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 Gorp\(^3\)

*Lab on Re-Engineering*
*Universiteit Antwerpen, Belgium*

**Abstract**

In this paper we provide a 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 [26,49,60]. 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

©2003 Published by Elsevier Science B. V.
as restructuring [2,47] or, in the case of object-oriented software development, refactoring [73,40].

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.

[46,101] 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. [101] 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 [90,12] 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 [30,36,35,72,28] 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 [69,17]. [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. [84] 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. [85] 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 be behaviourally equivalent with the original hierarchy. [94] uses the same technique to restructure software modules. [97] uses concept analysis to identify objects by semi-automatically restructuring legacy data structures.

Philipps and Rumpe [76] suggest to reconsider existing refinement approaches [102, 5, 75] 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 [80] 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 [75] 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].

Meta modelling is a useful technique because it makes refactoring less dependent on the implementation language [89]. Logic meta programming has been used to detect ”bad code smells” in object-oriented software to find
out where refactorings should be applied [62,96].

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 [59] and Haskell [88];
- the class-based object-oriented languages Smalltalk [79], Java [40] and C++ [74,93];
- the prototype-based object-oriented language Self [71].

A very recent research trend is to deal with refactoring at a higher level than source code, e.g. UML design models [3,17,86]. [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. Depending on the tool and the kind of support that is provided, the degree of automation can vary.

Tools such as the Refactoring Browser [79], XRefactory [103], jFactor [52] support a semi-automatic approach.

Some researchers demonstrated the feasibility of fully automated refactoringing. For example, Guru is a fully automated tool for restructuring inheritance hierarchies and refactoring methods in SELF programs [71]. Other automatic refactoring approaches are proposed in [19,54,81,23].

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 [91], 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. [84] 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 dynamic analysis of the runtime behaviour, and seems to be complementary to other approaches.

\[4\] For an extensive and up-to-date overview of refactoring tools, we refer to http://www.refactoring.com/.
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 [93].

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 [78, 63, 66].

In [87], 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 [67]. Therefore, a formal basis to detect and resolve such conflicts is essential [63, 64, 65]. 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 [77]: because of the so-called “ripple effect”, the application of a refactoring may require many other refactorings to be applied as well [96].

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. [68] provides a first attempt to define a general taxonomy for comparing software evolution tools. [83] 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 [82].

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 [41,61], 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 [3,15,17,86], 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 [81, 23, 92, 100]. 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 [95]. [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 [75], 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 [93]: 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, [59] 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 [84, 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]. [87] 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 [62, 96].

### 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
[63]. Merge techniques [67] 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 refac-
torings. Another fundamental question on when to apply refactorings concerns
the software development process. Refactoring is supported by agile develop-
ment processes such as eXtreme Programming (XP) software development
process [9]. The reason is that the activity of refactoring requires short iter-
ative 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 in-
cluded 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 com-
bination of refactoring support and unit testing, two core activities in XP. In
[98], the relationship between testing and refactoring is explored in more detail
to address the practical problem that refactorings often invalidate tests. [99]
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) [82] is to facil-
itate 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) [70]. 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.

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


URL www.borland.com/jbuilder/


URL www.cincomsmalltalk.com/

URL www.codagen.com/products/architect/


[27] Compuware, Optimalj pattern-driven generator (2002).
URL http://www.compuware.com/products/optimalj/


[34] eclipse.org, Eclipse (2002).
URL www.eclipse.org/


[46] Graham, W., extreme programming in a hostile environment, Sessionpaper at Int. Conf. on eXtreme Programming (2002).


[52] Instantiations, jFactor (2002).
   URL www.instantiations.com/jfactor/

   URL www.intellij.com/idea/


URL st-www.cs.uiuc.edu/users/opdyke/wfo.990201.c++.refac.html


URL www4.informatik.tu-muenchen.de/papers/PR01.html


