A Survey of Software Refactoring

Tom Mens, *Member*, *IEEE*, and Tom Tourwé

Abstract—This paper provides an extensive overview of existing research in the field of software refactoring. This research is compared and discussed based on a number of different criteria: the refactoring activities that are supported, the specific techniques and formalisms that are used for supporting these activities, the types of software artifacts that are being refactored, the important issues that need to be taken into account when building refactoring tool support, and the effect of refactoring on the software process. A running example is used throughout the paper to explain and illustrate the main concepts.

Index Terms—Coding tools and techniques, programming environments/construction tools, restructuring, reverse engineering, and reengineering.

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 complex and drifts away from its original design, thereby lowering the quality of the software. Because of this, the major part of the total software development cost is devoted to software maintenance [1], [2], [3]. 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 [4], 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 quality. The research domain that addresses this problem is referred to as *restructuring* [5], [79] or, in the specific case of object-oriented software development, *refactoring* [6], [7].

According to the taxonomy of Chikofsky and Cross [8], *restructuring* is defined as "*the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (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 term *refactoring* was originally introduced by Opdyke in his PhD dissertation [6]. *Refactoring* is basically the object-oriented variant of restructuring: "*the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*" [7]. The key idea here is to redistribute classes, variables, and methods across the class hierarchy in order to facilitate future adaptations and extensions.

In the context of *software evolution*, restructuring and refactoring are used to improve the quality of the software (e.g., extensibility, modularity, reusability, complexity, maintainability, efficiency). Refactoring and restructuring are also used in the context of *reengineering* [9], which is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [8]. In this context, restructuring is needed to convert legacy code or deteriorated code into a more modular or structured form [10] or even to migrate code to a different programming language or even language paradigm [11].

The remainder of this paper is structured as follows: Section 2 explains general ideas of refactoring by means of an illustrative example. Section 3 identifies and explains the different refactoring activities. Section 4 provides an overview of various formalisms and techniques that can be used to support these refactoring activities. Section 5 summarizes different types of software artifacts for which refactoring support has been provided. Section 6 discusses essential issues that have to be considered in developing refactoring tools. Section 7 discusses how refactoring fits in the software development process. Finally, Section 8 concludes.

2 Running Example

In this section, we introduce a running example that will be used throughout the paper. The example illustrates a typical nontrivial refactoring of an object-oriented design. The initial design depicted in Fig. 1 represents an object-oriented class hierarchy. It shows a *Document* class that is refined into three specific subclasses *ASCIIDoc*, *PSDoc*, and *PDFDoc*. A document provides *preview* and *print* facilities, which are realized by invoking the appropriate methods in the associated *Previewer* and *Printer* classes, respectively. Before these methods can be invoked, some preprocessing or conversion needs to be done, which is realized differently for each of the *Document* subclasses. In Fig. 1, this is

- *T. Mens is with the Université de Mons-Hainaut, Avenue du Champ de Mars 6, B 7000 Mons, Belgium. E-mail: tom.mens@umh.ac.be.*
- *T. Tourwé is with the Centrum voor Wiskunde en Informatica, PO Box 94079, NL 1090 GB Amsterdam, The Netherlands. E-mail: tom.tourwe@cwi.nl.*

Manuscript received 30 Apr. 2003; revised 30 Dec. 2003; accepted 6 Jan. 2004. Recommended for acceptance by J.-M. Jezequel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0047-0403.

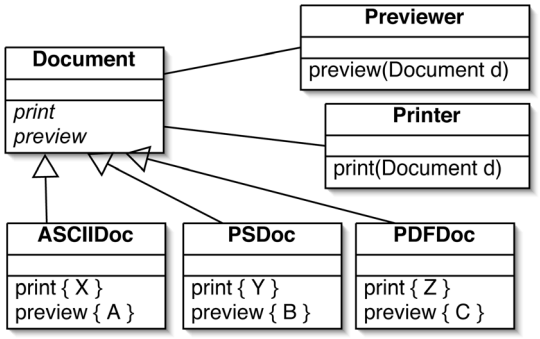

Fig. 1. *Document* class hierarchy and helper classes.

represented by the different code fragments `A`, `B`, `C` and `X`, `Y`, `Z`, respectively.

This design is not optimal because different functionalities of the *Document* class are distributed over all the subclasses. In order to add a new functionality to the *Document* class, such as a text search or a spell checker, we need to change every subclass of *Document* and we need to define the appropriate helper classes. Moreover, many such evolutions increase the complexity and reduce the understandability of the design because the *Document* class has many associations and no explicit relationship between all helper classes exists, although their roles are similar.

To overcome these problems, the design needs to be refactored. By introducing a so-called *Visitor* design pattern [12], the same functionality can be achieved in a more localized fashion, while, at the same time, the understandability of the design is improved. This is illustrated in Fig. 2. The idea is to introduce a *Visitor* class hierarchy that groups all helper classes and defines a common interface for them (the *visit** methods). At the same time, a generic *accept* method is implemented in all classes of the *Document* hierarchy. The *accept* method in each subclass calls a method, specifically defined for that subclass, of the *Visitor* hierarchy interface.

In this refactored design model, new functionality can be added by simply creating a new subclass of *Visitor*, and implementing the *visit** methods appropriately. As can be seen in Fig. 2, the implementations of the *print* and *preview* methods previously in subclasses of *Document* (i.e., `A, B, C, X, Y, Z`) have been moved to the *visit** methods of the *Printer* and *Previewer* classes (i.e., `A', B', C', X', Y', Z'`).

Although the above example is relatively simple, it already requires over 20 primitive refactorings to introduce the Visitor design pattern:

1. The *print* method in each *Document* subclass (three occurrences) is moved to class *Printer* using a **MoveMethod** refactoring.
2. To avoid name conflicts, each of the three moved *print* methods needs to be renamed first to a *visit** method using a **RenameMethod** refactoring.
3. The *preview* method in each *Document* subclass (three occurrences) is moved to class *Previewer* using a **MoveMethod** refactoring.
4. To avoid name conflicts, each of the three moved *preview* methods needs to be renamed first to a *visit** method using a **RenameMethod** refactoring.
5. An abstract *Visitor* class is introduced as a superclass for *Printer and Previewer* using an **AddClass** refactoring.
6. Three abstract *visit** methods are introduced in the new *Visitor* class using an **AddMethod** refactoring.
7. An *accept* method is introduced in all three subclasses of *Document* by extracting it from the *print* method and *preview* methods, using an **ExtractMethod** refactoring.
8. All *preview* and *print* methods now call the *accept* method with an instance of the appropriate *Visitor* subclass. Therefore, their definition can be pulled up to the *Document* class by using a **PullUpMethod** refactoring.

The refactorings in the above list are referred to as *primitive refactorings*. They are elementary behavior-preserving transformations that can be used as building blocks to create the so-called *composite refactorings* [6], [13]. These composite refactorings are usually defined as a sequence of primitive refactorings and reflect more complex behavior-preserving transformations that are more meaningful to the user. For example, the six refactorings in Steps 1 and 2 of the above enumeration can be combined into the single composite refactoring **MoveMethodsToVisitor** shown in Fig. 3. In a similar way, Steps 3 and 4 in the above enumeration can be combined into a single composite refactoring.

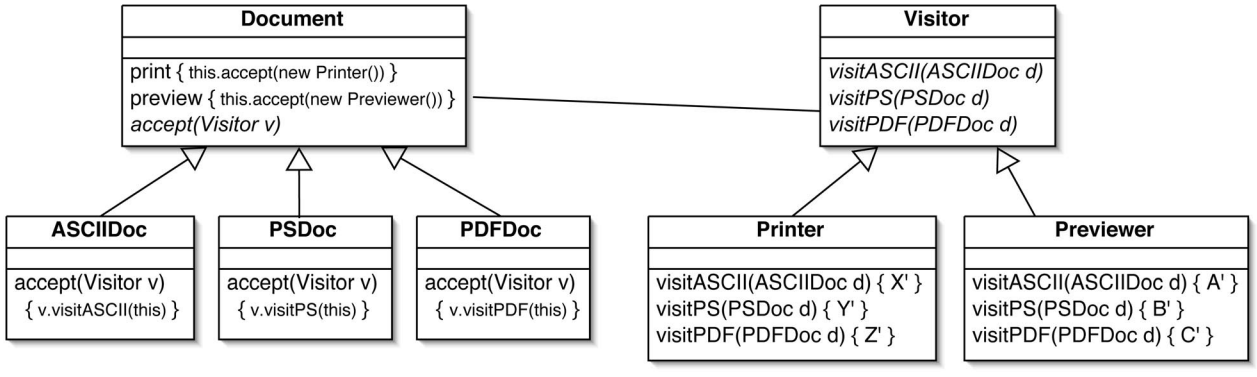

Fig. 2. Refactored design model for the *Document* class hierarchy.

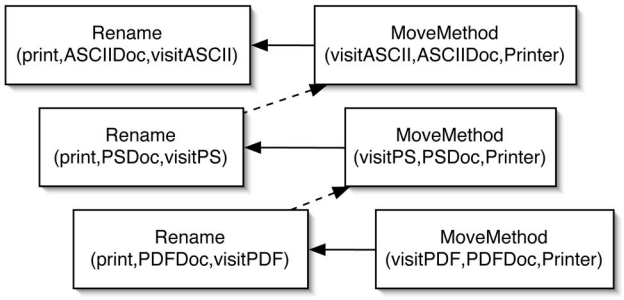

Fig. 3. Composite refactoring for renaming and moving *print* methods from the *Document* subclasses to the *Printer* class.

3 Refactoring Activities

The refactoring process consists of a number of distinct activities:

1. Identify where the software should be refactored.
2. Determine which refactoring(s) should be applied to the identified places.
3. Guarantee that the applied refactoring preserves behavior.
4. Apply the refactoring.
5. Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).
6. Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests, etc.).

As will be illustrated below, each of these activities can be supported by different tools, techniques or formalisms.

3.1 Identifying where to Apply which Refactorings

A first decision that needs to be made here is to determine the appropriate level of abstraction to apply the refactoring. Should the refactorings be applied to the program itself (i.e., the source code) or to more abstract software artifacts such as design models or requirements documents, for example?[1] We will tackle this particular question in detail in Section 5 and restrict ourselves to the subdomain of *program refactoring* here. In this subdomain, the activity of identifying the parts of the program that require refactoring (activity 1) and proposing refactorings that should be applied to these (activity 2) are usually combined.

Kataoka et al. implemented the *Daikon* tool to indicate where refactorings might be applicable by automatically detecting program invariants [14]. One invariant may be that a certain parameter of a method is always constant, or is a function of the other parameters of a method. In that case, it might be possible to apply a *removeParameter* refactoring. The main problem with this approach is that it requires dynamic analysis of the runtime behavior: The application needs to be executed to infer the program invariants. To this extent, the tool uses a representative set of test suites. It is, however, impossible to guarantee that a test suite covers all possible runs of a program. Therefore, the invariants may not hold in general. Nonetheless, very good results have been obtained in practice. Moreover, the approach is complementary to other approaches that rely on static information.

Probably the most widespread approach to detect program parts that require refactoring is the identification of *bad smells*. According to Beck, bad smells are "structures in the code that suggest (sometimes scream for) the possibility of refactoring" [7]. As a concrete example of a bad smell, reconsider the *Document* class hierarchy design in Fig. 1 of Section 2. By analyzing the code fragments `A`, `B`, `C` and `X`, `Y`, `Z`, respectively, it is very likely that one can detect a significant amount of code duplication. This is a typical example of a bad smell since code duplication should be avoided as it decreases maintainability. Balazinska et al. use a clone analysis tool to identify duplicated code that suggests candidates for refactoring [15]. Ducasse et al. sketch an approach to detect duplicated code in software and propose refactorings that can eliminate this duplication [16]. The approach is based on an object-oriented meta model of the source code and a tool that is capable of detecting duplication in code. The proposed refactorings consist of removing duplicated methods, extracting duplicated code from within a method and inserting an intermediate subclass to factor out the common code.

Fowler informally links bad smells to refactorings [7]. Tourwé and Mens use a semiautomated approach based on logic meta programming to formally specify and detect these bad smells and to propose refactoring opportunities that remove these bad smells [17]. A more ad hoc approach to detect structural weaknesses in object-oriented source code and solve them by refactorings is proposed by Dudziak and Wloka [19]. Van Emden and Moonen combine the detection of bad smells in Java with a visualization mechanism [18]. Simon et al. use object-oriented metrics to identify bad smells and propose adequate refactorings [20]. They focus on *use relations* to propose *move method/attribute* and *extract/inline class* refactorings. The key underlying concept is the *distance-based cohesion* metric, which measures the degree to which methods and variables of a class belong together. Especially in combination with software visualization, the use of object-oriented metrics seems well-suited to detect places in the source code that are in need of refactoring [20], [21].

A final but important issue is that identification of which refactorings to apply can be highly dependent on the particular application domain. If we restrict ourselves to, for example, Web-based software, the question of "where and why" to refactor is partially answered by the high-level refactorings from [22].

3.2 Guaranteeing that the Refactoring Preserves Software Behavior

By definition, a refactoring should not alter the behavior of the software. Unfortunately, a precise definition of behavior is rarely provided or may be inefficient to be checked in practice.

1. As a terminological side note, when we use the term *program* in the remainder of this paper, we specifically refer to the source code or executable code. In contrast, when we use the term *software*, we refer to any type of software artifact (including code, design models, requirements specifications, etc.).

The original definition of behavior preservation as suggested by Opdyke [6] states that, for the same set of input values, the resulting set of output values should be the same before and after the refactoring. Opdyke suggests to ensure this particular notion of behavior preservation by specifying *refactoring preconditions*. As a concrete example of such a refactoring precondition, reconsider the primitive refactorings in the running example of Section 2. The first refactoring suggested is **MoveMethod(print,ASCIIDoc, Printer)**. It has a number of necessary preconditions: The classes *ASCIIDoc* and *Printer* should be defined, the method *print* should be implemented in *ASCIIDoc*, and the method signature of *print* should not be present in class *Printer*. As can be seen in Fig. 1, the third precondition is *not* satisfied, which is precisely why the refactoring **RenameMethod (print,ASCIIDoc,visitASCII)** was suggested to avoid the method signature conflict.

In many application domains, requiring the preservation of input-output behavior is insufficient since many other aspects of the behavior may be relevant as well. This implies that we need a wider range of definitions of behavior that may or may not be preserved by a refactoring, depending on domain-specific or even user-specific concerns:

- For *real-time software*, an essential aspect of the behavior is the execution time of certain (sequences of) operations. In other words, refactorings should preserve all kinds of temporal constraints.
- For *embedded software*, memory constraints and power consumption are also important aspects of the behavior that may need to be preserved by a refactoring.
- For *safety-critical software*, there are concrete notions of safety (e.g., liveness) that need to be preserved by a refactoring.

One may deal with behavior preservation in a very pragmatic way, for example, by means of a rigorous testing discipline. If we have an extensive set of test cases and all these tests still pass after the refactoring, there is good evidence that the refactoring preserves the program behavior. Unfortunately, some refactorings will invalidate existing tests, even if the refactoring does not alter the behavior [23], [24]. The reason for this is that the tests may rely on the program structure that is modified by the refactoring.

Another pragmatic, but slightly more formal, approach is to adopt a weaker notion of behavior preservation that is insufficient to formally guarantee the full preservation of program semantics, but that works well in many practical situations. For example, we can define a notion of call preservation which guarantees that all method calls are preserved by the refactoring [25]. In the presence of a type system, one can show that a refactoring preserves type correctness [26].

A more fundamental approach is to formally prove that refactorings preserve the full program semantics. For a language with a simple and formally defined semantics, such as the logic programming language *Prolog*, one can prove that some refactorings that improve the efficiency actually preserve the program semantics [27]. For more complex languages such as C++, where a formal semantics is extremely difficult to define, we typically have to put restrictions on the allowed language constructs or refactorings, and the applicability of a refactoring tool may be limited to a particular version of a particular compiler [28].

3.3 Assessing the Effect of Refactoring on Quality

For any piece of software, we can specify its external quality attributes (such as robustness, extensibility, reusability, performance). Refactorings can be classified according to which of these quality attributes they affect. This allows us to improve the quality of software by applying the relevant refactorings at the right places. To achieve this, each refactoring has to be analyzed according to its particular purpose and effect. Some refactorings remove code redundancy, some raise the level of abstraction, some enhance the reusability, and so on. This effect can be estimated to a certain extent by expressing the refactorings in terms of the internal quality attributes they affect (such as size, complexity, coupling, and cohesion).

An important software quality characteristic that can be affected by refactoring is *performance*. It is a common misconception that improving the program structure has a negative effect on the program performance. In the context of logic and functional programs, restructuring transformations typically have the goal of improving program performance while preserving the program semantics [27], [29]. In the context of object-oriented programs, Demeyer [30] investigated the effect of refactorings that replace conditional logic by polymorphism. He concludes that the program performance gets better after the refactoring because of the efficient way in which current compiler technology optimizes polymorphic methods.

To measure or estimate the impact of a refactoring on quality characteristics, many different techniques can be used. Examples include, but are not limited to, software metrics, empirical measurements, controlled experiments, and statistical techniques. Kataoka et al. propose coupling metrics as an evaluation method to determine the effect of refactoring on the maintainability of the program [31]. Tahvildari and Kontogiannis. encode design decisions as *soft-goal graphs* to guide the application of the transformation process [32]. 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 composite refactorings. Tahvildari and Kontogiannis use a catalogue of object-oriented metrics as an indicator to automatically detect where a particular refactoring can be applied to improve the software quality [33]. This is achieved by analyzing the impact of each refactoring on these object-oriented metrics.

3.4 Maintaining Consistency of Refactored Software

Typically, software development involves a wide range of software artifacts such as requirements specifications, software architectures, design models, source code, documentation, test suites, and so on. If we refactor any of these software artifacts, we need mechanisms to maintain their consistency. Since the activity of inconsistency management

is a research area in its own right [34], [35], [36], we will not treat it in detail here. We only discuss a few approaches that relate consistency maintenance to refactoring.

Bottoni et al. propose maintaining consistency between the program and design models by describing refactoring as coordinated graph transformation schemes [37]. These schemes have to be instantiated according to the specific code modification and applied to the design models affected by the change.

Within the same level of abstraction, there is also a need to maintain consistency. For example, if we want to refactor source code, we have to ensure that the corresponding unit tests are kept consistent [23]. Similarly, if we have different kinds of UML design models and any of these is being refactored, the others have to be kept consistent. Van Der Straeten et al. suggest to do this by means of logic rules [39].

Rajlich uses the technique of *change propagation* to cope with inconsistencies between different software artifacts [38]. This technique deals with the phenomenon that, when one part of a software is changed, dependent parts of the software may need to be changed as well.

4 Refactoring Techniques and Formalisms

A wide variety of formalisms and techniques have been proposed and used to deal with one or more refactoring activities. We discuss two such techniques in detail: the use of assertions (preconditions, postconditions, and invariants) and the use of graph transformation. Next, we discuss how formalisms can help us to guarantee program correctness and preservation in the context of refactoring. Finally, we provide an indicative, but inevitably incomplete, list of other useful techniques to support refactoring activities.

4.1 Invariants, Pre and Postconditions

A refactoring's definition often includes *invariants* that should remain satisfied and *pre and postconditions* that should hold before and after the refactoring has been applied. These constitute a lightweight and automatically verifiable means to ensure that (certain parts of) the behavior of the software is preserved by the refactoring. A concrete example of the use of *preconditions* was already presented for the refactoring **MoveMethod (print,ASCII-Doc,Printer)** in Section 3.2. A set of *postconditions* for the same refactoring would be: 1) The *print* method must be implemented in *Printer* after the refactoring, 2) the method signature of *print* does not exist in *ASCIIDoc* after the refactoring. An example of an *invariant* is the fact that classes *ASCIIDoc* and *Printer* are defined before and after the refactoring.

The use of preconditions and invariants has been suggested repeatedly in research literature as a way to address the problem of behavior preservation when restructuring or refactoring software artifacts. In the context of object-oriented database schemas (which are similar to UML class diagrams), Banerjee and Kim identified a set of *invariants* that preserve the behavior of these schemas [40]. Opdyke adopted this approach to object-oriented programs and additionally provided *preconditions* or enabling conditions for each refactoring [6]. He argued that these preconditions preserve the invariants. Roberts used first order predicate calculus to specify these preconditions in a formal way [41]. The notion of preconditions or applicability conditions is also available in the formal restructuring approach of Ward and Bennett, using the formal language WSL [42].

Preconditions may vary depending on the complexity of the language studied. More complex languages typically require more preconditions on the refactoring in order to preserve the invariants. Unfortunately, there are some practical problems with preconditions. One problem is that the static checking of some preconditions may require very expensive analysis or may even be impossible. Another problem is that the preconditions do not consider the size or structure of the program [6]. For example, C++ programs may perform integer arithmetic with the address of a variable in a class, which is problematic if the refactoring changes the physical ordering of the variables in that class.

A number of suggestions have been made to overcome the above problems with preconditions. Tip et al. suggest using type constraints to efficiently verify preconditions that depend on interprocedural relationships between variable types [26]. This is particularly useful for refactorings that are concerned with generalization. Roberts suggests augmenting refactorings with *postconditions* [41]. These postconditions are particularly useful for those invariants that rely on dynamic information that is difficult to express, or expensive to check statically, with preconditions. Postconditions can also be used to increase the efficiency of a refactoring tool. From a theoretical point of view, it can be shown that a set of postconditions can be translated into an equivalent set of preconditions [43]. Roberts provided an algorithm to perform this translation for sequences of program transformations. Ó Cinnéide and Nixon extended this algorithm to deal with iteration and conditional constructs [13].

4.2 Graph Transformation

Traditionally, refactorings are specified as parameterized program transformations along with a set of pre and postconditions that guarantee behavior preservation if satisfied [6], [44]. If we adopt this view, there is a direct correspondence between refactorings and *graph transformations*. Programs (or other kinds of software artifacts) can be expressed as *graphs*, refactorings correspond to *graph production rules*, the application of a refactoring corresponds to a *graph transformation*, refactoring pre and postconditions can be expressed as *application pre and postconditions* [43], [45]. Table 1 summarizes some formal properties of graph transformation that may be used to address important issues in refactoring.

Hence, it is not surprising that the theory of graph transformations has been used to provide more formal support for software refactoring. Mens et al. use the graph rewriting formalism to prove that refactorings preserve certain kinds of relationships (updates, accesses, and invocations) that can be inferred statically from the source code [25]. Bottoni et al. describe refactorings as coordinated graph transformation schemes in order to maintain consistency between a program and its design when any of them evolves by means of a refactoring [37]. Heckel [43] uses graph transformations to formally prove the claim

TABLE 1
Correspondence between Refactoring and Graph Transformation

Refactoring	Graph transformation
software artifact	graph
refactoring	graph production
composite refactoring	composition of graph productions
refactoring application	graph transformation
refactoring precondition	application precondition
refactoring postcondition	application postcondition
(in)dependence between refactorings in a sequence	parallel or sequential (in)dependence
conflict between refactorings applied in parallel to the same software artifact	confluency and critical pair analysis

(and corresponding algorithm) of Roberts [41] that any set of refactoring postconditions can be translated into an equivalent set of preconditions. Van Eetvelde and Janssens [46] propose a hierarchical graph transformation approach to be able to view and manipulate the software and its refactorings at different levels of detail.

The properties of sequential and parallel (in)dependence of graph transformations are also extremely suitable to reason about the dependence between refactorings. Two refactorings are independent if they can be applied in any order, i.e., the order in which they are applied does not affect the end result. This gives rise to a whole range of useful application scenarios.

One scenario is the serialization of refactorings that have been applied in parallel to the same software artifact [102]. During this serialization process, it is possible that conflicts arise because the refactorings make incompatible changes. To detect and resolve such conflicts, one can rely on existing results about parallelism and confluence [110] and critical pair analysis [111].

Analysis of sequential dependencies can also be used to reorder a given sequence of refactorings, for example, to normalize the sequence, to identify refactorings that annihilate each other's effect, to regroup subsequences into predefined composite refactorings, and so on.

When building composite refactorings, it is useful to determine which refactorings have to be applied sequentially and which refactorings are mutually independent [41]. For example, the composite refactoring shown in Fig. 3 of Section 2 consists of a sequence of six primitive refactorings, but there are only three sequential dependencies (represented by straight arrows): Each **MoveMethod** refactoring has to be preceded by a **Rename** refactoring. The order in which the three **(Rename,MoveMethod)** pairs have to be applied, however, is irrelevant. This is represented by dashed arrows. This means that, to increase the efficiency of the refactoring, one may decide to apply these three pairs of primitive refactorings in parallel.

4.3 Formalisms for Program Correctness and Preservation

Formal approaches are needed to guarantee that certain program properties remain invariant to a program transformation. We will make a distinction between the property of *program correctness* and the property of *preservation*.[2] Program correctness is the property that a program will work without errors. The preservation property of a program transformation guarantees that (some aspect of) the program behavior is preserved by the transformation.

Obviously, any program transformation should preserve the syntactic rules (or well-formedness rules) of the programming language. After the transformation, the software should still be syntactically correct. This can be checked by using a scanner and a parser. The semantics of the program should also remain correct, i.e., the program should not give rise to runtime errors. Unfortunately, the correctness property is in general undecidable. Gupta et al. showed that we cannot prove, for an arbitrary running program and an arbitrary update to it, that the update is *valid* in the sense that it will eventually result in a reachable program state of the newly added program code [63]. Because of the undecidability of this property, we can only take a conservative approach. For example, if we only consider restructurings of the same algorithm (as opposed to changes to program functionality), a syntactic analysis of the old and new program code can identify program points that preserve update validity.

The *preservation* property can either be checked *statically* or *dynamically*. The checking of refactoring preconditions [6], [41] can be considered as a static approach. However, the preconditions that are expressed in first-order predicate logic are only a conservative approximation and, hence, rule out many legal refactorings. Mens et al. suggest other notions of behavior preservation that can be checked statically and show how this can be realized using a graph transformation formalism [25]. *Access preservation* means that all variable accesses should be preserved by the refactoring. *Update preservation* means that all variable updates should be preserved by the refactoring. *Call preservation* means that all method calls should be preserved by the refactoring. Another static way to check preservation of program behavior to a certain extent is by means of *type checking*: All typed software entities should still have the same type after the refactoring. This constraint can be loosened by allowing a type to be replaced by a subtype (in the presence of a subtyping mechanism).

To be able to check that more aspects of the program behavior are preserved, one needs to remove the restrictions imposed by static conservative approximations by taking more dynamic information into account. However, one should be aware that even then it is impossible to guarantee full behavior preservation in its generality. Moore and Bennett propose a more dynamic notion of call preservation, where the transformation guarantees that the same messages in a class will be sent in the same order [64].

2. This distinction is not made in the domain of program transformation for functional languages [29]. In this domain, the term *correctness* is used to indicate that a program transformation preserves the extensional meaning of programs. We will not use correctness in this sense because it leads to confusion with the more widely accepted definition of *program correctness*.

Mortimer presents an approach for restructuring program data types to identify, group, and/or restrict their possible values, while at the same time preserving the dynamic behavior of the software [65].

Bergstein defines a set of primitive refactorings that reorganize classes and methods across a class hierarchy in such a way that the same set of objects can still be instantiated after the refactoring [66]. This specific notion of behavior preservation is called *object equivalence*. Later, this work was extended to a special kind of graph transformations that have the property of being *language-preserving* [67]: The set of all acceptable program inputs before and after the transformations must be the same. As such, it provides a framework for refactoring with a theoretical basis in formal language theory. Hwang et al. propose another extension by using a notion of *object semiequivalence* between class hierarchies, which is a direct extension of *object equivalence* to take composite objects into account [68]. They propose a set of primitive transformations that preserves object semiequivalence, is complete (i.e., any transformation can be expressed as a sequence of the primitive transformations), and is minimal (i.e., no smaller set of primitive transformations can be found).

Hürsch and Seiter guarantee preservation by uncoupling the class structure from the object behavior, using class graphs and propagation patterns [69]. This allows them to define a restricted set of refactorings that change the structure without affecting the dynamic behavior.

Ward and Bennett provide a formal imperative language WSL and associated tool that comes with a library of program transformations that have been proven to preserve the dynamic behavior [42]. A disadvantage of the approach is that, if we want to apply it to some "informal" programming language, we first have to write a translator of this language to and from WSL and we cannot use formal methods to prove that this translation is correct.

For logic programming languages, there are several notions of program semantics: *least Herbrand model* semantics [70], *set of computed answer substitutions* semantics [71], *sequence of computed answer substitutions* semantics [72]. Transformation rules (such as Unfold/Fold) can be applied to restructure logic programs with the aim of improving efficiency. For these rules, it can be theoretically shown that they preserve program equivalence under the above notions of semantics, given some suitable restrictions [27]. Similar results have been obtained for functional programming languages [29].

4.4 Other Useful Techniques and Formalisms

Many other techniques and formalisms have been proposed and used to support restructuring and refactoring activities. We provide a brief overview below and refer to the literature for more detailed information.

Program slicing is a technique which extracts all statements that may possibly affect a certain set of variables in a program [47]. This technique, based on system dependence graphs, can be used to guarantee that a restructuring preserves some selected behavior of interest. For example, it has been proposed to deal with a specific kind of program restructurings: function or procedure extraction [48], [49]. A similar, but less formal approach is presented in [50], 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. Since program slicing can be applied to object-oriented programs too [51], it is likely that this technique can be used to deal with program refactorings as well.

Sands developed a *formal improvement theory* to be able to transform functional programs to improve their efficiency [29]. The transformations are guaranteed to be "meaning preserving," which boils down to the preservation of (global) equivalence. This is a nontrivial property since a sequence of transformations that preserve the local equivalence does not necessarily preserve (global) equivalence. Nevertheless, Sands provides a condition to achieve such global equivalence on recursive programs in higher-order functional languages, including lazy data structures.

Formal concept analysis provides a conceptual tool for the analysis of data [52]. The formalism uses lattice theory to provide a way to group and discuss objects based upon their common attributes. Snelting and Tip use concept analysis to refactor object-oriented class hierarchies, based on the "usage" of this hierarchy by a set of software systems [53]. The result is guaranteed to be behaviorally equivalent to the original hierarchy. Tonella uses the same technique to restructure software modules [54]. van Deursen and Kuipers use concept analysis to semiautomatically restructure legacy data structures into object-oriented software by identifying object structures [55].

Program refinement encompasses a collection of formal techniques to transform a program specification into an executable program in a stepwise fashion. Philipps and Rumpe [56] suggest using *refinement* approaches [57], [58] as a way to formally deal with the notions of behavior, behavioral equivalence, and behavior preservation. Ward and Bennett illustrate how to apply refinement in the context of program restructuring [42]. They define a formally defined imperative language WSL that provides three kinds of program transformations: (behavior-preserving) restructurings, (behavior-extending) refinements, and their opposite: abstractions. *Refinement of dataflow architectures* uses a clearly defined notion of observable behavior that allows us to precisely define what preservation and refinement of behavior means [59].

Software metrics can be used to deal with refactorings as well. Numerical measures can be used before applying a refactoring, to measure the (internal or external) quality of software, or after the refactoring, to measure improvements of the quality. Demeyer et al. propose using change metrics to detect refactorings between two successive software releases [60]. Simon et al. use distance-based cohesion metrics to detect where in a given piece of software there is a need for refactoring [20]. Kataoka et al. use coupling metrics to evaluate the effect of refactoring on maintainability [31]. Coleman et al. use a polynomial of multiple measures to define a maintainability index by which the effect of refactoring can be evaluated [61].

Software visualization is another technique that can help with software refactoring. Griswold et al. propose to use star diagrams for this purpose [62]. Ducasse et al. propose *DupLoc*, a graphical tool for detecting code duplication [16].

TABLE 2
Restructuring Support in Different Programming Languages

language paradigm	programming language
imperative	Fortran [73], [74]
	Cobol [75], [76]
	C [77], [78]
imperative formal	WSL [42]
functional	Scheme [79]
	Lisp [80]
	Haskell [82], [83]
logic [84], [85]	Prolog [72], [27]
class-based OO	Smalltalk [44]
	Java [7], [86], [13], [87]
	C++ [88], [89], [90], [91], [28]
prototype-based OO	Self [64]
aspect-oriented	AspectJ [92]

Simon et al. use a generic visualization framework based on static structure analysis and a cohesion-based distance metric to identify pathological situations that may be improved by applying refactorings [20]. Van Emden and Moonen detect bad smells in Java code and visualize them using the Rigi software visualization tool [18]. Lanza and Ducasse propose the *evolution matrix*, a lightweight combination of software metrics and visualization to identify patterns of evolution within object-oriented software [21]. Some of these patterns can be used to reveal where refactorings have been applied in the evolution history, or to identify places in the code that are in need of refactoring.

Dynamic program analysis is a useful technique when not all desired preconditions of a refactoring can be statically computed in a reasonable amount of time or with a reasonable computation effort. For these situations, Roberts suggests using dynamic program analysis to verify the preconditions of a refactoring or to deal with program invariants that cannot be specified or checked statically [41]. Runtime program information may also be used to identify where refactorings might be desirable. Kataoka et al. do this by dynamically inferring program invariants using the *Daikon* tool [14].

5 Types of Software Artifacts

Although contemporary IDEs limit support for refactoring to the source code only, refactoring can be applied to any type of software artifact. For example, it is possible and useful to refactor design models, database schemas, software architectures, and software requirements. Refactoring of these kinds of software artifacts rids the developer of many implementation-specific details and raises the expressive power of the changes that are made. On the other hand, applying refactorings to different types of software artifacts introduces the need to keep them all in sync.

5.1 Programs

Support for program restructuring and refactoring has been provided in a variety of different programming languages and programming paradigms. This is summarized in Table 2.

Programs that are not written in an object-oriented language are more difficult to restructure because data flow and control flow are tightly interwoven. Because of this, restructurings are typically limited to the level of a function or a block of code [79], [48], [49], [50]. On the other hand, the very nature of object-oriented principles makes some seemingly straightforward restructurings surprisingly hard to implement. The encountered difficulties typically have to do with the inheritance mechanism and, more in particular, the notions of dynamic binding, interfaces, subtyping, overriding, and polymorphism. For example, Najjar et al. mention the inconsistent use of super calls and the lack of scope when using interfaces as concrete problems in the context of program refactoring [87].

Also note that the more complex a language, the more difficult to automate the refactoring proces is. For example, a C or C++ program transformation tool cannot deal with preprocessor directives because they are not part of the actual language syntax [10], [28], [77]. This problem is tackled by *XRefactory* [78], a refactoring browser that allows one to refactor C programs in the presence of a C preprocessor.

5.2 Designs

A recent research trend is to deal with refactoring at design level, for example in the form of UML models [93], [37], [94]. Boger et al. developed a refactoring browser integrated with a UML modeling tool [95]. It supports refactoring of class diagrams, statechart diagrams, and activity diagrams. For each of these diagrams, the user can apply refactorings that cannot easily or naturally be expressed in other diagrams or in the source code. Van Gorp et al. propose a UML extension to express the pre and postconditions of source code refactorings using OCL [96]. The proposed extension allows an OCL empowered CASE tool to verify nontrivial pre and postconditions, to compose sequences of refactorings, and to use the OCL query engine to detect bad code smells. Such an approach is desirable as a way to refactor designs independent of the underlying programming language.

Design patterns provide a means to describe the program structure at a high level of abstraction [12]. Often, refactorings are used to introduce new design pattern instances into the software [88], [90], [91]. We already illustrated this in our running example of Section 2, where refactorings were used to introduce a *Visitor* design pattern. Design patterns also impose constraints on the software structure, which may limit applicability of certain refactorings. To detect this, Mens and Tourwé resort to logic reasoning [109]. Jahnke and Zündorf use graph transformation techniques to restructure/replace occurrences of poor design patterns in a legacy program by good design patterns [97].

Object-oriented database schemas can be seen as the predecessor of UML class diagrams. Because their main focus is on how data should be structured, they are an ideal candidate for refactoring. In fact, the research area of object-oriented software refactoring originates in the research on how to restructure object-oriented database schemas [40], [66], [69].

To deal with refactoring of *software architectures*, Philipps and Rumpe propose a promising approach where refactoring rules are based directly on the graphical representation of a system architecture [59]. These rules preserve the behavior specified by the causal relationship between the components. A more pragmatic approach is presented by Tokuda and Batory [28]: Architectural changes to two software systems are made by performing a sequence of primitive refactorings (81 refactorings in a first case study and 800 refactorings in a second case study).

5.3 Software Requirements

Restructuring can also be applied at the level of requirements specifications. For example, Russo et al. suggest restructuring natural language requirements specifications by decomposing them into a structure of *viewpoints* [98]. Each viewpoint encapsulates partial requirements of some system components, and interactions between these viewpoints are made explicit. This restructuring approach increases requirements understanding, and facilitates detecting inconsistencies and managing requirements evolution.

6 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.[3] In this section, we explore the different characteristics that affect the usability of a tool. More specifically, we discuss the notions of automation, reliability, configurability, coverage, and scalability of refactoring tools.

6.1 Automation

The degree of automation of a refactoring tool varies depending on which of the refactoring activities of Section 3 are supported by the tool, as well as the extent to which support for each of these activities is automated.

For example, contemporary IDEs often include a refactoring browser that supports a *semiautomatic approach* to refactoring. While it remains the task of the developer to identify which part of the software needs to be refactored and to select the most appropriate refactoring to apply, the actual application of the refactoring is automated. As indicated by Tokuda and Batory [28], a semiautomatic approach can drastically increase the productivity (in terms of coding and debugging time) when compared to refactoring by hand. Based on two nontrivial case studies, they estimate this to be a factor of 10 or more. Similarly, one can expect developer productivity to improve after the software has been refactored because the software generally is more understandable, maintainable, and evolvable. Another main advantage of refactoring tools from the viewpoint of the developer is that their behavior-preserving nature significantly reduces the need for debugging and testing, two activities that are known to be very time-consuming and labor intensive.

As an alternative to this semiautomatic approach, some researchers demonstrated the feasibility of *fully automated refactoring*. For example, *Guru* is a fully automated tool for refactoring inheritance hierarchies and refactoring methods in *SELF* programs [64]. Another automatic refactoring approach is proposed by Casais [99]. Optimization techniques as performed by compilers can also be considered as fully automated refactoring techniques. While these optimizing transformations are completely transparent to the user, their goal is to improve the performance of the program, yet preserve its behavior [100].

In many cases, automating refactoring activities gives rise to new activities or opportunities that were not possible without automation. For example, the added benefit of automatically applying refactorings is that its application can be easily undone to allow the software to return to its original state if it turns out the refactoring did not have the desired effect.

Compared to partial automation, fully automated refactoring and restructuring tools exhibit the disadvantage of doing too much work in the sense that certain parts of the refactored software become more difficult to understand than before. This is confirmed by Callis who identified some shortcomings of automatic program restructuring tools [101]. He pointed out that interactive restructuring tools do not have many of these shortcomings. On the other hand, the problem with interactive restructuring tools is that they involve a lot of human interaction when faced with large software, making it a time-consuming activity. Despite this problem, semiautomatic refactoring remains the most useful approach in practice, except in specific situations such as compiler optimization. The main reason for this is that a significant part of the knowledge required to perform the refactoring cannot be extracted from the software, but remains implicit in the developer's head.

6.2 Reliability

The reliability of a refactoring tool mainly depends on the ability to guarantee that its provided refactoring transformations are truly behavior preserving. As we have seen in Section 4.3, it is only possible to guarantee this in very specific cases (e.g., for simple languages, for a limited number of refactorings, given a clearly defined notion of semantics). Because of these restrictions, most tools check the refactoring preconditions before applying it and perform tests afterward.

In the absence of a full guarantee of behavior preservation, it is essential that a refactoring tool provides an *undo mechanism* to make undesired changes undone [41].

6.3 Configurability and Openness

There is a tendency to integrate refactoring tools directly into industrial strength IDEs. This is typically achieved using the built-in extensibility mechanisms of these tools (e.g., plug-ins, APIs, or wizards). Unfortunately, these extensibility mechanisms are often inadequate for the purpose of configuring the tools with user-specific or domain-specific information.

There are a variety of ways in which a user (or a group or users) should be able to configure a refactoring tool for a particular usage:

- by adding new or removing or modifying existing refactorings and bad smell specifications,

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

- by changing the way bad smells and refactorings are linked [17],
- by defining composite refactorings from primitive ones [13].

Having an open configurable tool is a necessity to allow one to configure the above information according to the needs in a user-friendly way. To make it easier for the user to specify and modify refactorings, Leitão suggests to use a pattern language (i.e., a collection of related patterns together with rules explaining how to apply them) to express refactorings [80]. Muñoz provides user-configurable threshold values to specify under which conditions a bad smell needs to be detected [81].

6.4 Coverage

As mentioned in Section 3, there is a wide range of refactoring activities that can be covered by a tool. An ideal refactoring tool should be as complete as possible, i.e., it should cover most of these activities. Unfortunately, most commercial refactoring tools only provide support for automatically applying refactorings, whereas the other activities of the refactoring process are neglected.

6.5 Scalability

Contemporary software development tools only support primitive refactorings. As illustrated in the example of Section 2, refactoring even the simplest design already requires applying a large number of primitive refactorings. To increase the scalability and performance of a refactoring tool, frequently used sequences of primitive refactorings should be combined into composite refactorings.

The use of composite refactorings has several advantages. First of all, 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. Second, composite refactorings 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 [41], [102]. A third advantage of composite refactorings is that they can weaken the behavior preservation requirements of its primitive constituents. The primitive refactorings in a sequence do not have to be behavior preserving as long as the net effect of their composition *is* behavior preserving. This interesting idea is referred to as *transactional refactoring* by Tokuda and Batory [28]. As an example, they show that the refactoring **DelegateMethod AcrossObjectBoundary** is a sequence of two primitive refactorings, **MoveMethodAcrossObjectBoundary** (which removes the method entirely from its original class) and **CreateMethod Accessor** (which reintroduces the method to the original class and delegates its execution to the moved method). While the net result of applying both refactorings in sequence is behavior preserving, the primitive refactorings are not. If clients of the original class reference the target method, the enabling conditions of the move method refactoring will prevent the method from being moved.

Similar to the use of composite refactorings, Ó Cinnéide and Nixon [103] propose using refactorings to introduce design patterns by first splitting up the design pattern into a sequence of *minipatterns* and then applying a sequence of corresponding *minitransformations* to introduce these minipatterns. Each minitransformation is expressed as a composition of primitive refactorings, using sequencing and iteration constructs [13].

Tokuda and Batory [28] also tested the scalability of refactorings on two nontrivial evolving software systems written in C++, a mainstream object-oriented language. The first system consisted of 11K lines of code and the code was refactored by executing 81 refactorings, modifying in total 486 lines of source code. The second system was modified by executing about 800 refactorings, resulting in 14K lines of code change. Despite these large numbers, the changes boiled down to eight (respectively, 20) conceptual transformation steps that had to be carried out sequentially.

A final aspect of scalability has to do with change propagation [38]. Because changes tend to propagate throughout the software, the application of a certain refactoring may suggest or even require other refactorings to be applied as well in order to achieve the goal intended by the original refactoring. Tourwé and Mens refer to this idea as *cascaded refactorings* [17] and provide tool support for it by means of logic rules that are implemented on top of an existing object-oriented IDE in Smalltalk. Like with composite refactorings, the main idea is to identify and specify sequential dependencies between refactorings.

6.6 Language Independence

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 behavior.

Lämmel [104] introduces the notion of *generic program refactoring* as an initial proposal toward a language-parametric framework that can be instantiated for a variety of different languages such as Java, Prolog, Haskell, and XML. The framework is implemented in the functional programming language Haskell and provides hot spots for the language-specific ingredients for refactoring. The underlying idea is that functional strategies are used to specify reusable parse tree traversal schemes.

Meta modeling is a useful technique to make refactoring less dependent on the implementation language. Tichelaar et al. [105] and Mens et al. [25] both propose a metamodel-based approach for language-independent refactoring.

Ward and Bennett suggest achieving language independence by translating code written in some language to the intermediate formal language WSL, where the code can be restructured, refined, and abstracted [42]. After transformation, the modified code can be translated again to the same language or to another language. This use of an intermediate language representation makes the approach language independent. For any new language, one only needs to write an automatic translator to or from WSL.

Language independence is not only important between different languages, but also if we have a language that is an extension of another one. For example, although *AspectJ* is an extension of *Java*, existing *Java* refactorings are not always valid in *AspectJ* as they do not consider the impact the modifications made by the refactoring have on the *AspectJ* code. Conversely, there are also new refactorings

that are needed to deal with the features in *AspectJ* that do not occur in *Java* [92].

7 Process Support

Refactoring is an important activity in the software development process. In this section, we discuss how refactoring fits into the processes of software reengineering, agile software development, and framework-based software development.

7.1 Software Reengineering

Refactoring naturally fits in the process of *software reengineering* [9], the aim of which is to restructure legacy software. In this process, refactoring is only the last stage and addresses the technical issue of (semi)automatically modifying the software to implement a new solution. The more important problems, however, are to determine which parts of the legacy software should be converted and exactly how to convert them, taking into account the constraints that reengineers are facing and the potential impact of the suggested changes. Even trying to understand what the legacy software does in the first place is already a significant problem.

Unlike forward engineering that is supported by a variety of processes such as the spiral and waterfall models of software development, no established process for reengineering is available. Due to the absence of such a process, *reengineering patterns* are the next best thing [9]. They codify and record best practice knowledge about modifying legacy software. They provide generic solutions based on recurring reengineering problems that were encountered in real-life situations. In this sense, they provide stable units of expertise that can be consulted in any reengineering effort.

Refactoring also seems to fit well into a *model-driven reengineering* process. One of the goals of model-driven architectures (MDA) is to facilitate platform migration by code generation from abstract models [106]. 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.

7.2 Agile Software Development

Typically, major reengineering efforts are carried out only when the software has already degraded so much that it has turned into legacy code. In contrast, the agile software development community, with *eXtreme Programming* (XP) as its main proponent [107], suggests supporting a culture of *continuous reengineering*. They propose a process where one develops and reengineers software in small iterations: You develop a little (to implement the desired behavior), reengineer a little (to improve the structure), develop a little more, and so on. Unfortunately, these short iterative development cycles do not seem to fit very well in a more classical software development process.

Refactoring is one of the cornerstones in the XP process. Many object-oriented IDEs provide considerable support for XP, using a combination of refactoring support and unit testing, two core activities in XP. In [23], [24], the relationship between testing and refactoring is explored in more detail to address the practical problem that refactorings often invalidate tests. Whenever this occurs, Pipka suggests modifying the tests first and apply the refactoring afterward to guarantee that we can still use the tests for program verification [23]. Van Deursen et al. [108] show that refactoring of test code is different from refactoring production code in two ways: 1) There is a distinct set of bad smells involved and 2) improving test code involves additional test-specific refactorings.

7.3 Framework-Based or Product Line Software Development

Parallel application of refactorings often leads to unexpected *evolution conflicts* [102]. This issue is particularly relevant for object-oriented *application frameworks*, where the framework may be instantiated into many different software systems while the framework itself is also subject to evolution. This implies that a refactoring of the framework may lead to evolution conflicts in each of its instantiations [102]. The same issue holds for *software families* or *product lines*.

As an example, consider what happens when a developer extends the design of Fig. 1 by implementing a spell checking algorithm for documents. This requires him to define *checkSpelling* methods in all classes of the *Document* class hierarchy, as well as a *SpellChecker* helper class. At the same time, and independent of the first change, another developer decides to refactor the design to introduce the Visitor design pattern, as depicted in Fig. 2. Both changes need to be combined into a single design that includes both the Visitor design pattern and the spell checking algorithm. We cannot simply merge both evolutions as this would lead to an inconsistent design: The print and preview algorithms would use the Visitor design pattern, whereas the spell checking algorithm does not. Tourwé and Mens [109] propose logic metaprogramming as a way to detect and resolve such problems.

8 Conclusions

This paper provides an extensive overview of existing research in the domain of software refactoring and software restructuring. We classified this research according to five different criteria: the refactoring activities that are supported, the specific techniques and formalisms that are used to support these activities, the kinds of software artifacts that are being refactored, the important characteristics that need to be taken into account when building refactoring tools, and the effect of refactoring on the software development process. In each of these categories, we indicated important open issues that remain to be solved. In general, we identified a need for formalisms, processes, methods, and tools that address refactoring in a more consistent, generic, scalable, and flexible way. Although

commercial refactoring tools have begun to proliferate, research into software restructuring and refactoring continues to be very active and remains essential to revealing and addressing the shortcomings of these tools.

ACKNOWLEDGMENTS

This research was funded by the FWO Project G.0452.03 "A formal foundation for software refactoring" and was 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 Jean-Marc Jézéquel and the anonymous reviewers for their excellent reviews that turned this paper into a far better paper than it would have been otherwise.

REFERENCES

[1] D.M. Coleman, D. Ash, B. Lowther, and P.W. Oman, "Using Metrics to Evaluate Software System Maintainability," *Computer,* vol. 27, no. 8, pp. 44-49, Aug. 1994.

[2] T. Guimaraes, "Managing Application Program Maintenance Expenditure," *Comm. ACM,* vol. 26, no. 10, pp. 739-746, 1983.

[3] B.P. Lientz and E.B. Swanson, *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, 1980.

[4] R.L. Glass, "Maintenance: Less Is Not More," *IEEE Software,* July/Aug. 1998.

[5] R.S. Arnold, "An Introduction to Software Restructuring," *Tutorial on Software Restructuring,* R.S. Arnold, ed., 1986.

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

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

[8] E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software,* vol. 7, no. 1, pp. 13-17, 1990.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns.* Morgan Kaufmann and DPunkt, 2002.

[10] R. Fanta and V. Rajlich, "Reengineering Object-Oriented Code," *Proc. Int'l Conf. Software Maintenance,* pp. 238-246, 1998.

[11] R. Fanta and V. Rajlich, "Restructuring Legacy C Code into C++," *Proc. Int'l Conf. Software Maintenance,* pp. 77-85, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems.* Addison-Wesley, 1994.

[13] M. Ó Cinnéide and P. Nixon, "Composite Refactorings for Java Programs," technical report, Dept. of Computer Science, Univ. College Dublin, 2000.

[14] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin, "Automated Support for Program Refactoring Using Invariants," *Proc. Int'l Conf. Software Maintenance,* pp. 736-743, 2001.

[15] M. Balazinska, E. Merlo, M. Dagenais, and B. Lagüe, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring," *Proc. Working Conf. Reverse Eng.,* pp. 98-107, 2000.

[16] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *Proc. Int'l Conf. Software Maintenance,* pp. 109-118, 1999.

[17] T. Tourwé and T. Mens, "Identifying Refactoring Opportunities Using Logic Meta Programming," *Proc. European Conf. Software Maintenance and Reeng.,* pp. 91-100, 2003.

[18] E. van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," *Proc. Working Conf. Reverse Eng.,* pp. 97-108, 2002.

[19] T. Dudziak and J. Wloka, "Tool-Supported Discovery and Refactoring of Structural Weaknesses in Code," MS thesis, Faculty of Computer Science, Technical Univ. of Berlin, Feb. 2002.

[20] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics Based Refactoring," *Proc. European Conf. Software Maintenance and Reeng.,* pp. 30-38, 2001.

[21] M. Lanza and S. Ducasse, "Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics," *Proc. Langages et Modèles à Objets,* pp. 135-149, Aug. 2002.

[22] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns.* Sun Microsystems Press, 2001.

[23] J.U. Pipka, "Refactoring in a 'Test First'-World," *Proc. Third Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng.,* 2002.

[24] A. van Deursen and L. Moonen, "The Video Store Revisited —Thoughts on Refactoring and Testing," *Proc. Third Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng.,* pp. 71-76, 2002.

[25] T. Mens, S. Demeyer, and D. Janssens, "Formalising Behavior Preserving Program Transformations," *Graph Transformation,* 2002.

[26] F. Tip, A. Kiezun, and D. Bäumer, "Refactoring for Generalization Using Type Constraints," *Proc. SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 13-26, 2003.

[27] M. Proietti and A. Pettorossi, "Semantics Preserving Transformation Rules for Prolog," *Proc. Symp. Partial Evaluation and Semantics-Based Program Evaluation,* vol. 26, no. 9, pp. 274-284, May 1991.

[28] L. Tokuda and D.S. Batory, "Evolving Object-Oriented Designs with Refactorings," *Automated Software Eng.,* vol. 8, no. 1, pp. 89-120, 2001.

[29] D. Sands, "Total Correctness by Local Improvement in the Transformation of Functional Programs," *Trans. Programming Languages and Systems,* vol. 18, no. 2, pp. 175-234, Mar. 1996.

[30] S. Demeyer, "Maintainability versus Performance: What's the Effect of Introducing Polymorphism?" technical report, Lab. on Reeng., Universiteit Antwerpen, Belgium, 2002.

[31] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring," *Proc. Int'l Conf. Software Maintenance,* pp. 576-585, Oct. 2002.

[32] L. Tahvildari and K. Kontogiannis, "A Methodology for Developing Transformations Using the Maintainability Soft-Goal Graph," *Proc. Working Conf. Reverse Eng.,* pp. 77-86, Oct. 2002.

[33] L. Tahvildari and K. Kontogiannis, "A Metric-Based Approach to Enhance Design Quality through Meta-Pattern Transformations," *Proc. European Conf. Software Maintenance and Reeng.,* pp. 183-192, 2003.

[34] J. Grundy, J. Hosking, and W. Mugridge, "Inconsistency Management for Multiple-View Software Development Environments," *IEEE Trans. Software Eng.,* vol. 24, no. 11, pp. 960-981, Nov. 1998.

[35] B. Nuseibeh, S. Easterbrook, and A. Russo, "Leveraging Inconsistency in Software Development," *Computer,* vol. 33, no. 4, pp. 24-29, Apr. 2000.

[36] G. Spanoudakis and A. Zisman, "Inconsistency Management in Software Engineering: Survey and Open Research Issues," *Handbook of Software Eng. and Knowledge Eng.,* vol. 1, pp. 24-29, 2001.

[37] P. Bottoni, F. Parisi-Presicce, and G. Taentzer, "Coordinated Distributed Diagram Transformation for Software Evolution," *Electronic Notes in Theoretical Computer Science,* vol. 72, no. 4, 2002.

[38] V. Rajlich, "A Model for Change Propagation Based on Graph Rewriting," *Proc. Int'l Conf. Software Maintenance,* pp. 84-91, 1997.

[39] R. Van Der Straeten, J. Simmonds, T. Mens, and V. Jonckers, "Using Description Logic to Maintain Consistency between UML Models," *Proc. Unified Modeling Language Conf. 2003,* 2003.

[40] J. Banerjee and W. Kim, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," *Proc. SIGMOD Conf.,* 1987.

[41] D. Roberts, "Practical Analysis for Refactoring," PhD thesis, Univ. of Illinois at Urbana-Champaign, 1999.

[42] M.P. Ward and K.H. Bennett, "Formal Methods to Aid the Evolution of Software," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 5, no. 1, pp. 25-47, 1995.

[43] R. Heckel, "Algebraic Graph Transformations with Application Conditions," MS thesis, TU Berlin, 1995.

[44] D. Roberts, J. Brant, and R.E. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems,* vol. 3, no. 4, pp. 253-263, 1997.

[45] A. Habel, R. Heckel, and G. Täntzer, "Graph Grammars with Negative Application Conditions," *Fundamenta Informaticae,* vol. 26, nos. 3 and 4, pp. 287-313, June 1996.

[46] N. Van Eetvelde and D. Janssens, "A Hierarchical Program Representation for Refactoring," *Proc. UniGra'03 Workshop,* 2003.

[47] M. Weiser, "Program Slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352-357, 1984.
[48] F. Lanubile and G. Visaggio, "Extracting Reusable Functions by Flow Graph-Based Program Slicing," *IEEE Trans. Software Eng.*, vol. 23, no. 4, pp. 246-258, Apr. 1997.
[49] A. Lakhotia and J.-C. Deprez, "Restructuring Programs by Tucking Statements into Functions," *Information and Software Technology*, special issue on program slicing, vol. 40, pp. 677-689, 1998.
[50] R. Komondoor and S. Horwitz, "Semantics-Preserving Procedure Extraction," technical report, Computer Sciences Dept., Univ. of Wisconsin-Madison, 2000.
[51] L. Larsen and M.J. Harrold, "Slicing Object-Oriented Software," *Proc. Int'l Conf. Software Eng.*, pp. 495-505, Mar. 1996.
[52] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations.* Springer-Verlag, 1999.
[53] G. Snelting and F. Tip, "Reengineering Class Hierarchies Using Concept Analysis," *Proc. Foundations of Software Eng.*, pp. 99-110, 1998.
[54] P. Tonella, "Concept Analysis for Module Restructuring," *Trans. Software Eng.*, vol. 27, no. 4, pp. 351-363, Apr. 2001.
[55] A. van Deursen and T. Kuipers, "Identifying Objects Using Cluster and Concept Analysis," *Proc. 21st Int'l Conf. Software Eng.*, pp. 246-255, 1999.
[56] J. Philipps and B. Rumpe, "Roots of Refactoring," *Proc. 10th OOPSLA Workshop Behavioral Semantics*, 2001.
[57] N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM*, vol. 14, pp. 221-227, 1971.
[58] R.-J. Back, "Correctness Preserving Program Refinements," technical report, Math. Centre Tracts #131, Mathematisch Centrum Amsterdam, 1980.
[59] J. Philipps and B. Rumpe, "Refinement of Information Flow Architectures," *Proc. Int'l Conf. Formal Eng. Methods*, 1997.
[60] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding Refactorings via Change Metrics," *Proc. Object-Oriented Programming, Systems, Languages, Applications Conf. 2000*, vol. 35, no. 10, pp. 166-177, Oct. 2000.
[61] D. Coleman, P. Arnold, S. Bdoff, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.
[62] W.G. Griswold, M.I. Chen, R.W. Bowdidge, and J.D. Morgenthaler, "Tool Support for Planning the Restructuring of Data Abstractions in Large Systems," *Proc. SIGSOFT Symp. Foundations of Software Eng.*, Oct. 1996.
[63] D. Gupta, P. Jalote, and G. Barua, "A Formal Framework for On-Line Software Version Change," *IEEE Trans. Software Eng.*, vol. 22, no. 2, pp. 120-131, Feb. 1996.
[64] I. Moore, "Automatic Inheritance Hierarchy Restructuring and Method Refactoring," *Proc. Object-Oriented Programming, Systems, Languages, Applications Conf.*, pp. 235-250, 1996.
[65] R.E. Mortimer and K.H. Bennett, "Maintenance and Abstraction of Program Data Using Formal Transformations," *Proc. Int'l Conf. Software Maintenance*, pp. 301-311, 1996.
[66] P.L. Bergstein, "Object-Preserving Class Transformations," *SIGPLAN Notices*, vol. 26, no. 11, pp. 299-313, Nov. 1991.
[67] P.L. Bergstein, "Maintenance of Object-Oriented Systems during Structural Evolution," *Theory and Practice of Object Systems*, vol. 3, no. 3, pp. 185-212, 1991.
[68] S.H. Hwang, Y. Tsujino, and N. Tokura, "A Reorganization Framework of the Object-Oriented Class Hierarchy," *Proc. Asia Pacific Conf. Software Eng.*, pp. 117-126, 1995.
[69] W.L. Hürsch and L.M. Seiter, "Automating the Evolution of Object-Oriented Systems," *Proc. Symp. Object Technology for Advanced Software*, pp. 2-21, 1996.
[70] H. Tamaki and T. Sato, "Unfold/Fold Transformation of Logic Programs," *Proc. Int'l Conf. Logic Programming*, pp. 127-138, 1984.
[71] T. Kawamura and T. Kanamori, "Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation," *Proc. Int'l Conf. Fifth Generation Computer Systems*, pp. 413-422, 1988.
[72] N. Jones and A. Mycroft, "Stepwise Development of Operational and Denotational Semantics for Prolog," *Proc. Int'l Symp. Logic Programming*, pp. 289-298, 1984.
[73] F. Bodin, "Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools," *Proc. Conf. Object-Oriented Numerics*, 1994.
[74] C.T.H. Everaars, F. Arbab, and F.J. Burger, "Restructuring Sequential Fortran Code into a Parallel/Distributed Application," *Proc. Int'l Conf. Software Maintenance*, pp. 13-22, 1996.
[75] J.C. Miller and B.M. Strauss, "Implications of Automatic Restructuring of Cobol," *SIGPLAN Notices*, vol. 22, pp. 76-82, June 1987.
[76] T.J. Harmer, P.J. McParland, and J.M. Boyle, "Using Knowledge-Based Transformations to Reverse-Engineer COBOL Programs," *Proc. Conf. Knowledge Based Software Eng.*, pp. 114-123, 1996.
[77] A. Garrido and R. Johnson, "Challenges of Refactoring C Programs," *Proc. Int'l Workshop Principles of Software Evolution*, 2002.
[78] M. Vittek, "Refactoring Browser with Preprocessor," *Proc. European Conf. Software Maintenance and Reeng.*, pp. 101-110, 2003.
[79] W.G. Griswold and D. Notkin, "Automated Assistance for Program Restructuring," *Trans. Software Eng. and Methodology*, vol. 2, no. 3, pp. 228-269, July 1993.
[80] A.M. Leitão, "A Formal Pattern Language for Refactoring of Lisp Programs," *Proc. European Conf. Software Maintenance and Reeng.*, pp. 186-192, 2002.
[81] F. Muñoz, "A Logic Metaprogramming Framework for Supporting the Refactoring Process," MS thesis, Vrije Universiteit Brussel, Sept. 2003.
[82] R. Lämmel, "Reuse by Program Transformation," *Functional Programming Trends*, 1999.
[83] H. Li, S. Thompson, and C. Reinke, "Tool Support for Refactoring Functional Programs," *Proc. SIGPLAN Workshop Haskell*, pp. 27-38, 2003.
[84] A. Pettorossi and M. Proietti, "Rules and Strategies for Transforming Functional and Logic Programs," *ACM Computing Surveys*, vol. 28, no. 2, pp. 360-414, June 1996.
[85] A. Power and L. Sterling, "A Notion of Map between Logic Programs," *Proc. Int'l Conf. Logic Programming*, pp. 390-404, 1990.
[86] J. Farrell, "Make Bad Code Good—Refactor Broken Java Code for Fun and Profit," *JavaWorld*, Mar. 2001.
[87] R. Najjar, S. Counsell, G. Loizou, and K. Mannock, "The Role of Constructors in the Context of Refactoring Large-Scale Object-Oriented Systems," *Proc. European Conf. Software Maintenance and Reeng.*, pp. 111-122, 2003.
[88] T. Genssler, B. Mohr, B. Schulz, and W. Zimmer, "On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems," *Proc. TOOLS Conf.*, 1998.
[89] W.F. Opdyke, "Refactoring C++ Programs," technical report, Lucent Technologies/Bell Labs, 1999.
[90] W. Scherlis, "Systematic Change of Data Representation: Program Manipulations and Case Study," *Proc. European Symp. Programming*, 1998.
[91] L. Tokuda and D.S. Batory, "Automated Software Evolution via Design Pattern Transformations," *Proc. Int'l Symp. Applied Corporate Computing*, Oct. 1995.
[92] P. Borba and S. Soares, "Refactoring and Code Generation Tools for AspectJ," *Proc. OOPSLA 2002 Workshop Tools for Aspect-Oriented Software Development*, Nov. 2002.
[93] D. Astels, "Refactoring with UML," *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng.*, pp. 67-70, 2002.
[94] G. Sunyé, D. Pollet, Y. LeTraon, and J.-M. Jézéquell, "Refactoring UML Models," *Proc. Unified Modeling Language Conf. 2001*, 2001.
[95] M. Boger, T. Sturm, and P. Fragemann, "Refactoring Browser for UML," *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng.*, pp. 77-81, 2002.
[96] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards Automating Source Consistent UML Refactorings," *Proc. Unified Modeling Language Conf.*, 2003.
[97] J.H. Jahnke and A. Zündorf, "Rewriting Poor Design Patterns by Good Design Patterns," *Proc. ESEC/FSE '97 Workshop Object-Oriented Reeng.*, 1997.
[98] A. Russo, B. Nuseibeh, and J. Kramer, "Restructuring Requirements Specifications for Managing Inconsistency and Change: A Case Study," *Proc. Int'l Conf. Requirements Eng.*, pp. 51-61, 1998.
[99] E. Casais, "Automatic Reorganization of Object-Oriented Hierarchies: A Case Study," *Object Oriented Systems*, vol. 1, pp. 95-115, 1994.
[100] T. Tourwé and W. De Meuter, "Optimizing Object-Oriented Languages through Architectural Transformations," *Proc. Int'l Conf. Compiler Construction*, 1999.
[101] F.W. Callis, "Problems with Automatic Restructurers," *SIGPLAN Notices*, vol. 23, pp. 13-21, Mar. 1988.

[102] T. Mens, "A Formal Foundation for Object-Oriented Software Evolution," PhD thesis, Dept. of Computer Science, Vrije Universiteit Brussel, Belgium, Sept. 1999.
[103] M. Ó Cinnéide and P. Nixon, "A Methodology for the Automated Introduction of Design Patterns," *Proc. Int'l Conf. Software Maintenance,* pp. 463-474, 1999.
[104] R. Lämmel, "Towards Generic Refactoring," *Proc. SIGPLAN Workshop Rule-Based Programming,* 2002.
[105] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A Meta-Model for Language-Independent Refactoring," *Proc. Int'l Symp. Principles of Software Evolution,* pp. 157-169, 2000.
[106] C. Atkinson and T. Kühne, "The Role of Meta-Modeling in MDA," *Proc. UML 2002 Workshop Software Model Eng.,* pp. 67-70, Oct. 2002.
[107] K. Beck, *Extreme Programming Explained: Embrace Change.* Addison Wesley, 2000.
[108] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring Test Code," *Extreme Programming Perspectives,* M. Marchesi, ed., pp. 92-95, 2001.
[109] T. Mens and T. Tourwé, "A Declarative Evolution Framework for Object-Oriented Design Patterns," *Proc. Int'l Conf. Software Maintenance,* pp. 570-579, 2001.
[110] P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi, *Handbook of Graph Grammars and Graph Transformation,* pp. 107-188, World Scientific, 1999.
[111] R. Heckel, J.M. Küster, and G. Taentzer, "Confluence of Typed Attributed Graph Transformation Systems," *Graph Transformation,* 2002.

Tom Mens received the Licentiate degree in mathematics in 1992, Advanced Masters degree in computer science in 1993, and PhD degree in science in 1999 at the Vrije Universiteit Brussel. He has been a teaching and research assistant, a research councellor for industrial research projects and a postdoctoral fellow of the Fund for Scientific Research Flanders (FWO). Since October 2003, he has been lecturing on software engineering and programming languages at the Université de Mons-Hainaut. He has published numerous peer-reviewed articles on the topic of software evolution, and has been a co-organizer, program committee member, and referee of many international workshops and conferences. He is a cofounder and coordinator of two international scientific research networks on software evolution, financed by the FWO and the European Science Foundation, respectively. He is a copromotor of a FWO interuniversity research project on software refactoring. He is a member of both the IEEE and the IEEE Computer Society.

Tom Tourwé received the degrees of Licentiate in computer science in 1997 and PhD in science in 2002 from the Vrije Universiteit Brussel. Since February 2002, he has been associated with the Department of Computer Science at the Vrije Universiteit Brussels as a postdoctoral research assistant. His research interests focus on advanced techniques to assess and improve the design quality of software in an automated way. He has published several peer-reviewed articles on this topic in international conferences and workshops. He is also involved as a research coordinator in two industrial research projects.