# Refactoring: Emerging Trends and Open Problems

Tom Mens
Service de Génie Logiciel
Université de Mons-Hainaut
B-7000 Mons, Belgique
tom.mens@umh.ac.be

Arie Van Deursen
CWI & Delft University of Technology
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands
Arie.van.Deursen@cwi.nl

October 22, 2003

**Abstract**

*This position paper identifies emerging trends in refactoring research, and enumerates a list of open questions, from a practical as well as a theoretical point of view. We suggest these directions for further research based on our own experience with refactoring, as well as on a detailed literature survey on software refactoring.*

## 1. Introduction

The term *refactoring*, first introduced by Opdyke in his PhD thesis [10] refers to "*the process of changing an [object-oriented] software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure*" [6]. Refactoring can be regarded as the object-oriented equivalent of *restructuring*, which is defined by Chikofsky and Cross [2] as "*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). [...] 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.*"

Refactoring as proposed by Fowler not only covers the mechanics of restructuring, but also addresses the following issues:

- Refactoring emphasizes that, in absence of more formal guarantees, testing should be used to ensure that each restructuring is behavior preserving. A rich test suite should be built, which must be run before and after each test is applied.
- Refactorings are described in a catalog, using a template reminiscent of design patterns.
- Refactorings are applied in small steps, one by one, running the test suite after every step.

Mens and Tourwé [9] have performed a very extensive literature survey on refactoring research. This survey has been used as a basis in this paper to identify future trends in refactoring research, and to suggest a number of open research questions and practical questions that remain to be solved.

## 2. Emerging trends

### 2.1. Refactoring Activities

The refactoring process consists of a number of different activities, each of which can be automated to a certain extent: (1) identify where the software should be refactored; (2) determine which refactorings should be applied to the identified places; (3) guarantee that the applied refactoring preserves behaviour; (4) apply the refactoring; (5) assess the effect of refactoring on software quality characteristics; (6) maintain consistency between refactored program code and other software artifacts (or vice versa).

If we look at contemporary tools for software refactoring, we see that support is ususally restricted to step 4 (and, to a lesser extent, step 2), while support for the remaining steps is largely neglected. Although some promising research for these other steps has been initiated, it still needs to make it into commercial development tools.

### 2.2. Techniques

As clearly identified in [9], a wide variety of techniques and formalisms have been proposed and used to deal with restructuring and refactoring.

- *Assertions* (invariants, pre-, and postconditions) can be used to express properties that should hold before and after the refactoring has been applied.
- *Graph transformations* provide an underlying theory of refactoring [8]. Each refactoring corresponds to a graph production rule, and each refactoring application

corresponds to a graph transformation. The theory of graph transformation can be used to reason about applying refactorings in parallel, using theoretical concepts such as confluence and critical pair analysis.

- *Program slicing* can be used to guarantee that a restructuring preserves some selected behaviour of interest. It has been used to deal with program restructurings such as function extraction.
- *Software metrics* can be used before a refactoring, to measure the quality of a software system, and to identify places that need refactoring. It can also be applied after the refactoring, to measure improvements of the software quality.
- *Formal concept analysis* can be used to restructure object-oriented class hierarchies in a behaviour preserving way.
- *Program refinement* can be used to formally express program refinements in a behaviour preserving way.

Initial results with each of these formalisms show that they are promising for supporting certain aspects of refactoring. However, further research and tool support remains necessary. We also need to identify other formalisms that might be used to support refactoring.

## 2.3. Types of software artifacts

Although refactorings have been applied primarily at the level of source code, the same techniques can also be used for a wide variety of different software artifacts: (object-oriented) database schemata; (UML) design models; software architectures; software requirements; executable code; test suites and many more. As an example, a series of specific refactorings on JUnit test suites are presented in [5].

For some of these types of software refactorings, initial research results have already been reported, but much more research in this direction is needed. Also, support for co-evolution between different types of artifacts is needed. For example, if the source code is refactored, how are the design models affected and vice versa? An initial analysis of the impact of source code refactorings on test suites is presented in [4].

## 2.4. Languages

Support for restructuring has been provided in a variety of different programming languages and language paradigms: imperative or procedural languages (*Fortran*, *Cobol*, *C*, ...); functional languages (*Scheme*, *Lisp*, *Haskell*, ...); logic languages (e.g., *Prolog*); class-based object-oriented languages (*Smalltalk*, *Java*, *C++*, ...); prototype-based object-oriented languages (e.g, *Self*); aspect-oriented languages (e.g., *AspectJ*).

Here the question can be raised whether techniques/formalisms/tools that have been conceived for one language/paradigm can be ported to another? Can we generalise the results over different languages? Can we specify and reason about refactorings in a language-independent way? Do we need particular refactorings for particular language concepts?

## 2.5. 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. For an extensive and up-to-date overview of refactoring tools, we refer to `http://www.refactoring.com/`.

Depending on the tool and the kind of support that is provided, the degree of automation can vary. Most commercial tools support a semi-automatic approach, where the user has to specify which refactoring he wants to apply to which part of the source code, and the refactoring tool applies the selected refactoring automatically. Some researchers also demonstrated the feasibility of *fully automated refactoring*, but these have the disadvantage that the refactored software becomes more difficult to understand. For example when removal of duplicated code is taken to the extreme, this tends to decrease understandability.

## 2.6. Process support

Refactoring is an important part of the software development process.

In the context of a *model-driven achitecture* (MDA) process, refactorings are a special kind of model-to-model transformations. They 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 an MDA process and what other techniques are complementary.

Refactoring naturally fits in an *agile software development* process. It forms even one of the cornerstones of the eXtreme Programming process, together with unit testing.

It is unclear how refactoring fits in the traditional waterfall or spiral models of software engineering [1]. In particular, the tight integration with software testing emphasized by Fowler is not easy to achieve in processes with a late testing activity. Therefore, it remains an open question if and how the activity of refactoring can be included in a more classical software development process.

In a *framework-based* or *product line software development* process, an entire range of products are developed/instantiated from a common core system. These leads to the problem that different instantiations may become inconsistent when this core system is refactored. Hence, adequate support for inconsistency maintenance is required.

*Software reengineering* projects aim at restructuring legacy software, possibly using a route starting with reverse engineering aimed at raising the level of abstraction on which the reengineering can take place. Reengineering purposes include support for platform migrations, migrations to new types of library, and so on. It does not often happen that a reengineering project takes place just for the sake of improving the quality — in which case it could be considered as a (large) refactoring activity. Within a reengineering project, refactoring can have its place: certain types of refactorings may help to simplify the reengineering itself. Moreover, refactorings can be applied after the migration has been conducted, in order to clean up the migrated code.

# 3. Open Questions

This section points out some future directions for refactoring research in the form of open questions. The questions have been subdivided into fundamental, research-directed questions on the one hand, and practical, tool-directed questions on the other hand (although the distinction is not always that clear).

## 3.1. Fundamental research questions

*Which formalisms and techniques are best suited for which purpose?* A wide range of formalisms and techniques can be used to support refactorings. Each one has its own merits and weaknesses. Hence, we need to identify which ones are most appropriate for which refactoring activity, and how the different formalisms and techniques can be combined.

*How can we analyse and manage dependencies between refactorings?* When building complex refactorings, it is crucial to determine which refactorings are mutually independent, and which refactorings have to be applied sequentially. Independent refactorings may be applied in parallel to speed up the refactoring. Detecting sequential dependencies between refactorings is also important to deal with change propagation, because the application of a refactoring may require many other refactorings to be applied as well.

*What is behaviour, and how can it be preserved by a refactoring?* Refactoring implies that program *behaviour* is preserved. The definition of refactoring talks about the external behaviour, which states that "for the same input, we should obtain exactly the same output". In many situations, however, other aspects of the behaviour are as least as important. 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 essential aspects of the behaviour. For *safety-critical systems*, there is a concrete notion of safety (e.g., the liveness property) that needs to be preserved by a refactoring. This implies that we need a wider range of notions of behaviour that may or may not be preserved by a refactoring, depending on domain-specific or even user-specific concerns.

## 3.2. Practical questions

*How can we tackle the scalability of refactorings?* To be able to implement tools for complex refactorings that scale up to industrial software, it is necessary to compose primitive refactorings into more complex refactorings. The use of composite refactorings has several advantages. Firstly, they better capture the specific intent of the software change induced by the refactoring. As such, it becomes easier to understand how the software has been refactored. Secondly, using composite refactorings will result in a performance gain because the tool needs to check the preconditions only once for the composite refactoring, rather than for each primitive refactoring in the sequence separately.Thirdly, composite refactorings weaken the behaviour preservation requirements of their primitive constituents. The primitive refactorings in a sequence do not have to be behaviour preserving, as long as we can ascertain that the net effect of their composition is behaviour preserving.

*How can we manage consistency between software artifacts at different levels during refactoring?* Software is composed of many different types of software artifacts (design models, analysis documents, architectures, source code, test suites, and so on) that all evolve. Therefore, all these software artifacts should be kept consistent when any of them is being refactored. It is also important to analyse the impact of a refactoring on all these artifacts, because a single change may propagate throughout the entire software system.

*How can we build more open/extensible refactoring tools?* Refactoring tools need better mechanisms to configure them with user-specific or domain-specific information about when and where to apply which kind of refactoring, or to extend the tool with new refactorings.

*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, adaptability, reusability, compatibility, performance, ease of use, portability and understandability). Refactorings can be classified according to which of these quality attributes they improve. This allows us to increase the quality of a software system by applying the relevant refactorings at the right places. To achieve this, we have to classify refactorings in terms of their measurable effect on internal quality metrics (such as coupling, cohesion and complexity), and relate these metrics to the external quality attributes to which the are correlated. We also need case studies, empirical studies and controlled experiments to provide anecdotal or statistical evidence about this correlation.

Quality in a refactoring context is often described using

the "code smell" metaphor, which has led to the creation of smell detection tools [14, 13]. Can these tools also be used to assess the quality improvements resulting from the application of refactorings?

*How can we compare tools, techniques and formalisms for refactoring?* Refactoring tools, techniques and formalisms need to be compared to evaluate whether a certain refactoring tool or formalism is more suitable (e.g., more performant, more correct, more scalable) than another one; to evaluate whether they are complementary; to compare the effects of a set of tools on the performance of the software that has been refactored; to compare the number of false positives or false negatives of different tools; and so on. A first prerequisite for comparison is a *taxonomy* of relevant evolution criteria. [7] provides a first attempt to define a general taxonomy for software evolution tools. [12] applied this taxonomy to compare four different refactoring tools. Secondly, we also need a *benchmark* for refactoring [3]. Such a benchmark provides a commonly agreed set of case studies that is freely available and that can be used by anyone who wishes to apply his own tool, technique or formalism to these case studies. The results of this should be freely available for other researchers to compare with their own results. For all the steps that need to be undertaken to come to an acceptable benchmark, we refer to [11].

# 4. Conclusion

The research in software 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. This list of questions can be used as a research agenda for future reseach within the area of software refactoring.

**Acknowledgements.** Tom Mens is partially funded by FWO research project G.0452.03 "A formal foundation for software refactoring". The research is carried out in the context of the scientific research 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.

Arie van Deursen received partial support from SENTER, project Ideals, hosted by the Embedded Systems Institute, Eindhoven, The Netherlands.

# References

[1] B. Boehm. Software engineering. *IEEE Transactions on Computers*, 12(25):1226–1242, 1976.

[2] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[3] S. Demeyer, T. Mens, and M. Wermelinger. Towards a software evolution benchmark. In *Proc. Int'l Workshop on Principles of Software Evolution*, September 2001.

[4] A. v. Deursen 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*, pages 71–76, 2002. Alghero, Sardinia, Italy.

[5] A. v. Deursen, L. Moonen, A. v. d. Bergh, and G. Kok. Refactoring test code. In G. Succi, M. Marchesi, D. Wells, and L. Williams, editors, *Extreme Programming Perspectives*, pages 141–152. Addison-Wesley, 2002.

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

[7] T. Mens, J. Buckley, A. Rashid, and M. Zenger. Towards a taxonomy of software evolution. In *Proc. Workshop on Unanticipated Software Evolution*, March 2003. Warshau, Poland.

[8] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002.

[9] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Software Engineering*, ?:?–?, ? ? Revised submission.

[10] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[11] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proc. Int'l Conf. Software Engineering*, 2003.

[12] J. Simmonds and T. Mens. A comparison of software refactoring tools. Technical Report vub-prog-tr-02-15, Programming Technology Lab, November 2002.

[13] T. Tourwé and T. Mens. Automatically identifying refactoring opportunities using logic meta programming. In *Proc. Int'l Conf. Software Maintenance and Re-engineering (CSMR)*, 2003.

[14] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, October 2002.