

Refactoring: Current Research and Future Trends

Tom Mens^{1,2}

*Programming Technology Lab
Vrije Universiteit Brussel, Belgium*

Serge Demeyer, Bart Du Bois, Hans Stenten, Pieter Van Gorp³

*Lab on Re-Engineering
Universiteit Antwerpen, Belgium*

Abstract

In this paper we provide an detailed overview of existing research in the field of software restructuring and refactoring, from a formal as well as a practical point of view. Next, we propose an extensive list of open questions that indicate future research directions, and we provide some partial answers to these questions.

1 Introduction

An intrinsic property of software in a real-world environment is its need to evolve. As the software is enhanced, modified and adapted to new requirements, the code becomes more and more complex and drifts away from its original design. Because of this, the major part of the total software development cost is devoted to software maintenance [26,49,61]. Better software development methods and tools do not solve this problem, because their increased capacity is used to implement more new requirements within the same time frame [44], making the software more complex again.

To cope with this spiral of complexity there is an urgent need for techniques that reduce software complexity by incrementally improving the internal software structure. The research domain that addresses this problem is referred to

¹ Tom Mens is a PostDoctoral Fellow of the Fund for Scientific Research - Flanders

² Email: tom.mens@vub.ac.be

³ Email: {serge.demeyer, bart.dubois, hans.stenten, pieter.vangorp}@ua.ac.be

as *restructuring* [2,47] or, in the case of object-oriented software development, *refactoring* [74,40].

According to the reverse engineering taxonomy of Chikofsky and Cross [21], restructuring is defined as follows:

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics). A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system.

The definition of refactoring is basically the same: “*the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure*” [40]. The key idea here is to redistribute classes, variables and methods in order to facilitate future adaptations and extensions.

[46,102] report on cases where refactoring was used in complex systems. In [46] refactoring is used to introduce tests to deal with the growing complexity of a system. [102] describes the effects of refactorings and unit testing on code in multiple versions of a large Java framework.

2 Current research

2.1 Formalisms

A wide variety of formalisms have been proposed and used to deal with restructuring and refactoring. We provide an indicative but (inevitably) incomplete list of such formalisms here.

Program slicing [91,12] has been proposed to deal with a specific kind of restructurings: function or procedure extraction [58,57]. These techniques based on system dependence graphs can be used to guarantee that a refactoring preserves some selected behaviour of interest. A similar, but less formal approach is taken in [56], where an algorithm is proposed to move a selected set of nodes in a control-flow graph together so that they become extractable while preserving program semantics.

Graph transformations [30,36,35,73,28] are another way to deal with restructuring: the software itself is represented as a graph, and restructurings correspond to transformation rules. Graph transformations have been proposed to provide support for software refactoring in [70,17]. [54] suggest to use graph transformation to replace occurrences of poor design patterns in a legacy program by good design patterns.

Software metrics [20,38,51] are another way to deal with refactorings. They

can be used before a refactoring, to measure the (internal or external) quality of a software system, or after a refactoring, to measure improvements of the quality. [31] proposes to use change metrics to detect refactorings between two successive releases of a software system. [85] uses metrics to detect where there is a need for refactoring a given software system. [25] defines a polynomial multiple measures, which provides a single maintainability index on which the effect of refactoring can be evaluated.

The technique of *formal concept analysis* [43] can be used to deal with restructuring. [86] uses concept analysis to restructure object-oriented class hierarchies, based on the “usage” of this hierarchy by a set of applications. The result is guaranteed to be behaviourally equivalent with the original hierarchy. [95] uses the same technique to restructure software modules. [98] uses concept analysis to identify objects by semi-automatically restructuring legacy data structures.

Philipps and Rumpé [77] suggest to reconsider existing refinement approaches [103,5,76] as a way to formally deal with the notions of behaviour, behaviour equivalence and behaviour preservation. Indeed, these notions are not specific to the domain of refactoring, but also occur in the areas of algebraic specification and refinement techniques that are much more mature:

- *Hidden sorts* [45] can be used to explicitly distinguish between internal and externally visible behaviour, and discusses its implications to the preservation of externally visible behaviour;
- *Behavioural refinement of state machines* [81] is needed because preservation of behavioural equivalence normally would be too restrictive. The idea is to add details to derive concrete implementations from abstractly specified behaviour.
- *Refinement of dataflow architectures* [76] uses a clearly defined notion of observable behaviour that allows to precisely define what preservation and refinement of behaviour means.
- *The refinement calculus* [6] is a framework for the stepwise derivation of imperative programs from a specification.

2.2 Techniques

In the context of *software re-engineering*, restructuring is often used to convert legacy code into a more modular or structured form, or even to migrate code to a different programming language or even language paradigm [37].

Software visualisation is another technique that can help with restructuring the software. [48] propose to use star diagrams for this purpose. *DupLoc* is a graphical tool for detecting code duplication [33]. *CodeCrawler* is a tool that visualises software metrics to identify places in the code that need restructuring [59].

Meta modelling is a useful technique because it makes refactoring less

dependent on the implementation language [90]. Logic meta programming has been used to detect "bad code smells" in object-oriented software to find out where refactorings should be applied [63,97].

2.3 Languages

Support for restructuring has been provided in a variety of different programming languages and language paradigms:

- the imperative programming language *Fortran* [13];
- the functional languages *Scheme* [47], *Lisp* [60] and *Haskell* [89];
- the class-based object-oriented languages *Smalltalk* [80], *Java* [40] and *C++* [75,94];
- the prototype-based object-oriented language *Self* [72].

A very recent research trend is to deal with refactoring at a higher level than source code, e.g. UML design models [3,17,87]. [15] developed a refactoring browser integrated in a UML modeling tool. It provides support for refactoring class diagrams, state diagrams and activity diagrams.

Last but not least, [8] applied restructuring operations in the context of database schema evolution.

2.4 Tool support

Although it is possible to refactor manually, tool support is considered crucial. Today, a wide range of tools is available that automate various aspects of refactoring.⁴ Depending on the tool and the kind of support that is provided, the degree of automation can vary.

Tools such as the *Refactoring Browser* [80], *XRefactory* [104], *jFactor* [52] support a semi-automatic approach.

Some researchers demonstrated the feasibility of *fully automated refactoring*. For example, *Guru* is a fully automated tool for restructuring inheritance hierarchies and refactoring methods in *SELF* programs [72]. Other automatic refactoring approaches are proposed in [19,54,82,23].

There is also a tendency to integrate refactoring tools directly into industrial strength software development environments. This is for example the case for *Smalltalk VisualWorks v7* [22], *Eclipse v2* [34], *Together ControlCenter v6* [92], *IntelliJ IDEA v3* [53], *Borland JBuilder v7* [16], etc...

The focus of all these tools is on applying a refactoring upon request of the user. There is much less support available for detecting where and when a refactoring can be applied. [85] proposes to do this by means of metrics, while [55] indicates where refactorings might be applicable by automatically detecting program invariants using the *Daikon* tool. This approach is based on

⁴ For an extensive and up-to-date overview of refactoring tools, we refer to <http://www.refactoring.com/>.

dynamic analysis of the runtime behaviour, and seems to be complementary to other approaches.

3 Future Trends

Despite all this work that has already been realised, from the practical as well as the formal side, there are still a lot of open issues that remain to be solved. In this section we summarise a number of these problems in the form of questions with partial answers. The questions have been subdivided into more fundamental, research-directed questions on the one hand, and more practical, tool-directed questions on the other hand (although the distinction is not always that clear).

3.1 Fundamental research questions

3.1.1 Which formalisms are most suited for our needs?

This question has already been answered partially before. We have seen that the following formalisms are used frequently in the context of refactoring: graph transformations; software metrics; program slicing; concept analysis; refinement techniques.

Obviously, other formalisms may be useful as well. For example, *statistical techniques* may be used to perform empirical measurements on the practical use of refactorings and the effect on the software quality.

3.1.2 How can we compose refactorings in a scalable way?

While support for complex refactorings is a necessity for large-scale re-engineering projects, current development tools and formalisms only support primitive refactorings. To be able to implement tools for complex refactorings that scale up to industrial software, it should be possible and easy to compose primitive refactorings into more complex refactorings [94].

One of the advantages of more complex refactorings is that they may be more efficient than an equivalent series of primitive refactorings. For example, we need to check the preconditions only once, rather than for each primitive refactoring in the sequence separately [79,64,67].

In [88], three ways to compose primitive design pattern transformations are expressed: *sequencing* signifies applying transformations in order one after another, *set iteration* means performing transformations iteratively on a set of program elements, and *concurrency* signifies a set of transformations performed concurrently.

3.1.3 How can we analyse dependencies between refactorings?

When building complex refactorings, it is also crucial to determine which refactorings are mutually independent, and which refactorings have to be applied sequentially. Parallel application of refactorings often leads to unexpected

conflicts [68]. Therefore, a formal basis to detect and resolve such conflicts is essential [64,65,66]. From a theoretical point of view, we can rely on existing results about parallelism and confluency in graph transformation systems [7] and critical pair analysis [50].

Detecting sequential dependencies between refactorings is also important to deal with change propagation [78]: because of the so-called “ripple effect”, the application of a refactoring may require many other refactorings to be applied as well [97].

3.1.4 What is behaviour, and how can it be preserved by a refactoring?

Refactoring implies that program “behaviour” is preserved, but a precise definition of behaviour is rarely provided, or may be too inefficient to be checked in practice.

Another problem is that the intuitive definition of observational behaviour equivalence, stating that “for the same input, we should obtain exactly the same output”, does not always suffice. In many application domains, the observable input-output semantics is not the only thing that is relevant:

- For *real-time systems*, an essential aspect of the behaviour is the execution time of certain (sequences of) operations;
- For *embedded systems*, memory constraints and power consumption are also important aspects of the behaviour that need to be taken into account;
- For *safety-critical systems*, there is a concrete notion of safety that needs to be preserved by a refactoring.

Therefore, in an ideal world, refactorings should be able to preserve these properties as well. In practice, not all these properties need to be preserved by all software entities, so we will in fact have a wide range of different notions of behaviour preservation.

From a practical side, one may deal with behaviour preservation in a very pragmatic way, for example by means of a rigorous testing discipline. If we have a rather complete set of test cases, and they all pass after the refactoring, there is good evidence that the refactoring is behaviour preserving.

From a formal side, one may attempt to determine a sufficiently expressive formal language of interesting program invariants, and relate this to the set of refactorings that guarantees the preservation of all expressible properties. This is similar to the well-known work of Courcelle [29] on monadic second order logic.

In the research literature, different notions of behaviour preservation have been used in the context of refactoring. [11] define *object-preserving class transformations*. [10] use a special kind of graph transformations that have the property of being *language-preserving*: the set of all acceptable program inputs before and after the transformations must be the same. As such, it provides a framework for restructuring with a theoretical basis in formal language theory.

3.1.5 *How can we guarantee consistency between software artifacts at different levels during refactoring?*

Code level refactoring affects higher level software artifacts (such as design models, analysis documents, architectures) and vice versa. Therefore, all levels of software description should be kept consistent when the software artifacts evolve. Consistency between artifacts at the same level (such as statecharts, interaction diagrams and class diagrams in a UML design) is also important in the context of refactoring [17].

3.1.6 *How can we compare tools, techniques and formalisms for refactoring?*

To evaluate whether a certain refactoring tool or formalism is more suitable than another one, they need to be compared in a disciplined way. This can be achieved on the basis of a taxonomy of relevant evolution criteria. [69] provides a first attempt to define a general taxonomy for comparing software evolution tools. [84] applied this taxonomy to compare four different refactoring tools.

3.2 *Practical questions*

3.2.1 *How can we specify refactorings in a language-independent manner?*

A tool or formal model for refactoring should be sufficiently abstract to be applicable to different programming languages, but should also provide the necessary hooks to add language-specific behaviour. OMG's *Model Driven Architecture* (MDA) promises to enable this kind of platform independence by explicitly separating out platform dependencies from the *Platform Specific Model* into a mapping from and to the *Platform Independent Model* [83].

Refactoring can also be very useful at an even more implementation-dependent level than programming languages, such as refactoring of bytecode, or even refactoring of executable code. Work on this is done by developers of compilers and compiler generators [41,62], the focus there lies on optimizing performance but the techniques used to transform e.g. bytecode are compatible with the definition of refactoring. Depending on the application domain these techniques could be used to improve other aspects (security, power consumption, code size, memory usage...) of a system. Research on runtime compilers [18] could be used in combination with refactoring to adapt running applications to changing requirements without restarting them.

3.2.2 *How can we apply refactorings at higher levels of abstraction?*

Refactorings are also useful at higher levels of abstraction than source code, such as design models, design patterns and software architectures:

Design models are typically specified using the Unified Modelling Language. Various attempts have already been made to provide support for refactoring of these UML models [3,15,17,87], but a more integrated approach is still needed. MDA extends the UML to enable the evolution of component models [39]. MDA code generators are based on model-to-model and

model-to-code transformations that are configurable by so-called “pattern” or “template” editors [27,24]. In the context of refactoring, the question that arises is how these code generators can be used for design optimization.

Design patterns provide a means to describe the program structure at a high level of abstraction [42]. Often, refactorings are used to introduce new design pattern instances into the software [82,23,93,101]. Design patterns also impose constraints on the software structure, which may limit applicability of certain refactorings. To detect this, we can resort to logic rules [96]. [54] suggest to use graph transformation techniques to restructure/replace occurrences of poor design patterns in a legacy program by a good design pattern.

To deal with refactoring of *software architectures*, an approach as proposed by [76], where refactoring rules are based directly on the graphical representation of a system architecture seems promising. These rules preserve the behaviour imposed by the causal relationship between the components’ behaviour. A more pragmatic approach is taken by [94]: architectural changes to two software systems are made by performing a sequence of primitive refactorings (81 refactorings in a first case study, 800 refactorings in a second case study).

3.2.3 *How can we build more open refactoring tools?*

Besides language independence, a refactoring tool should be easy to extend by the user with new refactoring operations. For this purpose, the extensibility mechanism of existing tools (e.g., plug-ins, APIs or wizards) do not suffice. Therefore, [60] suggests to use a pattern language to express refactorings.

3.2.4 *How can we determine where and why refactorings should be applied?*

Most refactoring tools only provide support for *applying* a refactoring, but not for finding out *where* and *why* a particular refactoring should be applied. Techniques such as the ones proposed in [85,55] may help to address these questions.

Quality is software domain specific. If we restrict ourselves to, for example, web applications the question of “where and why” to refactor is partially answered by the high-level refactorings from [1]. [88] encode design decisions as *soft-goal graphs* to guide the application of the transformation process. These soft-goal graphs describe correlations between quality attributes. The association of refactorings with a possible effect on soft-goals addresses maintainability enhancements through primitive and complex refactorings.

Meta programming techniques may be used to specify quality-related heuristics about object-oriented software (such as the detection of “bad code smells”) to find out where refactorings should be applied [63,97].

3.2.5 *How can we deal with evolution conflicts?*

Another issue has to do with *evolution conflicts* in presence of refactoring. This issue is particularly relevant for object-oriented application frameworks,

where the framework may be instantiated by many different customers, while the framework is also subject to evolution itself. This implies that a refactoring of the framework may lead to evolution conflicts in each of these instantiations [64]. Merge techniques [68] may be useful here.

3.2.6 *Where does refactoring fit in the software development process?*

As stated, tools can assist in determining how, where and why to apply refactorings. Another fundamental question on when to apply refactorings concerns the software development process. Refactoring is supported by agile development processes such as *eXtreme Programming* (XP) software development process [9]. The reason is that the activity of refactoring requires short iterative development cycles. For the same reason, refactoring does not fit very well in the waterfall or spiral models of software engineering [14]. Therefore, it remains an open question if and how the activity of refactoring can be included in a more classical software development process. [32] describes how refactoring fits in the software reengineering process.

Integrated development environments like *Smalltalk VisualWorks* [22] and *Eclipse* [34] or *IDEA* [53] provide considerable support for XP, using a combination of refactoring support and unit testing, two core activities in XP. In [99], the relationship between testing and refactoring is explored in more detail to address the practical problem that refactorings often invalidate tests. [100] shows that refactorings of test code is different from refactoring production code in two ways: (a) there is a distinct set of bad smells involved, and (b) improving test code involves additional test-specific refactorings.

A final question is how refactoring fits into a model-driven reengineering process. One of the goals of model-driven architectures (MDA) [83] is to facilitate platform migration by code generation from abstract models [4]. At first sight, this reduces the refactoring effort for platform migration substantially. However, code generation implies forward engineering and introduces a fixed architecture which typically is not present in hand-written code. Refactoring can be applied to transform the design of existing code into a form that can be understood by the reverse engineering facilities of an MDA tool. More research is required to decide which refactorings can be applied where and when in a model-driven reengineering process and what other techniques are complementary.

3.2.7 *What is the effect of a refactoring on the software quality?*

For any software system we can specify its external quality attributes (such as correctness, robustness, extensibility, reusability, compatibility, efficiency, ease of use, portability and functionality) [71]. Refactorings can be classified according to which of these quality attributes they affect. This allows us to improve the quality of a software system by applying the relevant refactorings at the right places.

There are many different refactorings, each having a particular purpose

and effect. Some refactorings remove code redundancy, some raise the level of abstraction, some enhance the reusability, some have a negative effect on the performance, etc... By classifying refactorings in terms of the internal quality attributes they affect, the effect of a refactoring on the software quality can be estimated (e.g., by means of software metrics, empirical studies, controlled experiments and statistical techniques).

4 Conclusion

The research in software restructuring and refactoring continues to be very active. Although commercial refactoring tools are beginning to proliferate, there are still a lot of open issues that remain to be solved. In general, there is a need for formalisms, processes, methods and tools that address refactoring in a more consistent, generic, scalable and flexible way. In this paper we raised a number of open questions, from a fundamental as well as from a practical perspective. We believe that this list of questions can be used as a research agenda for future research within the area of software refactoring.

Acknowledgement

This research is funded by the FWO Project G.0452.03 "A formal foundation for software refactoring" and is carried out in the context of the scientific networks "Formal Foundations of Software Evolution" and "Research Links to Explore and Advance Software Evolution" financed by the Fund for Scientific Research - Flanders and the European Science Foundation, respectively.

We thank Kim Mens, Johan Fabry, Johan Brichau for their comments on drafts of this paper.

References

- [1] Alur, D., J. Crupi and D. Malks, "Core J2EE Patterns," Sun Microsystems Press, 2001 .
- [2] Arnold, R. S., "Tutorial on Software Restructuring," IEEE Press, 1986 .
- [3] Astels, D., *Refactoring with UML*, in: *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, 2002, pp. 67–70, Alghero, Sardinia, Italy.
- [4] Atkinson, C. and T. Kühne, *The role of meta-modeling in MDA*, in: *Proc. UML 2002 Workshop on Software Model Engineering*, 2002, pp. 67–70, Dresden, Germany.
- [5] Back, R.-J., *Correctness preserving program refinements*, Technical Report Mathematical Centre Tracts #131, Mathematisch Centrum Amsterdam (1980).

- [6] Back, R.-J. and J. von Wright, “Refinement Calculus,” Springer Verlag, 1998.
- [7] Baldan, P., A. Corradini, H. Ehrig, M. Löwe, U. Montanari and F. Rossi, “Handbook of Graph Grammars and Graph Transformation,” World scientific, 1999 pp. 107–188.
- [8] Banerjee, J. and W. Kim, *Semantics and implementation of schema evolution in object-oriented databases*, in: *Proc. ACM SIGMOD Conference*, 1987.
- [9] Beck, K., “Extreme Programming Explained: Embrace Change,” Addison Wesley, 2000.
- [10] Bergstein, P. L., *Maintenance of object-oriented systems during structural evolution*, *Theory and Practice of Object Systems* **3(3)** (1991), pp. 185–212.
- [11] Bergstein, P. L., *Object-preserving class transformations*, in: *Proc. Conf. Object-oriented programming systems, languages, and applications* (1991), pp. 299–313.
- [12] Binkley, D. and K. Gallagher, *Program slicing*, *Advances of Computing* **43** (1996), pp. 1–50.
- [13] Bodin, F., *Sage++: an object-oriented toolkit and class library for building Fortran and C++ restructuring tools*, in: *Proc. 2nd Object-Oriented Numerics Conference*, 1994, Sunriver, Oregon.
- [14] Boehm, B., *Software engineering*, *IEEE Transactions on Computers* **12(25)** (1976), pp. 1226–1242.
- [15] Boger, M., T. Sturm and P. Fragemann, *Refactoring browser for UML*, in: *Proc. 3rd Int’l Conf. on eXtreme Programming and Flexible Processes in Software Engineering*, 2002, pp. 77–81, Alghero, Sardinia, Italy.
- [16] Borland, *JBuilder* (2002).
URL www.borland.com/jbuilder/
- [17] Bottoni, P., F. Parisi-Presicce and G. Taentzer, *Coordinated distributed diagram transformation for software evolution*, *Electronic Notes in Theoretical Computer Science* **72(4)** (2002).
- [18] Buytaert, D. and F. Arickx, *A selective runtime compiler for the wonka virtual machine*, Presentation at PACT Symposium at University of Ghent (2002).
- [19] Casais, E., *Automatic reorganization of object-oriented hierarchies: a case study*, *Object Oriented Systems* **1** (1994), pp. 95–115.
- [20] Chidamber, S. R. and C. F. Kemerer, *A metrics suite for object-oriented design*, *IEEE Trans. Software Engineering* **20(6)** (1994), pp. 476–493.
- [21] Chikofsky, E. J. and J. H. Cross, *Reverse engineering and design recovery: A taxonomy*, *IEEE Software* **7(1)** (1990), pp. 13–17.
- [22] Cincom, *Smalltalk VisualWorks* (2002).
URL www.cincomsmalltalk.com/

- [23] Cinnéide, M., “Automated Application of Design Patterns: A Refactoring Approach,” Ph.D. thesis, Department of Computer Science, Trinity College, University of Dublin (2000).
- [24] Codagen, *Codagen architect* (2002).
URL www.codagen.com/products/architect/
- [25] Coleman, D., P. Arnold, S. Boff, H. Gilchrist, F. Hayes and P. Jeremaes, “Object-oriented Development: the Fusion method,” Prentice Hall, Englewood Cliffs, NJ, 1994.
- [26] Coleman, D., D. Ash, B. Lowther and P. Oman, *Using metrics to evaluate software system maintainability*, IEEE Computer **27(8)** (1994), pp. 44–49.
- [27] Compuware, *Optimalj pattern-driven generator* (2002).
URL <http://www.compuware.com/products/optimalj/>
- [28] Corradini, A., H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors, “Graph Transformation,” Lecture Notes in Computer Science **2505**, Springer-Verlag, 2002.
- [29] Courcelle, B., *Graph rewriting: an algebraic and logic approach*, in: J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B* (1990), pp. 193–242.
- [30] Cuny, J., H. Ehrig, G. Engels and G. Rozenberg, editors, “Graph Grammars and Their Application to Computer Science,” Lecture Notes in Computer Science **1073**, Springer-Verlag, 1996.
- [31] Demeyer, S., S. Ducasse and O. Nierstrasz, *Finding refactorings via change metrics*, in: *Proc. Conf. Object-oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices **35(10)** (2000), pp. 166–177.
- [32] Demeyer, S., S. Ducasse and O. Nierstrasz, “Object-Oriented Reengineering Patterns,” Morgan Kaufmann and DPunkt, 2002.
- [33] Ducasse, S., M. Rieger and S. Demeyer, *A language independent approach for detecting duplicated code*, in: H. Yang and L. White, editors, *Proc. Int’l Conf. Software Maintenance* (1999), pp. 109–118.
- [34] eclipse.org, *Eclipse* (2002).
URL www.eclipse.org/
- [35] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, “Theory and Application to Graph Transformations,” Lecture Notes in Computer Science **1764**, Springer-Verlag, 2000.
- [36] Engels, G., E. Hartmut and G. Rozenberg, editors, “Special Issue on Graph Transformations,” *Fundamenta Informaticae* **26(3,4)**, IOS Press, 1996.
- [37] Fanta, R. and V. Rajlich, *Restructuring legacy C code into C++*, in: *Proc. Int’l Conf. Software Maintenance* (1999), pp. 77–85.

- [38] Fenton, N. and S. L. Pfleeger, “Software Metrics: A Rigorous and Practical Approach,” International Thomson Computer Press, London, UK, 1997, second edition.
- [39] Flater, D., *Impact of model-driven standards*, in: *35th Annual Hawaii International Conference on System Sciences (HICSS’02)*, Lecture Notes in Computer Science **9** (2002), p. 285.
- [40] Fowler, M., “Refactoring: Improving the Design of Existing Programs,” Addison-Wesley, 1999.
- [41] Fraser, C. W., D. R. Hanson and T. A. Proebsting, *Engineering a simple, efficient code-generator generator*, ACM Letters on Programming Languages and Systems **1** (1992), pp. 213–226.
- [42] Gamma, E., R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Languages and Systems,” Addison-Wesley, 1994.
- [43] Ganter, B. and R. Wille, “Formal Concept Analysis: Mathematical Foundations,” Springer-Verlag, 1999.
- [44] Glass, R. L., *Maintenance: Less is not more*, IEEE Software **15(4)** (1998), pp. 67–68.
- [45] Goguen, J., *Hidden algebra for software engineering*, in: *Proc. Conf. Discrete Mathematics and Theoretical Computer Science*, Australian Computer Science Communications **21** (1999), pp. 35–59.
- [46] Graham, W., *extreme programming in a hostile environment*, Sessionpaper at Int. Conf. on eXtreme Programming (2002).
- [47] Griswold, W. G., “Program Restructuring as an Aid to Software Maintenance,” Ph.D. thesis, University of Washington (1991).
- [48] Griswold, W. G., M. I. Chen, R. W. Bowdidge and J. D. Morgenthaler, *Tool support for planning the restructuring of data abstractions in large systems*, in: *Proc. 4th Symp. Foundations of Software Engineering*, ACM SIGSOFT Software Engineering Notes **21(6)** (1996), pp. 33–45.
- [49] Guimaraes, T., *Managing application program maintenance expenditure*, Comm. ACM **26(10)** (1983), pp. 739–746.
- [50] Heckel, R., J. M. Küster and G. Taentzer, *Confluence of typed attributed graph transformation systems*, in: *Graph Transformation*, Lecture Notes in Computer Science **2505** (2002), pp. 161–176.
- [51] Henderson-Sellers, B., “Object-Oriented Metrics: Measures of Complexity,” Prentice-Hall, 1996.
- [52] Instantiations, *jFactor* (2002).
URL www.instantiations.com/jfactor/
- [53] IntelliJ, *IDEA* (2002).
URL www.intellij.com/idea/

- [54] Jahnke, J. H. and A. Zündorf, *Rewriting poor design patterns by good design patterns*, in: S. Demeyer and H. Gall, editors, *Proc. of ESEC/FSE '97 Workshop on Object-Oriented Reengineering*, Technical University of Vienna, 1997, Technical Report TUV-1841-97-10.
- [55] Kataoka, Y., M. D. Ernst, W. G. Griswold and D. Notkin, *Automated support for program refactoring using invariants*, in: *Proceedings of the International Conference on Software Maintenance* (2001), pp. 736–743.
- [56] Komondoor, R. and S. Horwitz, *Semantics-preserving procedure extraction*, Technical report, Computer Sciences Department, University of Wisconsin-Madison (2000).
- [57] Lakhota, A. and J.-C. Deprez, *Restructuring programs by tucking statements into functions*, in: M. Harman and K. Gallagher, editors, *Special Issue on Program Slicing*, Information and Software Technology **40**, Elsevier, 1998 pp. 677–689.
- [58] Lanubile, F. and G. Visaggio, *Extracting reusable functions by flow graph-based program slicing*, *Trans. Software Engineering* **23(4)** (1997), pp. 246–258.
- [59] Lanza, M. and S. Ducasse, *Understanding software evolution using a combination of software visualization and software metrics*, in: *LMO 2002 Proceedings*, Hermes Publications, 2002 pp. 135–149.
- [60] Leitão, A. M., *A formal pattern language for refactoring of Lisp programs*, in: *Proc. 6th European Conf. Software Maintenance and Reengineering* (2002), pp. 186–192.
- [61] Lientz, B. P. and E. B. Swanson, “Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations,” Addison-Wesley, 1980.
- [62] Lindholm, T. and F. Yellin, “The Java Virtual Machine Specification,” Addison Wesley, 1996.
- [63] Mens, K., I. Michiels and R. Wuyts, *Supporting software development through declaratively codified programming patterns*, *Journal on Expert Systems with Applications* **23** (2002), pp. 405–431.
- [64] Mens, T., “A Formal Foundation for Object-Oriented Software Evolution,” Ph.D. thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium (1999).
- [65] Mens, T., *Conditional graph rewriting as a domain-independent formalism for software evolution*, in: *Proc. Int’l Conf. Active 1999: Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science **1779** (2000), pp. 127–143.
- [66] Mens, T., *A formal foundation for object-oriented software evolution*, in: *Proc. Int’l Conf. Software Maintenance* (2001), pp. 549–552.

- [67] Mens, T., *Transformational software evolution by assertions*, in: *CSMR Workshop on Formal Foundations of Software Evolution*, Lisbon, 2001.
- [68] Mens, T., *A state-of-the-art survey on software merging*, *IEEE Trans. Software Engineering* **28(5)** (2002), pp. 449–462.
- [69] Mens, T., J. Buckley, A. Rashid and M. Zenger, *Towards a taxonomy of software evolution*, in: *Proc. Workshop on Unanticipated Software Evolution*, 2003, Warshau, Poland.
- [70] Mens, T., S. Demeyer and D. Janssens, *Formalising behaviour preserving program transformations*, in: *Graph Transformation*, *Lecture Notes in Computer Science* **2505** (2002), pp. 286–301.
- [71] Meyer, B., “Object-Oriented Software Construction,” Prentice Hall, 1997, second edition.
- [72] Moore, I., *Automatic inheritance hierarchy restructuring and method refactoring*, in: *Proc. Int’l Conf. OOPSLA ’96*, *ACM SIGPLAN Notices* (1996), pp. 235–250.
- [73] Nagl, M., A. Schürr and M. Münch, editors, “Applications of Graph Transformations with Industrial Relevance,” *Lecture Notes in Computer Science* **1779**, Springer-Verlag, 2000.
- [74] Opdyke, W. F., “Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks,” Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).
- [75] Opdyke, W. F., *Refactoring C++ programs*, Technical report, Lucent Technologies/ Bell Labs (1999).
URL st-www.cs.uiuc.edu/users/opdyke/wfo.990201.c++.refac.html
- [76] Philipps, J. and B. Rumpe, *Refinement of information flow architectures*, in: M. Hinchey, editor, *Proc. ICFEM’97* (1997).
- [77] Philipps, J. and B. Rumpe, *Roots of refactoring*, in: K. Baclavski and H. Kiloy, editors, *Proc. 10th OOPSLA Workshop on Behavioral Semantics* (2001), Tampa Bay, Florida, USA.
URL www4.informatik.tu-muenchen.de/papers/PR01.html
- [78] Rajlich, V., *A model for change propagation based on graph rewriting*, in: *Proc. Int’l Conf. Software Maintenance* (1997), pp. 84–91.
- [79] Roberts, D., “Practical Analysis for Refactoring,” Ph.D. thesis, University of Illinois at Urbana-Champaign (1999).
- [80] Roberts, D., J. Brant and R. Johnson, *A refactoring tool for Smalltalk*, *Theory and Practice of Object Systems* **3(4)** (1997), pp. 253–263.
- [81] Scholz, P., “Design of reactive systems and their distributed implementation with statecharts,” Ph.D. thesis, Technische Universität München (1998).

- [82] Schulz, B., T. Genssler, B. Mohr and W. Zimmer, *On the computer aided introduction of design pattern into object-oriented systems*, in: *Technology of Object-Oriented Languages and Systems* (1998), pp. 258–267.
- [83] Siegel, J. and OMG Staff Strategy Group, *Developing in OMG’s model-driven architecture*, Technical Report White Paper Revision 2.6, Object Management Group (2001).
- [84] Simmonds, J. and T. Mens, *A comparison of software refactoring tools*, Technical Report vub-prog-tr-02-15, Programming Technology Lab (2002).
- [85] Simon, F., F. Steinbrückner and C. Lewerentz, *Metrics based refactoring*, in: *Proc. European Conf. Software Maintenance and Reengineering* (2001), pp. 30–38.
- [86] Snelling, G. and F. Tip, *Reengineering class hierarchies using concept analysis*, in: *Proc. Foundations of Software Engineering (FSE-6)*, SIGSOFT Software Engineering Notes **23(6)** (1998), pp. 99–110.
- [87] Sunyé, G., D. Pollet, Y. LeTraon and J.-M. Jézéquel, *Refactoring UML models*, in: *Proc. UML 2001*, Lecture Notes in Computer Science **2185** (2001), pp. 134–138.
- [88] Tahvildari, L. and K. Kontogiannis, *A methodology for developing transformations using the maintainability soft-goal graph*, in: *In Proceedings of the 9th IEEE Working Conference on Reverse Engineering (WCRE)* (2002), pp. 77–86.
- [89] Thompson, S. and C. Reinke, *Refactoring functional programs*, Technical report, Computing Laboratory, University of Kent (2001).
- [90] Tichelaar, S., “Modeling Object-Oriented Software for Reverse Engineering and Refactoring,” Ph.D. thesis, University of Bern (2001).
- [91] Tip, F., *A survey of program slicing techniques*, Journal of Programming Languages **3(3)** (1995), pp. 121–189.
- [92] TogetherSoft, *ControlCenter* (2002).
URL www.togethersoft.com/
- [93] Tokuda, L. and D. Batory, *Automated software evolution via design pattern transformations*, in: *Proc. 3rd Int. Symp. Applied Corporate Computing*, 1995.
- [94] Tokuda, L. and D. Batory, *Evolving object-oriented designs with refactorings*, Automated Software Engineering **8(1)** (2001), pp. 89–120.
- [95] Tonella, P., *Concept analysis for module restructuring*, Trans. Software Engineering **27(4)** (2001), pp. 351–363.
- [96] Tourwé, T., “Automated Support for Framework-Based Software Evolution,” Ph.D. thesis, Vrije Universiteit Brussel (2002).

- [97] Tourwé, T. and T. Mens, *Automatically identifying refactoring opportunities using logic meta programming*, in: *Proc. Int'l Conf. Software Maintenance and Re-engineering (CSMR)*, 2003.
- [98] van Deursen, A. and T. Kuipers, *Identifying objects using cluster and concept analysis* (1998).
- [99] van Deursen, A. and L. Moonen, *The video store revisited – thoughts on refactoring and testing*, in: *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, 2002, pp. 71–76, Alghero, Sardinia, Italy.
- [100] van Deursen, A., L. Moonen, A. van den Bergh and G. Kok, *Refactoring test code*, in: M. Marchesi, editor, *Proc. 2nd Int'l Conf. eXtreme Programming and Flexible Processes*, 2001.
- [101] van Winsen, P., “(Re)engineering with object-oriented design patterns,” Master’s thesis, Universiteit Utrecht (1996).
- [102] Wege, C. and M. Lippert, *Diagnosing evolution in test-infected code*, in: Succi, G., Marchesi, M. (eds.): *Extreme Programming Examined, Proc. 2nd Int. Conf. eXtreme Programming and Flexible Processes in Software Engineering* (2001), pp. 127–131.
- [103] Wirth, N., *Program development by stepwise refinement*, *Comm. ACM* **14** (1971), pp. 221–227.
- [104] XRef-Tech, *XRefactory* (2002).
URL xref-tech.com/speller/