

The idea of the *Dynamic Analysis* session was to compare the output of the presented tools, given the same subject system and use-case. The ArgoUML CASE tool was chosen because it is well documented and relatively mature. The use-case consisted of specifying a class diagram (involving multiple classes to ensure the existence of loops in a dynamic trace).

Andy Zaidman's tool uses a data-mining algorithm to discover key classes in a trace. Orla Greevy's tool maps functional roles to classes and methods by comparing multiple traces. For this approach, the use-case was divided in multiple scenarios. Michael Pacione's tool uses traces with static analysis to provide mappings between different levels of abstraction (e.g., to map code to activity diagrams), sometimes requiring human intervention. Neil Walkinshaw's tool uses landmark methods that must be executed for a particular function (as opposed to a full trace) to extract a sub-graph of the call graph corresponding to the use-case.

Neil Walkinshaw's tool had previously only been executed on relatively small systems and could not scale to produce usable results for a practical comparison with the other tools. Michael Pacione's tool required an amount of memory proportional to the

trace size, and so insufficient resources were available to analyse ArgoUML. Andy Zaidman's tool produced a ranked list of 50 classes that scored highest using the data-mining algorithm. Orla Greevy's tool successfully categorised classes depending on their contribution to the traces.

Generation and storage of the traces were handled differently by Andy Zaidman, Orla Greevy, and Michael Pacione's tools. Andy Zaidman and Orla Greevy both experimented with reducing the size of the traces, by removing loops and repetitions. This was relatively successful and led to trace sizes of 100 megabytes in Andy Zaidman's case. Michael Pacione had not carried out any compression on the trace, which resulted in a trace size of almost three gigabytes. This is too large considering that his tool loads the entire trace into memory. To remedy this, a caching approach was suggested, so that only relevant segments of the trace need to be stored in memory at any given time.

Neil Walkinshaw's tool also faced scaling problems, although it did not require a trace. The entire call graph is stored in memory, along with dependency graphs for each method in the call graph (this includes library methods). Again, a caching approach was suggested, where graphs can be loaded in memory when they are required.

Orla Greevy's tool produced a list of 30 "infrastructure" classes (classes that were accessed in 50% of the traces). This list was compared to Andy Zaidman's ranked list of 50 "key" classes. Out of the top 7 in Andy Zaidman's list, 5 were categorised by Orla Greevy's tool as infrastructure classes, but, beside these, there were very few other matches. On closer inspection, the classes that did match turned out to be mainly responsible for the user-interface, which is not surprising because a significant part of the use-case involves interacting with the GUI. Orla Greevy and Andy Zaidman plan further comparisons on other use-cases.

All the participants agreed that there is a significant potential for future collaborations, and drew up a series of suggestions for future comparative studies. For trace-based tools, it was decided that a common trace format should be used, to make traces interchangeable and to guarantee that tools are compared with one another appropriately. A wider selection of systems (and of use cases) was also suggested to ensure that all tools could handle the subject system (JHotdraw was a popular candidate because it is smaller and is used as experimental subject in a number of other projects).

3.2 Design Recovery

The goal of the working group on *Design recovery* was to devise a taxonomy of, or at least a comparative framework for, design recovery tools. During a very focused, intensive and productive group discussion, several dimensions of concern were identified and, for each of those, a number of sub-criteria were proposed. These concerns and criteria are relevant criteria against which different design recovery tools could be compared.

The list of concerns and criteria thereof was further refined after the workshop by three of the participants in this working group (Yann-Gaël Guéhéneuc, Kim Mens, and Roel Wuyts) with the goal of validating and refining it against different design recovery tools in existence today (and those presented at the workshop in particular). Although this is clearly ongoing work, in its current status the comparative framework distin-

guishes the following dimensions of concern :

Concern	Description
Context	In what context does the tool apply?
Intent	What is the purpose of the tool?
Users	What is expected from the users of the tool?
Input	What input is accepted by the tool?
Technique	What reasoning mechanism is used?
Output	What output is provided by the tool?
Implementation	How is the tool implemented?
Tool	How mature is the tool?

Each of those concerns is subdivided in different criteria, as summarised below :

Concerns	Criteria
Context	Methodology. Settings. Range of uses. Lifespan. Universe of discourse.
Intent	Short and Long term objectives.
Users	User knowledge. Targeted user. User background. Program acquaintance. Type of users. Cooperation.
Input	Kind of data. Type of data. Level of granularity. Precision. Underlying model. Representation. Versions. Data gathering automation. Assumptions about the software.
Technique	Level of automation. Semantic level. Method. Underlying model. Several quality criteria. . .
Output	Level of detail. Type of data. Representation. Underlying model. Manual interpretation required. Human- readable. Several quality criteria.
Implementation	Maintenance. Quality. Platforms. Language. External dependencies.
Tool	Language independence. Kind of license. Documentation. User base. Quality. Platforms. External dependencies.

A paper elaborating on an more detailed and improved version of this comparative framework is currently being written.

4 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. Therefore its vital that we organise such workshops outside of the traditional reengineering community (with conferences like ICSM, WCRE, CSMR, ...).
- *Research Community.* All participants agreed that it would be wise to organise a similar workshop at next year's ECOOP. There is an open invitation for everyone who wants to join in organising it: just contact the current organisers.

– *Workshop Format.* The workshop format, where some authors were invited to summarise position papers of others worked particularly well.
– *Joint Case Study.* The idea to compare research results by applying tools on the same case is a good idea. However, careful preparation is necessary to ensure that participants are able to obtain the results and compare notes *before* the actual workshop takes place.
– *Taxonomy.* Defining a taxonomy for design recovery proved worthwhile, although the time was too short to validate the result. Some of the workshop participants committed to continue the discussion after the workshop and write a paper about the result to be submitted to a conference or journal, as well as to next year's edition of this workshop.

References

1. 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.
2. S. Demeyer, S. Ducasse, and K. Mens. Workshop on object-oriented re-engineering (WOOR'03). In F. Buschmann, A. P. Buchmann, and M. Cilia, editors, *Object-Oriented Technology (ECOOP'03 Workshop Reader)*, volume 3013 of *Lecture Notes in Computer Science*, pages 72–85. Springer-Verlag, July 2003.
3. 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.
4. 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.
5. D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings for libraries and frameworks. In S. Demeyer, K. Mens, R. Wuyts, and S. Ducasse, editors, *proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. http://smallwiki.unibe.ch/WOOR, July 2005.
6. 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.
7. 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.
8. N. Flores and A. Aguiar. Jfreedom: a reverse engineering tool to recover framework design. In S. Demeyer, K. Mens, R. Wuyts, and S. Ducasse, editors, *proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. http://smallwiki.unibe.ch/WOOR, July 2005.
9. O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR*, pages 314–323. IEEE Computer Society, 2005.
10. Y.-G. Guéhéneuc and T. Ziadi. Automated reverse-engineering of UML v2.0 dynamic models. In S. Demeyer, K. Mens, R. Wuyts, and S. Ducasse, editors, *proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. http://smallwiki.unibe.ch/WOOR, July 2005.
11. D. Janakiram and J. Rajesh. Design patterns induction from multiple versions of software. In S. Demeyer, K. Mens, R. Wuyts, and S. Ducasse, editors, *proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. http://smallwiki.unibe.ch/WOOR, July 2005.

12. R. Marticorena, C. López, and Y. Crespo. Parallel inheritance hierarchy: Detection from a static view of the system. In S. Demeyer, K. Mens, R. Wuyts, and S. Ducasse, editors, *proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. http://smallwiki.unibe.ch/WOOR, July 2005.
13. N. Moha and Y.-G. Guéhéneuc. On the automatic detection and correction of design defects. In S. Demeyer, K. Mens, R. Wuyts, and S. Ducasse, editors, *proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. http://smallwiki.unibe.ch/WOOR, July 2005.
14. M. J. Pacione. VANESSA: Visualisation abstraction NEtwork for software systems analysis. In *International Conference on Software Maintenance (ICSM)*. IEEE, 2005.
15. N. Walkinshaw, M. Roper, and M. Wood. Understanding object-oriented source code from the behavioural perspective. In *IWPC*, pages 215–224. IEEE Computer Society, 2005.
16. R. Wuyts, S. Ducasse, S. Demeyer, and K. Mens. Workshop on object-oriented re-engineering (WOOR'04). In J. Malenfant and B. M. Østvold, editors, *Object-Oriented Technology (ECOOP'04 Workshop Reader)*, volume 3344 of *Lecture Notes in Computer Science*, pages 177–186. Springer-Verlag, June 2004.
17. A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR*, pages 134–142. IEEE Computer Society, 2005.