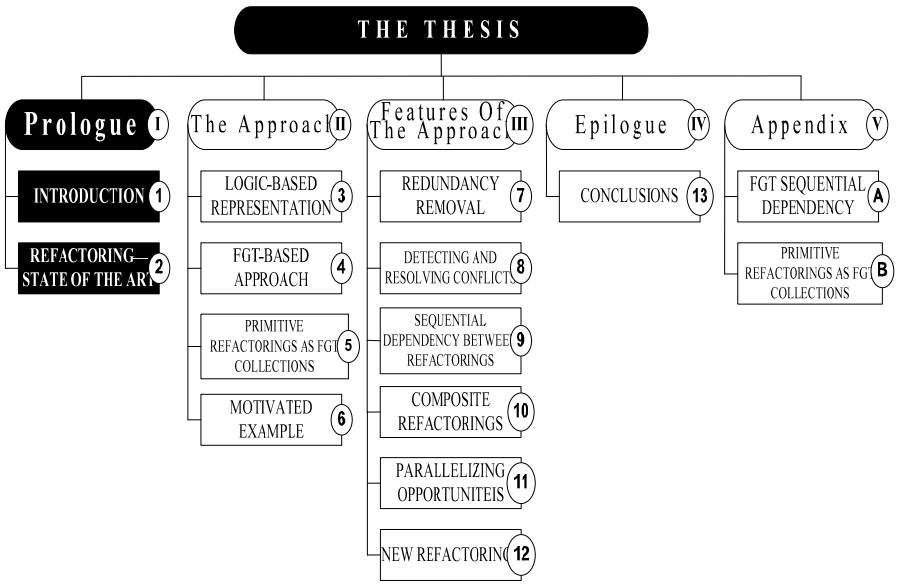
Part I

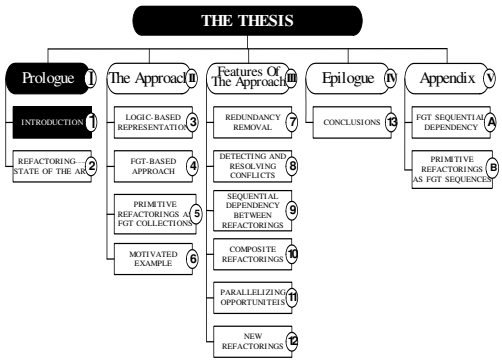
Prologue



CHAPTER 1

INTRODUCTION

1.1 The Problem



Software that is used in a real-world environment inevitably changes or becomes progressively less useful in that environment. As evolving software changes, its structure tends to become more complex [46]. “Because of this, the major part of the total software development cost is devoted to software maintenance [9, 29, and 47]. 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 [25], 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 [1, 28] or, in the specific case of object-oriented software development, refactoring [22, 65].” [59]

Refactoring is the process of improving the internal structure of the software while preserving its external behaviour [22, 65 and 70]. By improving the internal structure it is meant that refactoring will restructure the software in order to improve its quality by making it easier to understand, to extend, to find bugs, and to program faster [2, 60]. Preserving the external behaviour means, before and after applying the refactoring, the software will require the same preconditions and result in the same postconditions. The refactoring community assumes a set of precondition conjuncts for each refactoring that needs to be satisfied as a condition for applying that refactoring.

To give an idea about refactoring before going into the details of the thesis, Figure 1.1(a) shows a simple example of a UML class diagram with four classes: *HR*, *Employee* as a superclass, *Salesman* and *Engineer* as subclasses of *Employee*. The *HR* class has two association relations, one with each of the *Salesman* and *Engineer* classes. The *Salesman* and *Engineer* subclasses have the same method *getName* which is called by the method *report* in the *HR* class.

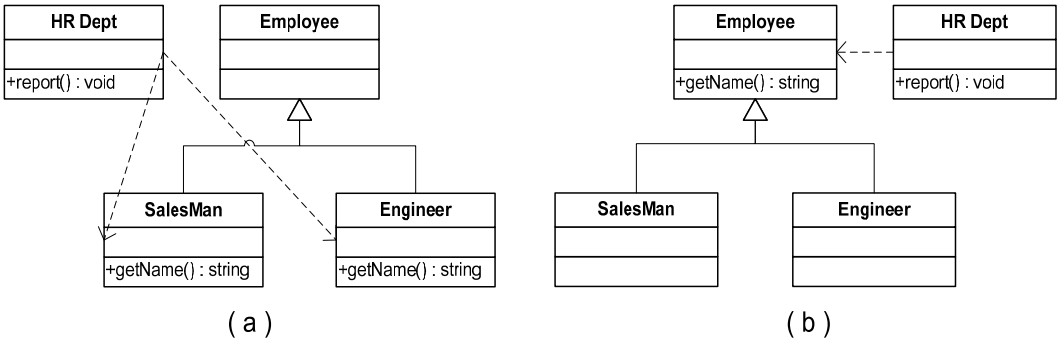


Figure 1.1: pullUpMethod Refactoring: (a) before refactoring, (b) after refactoring

Note that the duplication of the *getName* method in the two subclasses as shown in Figure 1.1(a) causes the following design problems:

1. More efforts and spaces are needed at the design and code levels.

2. There is an increased chance of inconsistency between the two copies. This can arise if the developer changes one of the two copies and forgets to change the other.

3. The design is complicated, because the same method appears two times in the design. This also causes two association relations to be created between *HR* class and each one of the two subclasses.

To solve these problems, it is preferred to change the design by deleting the *getName* method from the two subclasses and move it to their superclass, as shown in Figure 1.1(b). As a result, one copy of the *getName* method will appear in the design and also the two association relations between the *HR* class and the two subclasses will be replaced by one association relation between the *HR* class and the *Employee* class. Doing this restructuring will increase the quality of the internal design of the class diagram without changing the external behaviour of the system (The system will make the same services as before restructuring). This is because the *getName* method will be inherited to the two subclasses. The associated association relation also will be inherited.

The restructuring done in the previous example is an example of refactoring. In this case, it is a **pullUpMethod** refactoring. The precondition for the **pullUpMethod** refactoring that should be satisfied in order to apply the refactoring to the system, as a condition to preserve the behaviour of the system is:

1. The *getName* method should not be declared in the superclass (*Employee*) or any of its ancestors.

2. The access mode of the *getName* method in the subclasses is not *private.*

3. All the references made by the *getName* method must be visible from the superclass.

4. The signature of the method in all the subclasses should be the same.

A current research trend is to investigate refactorings at levels of abstraction above the code-level [23, 68, and 81]. This is because many people are visually oriented and prefer to visualize the relationships between classes rather than apprehend them textually. Furthermore, being able to directly manipulate code at a higher level of granularity (i.e. methods, variables, and classes rather than characters) can make refactoring more efficient [2]. Therefore, this thesis also focusses on refactorings at the design level.

Several approaches have been used to formalize such refactorings, as discussed in section 2.4. For example, the graph transformations approach [11, 18 and 19] represents software as a graph, and refactorings are formalized as graph-production rules [7, 34, 51-56, and 63]. As another approach, the logic-based conditional transformation approach [38, 39] represents software as logic-terms and refactorings are formalized as conditional transformations with pre- and postconditions.

In general, reasoning takes place at the level of refactorings themselves, and attention is not paid to the detailed transformational steps that must be applied to the model to achieve the refactoring. Such reasoning is with respect to a set of preconditions that must be satisfied in order to apply that refactoring, resulting in a set of postconditions. In this sense, a refactoring is treated as an abstraction, or as a black box as illustrated in Figure 1.2.

Figure 1.2: Refactorings as black box

Of course, to be of practical value, these conceptual ideas have to be implemented in refactoring tools. Such a tool would have to access some representation of an underlying system that is to be refactored. The refactorings themselves are implemented as hard coded parameterise procedures—i.e. as a sequence of code statements. To apply a particular

refactoring to the underlying system, the tool requires an interface that allows the user to select and invoke procedures which then execute the actual refactoring, thus changing the underlying system representation accordingly.

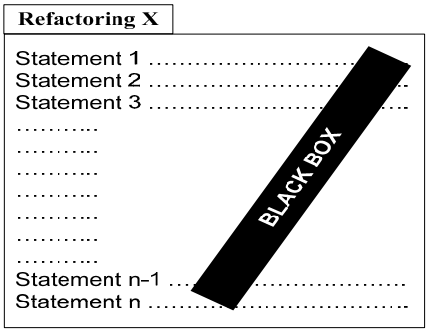
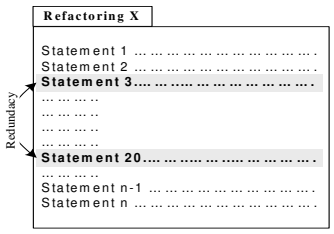


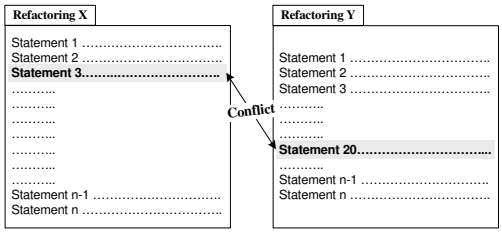
Figure 1.3: Refactorings as hard coded sequence of statements

Treating refactoring as a black box can be notionally conceived of as shown in Figure 1.3. Whenever a refactoring is applied, the hard coded sequence of statements is executed atomically. The inter-relationship between the different code statements both within and between refactorings cannot be determined. This has the following implications:

1. Where redundancy inside or between refactoring may exist, there is no possibility to remove it. As shown in the figure on the right, there could be a redundancy between statement 3 and 20 in the code. For example, if statement 3 adds an attribute to a specific class in the system and subsequently statement 20 deletes or changes the name or definition type of that attribute, the redundancy cannot be removed. This kind of scenario could arise, for example, when composing two or more refactorings into a single one.

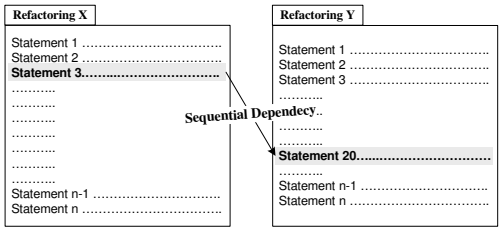


2. Where conflict occurs between two refactorings, it is not possible to determine which part of the two refactorings caused the conflict. The figure on the right side illustrates this by showing a conflict between statement 3 in refactoring X and statement 20 in refactoring Y. For example, statement 3 might add an attribute to a specific class in the system, based on a precondition of refactoring X that the class exists but does not have that attribute. On the other hand, statement 20 might delete that class from the system, based on the precondition—that the class exists and has no attributes.

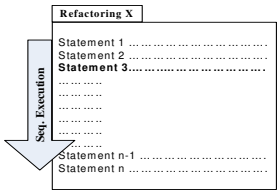


This would constitute a conflict between the two refactorings if they were to be applied as separate threads to the system.

3. Where there is a sequential dependency between two refactorings, there is no possibility to know at what specific point on it one of the two refactorings is sequentially dependent on the other. As shown in the figure on the right side if there is a sequential dependency between statement 3 in refactoring X and statement 20 in refactoring Y. Where statement 3, for example, adds a class to the system and statement 20 adds an attribute to that class. In this case refactoring Y is considered to be sequentially dependent on refactoring X and having to be applied to the system after refactoring X. Again, because the two refactorings are considered as code sequences, there is no possibility to know at what specific point in the code one of the two refactorings becomes sequentially dependent on the other.



4. Because refactorings are considered as code sequences, two or more refactorings can only be run in parallel if they are shown to be sequentially independent of each other. Because there is no meta-information about the nature of sequential dependency between their constituent code statements, it is not possible to determine whether parts of the refactorings could be run in parallel.



5. A new composite refactoring can be assembled by using previously-defined refactorings as building blocks. Its constituent elements can only be analysed for redundancy, conflicts, sequential dependency and possible parallelization with reference to the pre- and postconditions of these elements—i.e. with reference to the properties of the original refactorings. Nevertheless, as will be discussed later, such an analysis can suggest an ordering of the constituent refactorings which will avoid the so-called rollback problem.

6. If a tool allows a user to build new refactorings, the semantics of any new refactoring is necessarily constrained by the selection of refactorings that have been implemented in the tool. Any refactoring whose semantics goes beyond that will have to be hard coded as a task to be undertaken by the tool developer, rather than the tool user.

1.2 The Proposed Formalism

The refactoring formalism proposed in this thesis and described briefly in [73-75], is based on a predefined set of fine-grain transformations (FGTs) which are the basis for the construction of refactorings. These FGTs are derived from the general transformation actions that can be performed on elements of a UML class model. Each FGT can be applied to a UML model of a system, provided that the system satisfies the FGT's precondition. The FGT's postcondition is then realized on the system, which represents, in general, a small incremental change to the system. Note that this change need not preserve system behaviour.

Nevertheless, it will be shown that refactorings (which, of course, do preserve system behaviour) can be constructed by using a collection of these FGTs. As illustrated in Figure 1.4, a set of refactorings in the present approach is set of directed acyclic graphs (FGT-DAGs), each of which specifies an ordering of FGTs to be used in the refactoring. The order, effect, pre- and postcondition of each FGT in each FGT-DAG is known to the tool, and can be controlled at the time of refactoring. Of course, the final effects of refactoring X in Figure 1.4 is the same as the final effects of a hard coded version of refactoring X in Figure 1.3.

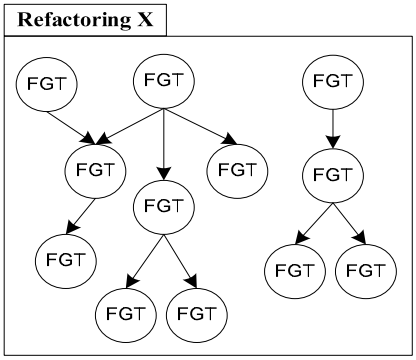


Figure 1.4: Refactoring as a set of FGT-DAGs

It will be shown that representing refactorings as a collection of FGTs allows for the following:

1. Where redundancy occurs between transformation operations that are carried out by the refactorings, the redundancy can be discovered and removed at the FGT-level. (This will be discussed in more detail in chapter 7)

2. In the case of conflict between two refactorings, the FGTs that cause the conflict can be discovered, and in some cases the conflict can be resolved without withdrawing one of the refactorings. (This will be discussed in more detail in chapter 8)

3. Sequential dependency between two refactorings can be discovered at the FGT-level. (This will be discussed in more detail in chapter 9)

4. Composite refactorings of more than one refactoring can be composed in a way that will avoid rollback problems. However, this is done by manipulating the ordering of FGT execution, rather than of refactoring execution. (This will be discussed in more detail in chapter 10)

5. Parallel execution can be exploited at the FGT-DAG level. Thus, all FGT-DAGs in one refactoring can be executed concurrently because there is no sequential dependency between the FGT-DAGs. For example, the refactoring in Figure 1.4 has two FGT-DAGs that can be manipulated concurrently. (This will be discussed in more detail in chapter 11)

6. An FGT-based tool can be built that will allow a user to build new refactorings whose semantics is constrained, not by the selection of existing refactorings that have been implemented in the tool, but rather by the semantics of the FGTs that have been predefined in the tool. (This will be discussed in more detail in chapter 12).

The discussion in this thesis is restricted to refactorings that relate to the simplified UML meta-model shown in Figure 1.5. In addition, it will be assumed that a limited amount of information derived from the source code of the system to be refactored is also available, as will be discussed in due course. Although the use of this code-based information goes beyond the requirements of existing approaches, it can be acquired fairly easily.

In deciding of which features of UML to include and which to exclude from the study, consideration had to be given to having a subset of the UML vocabulary that would be sufficiently large to lend credibility to the approach, yet not be so ambitious that it would prevent full coverage within the time available for this study. It was thought that the vocabulary represented by the simplified meta-model of FIgure 1.5 complied with this objective. Although, UML notations relating to interfaces, abstract classes, abstract methods, aggregations and so on, are not considered, extending the ideas developed in this thesis to these UML notations appears to be quite straightforward. However, a detailed investigation of this conjecture is a matter for future study.

It should be noted there are tools (such as IDEA by IntelliJ and Eclipse by IBM) that directly analyse and manipulate an existing code base. However, the types of refactorings that they address are generally of a different order to those addressed here (e.g. removal of declared but unused variables, or the identification of common code segments that can be turned into a method) and are beyond the scope of this thesis.

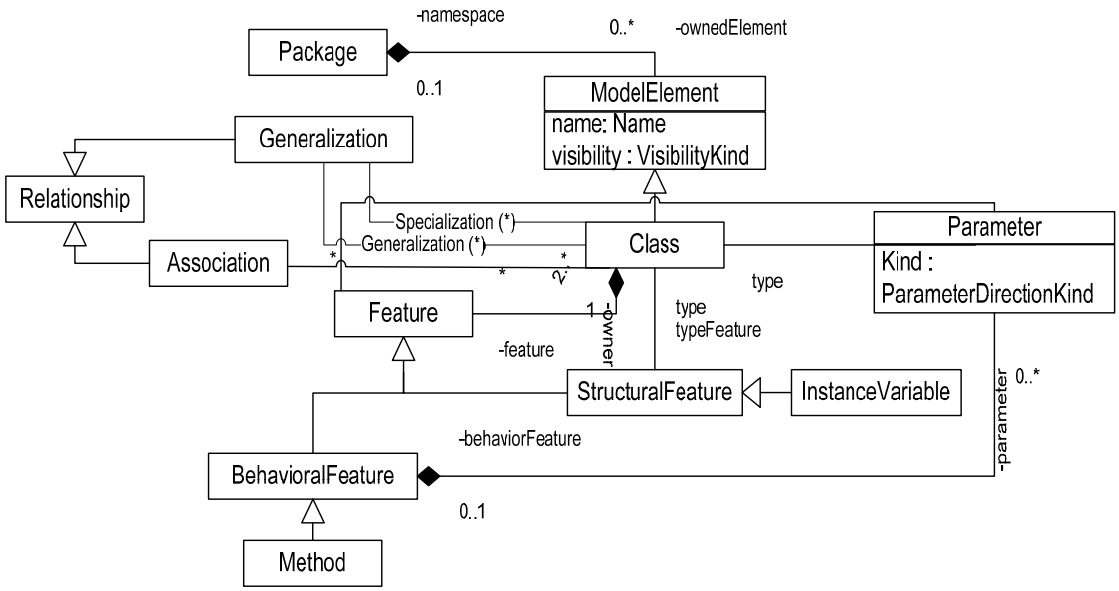


Figure 1.5: Simplified UML meta-model

1.3 Thesis Overview

In the next chapter, a survey of previous work in refactoring is presented. Thereafter, chapters three to six present the proposed approach and discuss the feasibility of the approach for formalizing refactorings. Then, chapters seven to twelve discuss the features that are obtained by adopting such approach.

The logic-based underlying representation of the UML class diagrams of the system under consideration is presented in chapter 3. Chapter 4 proposes an FGT-based methodology to construct model transformations in which FGTs are at the core of the refactoring system. Several common primitive refactorings that are frequently defined and used in the refactoring literature are presented in chapter 5. To illustrate the proposed approach, a motivated example is given in chapter 6.

Presenting features of the approach is started in chapter 7. The chapter introduces the idea of removing the redundancy between FGTs allocated in the same FGT-DAG. Chapter 8 shows

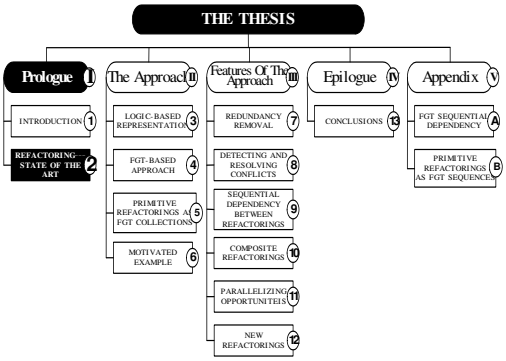
how to detect and resolve conflicts that may occur between two refactorings. The sequential dependency between two refactorings is discussed in chapter 9. Chapter 10 discussed the implications of using FGTs to deal with composite refactorings. The opportunities for parallelizing refactorings are presented in chapter 11. Chapter 12 presents the possibility for end users to build their own refactorings. Finally, chapter 13 summarizes the work, explores the contributions and identifies tasks for future work.

In summary, then, this thesis will show that when FGTs are used to build refactorings, all the well-known refactoring operations (such as determining redundancy, conflict and sequential dependency; and building composites) can take place at the FGT-level. In theory, this comes with certain advantages and disadvantages. Advantages include the fact that the user of an FGT-based tool will have enhanced flexibility in specifying new refactorings; redundancies and conflicts can be more accurately pin-pointed and removed; and opportunities for parallel execution are exposed at a more fine-grained level. It will be seen that these advantages come at the cost of having to carry out more computations because analysis has to take place at the FGT-level, rather than at what will later be called the “refactoring level". Although a prototype tool has been built to verify these claims, the full practical implications of this work are a matter for future study.

CHAPTER 2

REFACTORING — STATE OF THE ART

In this chapter, a survey of work related to refactoring is presented. First, the concept of software evolution and its relation to refactoring is introduced. Then, works related to different types of software artifacts that can be refactored is presented. Finally, works related to different refactoring formalisms is discussed.



2.1 Software Evolution

"Software evolution is an essential part of the software development process. Nearly all software inevitably undergoes changes during its lifetime. Changes can be large or small, simple or complex, important or trivial - all of which influence the effort needed to implement the changes" [51]. Sommerville [79] explains that proposals for change are the driver for system evolution. Change identification and evolution continue throughout the system's lifetime. Lehman & Belady [46] conducted empirical studies into software evolution and concluded the following eight laws:

1. Continuing change: Software that is used in a real-world environment necessarily must change or become progressively less useful in that environment.

2. Increasing complexity: As evolving software changes, its structure tends to become more complex. Extra resources must be devoted to preserve and simplify the structure.

3. Large program evolution: Software evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.

4. Organizational stability: Over a software lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.

5. Conservation of familiarity: Over the lifetime of a software, the incremental change in each release is approximately constant.

6. Continuing growth: The functionality offered by systems has to continually increase to maintain user satisfaction.

7. Declining quality: The quality of systems will appear to be declining, unless they are adapted to changes in their operational environment.

8. Feedback system: Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Experience over the last 30 years has shown that making software changes without visibility into their effects can lead to poor effort estimates, delays in release schedules, degraded software design, unreliable software products, and the premature retirement of the software system. The immaturity of current-day software evolution is clearly stated in the foreword of the international workshop on principles of software evolution [69]:

"*Software evolution is widely recognised as one of the most important problems in software engineering. Despite the significant amount of work that has been done, there are still fundamental problems to be solved. This is partly due to the inherent difficulties in software evolution, but also due to the lack of basic principles for evolving software systematically.*"

Software evolution is not restricted to the implementation phase only. Even in the earlier phases of requirements specification, analysis and design, evolution is a strict necessity. To date, most research on evolution has been dedicated to the implementation and maintenance phases, and to a lesser degree in the earlier phases of requirements specification and design [12, 15, 33, 41, 87, and 88]. However, there is a tendency to shift towards earlier phases.

2.2 Refactoring

Although in the context of *software reengineering*, refactoring is often used to convert legacy code into a more modular or structured form [20], refactoring can also 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 from 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[59].

In the following subsections, an introduction of refactorings at different types of software artifacts is given.

2.2.1 Codes Level

2.2.1.1 Non-Object-Oriented Programming Languages

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 [59].

In [27], Griswold proposes a technique to restructure programs written in a block-structured programming language. The language he worked on is Scheme. His transformations concern program restructuring for aiding maintenance. To insure that the transformations are meaning preserving, he uses Program Dependence Graphs to reason about the correctness of transformation rules.

Lakhotia and Deprez [42] present a transformation called *tuck* for restructuring programs by decomposing large functions into small functions. The transformation breaks large code fragments and tucks them into new functions. The challenge they faced was creating new functions that capture computations that are meaningfully related. There are three basic transformation to tuck functions.

4. Related code is gathered by driving a wedge (which is a program slice bounded with single-entry and a single exit point) into the function.

5. Then the code that is isolated by the wedge is split.

6. Finally, the split code is folded into a function.

These transformations can even create functions from non-contiguous code.

2.2.1.2 Object-Oriented Programming Languages

Opdyke, in his PhD thesis [65] was the first to introduce the term refactoring. His proposed refactorings were in the context of object-oriented programming languages. He identified twenty-three primitive refactorings and gave examples of three composite refactorings. He

arrived at his collection of refactorings by observing several systems and recording the types of refactorings that OO programmers applied.

The importance of the achievements of Opdyke is not only the identification of refactorings, but also the definition of the precondition that is required to apply a refactoring to a program without changing its behaviour. For that, he defined for each primitive refactoring a set of precondition conjuncts that would ensure that the refactoring would preserve behaviour.

Roberts, in his PhD thesis [70], improves the work of Opdyke. He gives a definition of refactoring that focuses on their pre- and postcondition conjuncts. The definition of postcondition conjuncts allows the elimination of program analysis that is required within a chain of refactorings. This comes from the observation that refactorings are typically applied in a sequence intended to set up precondition conjuncts for later refactorings.

In his book [22], Fowler presents a catalogue of refactorings. Each refactoring is given a name and short summary that describes it. A motivation describes why the refactoring should be done, a step-by-step description of how to carry out the refactoring and an example.

Back [3] propose a method called *stepwise feature introduction* for software construction. The method is based on incrementally extending the system with a new feature at a time. Introducing a new feature may destroy some already existing features, so the method must allow for checking that old features are preserved.

2.2.2 Design Level Models

A recent research trend is to deal with refactoring at a design level, for example, in the form of UML models [64]. Applying refactoring to models rather than to source code can encompass a number of benefits [23]. Firstly, software developers can simplify design evolution and maintenance, since the need for structural changes can be more easily identified and addressed on an abstract view of the system. Secondly, developers are able to address deficiencies uncovered by model evaluation, improving specific quality attributes directly on the model. Thirdly, a designer can explore alternative decision paths in a cheaper way (although small prototypes may be necessary). An apparent scenario for model refactorings is the incorporation of design patterns into a system's design model [37].

France *et al.* [23] identified two classes of model transformations: vertical and horizontal transformations. Vertical transformations change the level of abstraction, whereas horizontal transformations maintain the level of abstraction of the target model. A model refactoring is an

example of horizontal transformation. In contrast, the Model-Driven Architecture (MDA) approach [78], in which abstract models automatically derive implementation-specific models and source code, provides examples of a vertical transformation.

As the idea of refactoring models adds simplicity to software evolution, automatization and behaviour preservation are even more complex issues when dealing with models. Editing a class diagram may be as simple as adding a new line when introducing an association, but such changes must include identifying lines of affected source code, manually updating the source, testing the changes, fixing bugs and retesting the application until the original behaviour is recovered [83]. Methods and tools for partially or even totally removing human interaction in this process are invaluable for the refactoring practice.

Suny'e *et al*. [81] have provided a fundamental paradigm for model refactoring to improve the design of object-oriented applications. They present refactorings of class diagrams and state charts. In order to guarantee behaviour-preserving transformations of state charts, they specify the constraints that must be satisfied before and after the transformation using the OCL at the meta-model level.

Porres [68] implemented refactorings as a collection of transformation rules, which receives one or more model elements as parameters, and performs a basic transformation based on the parameters.

Boger *et al.* [6] present a refactoring browser integrated into a UML tool. They concentrate on the detection of conflicts that may be introduced after refactorings. They classify conflicts as warnings and errors. Warnings indicate that conflicts might cause a side effect. Errors indicate that an operation will cause damage to the model or code. They also address refactoring of state machines, like merging of states and formation of composite states.

Bottoni, Parisi and Taentzer [7] present an approach to maintain the consistency of specification and code after refactoring. They show that some refactorings require modifications in several diagrams at once. To ensure consistency between source code, structural and behavioural models, they use graph transformations.

Astel [2] proposes using an UML tool as an aid in finding smells—a structure in code that suggest the possibility of refactoring—and performing some elaborate refactorings. It is a tool that bases class diagrams directly on code, allowing code manipulation by the direct manipulation of the diagram.

Gorp *et al.* proposed a UML extension to express the pre- and postcondition of source code refactorings using OCL [26]. The proposed extension allows an OCL empowered CASE tool to verify non-trivial pre- and postcondition, 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.

2.2.3 Database Schemas Level

The main focus of database schemas is on how data should be structured. Therefore, they are ideal candidates for refactoring. In fact, the research area of object-oriented software refactoring originates from the research on how to restructure object-oriented database schemas.

Banerjee and Kim [4] applied refactoring in the context of database schema evolution. They defined a set of schema transformations, which are used for schema evolution and identified a set of invariant properties of an object-oriented schema which must be preserved across schema changes. An example of such an invariant is that attributes of a class, whether defined or inherited, have distinct names.

2.2.4 Software Architectural Level

In [67] Philipps and Rumpe propose a promising approach to deal with refactorings at the software architecture level. In their work, refactoring rules are based directly on the graphical representation of a system architecture. These rules preserve the behaviour specified by the causal relationship between the components.

Another approach is presented by Tokuda and Batory [83]: 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).

In [36] Kempen, Chaudron, and Kourie proposed an approach to refactoring at the software architectural level. In their approach, they use a CSP-based formalism to describe the refactoring and they show that the proposed refactorings indeed preserve behaviour of the system.

2.2.5 Software Requirements Level

Restructuring can also be applied at the requirements specifications level. For example, In [72], Russo *et al.* proposed an approach to refactor the requirement specifications of the

system. Their proposal is to restructure natural language requirement specifications by decomposing them into a structure of viewpoints. Each viewpoint encapsulates partial requirements of some system components, and interactions between these viewpoints are made explicit. This restructuring approach increases requirement understandings, and facilitates detecting inconsistencies and managing requirement evolutions.

2.3 Formalisms

A wide variety of formalisms have been proposed and used to deal with refactoring.

2.3.1 Graph Transformations

Graph transformation [10, 11, 18, 19, and 63] is one way to deal with restructuring. The software is represented as a graph, and restructuring corresponds to transformation rules. Mens [51] presents the formalization of refactoring using graph rewriting, a transformation that takes an initial graph as input and transforms it into a result graph. This transformation occurs according to some predefined rules that are described in a graph-production which is specified by means of left-hand and right-hand sides. The first one specifies which parts of the initial graph should be transformed, while the last one specifies the result after transformation.

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 [54]. 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 [7]. Heckel [31] uses graph transformations to formally prove the claim (and corresponding algorithm) of Roberts [70] that any set of refactoring postcondition conjuncts can be translated into an equivalent set of precondition conjuncts. Van Eetvelde and Janssens [17] propose a hierarchical graph transformation approach to be able to view and manipulate the software and its refactorings at different levels of detail.

2.3.2 Pre- and Postcondition

A refactoring's definition can be given in terms of an invariant in the form of a pre- and postcondition that should hold before and after the refactoring has been applied. This can form the basis of a lightweight and automatically verifiable means to ensure that the behaviour of the software is preserved by the refactoring.

The use of pre- and postcondition has been suggested repeatedly in research literature as a way to address the problem of behaviour 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 behaviour of these schemas [4]. Opdyke adopted this approach to object-oriented programs, and additionally provided precondition conjuncts or enabling conditions for each refactoring [65]. He argued that this precondition preserves the invariants. Roberts used first order predicate calculus to specify these precondition conjuncts in a formal way [70].

The notion of precondition or applicability condition is also available in the formal restructuring approach of Ward and Bennett, using the formal language WSL [86].

2.3.3 Program Slicing

Program slicing [5, 43, and 82] deals with specific kinds of restructurings: function or procedure extraction. These techniques based on system dependence graphs, can be used to guarantee that a refactoring preserves some selected behaviour of interest. Lakhotia and Deprez [42] present a transformation called *tuck* for restructuring programs by decomposing large functions into small functions. The approach breaks large code fragments and tucks them into new functions.

A similar approach is taken in [40], where an algorithm is proposed to move a selected set of nodes in a control flow graph, so that they become extractable while preserving program semantics. They identified conditions based on control and data dependence that are considered to be sufficient to guarantee semantic equivalence.

2.3.4 Formal Concept Analysis

In [24] a technique called *formal concept analysis (FCA)* is used to deal with restructuring. FCA involves clustering so-called objects (not necessarily software objects) according to their attributes. The result is a set of nodes (called concepts) that are hierarchically arranged in a lattice. Snelting in [77] uses FCA to restructure object-oriented class hierarchies. The result is guaranteed to be behaviourally equivalent with the original hierarchy. Tonella in [84] uses the same technique to restructure software modules. Deursen in [14] uses FCA to identify objects by semi-automatically restructuring legacy data structures.