Table of Contents

Workshop on Object-Oriented Reengineering 1
Serge Demeyer (University of Antwerp), Yann-Gaël Guéhéneuc (Université de Montréal), Anne Keller (University of Antwerp), Martin Kuhlemann (University of Magdeburg), Adrian Kuhn (University of Berne), Christian F.J. Lange (Eindhoven University of Technology), Kim Mens (Université catholique de Louvain)

Workshop on Object-Oriented Reengineering

10^{th} Anniversary Edition

Serge Demeyer[1], Yann-Gaël Guéhéneuc[2], Anne Keller[1], Christian F.J. Lange[4], Kim Mens[3], Adrian Kuhn[5], and Martin Kuhlemann[6]

[1] Department of Mathematics and Computer Science,
University of Antwerp — Belgium
[2] Department of Computer Science and Operations Research,
Université de Montréal — Canada
[3] Département d'Ingénierie Informatique,
Université catholique de Louvain — Belgium
[4] Software Engineering and Technology Group,
Eindhoven University of Technology — The Netherlands
[5] Software Composition Group,
University of Berne —Switzerland
[6] School of Computer Science,
University of Magdeburg — Germany

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 problem of transforming their object-oriented "legacy" systems into full-fledged frameworks. To address this problem, a series of workshops has been organised to set up a forum for exchanging experiences, discussing solutions, and exploring new ideas. Typically, these workshops were organised as satellite events of major software engineering conferences, such as ECOOP [1–11] and ESEC/FSE [12–14]. During the past 10 years, participants of this workshop series have been actively contributing to the state-of-the-art on reengineering of object-oriented systems. This special 10^{th} anniversary edition was no exception and this report summarises the key discussions and outcome of that workshop.

1 Introduction

In preparation to the workshop, participants were asked to submit a position paper that would help in steering the workshop discussions. Five position papers were accepted, of which eight authors were present during the workshop. Together with a number of organisers and participants without position paper, the workshop attracted eighteen participants. The position papers and other information about the workshop are available on the workshop web-site at http://smallwiki.unibe.ch/woor2007/ [15].

A format for the workshop day was chosen 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. The workshop was concluded with a plenary session during which the results of the working groups were exposed and discussed by *all* workshop participants in the larger group.

2 Position Papers

Taking into account positive feedback from participants from the previous years, we decided to require that each paper be presented by the authors of another paper. We have observed over the years that this system ensures that participants read the position papers of one another and are able to discuss them from different viewpoints. For the authors, it is also interesting to hear what another researcher in the field has understood from the submission and thinks about the approach. As in previous years, this format resulted in vivid discussions during the presentations, which formed a good foundation for the afternoon discussions. We now give a short summary of each of the five position papers that were presented at the workshop.

Discussion on the Results of the Detection of Design Defects [16]. In this paper, the authors present validation results of their design defect detection method (DECOR), which allows the systematic specification of design defects and subsequently the automatic generation of design defect detection algorithms from these specifications. The specification language is based on high-level key concepts identified from textual design defect descriptions. The validation is based on evaluating of 4 design defects together with their 15 smells in terms of precision and recall.

nMARPLE: .NET Reverse Engineering with MARPLE [17]. Detecting design pattern in a system is an aspect of reverse engineering that may contribute to the understanding of the overall system design. The authors of this paper present a tool (nMARPLE) that is an extension of the existing design pattern detection tool MARPLE. nMarple is specifically designed to reverse engineer .Net executables and the authors discuss peculiarities of reverse engineering in a .Net context.

Challenges in Model Refactoring [18]. While refactoring of source code is a well-known technique to improve maintainability, refactoring of models is a largely unexplored territory. The authors of this paper discuss nine challenges in model refactoring that they identified as the most important ones. The discussed challenges include traditional refactoring issues—such as (model) quality and behaviour preservation—and modelling-specific issues—such as model synchronisation and defining refactorings for domain specific languages.

Must Tool Building Remain a Craft? [19]. In the reverse-engineering community, tool building plays an important role for validating research results. In this position paper, the author argues that despite its importance as validation instrument and its cost, tool building is approached as a craft rather than as an engineering discipline. On the basis of an extensive study of research literature, the author identifies shortcomings in academic tool building in the fields of requirements engineering, design-related issues, and development process.

A Meta-model Approach to Inconsistency Management [20]. As models become increasingly important software development artefacts, consistency between models, *i.e.*, between different views, different versions, and different abstraction levels, also gains

importance. Consistency between models however should be managed rather than enforced. The authors of this paper introduce the idea of an inconsistency meta-model to support inconsistency management.

3 Working Groups

Taking into account the presented position papers and the discussions that followed them, we split up in three working groups for the afternoon. The next three subsections report on the discussions that were held in each of these working groups.

3.1 Working Group 1: Model Reengineering

The topic of the first working group was *modelling*. The main objective of the working group was to discuss how the interests of the WOOR workshop such as reengineering and refactoring relate to modelling. Originally, models were used in software engineering to describe and communicate about a system. However, in recent techniques, such as Model Driven Architecture (MDA [21]), automated transformations are performed on models to create new models. The target model is based on the source model, but it contains more details, *i.e.*, it is more specific than the source model. The transformation is called *code generation* in case the target model is a representation of the original model in the form of source code. An overall result of the discussion is that, in the context of modelling, there are three areas of interest for reengineering research. We present the three areas of interest together with open questions that were raised during the discussions:

- **Source model**. Which flaws exist in a model? What are the quality requirements for the model to be useful to its audience or to be 'transformable'?
- **Target model**. Which flaws exist in a model? To what extent do flaws in the source model lead to flaws in the target model?
- **Transformation**. Is 'reengineering-by-transformation' possible (*i.e.*, can source model flaws be resolved by transformations)? Can transformations be improved by means of reengineering?

The discussions were divided into two parts that are summarised below. The major part of the discussion addressed *model smells*, similar to code smells (as proposed by Fowler [22]). A smaller part of the discussion was devoted to *code generation*.

Model Smells. Model smells are flaws in a model. To distinguish model smells from bugs or errors in general, the working group defined model smells as being specific for models, *i.e.*, they are caused by characteristics that are inherent to models. Additionally, it was agreed upon that model smells can be distinguished from "simple errors" by the amount of human effort that is involved in dealing with them. Usually the amount of human effort involved in detecting, analysing, prioritising, and resolving model smells is rather important, in comparison to simple errors, where automated techniques reduce the amount of human effort involved in dealing with them.

Furthermore the discussion addressed the resolution of model smells. The following set of basic model operations was identified, that can be used to resolve model smells: *add*, *remove*, *delete*, *merge*, *collapse*, *flatten*, and *split*. Depending on the type of a model smell and its location, its resolution consists of applying a subset of these operations. Future work should relate model smell types to particular resolution strategies based on these operations.

Two substantial questions regarding resolution were addressed:

- *Can* all smells be resolved?
- *Must* all smells be resolved?

The first question could not be answered at all in the discussion, because there did not exist an exhaustive list of model smells and because there exists only little work in the area of model smell resolution. The working group formed an opinion about the latter question. It is assumed that for automated processing of models (*e.g.*, model transformation) "small errors" must be resolved, and model smells should be resolved. However, for a human-only use of the models, errors and smells could remain unresolved, depending on the purpose of the model usage. More research is required to investigate which types of model smells must be resolved for particular purposes and which ones do not necessarily need to be resolved.

Besides these observations concerning model smells, the main outcome of the discussion was a list of types of model smells:

- *Inconsistency*. Inconsistencies are contradictions between different parts of the models, *e.g.*, between diagrams.
- *Incompleteness*. Incompleteness is the absence of necessary model elements. Typically, a model is incomplete with respect to some other artefacts, for example with respect to the requirements, *i.e.*, not all functionalities are modelled, or with respect to the information needed by a stake-holder.
- *Imbalance*. Imbalance refers to differences between parts of a model with respect to their properties, such as level of abstraction or completeness.
- *Redundancy*. Redundancy is present if a concept is described in different parts of a model. Redundant parts can be inconsistent, they can differ with one another, *e.g.*, with respect to their levels of abstraction or versions.
- *Violations of Modelling Conventions*. Modelling conventions [23] are a means to assure a uniform style of modelling amongst several modellers. Violations of these conventions are considered model smells.
- *(Too high) Complexity*. In modelling, usually different ways of modelling can be used to describe the same system. If a complex way of modelling is chosen while a simpler model would be possible, the high complexity is regarded as a model smell. This complexity differs from the complexity that is inherent to a problem, which cannot be reduced by a particular way of modelling.
- *Too abstract–Too detailed*. The appropriate level of abstraction depends on the goal of a model. A wrong choice in the abstraction level is regarded as a model smell.
- *Dead model parts*. Dead model parts are parts of a model that are outdated or that are not used anymore.

– *Layout flaws*. Several models offer diagrams as graphical representations. In case the layout of the diagrams does not follow established layout guidelines (*e.g.*, Sun et al. [24]), the flaw is regarded as a model smell.

Code Generators. A smaller part of the discussion was devoted to code generators. It was noticed that implementations that are based on the same source model can have different behaviours, depending on the code generator that was used. Additionally, participants reported that the choice of a particular code generator would strongly affect whether model refactorings are behaviour preserving. Therefore the comparison with respect to differences in behaviour between code generators, such as AndroMDA or GMF, was identified as an area for future research.

3.2 Working Group 2: Tool Building Issues

The second working group discussed a number of issues relevant to builders of reengineering tools. First of all, based on Kienle's position paper "Must tool building remain a craft?" [19], which was presented in the morning session, the group discussed whether or not tool building should remain a craft and highlighted some points in favour and against. Next they discussed what lessons could be learned from experienced tool builders: what are the success stories and what are the pitfalls to avoid? Finally, they briefly discussed the issue of how to compare and benchmark tools.

Should tool building remain a craft? Several arguments in favour of considering tool building as a craft were put forward. One of the strongest arguments is that building research tools is often an exploratory process. In many cases, neither the requirements nor the solution space of the problem at hand are known completely up front. Flexibility in research is needed. In such a situation, the process of building a tool can actually help researchers to shape the way they think about the problem and to discover what their tool can and should do. Exploratory modelling is often a key task in research tool building.

On the other hand, sometimes making just a proof of concept is not enough and there are some strong reasons for tool building not to be an ad-hoc process. First of all, having a more rigid process can increase the productivity of tool builders and may lead to better quality tools. It may even allow to automate part of the process. Also, if you do not want a tool that is just built for one occasion or experiment but rather a tool or a tool framework that remains for many years, then maybe following an ad-hoc process is not the right choice.

In conclusion, there seems to be a kind of trade-off to be made by the tool builders. But maybe these two alternative ways of building tools should not be regarded as two independent and separate approaches, but rather as complementary. Maybe the best process to follow is to first make a "quick and dirty" prototype of the tool that you would like to develop, possibly reusing some existing building blocks or parts of other tools built earlier. Having made such a prototype allows you to evaluate it and improve on it. This process can be repeated until you are happy with the developed tool or until some important limitations are encountered. At that point, it is probably wise to start adopting a more rigourous tool building process.

Positive lessons learned in building tools. There was a consensus among tool builders present at the workshop that, in the long run, it pays off to invest into shared infrastructure and common exchange formats. This facilitates both reuse of existing work and co-operation between researchers and research groups.

A well-known example of such a *common exchange format* in the domain of object-oriented reengineering is the FAMIX exchange model [25, 26]. This model allows reengineering tools or prototypes to exchange information concerning object-oriented source code. Building a common exchange format has the advantage that it is easier to get accepted than a common tool infrastructure, yet might lead to a common infrastructure once everyone starts using that common format.

To avoid reinvestigating effort in earlier tools, it is often better to build a bridge (like for instance mentioned in the reengineering pattern "build a bridge to the new town" [27]) to prior tools rather than reimplement them whenever you want to apply them in a new context or port them to a new environment. A noteworthy example hereof is the IntensiVE tool-suite [28], implemented in the Smalltalk language, which was originally built to manage and reason about evolution of Smalltalk programs. Since the underlying ideas of the tool were applicable to any object-oriented language, the decision was taken to extend the tool-suite to deal with Java programs as well. One way of doing so would have been to reimplement from scratch the entire tool in Java. However, this would have lead to a major implementation effort and two separate branches of the tool to be maintained: one for Smalltalk and one for Java. Instead, a bridge was implemented to allow the Smalltalk tool to access Java programs in an Eclipse workspace. Although implementing this bridge was not a trivial endeavour; in the end, it allowed having one tool that can deal with several languages. In addition, the bridge itself could be reused for similar Smalltalk programs that reason over Java programs.

A third proven recipe for success in reverse engineering tools is to make a common framework or tool that can serve as a backbone for several other tools. We can mention several examples of such a common backbone:

- The MOOSE platform that implements the language-independent FAMIX model mentioned above.
- The SOUL declarative meta-programming language [29] that was used as a backbone for the IntensiVE tool-suite [28] and many other tools that require reasoning over the structure of object-oriented source code.
- The (ASF+SDF) Meta-Environment [30] that is a framework for language development, source code analysis, and source code transformation. It has been successfully used in a wide variety of analysis, transformation, and renovation projects.
- the PADL meta-model of the Ptidej tool suite that is a language-independent meta-model to describe object-oriented systems and patterns, and that has been used in several research work, see for example [31].

Pitfalls in tool building. Tool development in research differs from tool building in industrial environments. The main difference is that research tools are mainly built as proofs of concept of techniques, solutions, and algorithms, rather than for commercial use. That particular set-up gives rise to several pitfalls and drawbacks, some of which are listed below.

- Research tools are usually built by Ph.D. candidates, and when the original tool builder leaves the research group, the tool often dies.
- Researchers often implement their tools from scratch, rather than building on prior tools or frameworks.
- Researchers tend to reimplement algorithms and tools that have already been implemented earlier. We have to remember that we are tool builders and not algorithm implementers.
- To be dependent on tools maintained by other researchers can be dangerous, as there is no guaranteed maintenance of research tools (see the first item above). This is a trade-off with the previous item: to avoid such dependencies, researchers often prefer to reimplement things themselves.
- We should use the right tool for the job. Some tools and environments are better for achieving your goal than others (try to make the right choice up front). For example, every so many years, a new language or environment hype appears in research or industry. As researchers, we should not switch to those new languages or environments when there is no real need to. On the one hand, remember that we are not commercial tool builders: techniques, solutions, and algorithms can also be proven and verified in more dynamic and lightweight environments. On the other hand, a good reason for jumping on the mainstream could be to have a large user base for conducting empirical studies.

Tool building patterns. We identified the need to distil recurring patterns from these pitfalls and positive lessons. There have been some interesting approaches in that direction recently. For example, to overcome the tendency of building monolithic, single-purpose software development tools, Vainsencher and Black [32] proposed a pattern language that would enable the integration of multiple analyses and tools in a more modular fashion.

An example that was discussed in the working group was the "tool bridge pattern". Many tool builders use a bridge pattern to reuse existing subtools with an "ad-hoc" bridge (for example, a bridge from your tool to the Eclipse API to access the right data in the Eclipse UI, or a bridge to a dedicated program like Mathematica to do some advanced calculations). Advantages of such a bridge are that it becomes much faster to implement the tool (because you do not have to reimplement the subtools) and that it is easy to replace the subtools you rely on by other ones later on. Disadvantages are that the tool(s) may be considered as less coupled from a user point of view (you need two or more subtools rather than having one homogeneous tool) and that the bridge may bring some runtime overhead. Implementing such a bridge may also require quite some hacks and technicalities, thus leading to a more complex maintenance process.

Good practices. From the above discussions, we then distilled some "good practices" for tool builders to take into account:

- Probably a good strategy is to start by implementing an ad-hoc prototype first, to get the ideas straight, and then gradually adhere to a more rigourous development process.

- Developing a backbone tool infrastructure, or at least a common exchange format, is generally a good idea.
- Give the tool an extensible (data) architecture, to make it easy to extend or reuse later on.
- Overcome the "not invented here" syndrome and try to avoid reinventing or reimplementing existing tools. Rather, reuse or extend (mature) existing ones.
- Avoid "cowboy coding" where tools are implemented from scratch in an ad-hoc fashion.
- Create a community of researchers that use or work on the tool (either a single research group or bigger).

In addition to these good practices the importance of publishing scientific articles on the tool itself (and not only on the results that were obtained with it) was stressed. In spite of what is generally believed, there do exist several journals where such articles can be published, such as Elsevier's "Experimental Software and Toolkits" (EST) or Wiley's "Software—Practice and Experience" (SPE). There exist several workshops in domains related to object-oriented reengineering, where tool papers can be presented and published, sometimes as a special issue of some journal.

How to compare/benchmark tools? To conclude our working group session a final discussion was held on how to compare and benchmark tools. Although many articles and books have been written on "evaluation", tool builders do not seem to be sufficiently aware of that literature. They should get acquainted with that literature and use it in their articles, mentioning clearly what kind of evaluation was conducted (just some argumentation, benchmarks, prototypical examples, interviews and surveys, case studies, user studies, ...). Part of the problem is the lack of background of many researchers in empirical studies. Ideally, every researcher should have had a course on doing good formal empirical studies during his studies (statistics, null hypothesis, independent variables...). Unfortunately, often this is not the case.

3.3 Working Group 3: Language Independence for Reverse Engineering Tools

The subject of this working group was language independence for reverse engineering tools. The problem is that reverse engineering tools must accommodate many existing programming languages in (as well as making room for the new ones that appear) without having the need to "reinvent the wheel". This problem was raised during the morning session and in previous WOOR workshops and there was some overlap between this working group on the previous working group on tool building. Nevertheless, it was felt that language independence deserved dedicated focus because it is a problem that exists for many years and no solution, to the best of our knowledge, is yet within grasp. The discussions attempted to formalise the problem and to provide guidelines for future research on language independence. The discussions, although intense, led to a smaller body of knowledge than other working sessions because of the reduced number of participants and of the need for further research.

Conclusion of the discussions. During the discussions of this working group four main points came out.

- Language independent representations of source code need an agreed-upon level of abstraction, which could be an intermediate representation or bytecode.
- Language independence depends on the kind of analyses that will be performed on the program representations. Indeed, to define a language independent representation, we need to know what kind of analyses will be performed, because these analyses may *not* be language independent, possible only on a subset of major languages. In any case, there must exist mechanisms to extend the language independent representation of a program with language-dependent data, some context.
- The representation must be fit both for representing programs and for being queried. It is possible that a true language independent representation will be defined only as a set of low-level queries, which results can then be aggregated. Queries allow the integration of the concept of views.
- Language independent representation must itself be language independent. The use of XML Schema seems like promising to define such a representation, if used Schema must be instantiated/implemented in different programming languages according to the needs of its users.

Future directions of research. Future work includes studying existing program representations (whether they claim to be language independent or not) to compare them with one another and possibly identify common primitives. A representation must include a mechanism to allow new data to be included incrementally but applying successive analyses. A first list of existing representations includes (but is certainly *not* limited to):

- Basic elements, for example Elemental Design Pattern, inheritance, method calls...
- ASTs but one for each version of each language.
- MOF-based meta-model for Java in the process of combining with C++ (cf. Helmut)
- Queries-based, SOUL (Smalltalk, Java)
- UML.
- PADL.
- Rigi Standard Format but maybe language dependent because does not carry any semantics.
- FAMIX.
- AOL.

Future work also includes researching all known analyses developed or performed on a regular basis by researchers and practitioners to identify common and opposing requirements with respect to language independence. Then, among all possible candidates, a choice should be made based on the decision whether to support an analysis in the representation or not. A first list of existing analyses includes (but is certainly *not* limited to):

- Design pattern detection.

– Code smell detection.
– Association/aggregation/composition (but may be too low level).
– Point-to analysis.
– Clustering.
– Metric computations (which might enrich a representation).
– Slicing (dynamic and static).
– Test suite generation.
– Type inference (tagging nodes, refining nodes, instantiating new nodes).

4 Conclusion: What next ?

In this report, we have listed the main ideas that were generated during the workshop on object-oriented reengineering organised in conjunction with ECOOP 2007. After this tenth anniversary edition, the main question was whether it was worthwhile to continue working on reengineering, or whether it was better to continue on a new topic. The question remained open, but some participants suggested to organise a tool builders workshop, because tool building (integration, exchange formats, benchmarks...) has been an active topic of discussions that appeared throughout the whole WOOR series.

Acknowledgements

This workshop was sponsored by the Interuniversity Attraction Poles Programme (IUAP) on "Modeling, Verification, and Evolution of Software" (MOVES), financed by the *Belgian State – Belgian Science Policy* from January 2007 to December 2011.

References

1. Casais, E., Jaasksi, A., Lindner, T.: FAMOOS workshop on object-oriented software evolution and re-engineering. In Bosch, J., Mitchell, S., eds.: Object-Oriented Technology (ECOOP'97 Workshop Reader). Volume 1357 of Lecture Notes in Computer Science., Springer-Verlag (December 1997) 256–288
2. Ducasse, S., Weisbrod, J., eds.: Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-engineering. FZI report 6/7/98, FZI Forschungszentrum Informatik (July 1998)
3. Ducasse, S., Weisbrod, J.: Experiences in object-oriented reengineering. In Demeyer, S., Bosch, J., eds.: Object-Oriented Technology (ECOOP'98 Workshop Reader). Volume 1543 of Lecture Notes in Computer Science., Springer-Verlag (December 1998) 72–98
4. Ducasse, S., Ciupke, O., eds.: Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-engineering. FZI report 2-6-6/99, FZI Forschungszentrum Informatik (June 1999)
5. Ducasse, S., Ciupke, O.: Experiences in object-oriented re-engineering. In Moreira, A., Demeyer, S., eds.: Object-Oriented Technology (ECOOP'99 Workshop Reader). Volume 1743 of Lecture Notes in Computer Science., Springer-Verlag (December 1999) 164–183
6. Demeyer, S., Ducasse, S., Mens, K., eds.: 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)

7. Demeyer, S., Ducasse, S., Mens, K.: Workshop on object-oriented re-engineering (WOOR'03). In Buschmann, F., Buchmann, A.P., Cilia, M., eds.: Object-Oriented Technology (ECOOP'03 Workshop Reader). Volume 3013 of Lecture Notes in Computer Science., Springer-Verlag (July 2003) 72–85
8. Wuyts, R., Ducasse, S., Demeyer, S., Mens, K., eds.: Proceedings of the ECOOP'04 Workshop on Object-Oriented Re-engineering (WOOR'04). Technical Report, University of Antwerp - Department of Mathematics and Computer Science (June 2004)
9. Wuyts, R., Ducasse, S., Demeyer, S., Mens, K.: Workshop on object-oriented re-engineering (WOOR'04). In Malenfant, J., Østvold, B.M., eds.: Object-Oriented Technology (ECOOP'04 Workshop Reader). Volume 3344 of Lecture Notes in Computer Science., Springer-Verlag (June 2004) 177–186
10. Wuyts, R., Ducasse, S., Demeyer, S., Mens, K.: Workshop on object-oriented re-engineering (WOOR'05). In: Object-Oriented Technology (ECOOP'05 Workshop Reader). Lecture Notes in Computer Science, Springer-Verlag (2005)
11. Wuyts, R., Ducasse, S., Demeyer, S., Mens, K.: Workshop on object-oriented re-engineering (WOOR'06). In: Object-Oriented Technology (ECOOP'06 Workshop Reader). Lecture Notes in Computer Science, Springer-Verlag (2006)
12. Demeyer, S., Gall, H., eds.: 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 (September 1997)
13. Demeyer, S., Gall, H.: Report: Workshop on object-oriented re-engineering (WOOR'97). ACM SIGSOFT Software Engineering Notes **23**(1) (January 1998) 28–29
14. Demeyer, S., Gall, H., eds.: 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 (September 1999)
15. Demeyer, S., Guéhéneuc, Y.G., Mens, K., Wuyts, R., Ducasse, S., Gall, H., eds.: Proceedings of the ECOOP'07 Workshop on Object-Oriented Re-engineering (WOOR'07) – 10th anniversary edition, http://smallwiki.unibe.ch/woor2007/ (June 2007)
16. Moha, N., Guéhéneuc, Y.G., Duchien, L., Meur, A.F.L.: Discussion on the results of the detection of design defects. In Demeyer, S., Guéhéneuc, Y.G., Mens, K., Wuyts, R., Ducasse, S., Gall, H., eds.: Proceedings of the ECOOP'07 Workshop on Object-Oriented Re-engineering (WOOR'07) – 10th anniversary edition. (2007)
17. Arcelli, F., Cristina, L., Franzosi, D.: nMARPLE: .NET reverse engineering with MARPLE. In Demeyer, S., Guéhéneuc, Y.G., Mens, K., Wuyts, R., Ducasse, S., Gall, H., eds.: Proceedings of the ECOOP'07 Workshop on Object-Oriented Re-engineering (WOOR'07) – 10th anniversary edition. (2007)
18. Mens, T., Taentzer, G., Müller, D.: Challenges in model refactoring. In Demeyer, S., Guéhéneuc, Y.G., Mens, K., Wuyts, R., Ducasse, S., Gall, H., eds.: Proceedings of the ECOOP'07 Workshop on Object-Oriented Re-engineering (WOOR'07) – 10th anniversary edition. (2007)
19. Kienle, H.M.: Must tool building remain a craft? In Demeyer, S., Guéhéneuc, Y.G., Mens, K., Wuyts, R., Ducasse, S., Gall, H., eds.: Proceedings of the ECOOP'07 Workshop on Object-Oriented Re-engineering (WOOR'07) – 10th anniversary edition. (2007)
20. Keller, A., Demeyer, S.: A meta-model approach to inconsistency management. In Demeyer, S., Guéhéneuc, Y.G., Mens, K., Wuyts, R., Ducasse, S., Gall, H., eds.: Proceedings of the ECOOP'07 Workshop on Object-Oriented Re-engineering (WOOR'07) – 10th anniversary edition. (2007)
21. Object Management Group: MDA Guide, Version 1.0.1. omg/03-06-01 edn. (June 2003)
22. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Co., Inc. (November 1999)

23. Lange, C.F.J., DuBois, B., Chaudron, M.R.V., Demeyer, S.: An experimental investigation of UML modeling conventions. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006). Volume 4199 of Lecture Notes in Computer Science., Heidelberg, Springer (October 2006) 27–41
24. Wong, K., Sun, D.: On evaluating the layout of UML diagrams for program comprehension. Software Quality Journal **14**(3) (September 2006) 233–259
25. Serge Demeyer, S.T., Steyaert, P.: FAMIX 2.0 - the FAMOOS information exchange model (August 1999) Technical report, University of Berne.
26. Tichelaar, S., Ducasse, S., Demeyer, S.: FAMIX and XMI. In: Proceedings of the Seventh Working Conference of Reverse Engineering, IEEE Computer Society (2000) 296–298
27. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann, San Francisco, CA, USA (2002)
28. Mens, K., Kellens, A., Pluquet, F., Wuyts, R.: Co-evolving code and design with intensional views — a case study. Journal on Computer Languages, Systems and Structures **32**(2–3) (July-October 2006) 140–156 Special Issue: Smalltalk.
29. Mens, K., Michiels, I., Wuyts, R.: Supporting software development through declaratively codified programming patterns. Journal on Expert Systems with Applications (23) (November 2002) 405–431
30. Klint, P.: A meta-environment for generating programming environments. ACM Transactions on Software Engineering nd Methodology **2**(2) (1993) 176–201
31. Antoniol, G., Guéhéneuc, Y.G.: Feature identification: An epidemiological metaphor. Transactions on Software Engineering **32**(9) (September 2006) 627–641
32. Vainsencher, D., Black, A.P.: A pattern language for extensible program representation (2006) Pattern Languages of Programming Confernce (PLoP2006).