Chapter 2

Background

In this chapter we explore the background to this research, with the aim of putting our work in context. We survey the two research fields that form the foundation of this work, namely program restructuring (2.1) and design patterns (2.2). In section 2.3 we state precisely the gaps our work aims to fill in the existing literature, and, in section 2.4, the chapter is summarised.

Detailed analyses of very closely related work and comparisons between our work and others are not covered in this chapter, but appear in later chapters.

2.1 Program Restructuring and Refactoring

2.1.1 Definitions

In their widely-used taxonomy of reengineering terms, Chikofsky and Cross define *restructuring* in this way:

> Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserv-

> ing the subject system's external behaviour (functionality and semantics).[19]

Program restructuring then is a source-to-source restructuring that preserves the semantics and external behaviour of the program.

The first use of the term "refactoring" in the literature was in the work of Opdyke and Johnson [78], though the practice was in use well before this. Opdyke defines refactorings as "behaviour-preserving program restructurings[1]," which is the definition we use in this work. Fowler uses a similar definition, though emphasizes that we expect the process of refactoring to improve the design:

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure. [38, p.xvi]

Roberts changes the definition radically by also permitting "refactorings" that change program behaviour [84]. While it is valuable to allow program transformations that are not behaviour-preserving, the redefinition of a standard term seems very unnecessary, especially in a field that is already dogged by confusing terminology [5].

We have used the term "behaviour preserving" without being specific as to what is meant. Opdyke defined it in terms of observable behaviour, i.e., that the program must produce the same externally observable behaviour for any legal input before and after the refactoring [77]. Roberts correctly points out that if timing constraints are taken to be part of program behaviour, it becomes extremely difficult to argue behaviour preservation. Other non-functional properties of a program, for example memory usage or patterns

[1]This is tautological, since restructurings are, by definition, behaviour preserving.

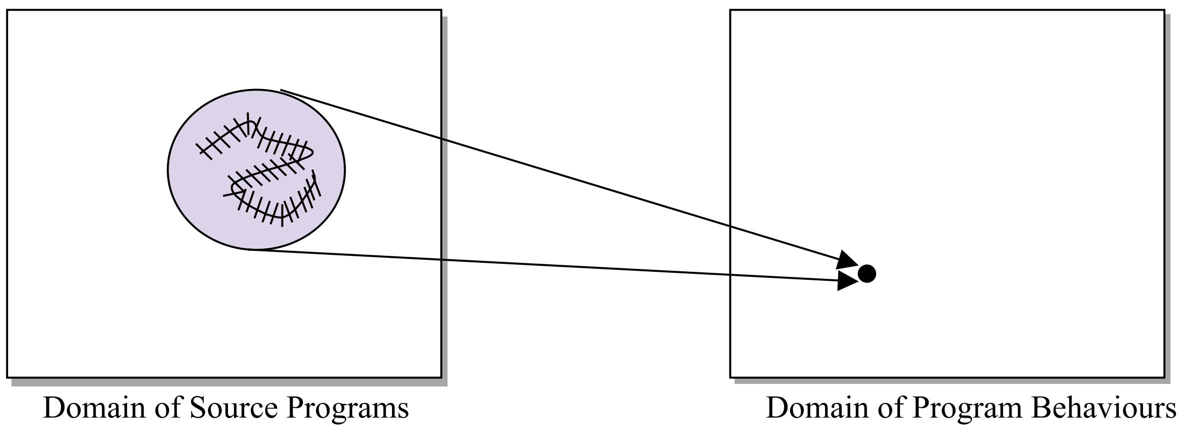

Figure 2.1: Graphical Image of Refactorings

of network access, would also be very difficult to maintain in a refactoring[2]. For these reasons, we do not consider in this work programs where timing constraints or other non-functional requirements are part of their specification.

2.1.2 A Global View of Refactoring

Figure 2.1 is a graphical depiction of refactoring and what it aims to achieve. The domain on the left is the set of all source programs (e.g., all legal Java programs) while the domain on the right depicts the set of all possible program behaviours. The shaded subset on the left is a set of programs that all exhibit the same behaviour, depicted by their all mapping to the same point in the behaviour domain.

Refactoring research aims to show how, given a program in the shaded set, it is possible to transform it to other programs in the same set. Of

[2]In a practical sense, the behaviour of a program that has been optimised to run in a particular hardware/software environment could be affected by refactoring.

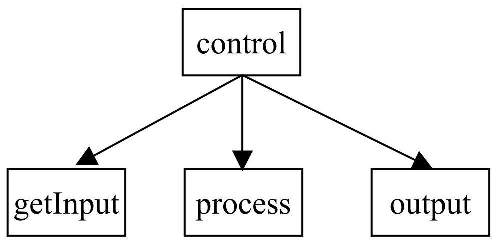

Figure 2.2: A Generic Structure Chart

course, it is not interesting to do this in a random fashion[3]; the aim is to improve the design of the program according to some criteria. Refactoring research aims then to build the "train tracks" that connect one program to another program with the same behaviour. In the diagram, applying a composition of refactorings is equivalent to moving along the track to a possibly very different program structure, but one that nevertheless exhibits the same external behaviour.

Refactoring research has really only taken place in the past decade, and has been focused on the transformation of object-oriented programs. To understand why it never received much attention in the context of structured programming, consider the generic structure chart depicted in figure 2.2, and what sort of refactorings could be applied to it. It is hard to propose much, other than that data that is passed around the chart a lot could be moved to a shared data structure. The problem to be solved has been factored into a number of functions and these have been fixed in a tight control structure where little movement is possible.

By way of contrast, consider the generic class diagram of figure 2.3. Even without any knowledge of the actual application, many possible refactorings

[3]A quirky notion would be to apply refactorings to a program pseudo-randomly, perhaps using simulated annealing, and use some metrics suite to decide if the design was improved.

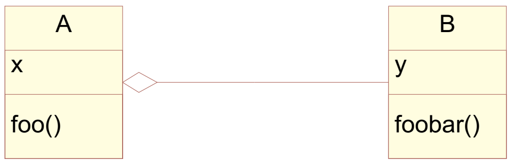

Figure 2.3: A Generic UML Class Diagram

come to mind. An interface could be added to the class B and the class A updated to access B only via this interface. The method foo could be moved from the class A to the class B and replaced by a delegating method. Perhaps foo could be moved to another class entirely and A updated to inherit it from that class. Similar refactorings could be applied to the method foobar. We could even contemplate replacing the aggregation relationship from A to B with an inheritance relationship in the same direction. The fact that so many potential refactorings spring from a simple class diagram is a consequence of the much richer set of abstractions available in the object-oriented approach when compared with the structured approach.

2.1.3 Formal and Informal Approaches to Behaviour Preservation

It is theoretically impossible for a refactoring technique to relate all programs that exhibit the same behaviour. In practice, we have to be very modest in our aims. Few industrial languages have a formal semantics. Even rarer are those that have a formal semantics and a compiler that verifiably implements those semantics. Even given a formal semantics for an industrial language, the complexity of the behaviour preservation proofs for non-trivial transformations will be intractable. Approaches based on a formal semantics of the programming language cannot therefore be currently expected to produce a working software tool[4].

Existing refactoring work has generally relied on either a semi-formal demonstration of behaviour preservation [77], or indeed no demonstration of behaviour preservation at all [38]. The former approach is appealing, in that it mimics to some degree what a disciplined programmer will do in practice when refactoring a program. They will certainly not just change it and hope for the best; they will reason logically that the change they intend to make is behaviour preserving. This is an interesting middle-ground between a fully-formal approach to proving behaviour preservation and ignoring the issue completely. By constructing a semi-formal proof of behaviour preservation we improve our confidence that the transformations we build are indeed refactorings. Also, if in testing an error is found in that a supposed refactoring changes the behaviour of the program it has been applied to, the error can be traced back to the proof and corrected there.

This notion of behaviour preservation admits many simple program refactorings. Assuming certain pre-conditions are met by the program being transformed, classes, methods and interfaces may be added or removed; invocations of a method may be replaced by invocations of another method; access to a field may be replaced by a method invocation, and so on. We will see later in this work how such simple refactorings can be combined to produce complex transformations that have a profound effect on program structure.

[4]For an interesting example of a formal, correctness-preserving approach to program restructuring applied to a small-scale software engineering problem, see [42]. This approach requires significant work in reverse engineering the program, and it is not apparent whether the transformations used can be automated.

2.1.4 Existing Work in Automated Refactoring

So far we have discussed refactoring in general, but the main focus of this thesis is specifically *automated* refactoring. Obviously automation is valuable: once the programmer decides that a certain refactoring should take place, much of what remains is tedious and error-prone work. Such work should, where possible, be automated. At the simplest level, the programmer should be able, for example, to rename a class, and have the refactoring tool check that the new name is not already in use and update all uses of the old class name to the new class name. At a much more complex level, the programmer should be able to select a number of program elements and apply a sophisticated, high-level restructuring to them; this is the direction this thesis will take.

The work of Opdyke and Roberts forms the basis for the automated refactoring approach taken in this thesis. Opdyke defined a set of refactorings that could be applied to a C++ program [77] and in further work showed how they could be used to construct higher-level refactorings, for example, to convert an inheritance relationship to an aggregation one, and vice versa [51]. Roberts [84] extended Opdyke's work by providing a more formal basis for composing refactorings, and examined the use of dynamic information in refactoring. This work will be extensively cited throughout this thesis, so it is not discussed further here. In the following subsections we consider some of the other approaches that have been taken to automated refactoring. In many cases the term refactoring has not actually been used, but the work nevertheless involves behaviour preserving restructuring of object-oriented programs.

Approaches to Inheritance Hierarchy Reorganisation

One of the significant contributions of the object-oriented approach was that it made inheritance a firm part of mainstream software development. Designing a class hierarchy is a difficult task however, so many attempts have been made to provide automated support for this process. Probably the earliest work that addressed this issue was that of Pun and Winder [80]. When a designer adds a class to a hierarchy, the design of the hierarchy may cause the class to inherit unwanted attributes. This indicates that the hierarchy should be reorganised to separate the attributes that the designer would like to be inherited from the undesirable ones. Pun and Winder show how this reorganisation process can be automated and partly formalise their work using an algebraic manipulation system.

Casais solves the "inheritance of unwanted features" problem in a somewhat different way, specifying both global and incremental algorithms that reorganise a class hierarchy so as to remove the inheritance of unwanted features [16, 17]. This improves on Pun and Winder's work in that it allows incremental reorganisation of a class library whenever a class is added to it. Casais also defines how to automate this restructuring algorithm precisely and, in [18], presents the results of applying his restructuring algorithms to the Eiffel libraries. His restructurings are intended to operate in automatic mode, which has the benefit that they can be applied to very large hierarchies, but the disadvantage that they will, in some cases, produce a result that is either incomprehensible, or of no software engineering impact.

Lieberherr, Bergstein and Silva-Lepe describe an algorithm that learns a class library from a set of object examples, and minimises the number of aggregation and inheritance relationships[5] in this library, while preserving the

[5]These are the usual interpretation of the construction and alternation relationships in

set of objects defined by the library [59, 7]. This work is based on the accepted philosophy that abstractions are discovered rather than invented [50], so it makes sense to allow a designer to define the concrete objects they want to use, and then to learn the class hierarchy from these examples. More recent work by Hürsch and Seiter in the same area describes a set of behaviour-preserving transformations that can be applied to a class library [45]. This work has never achieved popularity in mainstream software development, probably due to the fact that it is tightly bound to the seldom-used *adaptive* software model, where class structure (the *class graph*) is modelled separately from behaviour (*propagation patterns*). This contrasts strongly with the work of Opdyke and Roberts, and the work presented in this thesis, that simply assumes the class library to be specified in a mainstream programming language[6].

Ivan Moore has developed a tool called Guru that can analyse and restructure an inheritance hierarchy expressed in the Self programming language [67, 69]. The inheritance hierarchy is optimised in a certain way, whilst preserving program behaviour. Optimal is taken to mean that duplicate methods are removed, method sharing is maximised, and redefinition of methods is avoided. Moore found that in general some manual intervention was necessary to produce a good result, and that given an incompetently-developed hierarchy as input, the restructuring could not improve it ("garbage in, garbage out"). There is also the risk with this sort of automated restructuring that the essential abstractions that the programmer defined in the hierarchy will be removed by the restructuring, if they have not yet actually been made use of. In [68] Moore extends this restructuring algorithm to refactor meth-

the *Demeter* notation.

[6]Opdyke's refactorings transformed C++ programs, Roberts developed the Smalltalk Refactoring Browser, while this thesis will focus on transforming Java programs.

ods by moving common expressions to separate methods and invoking them there. While this method-level refactoring can reduce the amount of code in the application and increase reuse, the new methods it introduces will not necessarily appear cohesive to the programmer.

Snelting and Tip propose reengineering class hierarchies using *concept analysis* [91]. When a designer creates a class hierarchy, they are in effect describing their perception of the key classes and relationships in the domain they are modelling. A programmer who uses this hierarchy may find that the classes provided are not quite what are required in their application, and this will appear as anomalies in their code. For example, a class may not use all the functionality of its superclass, or the application may create several objects of the same class, but use different subsets of the class's functionality in different contexts. In both these examples, the user of the hierarchy requires different classes (or concepts) from the ones provided by the designer of the hierarchy. In this work a *concept lattice* is constructed that highlights the concepts that the programmer has actually made use of. This provides valuable guidance in reengineering the class hierarchy; in the examples described above, the classes in question probably need to be split. The type of transformations this analysis produces would have the effect of making the class hierarchy represent more truly the programmers' view of the domain. In the context of this thesis, the reengineering described in this paper could be undertaken prior to the introduction of a design pattern.

Other Approaches

Ducasse, Rieger and Demeyer describe a technique for detecting duplicated code based on simple string comparisons to detect identical lines of code, and the use of a *scatter plot* to visualise the results of the comparisons [28].

For a program with n lines of code, the corresponding scatter plot would be an n-by-n matrix where a dot is present at location (i, j) only when line i in the program is identical to line j. This work is used as a basis in [29], where a preliminary proposal is made for tool support for refactoring to remove duplicated code. They suggest that full automation is possible only in simple cases of exact code cloning, and that programmer intervention will be required in most cases.

Sweeney and Tip developed an automated approach to detecting dead data members in C++ applications [95]. A data member m is defined to be dead if there is no object in the program that contains m such that the value of m can affect the program's external behaviour. Naturally, detecting such dead data members paves the way for a simple refactoring that removes them. This type of refactoring appears unremarkable but the results achieved were dramatic. On the benchmarks tested, an average of 12.5% of the data members were found to be dead, and the average occupancy of run-time object space by dead data was found to be 4.4%. This suggests that refactoring research is still in its infancy, and that a lot can still be achieved with quite simple techniques.

Maruyama and Shima present an approach to method refactoring based on the usage patterns of a framework [63]. The basis is that a method in a framework has dependencies on other framework methods that to a greater or lesser degree match how programmers using the framework will override the method. If the method is normally overridden in such a way as to preserve these dependencies, it suggests that the interaction with the other methods is invariant and can be captured in a template method. Conversely, if the method is normally overridden in such a way as to destroy these dependencies, it suggests that the method represents a "hot spot" [79] and is better modelled as a hook method. In the first case, the transformation will mean that a programmer using the framework has less code to write; in the latter case it will mean that the programmer has less code to read. Experimental results presented in [63] produced a reduction of up to 22% in the number of statements a programmer has to write when using the framework to develop new applications. Because the refactoring process operates in automatic mode, it exhibits the attendant problem of creating new methods that may appear meaningless to the programmer. Nevertheless the results of this approach seem very valuable, probably because using the modification histories of the methods in the framework is in effect giving the programmer indirect control over what refactorings take place.

2.1.5 Categorisation of Refactoring Approaches

There are a number of attributes that can be used for categorising approaches to refactoring. The most significant ones are as follows:

- *Method of Application*: In a *fully-automated* approach a software tool is used that applies a large scale restructuring to the program. A *semi-automated* approach also involves a software tool, but involves the user choosing what refactorings are to be applied. Finally, the user can simply apply the refactoring *by hand*.

- *Approach to Behaviour Preservation*: The simplest approach is where *no proof* of behaviour preservation is presented; it is simply taken for granted or assumed to be obvious. A *semi-formal* proof means that some formal model (usually first-order predicate logic) is used to support the behaviour preservation arguments, but the reasoning used is not limited to syntactic deduction. In a fully *formal* approach, a formal model is used that reflects the semantics of the programming language sufficiently strongly that an entire behaviour preservation proof can be constructed in the formal domain.

- *Method of Composition*: A refactoring approach that provides a suite of refactorings will usually also provide a method for composing them. In *dynamic* composition the user is allowed to combine refactorings freely as they are working on the code, while *static* composition approaches provide the user with a set of higher-level (composite) refactorings.

For example, Fowler presents a catalogue of refactorings [38] that are to be applied by hand, no proof of behaviour preservation is provided, and nothing is said about composing these refactorings. On the other hand Robert's refactorings [84] are applied semi-automatically (the user states where to apply them), a semi-formal proof of behaviour preservation is provided, and a dynamic method of refactoring composition is provided.

In general, the fully automatic method of application has the advantage that it may be left run in batch mode on a large system without requiring user intervention. It may however perform refactorings that are of little or no real significance, and the ultimate results may be hard to comprehend.

As discussed earlier, a behaviour preservation argument is desirable, though the fully-formal approach is not promising.

As regards composition of refactorings, the dynamic approach is the freer and more expressive one. However the static approach allows powerful refactorings to be developed, tested extensively and then presented to the user as a reliable refactoring option.

The approach we take in this thesis is to statically develop semi-automated, composite refactorings, and to develop for each one a semi-formal proof of behaviour preservation.

2.2 Design Patterns

Patterns have been one of the most significant developments in software engineering in the past decade. The aim of this field is to identify and catalogue the knowledge and expertise that has been built up over many years of software engineering. Patterns can be identified in all parts of the development process: architecture, analysis, design, coding, reengineering, as well as in specific application areas such as real-time programming or user interface construction. Patterns are in no way invented; they are discovered or "mined" from existing systems. The motivation is to uncover proven designs that experts have already used and reused, and to distill from these the essence of the solution with domain-specific detail removed. The resulting nugget of design wisdom can then be documented and made generally available. This pattern can be assimilated by other designers and applied in other domains.

The notion of a pattern in software was borrowed from the work of the architect Christopher Alexander, who described the process of architecting living space (be it the corner of a room or an entire city) in terms of patterns. He defined the notion of a pattern in the following way:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. [1, p.247]

Varying definitions of the term pattern abound, but this "three-part" version suits our current purposes. Richard Gabriel puts the Alexandrian definition into a software context in this way:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs

> repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves. [39]

This thesis is concerned with the *automated* application of *design* patterns. We choose to work with patterns at the design level for two reasons:

- It is a richer set than the program-language specific patterns found at the coding level.
- They are more concrete than those found at the analysis level so automating their application to source code is realistic.

The notions of formalisation and automation are not generally welcomed in the patterns community. Jim Coplien expressed this distaste clearly:

> Patterns aren't designed to be executed or analyzed by computers, as one might imagine to be true for rules: patterns are to be executed by architects with insight, taste, experience, and a sense of aesthetics. [23]

We concur with this position in terms of the first two parts of the Alexandrian definition. Deciding that a context is appropriate for the application of a pattern and assessing that the forces acting in this context will be resolved by the pattern is a matter of "insight, taste, experience, and a sense of aesthetics." However, the third part of the pattern definition, that of applying the software configuration that resolves the forces, is clearly a potential candidate for automation. In chapter 4 we will present a methodology for the development of automated design pattern transformations where the designer defines the context to which the pattern is to be applied and the actual application of the software structure is automated. Other work in the area of automated pattern application is considered in that chapter as well, so in this chapter we focus on other uses of formalisation and automation in the context of design patterns.

2.2.1 Formalisation of Design Patterns

Anthony Lauder and Stuart Kent argue that existing pattern descriptions suffer from being expressed in informal language and being overly-dependent on a specific example to convey the essence of the pattern [56]. They consequently develop a formal three-part model to describe a pattern, viz:

- Role model. This is the most abstract representation of the pattern. The actors involved in the pattern are identified as well as their abstract state and the essential collaborations between them. These definitions are abstract and imply constraints that any refinement of the pattern must respect.

- Type model. This is a refinement of the role model where roles are replaced by domain-specific types that define concrete syntax for operations and add to the abstract semantics of the role model.

- Class model. This final refinement is the actual deployment of the pattern in terms of concrete classes.

In each model, system dynamics can be expressed using a variant of the UML sequence diagram. As each of the three models is formalised in terms of sets and constraints, it has the potential to be used in the development of automated tool support for patterns.

Amnon Eden *et al* have developed a declarative language called LePUS that is specifically geared towards expressing the object-oriented motifs that typically recur in design patterns [33, 32]. In LePUS a program is modelled as sets of entities (classes and methods) and various relationships/collaborations between these entities (inheritance, method invocation, method forwarding etc.). In [33] LePUS is used to describe a set of the Gamma *et al* design patterns [41] and to explore the relationships between patterns.

LePUS has both a graphical format and a textual one that closely resembles Prolog. This latter fact makes it easy to implement a LePUS model as a Prolog facts database and use it in various pattern activities [31]:

- *Validation.* Testing if a certain set of classes/methods fit a certain pattern can be achieved by executing a query with these elements as arguments to the query.
- *Discovery.* To discover an instance of a certain pattern in a model, the query can be executed with variables instead of program elements. This will attempt to match the pattern across the entire database.
- *Application.* Rather than searching for the pattern in the database, the assertions representing the pattern are themselves added to the database[7].

A formal model of patterns certainly has potential to serve as a sound foundation for automated pattern application. Work in this area is ongoing, though as yet few working prototypes have been developed. One exception is the work of Gert Florijn and his group, which is discussed on page 87.

2.2.2 Automated Detection of Design Patterns

Automated detection of design patterns is related to automated design pattern application and has received some attention by researchers. The idea is

[7]Note that in our opinion this work does not fully address the issues involved in pattern application, a position we outline in section 4.5

very tempting: leave an automated tool roam over a large software repository and see what patterns it may find. There is potential to uncover new patterns, or to find known patterns thus enhancing the comprehension of the system.

Kyle Brown developed a tool that reverse engineers Smalltalk programs and can recognise certain design patterns in the code [13]. In the tests he conducted, it found several of the Gamma *et al* patterns [41] with good success. In each case, the pattern structure it detected was later verified to indeed be an instance of the relevant pattern. His case study was quite small so it is hard to draw a firm conclusion from this.

Tonella and Antoniol use concept analysis to identify groups of classes sharing common patterns of relationships, both structural (inheritance and association) and non-structural (method invocation etc.) [98]. Their claim is that these groupings are likely to represent design patterns that are present in the code. In a case study, their approach successfully identified several instances of the well-known Adapter pattern, and also aided in identifying a domain-specific pattern related to input/output. Of course applying this approach to poorly-written code would more likely uncover poor patterns rather than good patterns.

Jahnke and Zündorf propose a method precisely for the identification of poor patterns, with the intention of transforming them to good design patterns[8] [49]. They use Generic Fuzzy Reasoning Nets (GFRNs) to describe the poor pattern structure that is to be transformed. Because it is "fuzzy," the description does not define one precise structure, but a more vague set of structures that indicate that a certain pattern should be applied. The poor pattern identification tool is intended to be used interactively: the user

[8]Their novel approach to pattern application is discussed in section 4.5.

identifies where they suspect a poor pattern to be and the GFRN uses fuzzy inference to assess if the user is correct. They give an example of using their approach to detect a set of global variables to which the Singleton pattern could be applied, but otherwise this innovative work does not appear to have been developed further.

Keller *et al* have developed the SPOOL environment for the reverse-engineering of C++ code [52]. This is a collection of off-the-shelf tools (parsers, browsers, layout generators etc.) that are combined to produce an environment that can provide several abstract views of a software system. In [52] SPOOL is used to recognise patterns during the process of reverse engineering. They argue that rather than simply extracting a design from source code, the rationale behind this design must also be uncovered[9]. Some patterns can be recognised in a purely automatic way, while some require user intervention. In [87] SPOOL is also used for the detection of hot spots in a framework.

Considering pattern detection in terms of the three-part definition of pattern given above, we see that fully automated approaches can only ever deal with recognition of pattern structure. Pattern structure is insufficient in exact design pattern recognition as the pattern structure may be present, but not dynamic relationships or the intent. Also, several patterns have the same pattern structure, and it is only the non-structural characteristics that differentiate between them. Apart from the first approach above (that of Brown), all the pattern recognition and detection work operates in a semi-automatic way, where the user is involved in the process as well. This again brings the "insight, taste, experience, and a sense of aesthetics" into play and means

[9]Extracting rationale as well as architecture is also the major theme in the work of Woods *et al* [100].

that full pattern recognition is possible.

2.2.3 Patterns in Reengineering, Reverse Engineering and Evolution

Automated introduction of design patterns has a clear application in reengineering. In making a system more flexible to cope with future developments, introduction of a design pattern is a likely task to undertake. There can also be patterns in the actual process of evolution and reengineering itself, and it is this work that we look at in this section.

Foote and Opdyke propose a nascent pattern language to describe the process of developing usable software [37]. The topmost pattern, "Develop Software that Is Usable Today and Reusable Tomorrow," gives rise to three patterns on the next layer:

- "Prototype a First-Pass Design."
- "Expand the Initial Prototype."
- "Consolidate the Program to Support Evolution and Reuse."

Their work focuses then on further patterns that form part of the consolidation pattern, ultimately leading to the low-level refactorings proposed by Opdyke [77]. Although not explicitly mentioned, the pattern "Apply a Design Pattern" would be part of consolidation as well, and this thesis provides automated support for this process.

Demeyer, Ducasse and Nierstrasz propose a pattern language for reverse engineering [24]. They subdivide these patterns into four clusters:

- First Contact: what to do when first approaching an unknown software system.

- Initial Understanding: how to obtain a preliminary understanding of the software system, mainly based on class diagrams.
- Detailed Model Capture: how to obtain a detailed understanding of (part of) the software system.
- Prepare Reengineering: since reverse engineering is normally a precursor to reengineering, this cluster of patterns shows how to prepare for subsequent reengineering.

The patterns developed include the self-explanatory "Read all the Code in One Hour" and "Recover the Refactorings," which aims to recover what the original developers learned during the iterative process of development. This pattern language expresses the reverse engineering expertise developed by the authors over several years of academic and practical experience, and so reflects a classic use of the pattern approach. In relation to this thesis, the focus is on reverse engineering rather than software evolution or reengineering.

Stevens *et al* argue that one of the main reasons why software reengineering research has had little impact on software reengineering practice is the difficulty in communicating the research results to the practicing community [92, 26]. They consequently propose *system reengineering patterns* as an approach to package and transfer this expertise. For example, the Deprecation pattern captures the well-established practice of updating an unsatisfactory interface by defining the new interface but also leaving the existing interface intact. A "deprecated" flag is added to the old interface, advising users to move to the new one in preference. In time, the unsatisfactory deprecated interface can be removed. As argued in section 1.2, this thesis can also be viewed as providing automated support for the reengineering process.

2.3 Thesis Context

This thesis merges the two strands of research described in this chapter. Program restructuring (section 2.1) is used in order to automate the application of design patterns (section 2.2) to an existing program. This merging is timely, as program restructuring research has suffered from the lack of a firm basis for deciding what sort of structures it should be targetting. Design patterns are solutions that have proven their worth in practice, and so provide an excellent domain in which to find such target structures.

The existing work in program restructuring is inadequate for our purposes. It is only that of Roberts [84] that deals with a rigorous approach to refactoring composition. However he only allows compositions that are simple sequences of refactorings, and many design pattern transformations are too complicated to be described this way. Accordingly we have extended his method in several ways, the principle one being that we allow a set iteration construct in the definition of a composite refactoring.

It is also clear that existing design pattern work is not sufficient for our purposes. Building a restructuring that applies a design pattern leads us to consider questions about the pattern that have not been addressed in existing work. Firstly, it must be decided what the starting point of the transformation should be, i.e., what type of program structure the transformation can be applied to. Secondly, the commonality between design patterns must be identified and exploited in the development of the transformations, to avoid the wholesale duplication in the transformation definitions that would occur otherwise.

2.4 Summary

In this chapter we have described the two principle research fields upon which this thesis is founded: program restructuring and design patterns. The aim of this is to provide a general background to existing and ongoing research in these areas. In subsequent chapters we present our own contributions in more detail, and also present detailed analysis of our approach in comparison to closely related work.