Automated Application of Design Patterns: A Refactoring Approach

A thesis submitted to the

University of Dublin, Trinity College,

for the degree of

Doctor of Philosophy.

Mel Ó Cinnéide, B.Sc., M.Sc.,

Department of Computer Science,

Trinity College,

Dublin.

October 2000

Thesis
6126

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this or any other University, and that unless otherwise stated, it is entirely my own work.

Mel Ó Cinnéide

Mel Ó Cinnéide

October 2000

Permission to lend or copy

Thesis Summary

Software systems have to be flexible in order to cope with evolving requirements. However, since it is impossible to predict with certainty what future requirements will emerge, it is also impossible to determine exactly what flexibility to build into a system. Design patterns are often used to build this flexibility into a program, so this question frequently reduces to whether or not a particular design pattern should be applied to the program. The original programmer faces this dilemma, and the maintenance programmer must work with the consequences of the decision made.

We address this problem by developing a methodology for the construction of automated transformations that introduce design patterns to an existing program. This enables a programmer to safely postpone the application of a design pattern until the flexibility it provides becomes necessary.

Our methodology deals with the issues of reuse of existing transformations, preservation of program behaviour, and the application of the transformations to existing program code. We apply the methodology to the Gamma *et al* pattern catalogue [41], and find that in almost 75% of cases a satisfactory transformation is developed, and that considerable reuse of existing transformations is achieved.

Acknowledgements

First and foremost I wish to thank my supervisor, Professor Paddy Nixon, for his clear advice and encouragement during this project. His enthusiasm for research is infectious and I left every meeting with him full of ideas and energy for my work.

I am grateful to the other members of the Distributed Systems Group in Trinity College Dublin who accommodated my pursuing a research topic tangential to the main interests of the group, and especially for contriving to make ECOOP 2000 in Cannes a very memorable conference!

Much of this work was carried out in the Department of Computer Science, University College Dublin, and I wish to thank my colleagues there for creating such a pleasant and stimulating environment for conducting research work. I am especially grateful to the departmental heads during my time working on this thesis: Dr. Michael Sherwood-Smith and Professor Mark Keane. Both contributed greatly by subventing my fees, and refactoring my lecture load at crucial times.

Dr. Nick Kushmerick, Dr. Ronan Reilly and Dr. Neil Hurley proofread various parts of the final thesis and I thank them for valuable feedback.

Thanks to my friends for either cajoling me into action when I needed it, or just being totally unaware of my tribulations and being fun to be with!

Finally, thanks to my parents, for everything.

Contents

List of Figures

Chapter 1

Introduction

Getting a design right first time is impossible. One of the major advances in software development thinking in the past decade has been the notion that the process of building a software system should be an evolutionary one [10, 81, 48, 3]. Rather than the classical waterfall model where analysis is fully completed before design, and design fully completed before implementation, evolutionary approaches are based on building a simple version of what is required and extending this iteratively to build a more complicated system. As John Gall put it:

> "A complex system that works is invariably found to have evolved from a simple system that worked." [40, p.50]

Or in Kent Beck's inimitable style:

> "Start stupid and evolve." (quoted in [96])

We are interested in developing a particular type of automated transformation to provide support for software evolution. In section 1.1 we explain more exactly what type of transformations we will focus on and describe this in the context of software evolution. In section 1.2 we show how our approach also addresses problems faced in the reengineering of legacy systems. In section 1.3 we state both the thesis and principle contributions of our work, and finally, in section 1.4, we provide a road map of this dissertation.

1.1 Evolutionary Approaches to Software Development

In an evolutionary approach to software development, a simple working system is built which subsequently undergoes many evolutions until the desired system is reached[1]. At each stage there is a working system which is to be extended with a new requirement or set of requirements. It is very unlikely that the design of the initial system will be flexible enough to elegantly support the later requirements to be added in. Consequently, it is to be expected that when the system is to be extended with a new requirement, its design will also have to be made more flexible in order to accommodate the new requirement elegantly. Current thinking recommends breaking this process of extending a system into two stages [5, 35, 45], [38, p.7]:

1. Program Restructuring: This involves changing the design of the program so as to make it more amenable to the new requirement, while not changing the behaviour of the program.

2. Actual Updating: Here the program is changed to fulfill the new requirement. If the restructuring step has been successful, this step will be considerably simplified.

[1]As remarked in [92], one can never speak of the "final" system. Useful systems tend to evolve continually during their lifetime.

This thesis will present a novel approach to providing sophisticated automated support for the restructuring step.

Let us consider now what type of restructurings a designer may want to perform in order to make a system more flexible and able to accommodate a new requirement. A designer usually has an architectural view of how they wish the program to evolve that is at a higher level than, for example, simply creating a new class or moving an existing method. Probably the most interesting and challenging category of higher-level transformation that a designer may wish to apply comprises those transformations that introduce a *design pattern*[2] [41]. Design patterns typically loosen the coupling between program components, thus enabling certain types of program evolution to occur with minimal change to the program itself. For example, the instantiation of a Product class within a Creator class could be replaced by an application of the Factory Method pattern[3]. This would enable the Creator class to be extended to instantiate a subclass of the Product class without significant reworking of the existing code.

The restructurings we develop in this thesis will be those that automate the introduction of design patterns to an existing object-oriented program. The scenario we consider is as follows: An existing program is being extended with a new requirement. After studying the code and the new requirement, the designer concludes that the existing program structure makes the desired extension difficult to achieve, and that the application of some particular design pattern would introduce the necessary flexibility to the program. It is at this point that we aim to provide automated tool support. The designer selects the design pattern to be applied and the program components that

[2]See section 2.2 for a more detailed description of design patterns.

[3]See appendix A for a description the Factory Method pattern

are to take part in the restructuring, and our tool applies that design pattern to the given program components in such a way that program behaviour is maintained.

A key aspect of this approach is that the intellectual decision of what design pattern to apply, and where to apply it, remains with the designer. We are not attempting to formalise or automate quality; our aim is to remove the burden of tedious and error-prone code reorganisation from the designer. In this thesis we will present and validate a methodology for the development of automated design pattern transformations.

1.2 Legacy Systems

Brodie and Stonebraker provide a widely-accepted definition of a legacy system:

> "[A legacy system is one] that significantly resists modification and evolution to meet new and constantly changing business requirements." [12, p.xv]

Legacy systems frequently require restructuring in order to make them more amenable to changes in requirements. This restructuring is performed either by hand, or through the use of automated tools, for example, [6]. In the latter case, the designer usually specifies certain operations to be carried out, for example, to extract a method from existing code or to move a method from one class to another, and the tool handles the mundane details of performing the transformation itself.

There are clear similarities between a designer restructuring a program that is still under development as described in the previous section, and the restructuring of a legacy system. In both cases the following conditions exist:

- A new requirement (or requirements) has arisen that the program must fulfill.
- The structure of the program is not flexible enough to accommodate the new requirement(s) easily and elegantly.
- The existing program exhibits useful behaviour that must be maintained by any reorganisation that takes place.

The similarity between the forward engineering scenario and the restructuring of a legacy system becomes even clearer when the following points are considered:

- The notion of a legacy system usually evokes an image of an aged system developed with now-defunct technology. However, in the above definition there is no mention of age; a week-old program developed using the latest technology can perfectly fit the definition of a legacy system.
- An evolutionary-centric development methodology such as Extreme Programming[4] can be viewed as actually encouraging the creation of a series of legacy systems. Little up-front design is performed, so with each new requirement that is added, the program is restructured just enough to elegantly accommodate the new requirement.

The conclusion is that evolutionary software engineering and legacy systems reengineering are not such different processes. The design pattern transformations described in this thesis are applicable in both cases.

[4]Extreme Programming is discussed further on page 60.

1.3 Thesis and Contributions

In the last two sections we described how introducing design patterns to a program is part both of forward software engineering and of reengineering of a legacy system. The fundamental thesis of this work can be stated as follows:

> *Automating the application of design patterns to an existing program in a behaviour preserving way is feasible.*

The following are the principle contributions of this thesis:

- *A methodology for developing design pattern transformations.* This is the essential contribution of this work. The methodology we have developed has been applied with full rigour to seven common design patterns[5], and a prototype software tool has been built that can apply these seven design patterns to Java programs[6]. The methodology has also been applied to the remaining patterns in the Gamma *et al* pattern catalogue [41], though these pattern transformations have not been prototyped. The essence of our methodology has been published in summary form in [74, 72], and more completely in [75].

- *A minitransformation library.* Design pattern transformations have a strong degree of commonality and this has been captured in a set of six minitransformations. These minitransformations have been implemented and demonstrated to be widely applicable in developing design pattern transformations.

[5]The seven design patterns to which the methodology has been fully applied are Abstract Factory, Factory Method, Singleton, Builder, Prototype, Bridge and Strategy [41].

[6]We have used Java as the vehicle language for this work. The possibility of language independent approaches is discussed on page 165 in section 6.2.

- *A model for behaviour-preservation proofs.* The transformations we develop must be invariant with respect to program behaviour. In order to prove this rigorously for the sophisticated program transformations that we develop, we have extended existing refactoring work by allowing the transformation definition to contain not only simple sequences, but also iteration and conditional statements. This model has been applied in full rigour to several examples, and has been published in [76].

1.4 Thesis Outline

This thesis is structured as follows:

Chapter 1 (this chapter) introduces the topic of automated design pattern transformations and places it in the context of evolutionary approaches to software engineering and legacy system reengineering.

Chapter 2 describes in detail the background to this work, namely program restructuring and design patterns. Note that research that is very directly related to our work is discussed in the relevant later chapter.

Chapter 3 presents our approach to demonstrating that a program transformation preserves the behaviour of the program and applies it in full rigour to a realistic example.

Chapter 4 describes our methodology for the development of automated design pattern transformations by applying it in detail to a single flagship example.

Chapter 5 applies the methodology to the entire Gamma *et al* design pattern catalogue [41] and analyses the results.

Chapter 6 contains our overall conclusions and presents future work in the area of automated design pattern transformations.

Appendix A contains a description of the Factory Method design pattern, which is the subject of chapter 4.

Appendix B contains the complete specification of all analysis functions, helper functions and primitive refactorings that are used in this work.

Appendix C describes briefly the minitransformations that we developed, and provides a reference to the more detailed description in the main text.

Appendix D describes the architecture of the software prototype developed in this work and presents an example of its application.

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.

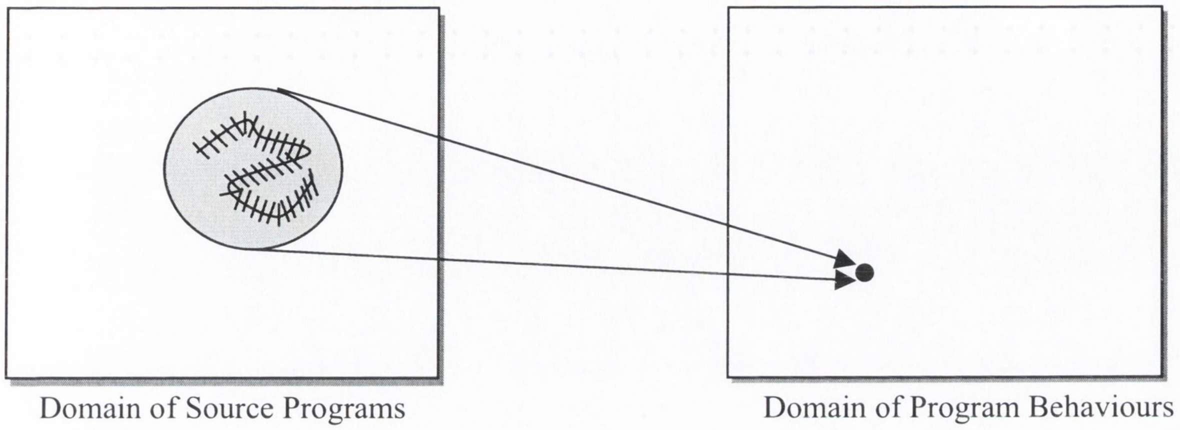


Figure 2.1: Graphical Image of Refactorings

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.

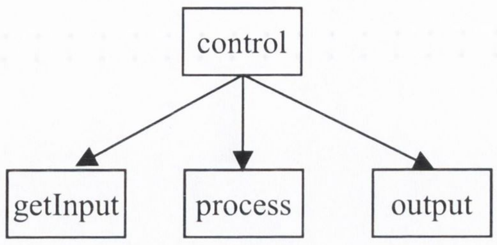


Figure 2.2: A Generic Structure Chart

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.

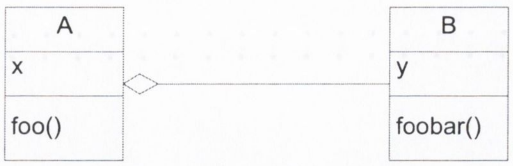


Figure 2.3: A Generic UML Class Diagram

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.

Chapter 3

Foundations of Refactoring: Behaviour Preservation

In the previous chapter we described the notion of behaviour preservation and hinted at the approach that will be adopted in this thesis. In this chapter we present our approach to demonstrating behaviour preservation in detail and apply it with full rigour to a concrete transformation.

In section 3.1 we describe our approach to defining primitive refactorings, stating their pre- and postconditions, and arguing behaviour preservation. In section 3.2 a method for the derivation of the pre- and postconditions of composite refactorings is presented and applied to a concrete example. In section 3.3 our approach is compared to other work in the field and finally, in section 3.4, the results of this chapter are summarised. The approach presented in this chapter has been published in [76].

3.1 Primitive Refactorings and Behaviour Preservation

A *primitive refactoring* is a refactoring that is not decomposed into simpler refactorings. Our transformation approach is based upon a layer of primitive refactorings. Section 3.1.4 describes how we define a primitive refactoring, while in appendix B.3 a list of the actual primitive refactorings used in this work is provided.

As stated previously, it is necessary in defining a primitive refactoring to state what the precondition of the refactoring is. In defining this precondition, assertions are made about the program, for example, that a certain class exists or a given name is not already in use. We define a set of *analysis functions* to enable these assertions to be made. Analysis functions are described further in section 3.1.2.

In developing higher-level refactorings we frequently need to extract certain information from the program, for example, to build an interface from a class based on the signatures of its public methods. This type of function does not affect the program in any way, but performs a more significant task than what an analysis function does. These functions are referred to as *helper functions* and are elaborated further upon in section 3.1.3.

Certain general assumptions are made about the program being transformed and these are described in section 3.1.5. Also, the mathematical preliminaries for this chapter are described in section 3.1.1.

3.1.1 Mathematical Preliminaries

We use the following notation based on [62], also used in [84]. This will be used extensively in section 3.2 where it will be necessary to be precise about the effect of a refactoring on a program.

- P: This is the program to be refactored.
- $\mathcal{I}_P$: Denotes an interpretation of first-order predicate logic where the universe of discourse comprises the program elements of P, and the functions and predicates of the calculus reflect the analysis functions as applied to the program P.
- $\models_{\mathcal{I}_P} pre_R$: Denotes the evaluation of the precondition of the refactoring R on the program interpretation $\mathcal{I}_P$.
- $post_R(\mathcal{I}_P)$: Denotes the program interpretation $\mathcal{I}_P$, rewritten with the postcondition of the refactoring R.
- $f[x/y]$: Denotes an analysis function that is precisely the same as the analysis function f, except that it maps the element x to y. This syntax is used in postconditions to describe the effect of the refactoring on the analysis functions. Note that the name of a new analysis function produced as the result of applying a refactoring is written with a prime ($'$), so stating that an analysis function f is updated with the new element (x, y) would be written thus: $f' = f[x/y]$.
- $\perp$: Is used in a postcondition to mean an undefined value. For example, if a transformation removes a method called m, the updating of the $classOf$ analysis function to indicate that m no longer belongs to any class would be written thus: $classOf' = classOf[m/\perp]$.

3.1.2 Analysis Functions

Analysis functions serve two related roles in our work. Firstly, they are used as functions and predicates in the first-order predicate calculus expressions that define the precondition of a refactoring. Secondly, they are implemented as actual operations that can be applied to a Java program to extract some information about the program, for example, to test if a method is in a certain class or to find the signature of a given method. The relationship between these two roles is that the latter is the implementation of the interpretation of the former. We will simply speak of "analysis functions" and rely on the context to make it clear whether we are referring to a function in first-order predicate logic, or a concrete operation, or both.

The analysis functions used in this work are defined in appendix B.1. There are also dependencies between the analysis functions and these are described in appendix B.1.1. For example, if one class inherits from another class, the type of the former class must also be a subtype of the type of the latter class. In computing the precondition of a composite refactoring in section 3.2, it will be necessary to make use of these dependencies.

Some of the analysis functions are obviously easy to evaluate, for example, the $classOf$ function that tests if a method is a member of a class. Others are more difficult, and a number of them are generally undecidable. In the latter case, there are three possible ways the situation can be dealt with:

1. *An implementation may not be necessary.* Some analysis functions are only used in a precondition when a previous refactoring has already established the condition. This type of analysis function will appear in precondition specifications, and in behaviour preservation arguments, but the necessity for an implementation will never arise. An example of this is the $createsSameObject$ analysis function, that tests if a given method and constructor return identical objects given the same arguments. It is necessary to implement a refactoring (in fact $makeAbstract$, a *helper function*, see section 3.1.3) that sets up this

condition, but this is a straightforward task.

2. *A conservative estimation can be made.* For some undecidable analysis functions a useful conservative estimation exists. For example, the $uses(method1, method2)$ analysis function that determines if $method1$ may invoke $method2$ can only be determined precisely by using an expensive dynamic analysis of the program. However, a conservative estimation that probably includes some false positives can be easily made based on the program text.

3. *The programmer may be queried.* Asking the programmer to assess if a given precondition holds is not an unreasonable approach. They would have to make this assessment anyway were they to perform the refactoring by hand, so their workload is not being increased. Indeed, this approach encourages them to think about program conditions that they might otherwise have overlooked.

Program Entities

In describing a refactoring or its precondition, it is necessary to refer to various program elements: classes, methods, interfaces etc. The principle elements that we make use of, and their interrelationships, are depicted as a UML class model in figure 3.1. Other program entities that are used in defining refactorings and analysis functions are: Interface, Argument, ObjectReference, Field, Parameter, Expression, Variable and MethodInvocation.

For any entity X, we also define an entity SetOfX that represents a set of entities of the type X. Note that for purposes of brevity, a program entity and its name may be used interchangeably. For example, a refactoring that operates on a *Class* may instead be passed a *String* that represents a class

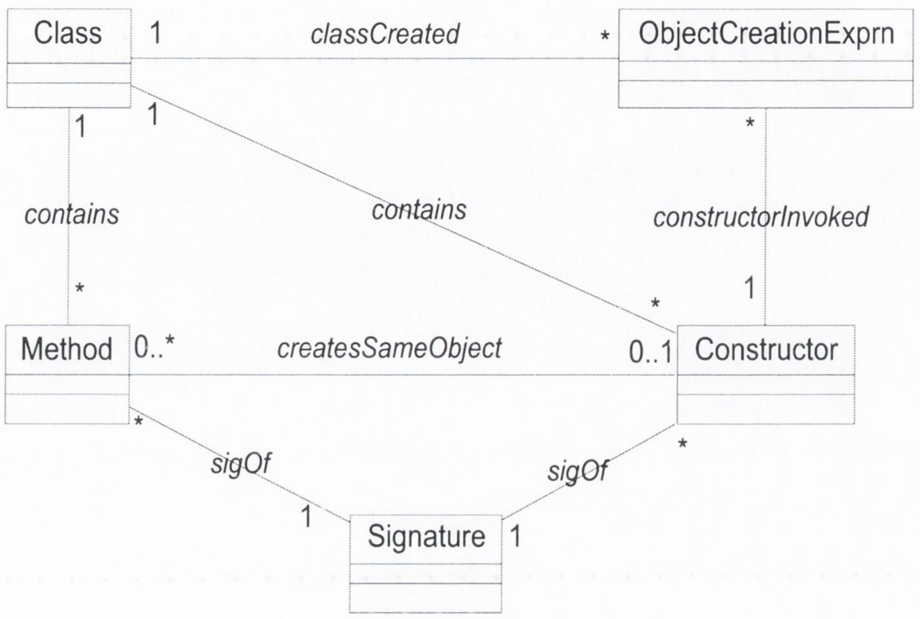


Figure 3.1: Principal Program Entities and their Relationships

name. *getClass*(*String*) could be used to make this relationship precise, but this adds unnecessary bulk to the descriptions.

3.1.3 Helper Functions

In describing a refactoring it may be necessary to extract richer content from the program code than is provided by the analysis functions. For example, we may wish to build an interface from a class based on the signatures of its public methods. Helper functions are used to perform this type of task. Because they are not at the primitive level of the analysis functions, we provide them with a pre- and postcondition. Helper functions are proper functions without side-effects on the program, so the postcondition invariably involves the return value of the helper function itself. The complete list of helper functions used in this work is presented in appendix B.2.

3.1.4 Primitive Refactorings

The aim of this work is to develop composite refactorings that introduce design patterns, not to develop a complete set of primitive refactorings as such. For this reason, we have not defined refactorings that we assessed might transpire to be useful; rather we have defined a new refactoring only when the need for it arose. The complete list of refactorings used in this work is presented in appendix B.3. Some of them are standard and would be part of any refactoring suite, for example, *addClass*. Others are idiosyncratic and quite peculiar to the present work, for example, *replaceObjCreationWithMethInvocation*, which replaces a given object creation expression with an invocation of a given method using the same argument list.

Each primitive refactoring is described in the following way:

- *Name, return type, argument types and informal description*: The return and argument types may be *boolean* or *void*, or one of the program entities described in section 3.1.2. Name and informal description are self-explanatory.

- *Precondition*: This is an assertion, written in first-order predicate logic, that must be true in order for the refactoring to be behaviour preserving. If a precondition fails, and the transformation is nevertheless performed, the resulting program may not be legal Java or may behave differently from the original program.

- *Postcondition*: This is a mapping from analysis functions to analysis functions. It describes the effect of applying the refactoring in terms of changes to the analysis functions defined in appendix B.1.

- *Behaviour preservation argument*: Opdyke [77] presents behaviour preservation arguments in terms of seven program properties that he proposes are easily violated during refactoring[1]. We take a similar approach, but rather than limiting the properties that are maintained to a fixed few, we consider what properties can possibly be violated by each individual refactoring and argue that they are not. The arguments are non-formal in style and cannot guarantee that no behaviour violations occur, but they are rigorous and are intended to be stronger than the argument a programmer would typically make internally were they to perform the refactoring by hand. A key advantage to our approach is that the behaviour preservation argument is made only once by the creator of the primitive refactoring, and need not be repeated by the many programmers who will apply the transformation in practice.

3.1.5 Assumptions and Limitations

It is assumed that certain constraints hold on the Java programs that are transformed in this work. The assumptions we make are as follows:

1. The initial program must compile correctly. If this was not the case, then, for example, the refactoring *addMethod* could change the program behaviour by causing an illegal program to become a legal one.

2. Reflective programs cannot be transformed safely using the approach in this work. For example, the following code invokes a method called foo() on object obj:

   ```
   obj.getClass().getMethod("foo",null).invoke(obj);
   ```

   It is clear that if the program is transformed to rename the method

[1]Tokuda and Batory use an approach based on Opdyke's, and point out that at least three more program properties are necessary to maintain program behaviour [96].

foo, this code excerpt will not execute as before, but will throw an exception.

3. We have assumed that objects are only created using the new operator. The issues surrounding object cloning have not been dealt with in detail in this work[2].

4. Private classes are not considered. We disallow the creation of a new class if its name clashes with an existing class, even if the existing class is private and no real clash exists.

5. Packages are not dealt with in this work, so a class or interface can be safely identified by just its name.

6. The interface to a method is described by its name, return type, and parameter types. Exceptions also form part of the interface to a method, but for simplicity we have ignored them in this work.

The first two constraints are fundamental to our approach, the third involves an issue that we have not yet addressed, while the last three are simplifications that would be burdensome to do without, but are not essential to our approach.

3.2 Composite Refactorings

The ultimate goal of this work is to use the refactorings, helper functions, and analysis functions described in the last section to define behaviour preserving

[2]For example in a new expression, the class of the created object is given explicitly. However, in a clone expression, the class of the created object is not known statically, but depends on the type of the receiving object. This would be an issue when designing transformations for creational patterns, as they have an impact on how objects are created.

design pattern transformations. As will be presented in the next chapter, the process of constructing a design pattern transformation is essentially a top-down one, but there is also an element of bottom-up composition of existing refactorings. In this section we describe the way in which we compose refactorings, and present a technique for computing the pre- and postconditions of a composite refactoring. The importance of these techniques lies in the fact that they allow us to implement a design pattern transformation as a composition of refactorings and then to check the legality of the composition and calculate its overall precondition.

We could avoid the necessity of calculating the overall precondition of a composite refactoring by checking the precondition for each component refactoring just before it is applied. If a precondition fails, we simply rollback to the starting point and inform the user. This approach is undesirable whether the composition is legal or illegal:

- If the composite refactoring is legal, testing its precondition will normally be faster, and never slower, than testing the precondition of each component refactoring separately.

- If the composite refactoring is illegal, testing its precondition will be considerably faster than applying several of the component refactorings and then being obliged to rollback to the starting point. Note that some refactorings are not undoable, so supporting rollback would involve checkpointing.

Since we aim to build refactorings statically, the program P is not available for a "try it and see if it works" approach. No assumptions can be made about P, other than those described in section 3.1.5.

In our work, we have discovered that there are two ways in which we need to compose refactorings:

1. Chaining.

2. Set iteration.

Chaining is where a sequence of refactorings are applied one after the other. For example, the following chain adds methods foo and goo to the class c.

```
addMethod(c,foo)
addMethod(c,goo)
```

Set iteration is where a refactoring or a refactoring chain is performed on a set of program elements. For example, the following set iteration copies all the methods of the class a to the class b.

```
ForAll m:Method, classOf(m)=a {
    addMethod(b,m)
}
```

Other forms of composition are possible as well of course, the most obvious one being a selection statement. Although this is straightforward to deal with, it is omitted here as we have found that in the construction of design pattern transformations in this work, chaining and set iteration suffice.

3.2.1 Computing Pre- and Postconditions for a Chain of Refactorings

A chain of refactorings may be of any length, but we can simplify the computation of its pre- and postconditions by observing that we need only solve the problem for a chain of length 2. This procedure can then be iteratively applied to the remaining chain until the full pre- and postconditions have been computed. For a chain of length n, n-1 applications of this process will be required.

The two refactorings to be composed are referred to as R_1 and R_2. For a general refactoring R_i, its precondition and postcondition are denoted by pre_{R_i} and $post_{R_i}$ respectively. See figure 3.2.

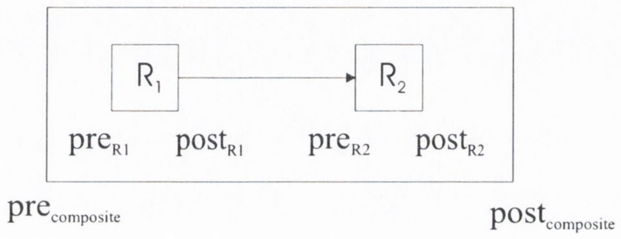


Figure 3.2: A Composite Refactoring with Pre- and Postconditions

The naïve approach to computing the precondition is simply to logically AND the preconditions, i.e.,

$$pre_{composite} = pre_{R_1} \wedge pre_{R_2},$$

however there are several problems with this. Firstly, $post_{R_1}$ may guarantee pre_{R_2} which means that an unnecessarily strong precondition results (or indeed typically a contradictory precondition), for example,

```
addClass(c)
addMethod(c,m)
```

ANDing the preconditions produces, among other clauses, $\neg isClass(c) \wedge isClass(c)$, even though this chain is perfectly correct. The source of this contradiction lies in the fact that the two preconditions should be valid at different points in the transformation.

Secondly a composition may be simply illegal, e.g., deleteClass(c)

addMethod(c,m)

ANDing the preconditions here gives simply $isClass(c)$ even though this chain is illegal! Although the precondition for addMethod is valid at the start of the chain, deleteClass breaks it so this composition of refactorings can never be legal.

The precondition of the chain is computed first[3]. During this computation it may emerge that the chain is in fact illegal. If the chain is legal, the postcondition is then computed. We describe how these computations are performed in the following two subsections.

Legality test and precondition computation

Assuming the chain is legal, its precondition is obtained by logically ANDing pre_{R_1} with whatever parts of pre_{R_2} that are not guaranteed by $post_{R_1}$. The parts of pre_{R_2} that are not guaranteed by $post_{R_1}$ are obtained by evaluating:

$$\models_{post_{R_1}(\mathcal{I}_P)} pre_{R_2}$$

If a contradiction arises in this evaluation, the chain is illegal. The post-condition of the first refactoring sets up a condition that contradicts the precondition to the second refactoring.

The precondition of the complete chain is obtained by evaluating:

$$pre_{R_1} \wedge \models_{post_{R_1}(\mathcal{I}_P)} pre_{R_2}$$

A contradiction can arise in this evaluation as well, and this also means that the chain is illegal. In this case the precondition to the first refactoring

[3]It is valuable to compute the precondition first, because if the chain requires a stronger precondition than simply pre_{R_1}, it can be useful to use this stronger condition in later computations.

demands a certain condition that contradicts the precondition to the second refactoring, and the first refactoring does not change this condition.

Postcondition computation

In our approach[4] a postcondition is described as a set of updates to analysis functions in the following form:

$$f' = f[x/y]$$
$$g' = g[p/q]$$
$$\dots$$

Any analysis function not mentioned in the postcondition is implicitly not affected by the refactoring.

The postcondition of a refactoring chain is obtained by concatenating the function updates described in the postconditions. For example, if $post_{R_1}$ contains the mapping:

$$classOf' = classOf[foo/c_1]$$

and $post_{R_2}$ contains the mapping:

$$classOf' = classOf[foo/c_2]$$

then naturally $classOf' = classOf[foo/c_2]$ becomes part of the postcondition of the chain. Denoting this concatenation operation as $|$ we state the postcondition of the chain to be:

$$post_{R_1} \mid post_{R_2}$$

Table 3.1 describes how this operator works in general.

A complete example of the application of this algorithm is given in section 3.2.3.

[4]I am grateful to Dr. John Boyland of the University of Wisconsin for pointing out problems in my original approach to postcondition computation.

$post_{R_1}$	$post_{R_2}$	$post_{R_1} \mid post_{R_2}$
$f' = f[x/y]$	$g' = g[p/q]$	$f' = f[x/y] \quad g' = g[p/q]$
$f' = f[x/y]$	$f' = f[p/q]$	$f' = f[x/y][p/q]$
$f' = f[x/y]$	$f' = f[x/z]$	$f' = f[x/z]$

Table 3.1: Concatenation of Postconditions

3.2.2 Computing Pre- and postconditions for a Set Iteration

A set iteration has the following format:

```
ForAll x:Entity, Pred(x,...) {
    R(x,...)
}
```

where *Entity* is some type of program entity, *Pred* is some predicate and "..." denotes the program entities that are arguments to the predicate and/or arguments to the refactoring. If the set of x of type *Entity* that satisfies *Pred*$(x, \ldots)$ is given as $\{x_1, x_2, \ldots, x_n\}$, and writing R_i as a shorthand for $R(x_i, \ldots)$, then this iteration may be viewed as the following chain:

$R_1,\ R_2,\ \ldots,\ R_n$

However this is a *set* iteration, so the refactorings could take place in any order. That is to say, they must be able to commute and this fact enables us to define when a set iteration is legal and what its pre- and postconditions should be.

1. *Legality test*: A set iteration is illegal if the precondition of any component refactoring depends on the postcondition of another component

refactoring. It is also illegal if the postcondition of any component refactoring contradicts the precondition of another component refactoring[5]. Both these conditions are captured by requiring that for any refactoring R_i in the set iteration, the evaluation of the precondition is not affected by the prior application of any sequence of $R_j, j \neq i$. We express this using the notation of section 3.1.1 as:

$$\begin{array}{l} \forall R_i, i \in \{1..n\}, \models_{\mathcal{I}_P} pre_{R_i} = \models_{\mathcal{I}_{P'}} pre_{R_i} \\ \text{where } P' = post_{R_{jm}}(\ldots post_{R_{j2}}(post_{R_{j1}}(\mathcal{I}_P))), \\ j_m \in \{1..n\} - \{i\}, j_x = j_y \Rightarrow x = y \end{array}$$

Roberts [84] looks at the issue of commutativity of general refactorings in detail, however we are only concerned with the constrained case of set iterations. A very conservative approach to take is to demand that the postcondition of a component refactoring in a set iteration should not refer to the analysis functions used in its precondition. This has transpired to be too constraining to be of use, so it will often prove necessary to examine the semantics of the iteration performed to ascertain if the above property holds. The legality test performed on page 50 is an example of this.

2. *precondition computation*: Any of the R_i could be the first in the chain. Since the precondition of the first refactoring of a chain must form part of the precondition for the whole chain, the precondition of the set iteration must be at least the ANDing of the preconditions of each of the component refactorings. Nothing stronger is required, so the

[5]The component postconditions could be allowed to contradict each other. However the postcondition notation would have to be extended to allow disjunction between the function updates.

precondition for the above chain can be expressed as:

$$\bigwedge_{i=1}^{i=n} pre_{R_i}$$

or in a more useful form as:

$$\forall x : Entity, Pred(x) \bullet pre_{R(x,\ldots)}$$

3. *postcondition computation*: By a similar argument, the postcondition for the above chain can be expressed as:

$$post_{R_1} \mid post_{R_2} \mid \ldots \mid post_{R_n}$$

We have described how pre- and postconditions can be computed for refactoring sequences and set iterations. In the next section we apply these techniques to a non-trivial example.

3.2.3 A Worked Example

In this section we take a typical composite transformation that involves both chaining and set iterations and compute its pre- and postconditions. The calculations are performed in all detail in this example, but in future we will only summarise the derivation.

The example we use is the algorithm that describes how to apply the EncapsulateConstruction minitransformation[6]. The purpose of this minitransformation is to loosen the binding between one class (*creator*) and another class that it instantiates (*product*). It does this by adding new construction methods to the *creator* class that perform the creation of *product* objects. Each new method is given the name *createP*, and all expressions that

[6]Minitransformations are described in detail in section 4.3. For the purposes of the current chapter, they may be thought of simply as composite refactorings.

create *product* objects in the *creator* class are updated to use the appropriate construction method. The impact of applying this minitransformation is that extending the *creator* class to work with a new type of *product* class is simply achievable by subclassing *creator* and overriding the *createP* method.

The algorithm for this minitransformation is defined as follows using the analysis functions, helper functions and refactorings described in earlier sections:

```
EncapsulateConstruction(Class creator, Class product, String createP){
    ForAll c:Constructor, classOf(c)=product {
        Method m = makeAbstract(c, createP);
        addMethod(creator, m);
    }
    ForAll e:ObjectCreationExprn, classCreated(e) = product $\wedge$
      containingClass(e) = creator $\wedge$
      nameOf(containingMethod(e)) $\neq$ createP {
        replaceObjCreationWithMethInvocation(e, createP);
    }
}
```

Computing the pre-and postconditions of this composite refactoring proceeds in several steps:

1. Compute *pre* and *post* for the chain in the first set iteration body

2. Compute *pre* and *post* for the first set iteration

3. Compute *pre* and *post* for the second set iteration

4. Compute *pre* and *post* for the overall chain

Computing *pre* and *post* for the chain in the first set iteration body

1. *Legality test and precondition computation*: This involves first rewriting the precondition of addMethod(creator, m) with the postcondition of makeAbstract(c, createP):

$$\begin{aligned}&\models_{post_{makeAbstract}(\mathcal{I}_P)} pre_{addMethod}\\ &= isClass(creator) \wedge \neg defines(creator, nameOf[m/createP](m), sigOf(m))\\ &= isClass(creator) \wedge \neg defines(creator, createP, sigOf(m))\end{aligned}$$

   and then ANDing this with the precondition for Method m = makeAbstract(c). The latter is simply true, so the final precondition for this chain is:

$$isClass(creator) \wedge \neg defines(creator, createP, sigOf(m)) (3.1)$$

   No contradiction occurred so the chain is legal.

2. *postcondition computation*: There is no analysis function updated in both $post_{addMethod}$ and $post_{makeAbstract}$ so we can simply concatenate the postconditions to obtain:

$$\begin{aligned}&createsSameObject' = createsSameObject[(c, m)/true]\\ &nameOf' = nameOf[m/createP]\\ &classOf' = classOf[m/creator]\\ &\forall a : Class, a \neq creator \bullet equalInterface(a, creator) \Rightarrow\\ &equalInterface' = equalInterface[(a, creator)/false] \quad (3.2)\end{aligned}$$

Computing *pre* and *post* for the first set iteration

1. *Legality test*: On first glance the postcondition of the body of this iteration (3.2 above) appears to have no impact on the precondition (3.1 above). However from appendix B.1.1 we know that

$$classOf(m) = creator \wedge nameOf(m) = createP$$
$$\Rightarrow defines(creator, createP, sigOf(m))$$

and this may contradict the second conjunct of 3.1. This would only occur if there were two methods m with the same signature. However, m is a method whose signature is derived from iterating through the constructors of the *product* class. Since no two constructors in the same class can have the same signature, neither can two methods in the set iteration have the same signature. This means that the value for *sigOf(m)* will vary on each iteration so there is no risk that the precondition will be violated.

2. *precondition computation*: On every iteration, the precondition must be true, i.e.,

$$isClass(creator) \wedge \neg defines(creator, createP, sigOf(m))$$

must be valid for every constructor processed. The first conjunct is not affected by the iteration, so it simply becomes part of the precondition of the iteration. The second conjunct presents a problem as m is only calculated in the body of the iteration and so cannot be used in the precondition. However, $sigOf(m)$ is the same as the signature of the constructor being processed, so we can write the precondition as:

$$isClass(creator) \wedge \forall c : Constructor,\ c \in product \bullet$$
$$\neg defines(creator, createP, sigOf(c)) \qquad (3.3)$$

This precondition ensures that no method called *createP* already exists in the *creator* class with a signature that matches any of the constructors of the *product* class. If for practical reasons we prefer not to allow a method called *createP* to exist in the *creator* class at all, then this simpler precondition may be used:

$$isClass(creator) \wedge \neg defines(creator, createP)$$

3. *postcondition computation*: The postcondition for the body of this iteration is given in (3.2) above. The iteration creates a new m each time, so the full postcondition is:

$$\begin{aligned}
&\forall c : Constructor, c \in product \bullet \exists m : Method \text{ such that}\\
&\qquad createsSameObject' = createsSameObject[(c, m)/true]\\
&\qquad nameOf' = nameOf[m/createP]\\
&\qquad classOf' = classOf[m/creator]\\
&\forall a : Class, a \neq creator \bullet equalInterface(a, creator) \Rightarrow\\
&equalInterface' = equalInterface[(a, creator)/false] \quad (3.4)
\end{aligned}$$

Computing *pre* and *post* for the second set iteration

1. *Legality test*: The postcondition of the refactoring

   replaceObjCreationWithMethInvocation(e,createP)

   is that e is deleted, i.e.,

   $containingMethod' = containingMethod[e/\perp]$.

   This can only have an impact on the precondition[7]

   $createsSameObject(constructorInvoked(e), createP) \wedge$

   $containingMethod(e) \neq createP$

[7]Where there is a disjunctive in the precondition as here, it may be clear that only one of the disjuncts is relevant and we can safely choose that one to work with. In this case $returnsSameObject(constructorInvoked(e), m) \wedge hasSingleInstance(product)$ is dropped in favour of $createsSameObject(constructorInvoked(e), m)$. The dropped disjunct relates to the very rare case where the *product* class is only instantiated once.

if e refers to the same object creation expression. However, the set iteration processes each *product* creation expression in the class *creator*, so e will refer to a different expression on each iteration. This set iteration is therefore legal.

2. *precondition computation*: For each object creation expression processed in the iteration, there must be a suitable method called *createP* defined in the *creator* class:

$$\begin{aligned}
&\forall e : ObjectCreationExprn, classCreated(e) = product \wedge \\
&containingClass(e) = creator \wedge \\
&nameOf(containingMethod(e)) \neq createP \bullet \\
&\exists m : Method, nameOf(m) = createP, defines(creator, m) \text{ such that} \\
&createsSameObject(constructorInvoked(e), m) \qquad (3.5)
\end{aligned}$$

   Note that the precondition conjunct $containingMethod(e) \neq m$ is dropped as this is guaranteed by the fact that $nameOf(m) = createP$ and $nameOf(containingMethod(e)) \neq createP$.

3. *postcondition computation*: All the *product* creation expressions in the *creator* class that are not in a method called *createP* have been removed:

$$\begin{aligned}
&\forall e : ObjectCreationExprn, classCreated(e) = product \wedge \\
&containingClass(e) = creator \wedge \\
&nameOf(containingMethod(e)) \neq createP \bullet \\
&containingMethod' = containingMethod[e/\perp] \qquad (3.6)
\end{aligned}$$

Computing *pre* and *post* for the overall chain

1. *Legality test and precondition computation*: Precondition 3.5 must be rewritten with postcondition 3.4 and the remaining conjuncts made part of the precondition of the whole minitransformation. Before this can be performed, postcondition 3.4 must be massaged to a suitable form.

   Postcondition 3.4 makes a universally quantified statement about all the constructors of the class *product*. For every *product* creation expression in the *creator* class there is a corresponding constructor in the *product* class. We can therefore safely replace the quantification over the constructors of the *product* class with quantification over the *product* creation expression in the *creator* class. If the *product* class has constructors that are not used in the *creator* class, this change will weaken the postcondition. Using a weaker postcondition than is actually guaranteed is fortunately a safe substitution.

   Postcondition 3.4 may therefore be rewritten thus[8]:

$$\begin{array}{l} \forall e : ObjectCreationExprn, classCreated(e) = product, \\ containingClass(e) = creator \bullet \exists m : Method \text{ such that} \\ \qquad createsSameObject' = \\ \qquad createsSameObject[(constructorInvoked(e), m)/true] \\ \qquad nameOf' = nameOf[m/createP] \\ \qquad classOf' = classOf[m/creator] \end{array}$$

   The transformation of the *classOf* relationship may be replaced by a similar transformation to the *defines* relationship (see section B.1.1) to

[8]The final part of the postcondition has been dropped as it is clear that the effect of this refactoring on the *equalInterface* analysis function is irrelevant in this context.

give:

$$\begin{aligned}
&\forall e : ObjectCreationExprn, classCreated(e) = product,\\
&containingClass(e) = creator \bullet \exists m : Method \text{ such that}\\
&\quad createsSameObject' =\\
&\quad createsSameObject[(constructorInvoked(e), m)/true]\\
&\quad nameOf' = nameOf[m/createP]\\
&\quad defines' = defines[(creator, m)/true] \qquad (3.7)
\end{aligned}$$

This postcondition is now in a suitable format to rewrite precondition 3.5 as follows:

$$\begin{aligned}
&\forall e : ObjectCreationExprn, classCreated(e) = product \wedge\\
&containingClass(e) = creator \wedge\\
&nameOf[m/createP](containingMethod(e)) \neq createP \bullet\\
&\exists m : Method, nameOf[m/createP](m) = createP,\\
&defines[(creator, m)/true](creator, m) \text{ such that}\\
&createsSameObject[(constructorInvoked(e), m)/true](constructorInvoked(e), m)
\end{aligned}$$

Simplifying this out gives:

$$\begin{aligned}
&\forall e : ObjectCreationExprn, classCreated(e) = product \wedge\\
&containingClass(e) = creator \wedge\\
&nameOf(containingMethod(e)) \neq createP \bullet\\
&\exists m : Method, true
\end{aligned}$$

This simplifies to just *true*, so in fact the precondition for the second set iteration is fully guaranteed by the postcondition of the first set iteration. This means that the precondition of the second set iteration does not contribute anything to the overall precondition for this minitransformation, so the overall precondition is simply the precondition to the first set iteration, namely precondition 3.3.

2. *postcondition computation*: The postcondition for the first set iteration (3.7) and the second (3.6) are combined as follows:

$$\begin{array}{l} \forall e : ObjectCreationExprn, classCreated(e) = product, \\ containingClass(e) = creator \bullet \exists m : Method \text{ such that} \\ \qquad createsSameObject' = \\ \qquad createsSameObject[(constructorInvoked(e), m)/true] \\ \qquad nameOf' = nameOf[m/createP] \\ \qquad defines' = defines[(creator, m)/true] \\ \forall e : ObjectCreationExprn, classCreated(e) = product, \\ containingClass(e) = creator, \\ nameOf(containingMethod(e)) \neq createP \bullet \\ \qquad containingMethod' = containingMethod[e/\perp] \end{array} \quad (3.8)$$

   Note that the first set iteration adds a construction method to the *creator* class, regardless of whether it used in the *product* class or not. Constructors of the *product* class that are not used in the *creator* class could be omitted from the transformation, but this was not done as it is likely that a future evolution of the program would make it necessary to include them again.

It is interesting to observe that in the overall precondition the *product* class was not required to exist. This is correct, in that the ENCAPSULATE-CONSTRUCTION transformation reduces in this case to the null transformation, which is of course behaviour preserving. However, for this transformation to be *useful*, the *product* class must indeed exist. For this reason we will sometimes add such extra conditions to the precondition of a transformation.

3.2.4 Commentary

We have demonstrated that if precondition 3.3 holds in a given program, then the EncapsulateConstruction transformation can be safely applied without changing the behaviour of the program. Also, in the final program state, postcondition 3.8 will be valid.

The argument was non-trivial and required a considerable amount of effort. However this need only be done once, and then the minitransformation can be added to a library and reused in any number of future design pattern transformations. The existence of this argument enhances our confidence that the transformation is indeed behaviour preserving. If during prototype evaluation it transpires that the implemented transformation is not behaviour preserving, the error can be traced back and, if it is present in the behaviour preservation argument, it may be corrected there.

Constructing the behaviour preservation argument also caused us to give consideration to factors that were not immediately apparent from the minitransformation description. For example, the fact that the *creator* class might already have methods called *createP* and that this is not a problem unless the signature of one of them clashes with the signature of a constructor in the *product* class was made very clear during the computation of the pre- and postconditions.

Finally, this method of arguing behaviour preservation is not formal[9]. First-order predicate logic is used in defining the preconditions and some of the reasoning performed is formal and based purely on the laws of first-order logic. However, it was frequently necessary to use our knowledge of the semantic domain (Java programs) in computing the pre- and postconditions. For example, the transformation of postcondition 3.4 to the more

[9]It is for this reason we avoid using the term "proof" in this chapter.

useful postcondition 3.7 required this knowledge. Since our purpose is to provide a method of argument that reflects in some way how a programmer reasons about a program, this is a valid approach. Were we to attempt to automate the process of computing the pre- and postconditions for a composite refactoring, then this approach would of course need to be strengthened.

3.3 Related Work

Donald Roberts [84, 85] describes a similar approach to computing the pre- and postconditions of a composite refactoring to the one we have presented here. However he does not demand that a refactoring be behaviour preserving[10] [84, p.19] and so does not argue this for his refactorings. The algorithm we present differs from his in several ways:

- it tests if the chain is legal rather than assuming it is [84, p.39];
- it allows set iterations over refactorings and chains;
- it makes use of the relationships between analysis functions[11];
- it computes the postcondition for a composite refactoring, as we intend to use the composite refactoring in further compositions.

Tokuda and Batory use a set of Opdyke-style refactorings in order to build higher-level refactorings [96] and to study the use of refactorings in the evolution of object-oriented programs. A very interesting feature of this work is that they present the first ever case study that actually takes an existing system that has been reengineered, and attempts to perform the

[10] An unfortunate redefinition of an existing term.

[11] Roberts neglects this in his work and, for example, does not identify the relationship between *isClass* and *isGlobal*, i.e., that $IsClass(class) \Rightarrow IsGlobal(nameOf(class))$.

reengineering that took place using a refactoring tool. They estimate that were they to perform the changes involved in the reengineering by hand, it would take them approximately ten times longer than it took them to perform the changes using automated refactorings. This improvement is attributed to the obvious reduction in the amount of manual work required, and the fact that reliable automated refactorings reduce the amount of testing required. This result has provided some concrete evidence favouring the use of automated refactoring approaches.

Schulz [88] proposes arguing behaviour preservation by first transforming a legacy object-oriented program into an adaptive program [61]. This adaptive program can be reasoned about more easily and the transformations performed on this program. Finally the transformed adaptive program is converted back to a non-adaptive program. He does not describe this last conversion and it is not clear that it is feasible. In other work Schulz [90] proposes using Opdyke's approach [77, 51] to prove behaviour preservation of design pattern transformations.

Elbereth is a tool developed for refactoring Java programs [54] that uses the notion of a *star diagram*. A star diagram allows the programmer to easily view all uses of a construct (method, field etc.) across the entire program without having to also view unrelated code. Korman describes how the programmer can be supported in performing a variety of refactoring tasks, such as adding a new subclass or replacing an existing class with an enhanced version. While these tasks are intended to be refactorings, he does not address the issue of arguing that they are behaviour preserving.

Developing the pre- and postcondition for a composite refactoring bears an obvious resemblance to the weakest precondition calculus of Dijkstra's guarded command language [27]. In that approach, if we wish the composition of two transformations T_1 and T_2 to leave the program in the state $post_{composition}$, then the weakest precondition necessary is given by:

$$wp(T_1, wp(T_2, post_{composition}))$$

where $wp(T, post)$ is the weakest precondition that will ensure that the transformation T will leave the program in a state where *post* is true. The aim of this work is that given a postcondition, it should be possible to derive an algorithm (a composition of transformations) that can reach this postcondition, and work out what precondition must hold in the initial state.

The problem we faced in demonstrating behaviour preservation is different. We use postconditions to describe the result of applying a refactoring only in sufficient detail that it is possible to determine what subsequent refactorings are legal. The refactoring itself has a richer meaning, but that is only described informally in the refactoring description and not captured in the formal postcondition. In composing these refactorings, we have a notion of what is to be achieved, and the purpose of the pre- and postcondition computation is to determine whether the composed refactoring is legal, what types of program it can be applied to, and what subsequent refactorings can be legally applied. The possibility of extending this work to the formal derivation of the complete design pattern transformation will be discussed in section 6.2.

Refactoring is a key part of Kent Beck's *Extreme Programming* methodology [3]. Extreme programming requires many rapid iterations through the development process, each time extending the system functionality a little further. As little up-front design is performed, it is necessary to refactor the program whenever a new requirement makes the existing design inadequate. Behaviour preservation is not discussed in this approach, but in effect it is demonstrated through the use of automated corrective regression testing [58]. After refactoring, the programmer runs an automated test suite on the program. If the program produces the same test results as it did before the refactoring, it is concluded that the behaviour of the program has not changed. Obviously this approach is dependent on the completeness on the test suite, and thus can never be fully relied upon.

Test suites are used in a different way to demonstrate behaviour preservation in the Smalltalk Refactoring Browser [11]. For example, in the $renameMethod$ refactoring, all methods that call the renamed method must also be updated. However, in Smalltalk it is impossible to find all the callers of a method statically, so the authors use dynamic analysis to compute this. The program code is instrumented, run on a test suite, and it is calculated from the execution trace what methods called the given method. As in the previous case, this approach is only as effective as the test suite used in the dynamic analysis.

Finally, in a recent text on the topic of refactoring by Martin Fowler [38], only scant attention is paid to the topic of behaviour preservation, and that is in two chapters written by Opdyke and Roberts respectively, whose work has been extensively cited in this chapter. This text does however provide a detailed listing of low-level refactorings that can be performed by hand, and gives useful informal advice on where they should be applied and what steps should be taken to achieve a safe refactoring.

3.4 Summary

In this chapter we presented our approach to defining primitive refactorings and composing these to create more complicated refactorings. Two methods of composition were allowed: sequencing (or chaining), and iteration over a set of program elements. A method for computing the pre- and postconditions of such composite refactorings was also described. This approach to behaviour preservation is undecidable in general, but for the simple preconditions we work with this will prove not to be an issue.

In the next two chapters we will show how these forms of composition can be used to build sophisticated design pattern transformations.

Chapter 4

A Methodology for the Development of Design Pattern Transformations

4.1 Introduction

In this chapter we describe in detail the methodology we propose for the development of design pattern transformations. The motivations for our approach are presented in section 4.1.1 followed by a brief overview of the entire methodology in section 4.1.2. The approach we take in describing the methodology is to describe each part in a general way and then to apply it to one design pattern. The flagship pattern we use is the Factory Method pattern (see appendix A), as it is sufficiently complicated to exercise the methodology and yet yields an elegant result. The details of the methodology appear in sections 4.2 and 4.3, culminating in the final specification of the Factory Method design pattern transformation in section 4.4. In section 4.5 we evaluate related work in the area of design pattern application and finally, in section 4.6, a summary of this chapter is presented.

The essence of the approach presented here has been published in summary form in [74, 72], and in more detail in [75].

4.1.1 Motivation

There are several criteria we wish our methodology to fulfill:

1. The design pattern transformations developed must preserve program behaviour.

2. The transformations are to be applicable to real programs.

3. Reuse of portions of existing transformations should be feasible and encouraged.

4. Judging where a pattern should be applied remains the domain of the programmer.

We expand on these criteria in the following paragraphs.

1. *Behaviour Preservation*

For any form of automated refactoring to be successful in practice, the programmer must have a strong degree of confidence that the transformations being applied do indeed preserve program behaviour [89]. In our approach, we therefore place a heavy emphasis on demonstrating that the design pattern transformations are behaviour preserving. The foundations of our approach to behaviour preservation were introduced in chapter 2 and presented in detail in chapter 3. In this chapter we use these foundations to show how behaviour preservation can be demonstrated for a complete design pattern transformation.

2. *Applicability to real programs*

The transformations developed should be applicable to real programs and be able to cope with the complexities of source code representation of design structures. This is especially important if they are to be used in practice for transforming existing legacy systems, where formal design documentation frequently does not exist. This criterion conflicts to a certain extent with the previous point, in that formally proving complex behavioural properties of programs written in industrial-strength languages is currently impractical. We have resolved this by working with an industrial language, Java, and taking a semi-formal approach to demonstrating behaviour preservation.

3. *Reuse where possible*

Design patterns have a lot in common so it is to be expected that design pattern transformations will have a lot in common as well. In our methodology we seek to decompose the transformations into reusable units and to make these units available to later developments of design pattern transformations.

4. *Programmer controls quality*

One of the pitfalls in attempting to automate patterns is to treat them completely formally and not allow for the fact that their "goodness" is something essentially informal [26]. In section 2.2 we described the design insight necessary to assess what pattern to apply and where to apply it. In our methodology the programmer remains in control of these issues.

4.1.2 Outline of the Methodology

The complete methodology is depicted as a UML activity chart in figure 4.1. Initially a design pattern is chosen that will serve as a target for the design pattern transformation under development. We then consider what

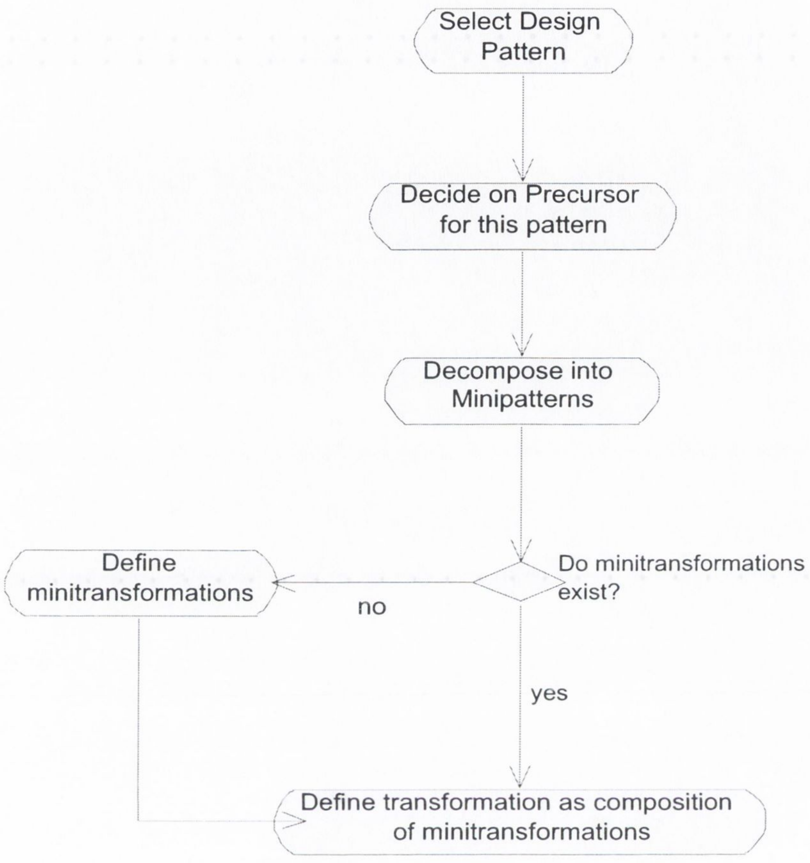

Select Design Pattern
Decide on Precursor for this pattern
Decompose into Minipatterns
Do minitransformations exist?
no
Define minitransformations
yes
Define transformation as composition of minitransformations

Figure 4.1: The Design Pattern Methodology

the starting point for this transformation will be, that is, what sort of design structures it may be applied to. This starting point is termed a *precursor*, which is described in more detail in section 4.2. It has now been determined where the transformation begins, (the precursor) and where it ends (the design pattern itself). This transformation is then decomposed into a sequence of *minipatterns*. A minipattern is a design motif that occurs frequently; in this way it is similar to a design pattern but is a lower-level construct.

For every minipattern discovered a corresponding *minitransformation* that can apply this minipattern must also be developed. A minitransformation comprises a set of preconditions, an algorithmic description of the transformation, and a set of postconditions. The algorithm is expressed in terms of the primitive refactorings and helper functions defined in appendix B. It is built by hand, using the precursor and the design pattern structure as a guide[1]. The pre- and postconditions are computed by applying the method described in chapter 3 to this algorithm.

Minitransformations are our unit of reuse, so for any minipattern identified we first check if a minitransformation for it has already been built as part of the development of a previous design pattern transformation. If so, that minitransformation can be reused now, otherwise a new minitransformation must be developed. Section 4.3 examines minipatterns and minitransformations in more detail, and in particular specifies precisely the minitransformations that comprise the Factory Method transformation.

The final design pattern transformation can now be defined as a composition of minitransformations. The pre- and postconditions for this design pattern transformation are computed in the same way as they are computed for a minitransformation. In the following sections we describe this entire process in full detail, finally providing the complete specification of the Factory Method transformation in section 4.4. In particular, the concepts of *precursor*, *minipattern* and *minitransformation* are discussed in detail.

4.2 Precursors

Much of the existing work on design pattern transformations [14, 30, 36, 96, 55, 9] assumes as a starting point what can be termed a *green field sit-*

[1]By this we simply mean that implementing a minitransformation is similar to the normal process of informal program development, where the program specification has been given rigorously, though not formally.

uation. By this we mean that when the design pattern transformation is applied to the program, the components that take part in the transformation do not already have any existing relationships pertaining to the pattern. Consequently these approaches do not support the breaking of existing relationships as part of the transformation process. From a software evolution perspective this is inadequate because in an existing program the basic intent of the pattern may well exist in the code already, but in a way that is not amenable to further program evolution. For example, in the case of the Factory Method pattern, the Creator class may already create and use instances of a Product class, but not in the flexible manner that allows easy extension to other Product classes.

At the other extreme there is the *antipattern* approach [53, 70], which was investigated in our earlier work [71, 73] and is also used in [25]. In this approach the assumption is made that the programmer has failed to appreciate the need for the pattern in the first instance, and has used some inadequate design structure to deal with the situation. The philosophy behind this approach is that the code may have been developed by a programmer who was not aware of patterns. For example, in the case of the Factory Method pattern, the client of the Creator class may have to configure it with a flag to tell it what type of Product class to create. We discovered several problems with the antipattern approach:

- For any pattern there are several variants and for each variant there can be several antipatterns. The volume of antipatterns rises sharply and each one has to be dealt with individually.

- The design knowledge encapsulated in design patterns has been developed over many decades of software development. A programmer who is "not aware of patterns" and chooses an inappropriate solution

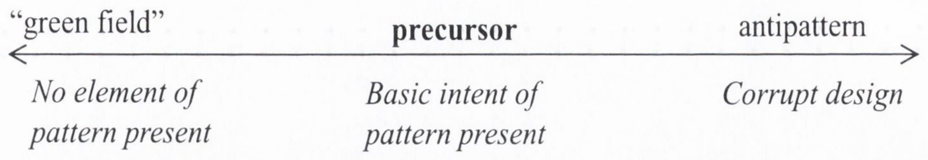

"green field"
precursor
antipattern
No element of pattern present
Basic intent of pattern present
Corrupt design

Figure 4.2: Possible starting points for a Design Pattern Transformation

to a design problem has really just made a mistake[2]. The problem of transforming an antipattern to a design pattern then becomes that of transforming poor design to good design, which cannot of course be solved generically.

For these reasons we use a different starting point for our transformations. For a large class of design patterns, the effect of the pattern may be viewed as making certain program evolutions easier. This suggests that in the simple case the design pattern is not needed, but as future changes in requirements demand greater flexibility from the software, it becomes necessary. For example, it is frequently the case that a class A creates an instance of a class B, but normally this relationship does not require the application of a design pattern. However a future change in the requirements may well require that the class A have the flexibility to work with any one of a number of different subclasses of B, and so the need for the Factory Method pattern arises. The programmer of the original system did not make an error of judgement; software systems will always evolve in ways that the original creators simply cannot foresee[3]. Indeed, applying a design pattern where it is not needed is highly undesirable as it introduces an unnecessary complexity to the system.

[2]The author's position is that a programmer who is faced at some point with the prospect of using an antipattern solution will baulk, and restructure the design in order to enable a more elegant solution.

[3]As Lucy Berlin commented, "Prescience is not an exact science" [8].

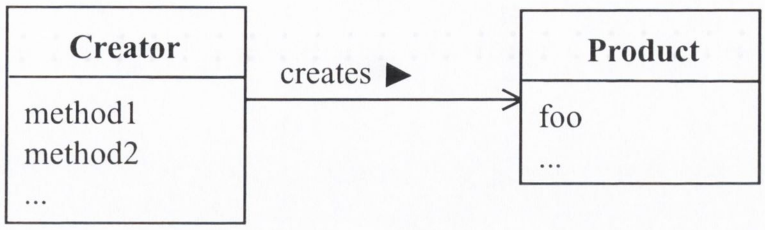


Figure 4.3: Precursor for the Factory Method Transformation

This leads us to our description of a precursor: a precursor is a design structure that expresses the intent of a design pattern in a simple way, but that would not be regarded as an example of poor design. This is not a formal definition, but it serves to exclude both the green field situation where there is no trace of the intent of the pattern in the code, and the antipattern situation where the programmer has tried to resolve the problem in an inadequate way. Figure 4.2 illustrates the relationship between these various starting points.

For example, the precursor we use for the Factory Method pattern is simply this: the Creator class must create an instance of the Product class. This is specified using an analysis function thus:

```
creates(creator, product)
```

Figure 4.3 depicts this precursor in a UML class diagram. This condition may appear to be trivial, but it is a natural precursor to the Factory Method pattern. The Creator class creates and uses an instance of the Product class and while this is adequate for the moment, a new requirement may demand that the Creator class be able to work with other types of Product class and this will require the application of the Factory Method pattern.

4.3 Minipatterns and Minitransformations

In developing a transformation for a particular design pattern we naturally wish to reuse our previous efforts as much as possible. To obtain maximum leverage, this reuse should be at the highest level possible. Examining the design pattern catalogues [41, 15, 43, 44], it is clear that certain motifs occur repeatedly across the catalogues. For example, a class may know of another one only via an interface, or the messages received by an object may be delegated to a component object for detailed processing. These design motifs, or minipatterns, are combined in various ways to produce different design patterns. In this way a pattern can be viewed as a composition of minipatterns. By focusing on developing transformations for minipatterns, we are able to develop a library of useful transformations that can be reused whenever that minipattern is identified again in a later development. The transformation that corresponds to a minipattern is naturally called a minitransformation. In the case of the Factory Method pattern we can identify four component minipatterns:

1. ABSTRACTION: The Product class must have an interface that reflects how the Creator class uses the instances of Product that it creates.

2. ENCAPSULATECONSTRUCTION: In the Creator class, the construction of Product objects must be encapsulated inside dedicated, overrideable methods, which we term construction methods.

3. ABSTRACTACCESS: Apart from within the construction methods described in (2) the Creator class must have no knowledge of the Product class except via the interface described in (1).

4. PARTIALABSTRACTION: The Creator class must inherit from an ab-

stract class where the construction methods are declared abstractly.

This amounts to a declarative description of the structure of the Factory Method pattern. It is obvious that other patterns use some of these minipatterns as well. For example, Abstract Factory uses all of them, while many design patterns make use of the ABSTRACTION minipattern. In the following subsections each of the above minipatterns is taken in turn and processed as follows:

1. A minitransformation for this minipattern is specified in terms of the primitive refactorings and helper functions defined in appendix B;

2. The pre- and postconditions for this minitransformation are computed using the method described in chapter 3.

In appendix C a complete list of all the minitransformations developed in this work is presented, together with a reference to the thesis section where more detail can be found.

4.3.1 The Abstraction Minitransformation

The ABSTRACTION minitransformation is used to add an interface to a class. This enables another class to take a more abstract view of this class by accessing it via this interface. This minitransformation is implemented in the following way as a chain of refactorings:

```
Abstraction(Class c, String newName){
    Interface inf = abstractClass(c, newName);
    addInterface(inf);
    addImplementsLink(c, inf);
}
```

An interface is first created that reflects the public methods of this class[4]. This interface is then added to the program and an implements link is added from the class to this interface.

To demonstrate legality of this chain and to compute its pre- and post-conditions, we apply the method described in section 3.2.1. The computation is straightforward and produces the following:

precondition:

The class c exists:

$\text{isClass}(c)$

No class or interface with the name *newName* exists:

$\neg\text{isClass}(\textit{newName}) \wedge \neg\text{isInterface}(\textit{newName})$

postcondition:

A new interface *inf* called *newName* exists:

$\text{nameOf}' = \text{nameOf}[\textit{inf}/\textit{newName}]$

$\text{isInterface}' = \text{isInterface}[\textit{inf}/\text{true}]$

The class c and the interface *inf* have the same public interface:

$\text{equalInterface}' = \text{equalInterface}[(c,\textit{inf})/\text{true}]$

An implements link exists from the class c to the interface *inf*:

$\text{implementsInterface}' = \text{implementsInterface}[(c,\textit{inf})/\text{true}]$

The effect of applying this minitransformation to the Factory Method precursor (figure 4.3) is depicted in figure 4.4. An interface has been added that provides an abstract view of the Product class.

[4]The new interface created here reflects the entire public interface of the class, even though all that is required are the parts of the public interface that are actually used in whatever context is going to use this interface. However, if this context happens not to use an essential part of the class, this transformation would result in the creation of an unintuitive interface. A consequence of our approach is that the declared type of some variables will be broader than how they are actually used.

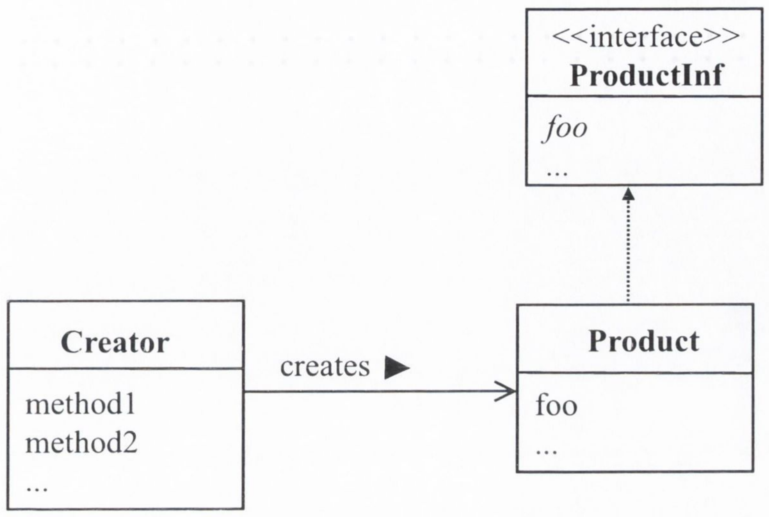


Figure 4.4: Application of the ABSTRACTION Minitransformation

4.3.2 The EncapsulateConstruction Minitransformation

This minitransformation is used when one class creates instances of another, and it is required to weaken the binding between the two classes by packaging the object creation statements into dedicated methods. It was already considered in great detail in section 3.2.3. The algorithm is given on page 49, so here we simply restate, with some extra supporting text, the pre- and postconditions.

EncapsulateConstruction(Class *creator*, Class *product*, String *createProduct*)
precondition:
The class *creator* exists:

isClass(*creator*)

The *creator* class defines no methods called *createProduct* that have the same signature as a constructor in the class *product*:

$\forall$ c:Constructor, c $\in$ *product* •

$\neg$ defines(*creator*, *createProduct*, sigOf(c))

postcondition:

For every *product* object creation expression in the *creator* class, a method called *createProduct* that creates the same object is added to the *creator* class:

$\forall$ e:ObjectCreationExprn, classCreated(e) = *product*,

containingClass(e) = *creator* $\bullet$ $\exists$ m:Method such that

createsSameObject′ =

createsSameObject[(constructorInvoked(e),m)/true]

nameOf′ = nameOf[m/*createProduct*] $\wedge$

defines′ = defines[(*creator*,m)/true]

Every *product* object creation expression in the *creator* class that is not contained in a method called *createProduct* is deleted:

$\forall$ e:ObjectCreationExprn, classCreated(e) = *product*,

containingClass(e) = *creator*,

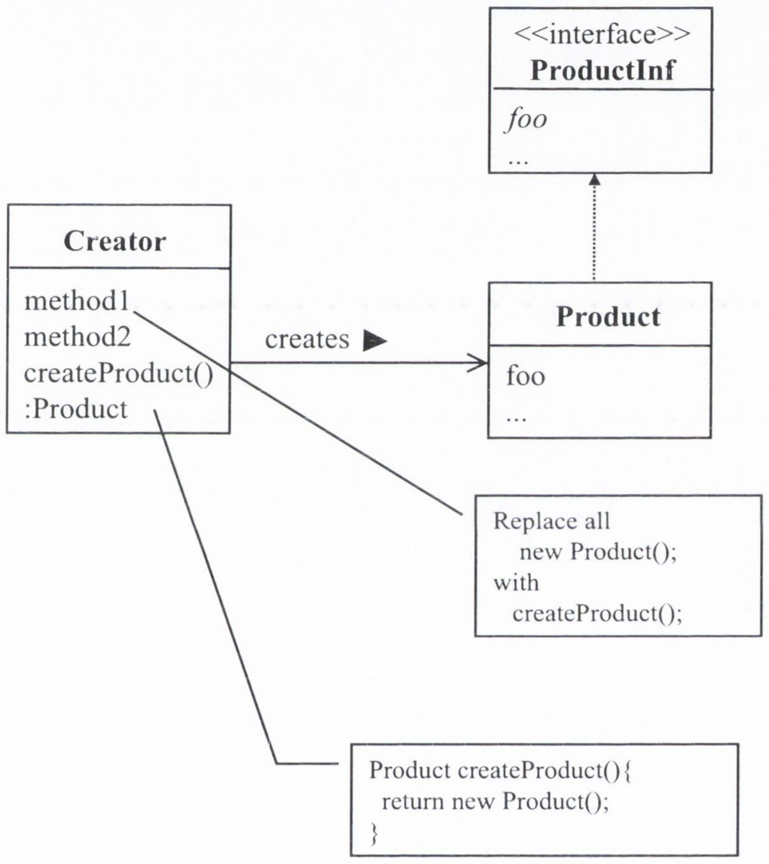
nameOf(containingMethod(e)) $\neq$ *createProduct* $\bullet$

containingMethod′ = containingMethod[$e/\perp$]

Applying this minitransformation to the structure depicted in figure 4.4 results in the structure depicted in figure 4.5. For each constructor of the Product class, a method of the same signature has been added to the Creator class that returns the same object as the corresponding constructor. All creations of Product objects in the Creator class have been updated to invoke these methods instead.

4.3.3 The AbstractAccess Minitransformation

The AbstractAccess minitransformation is used when one class (*context*) uses, or has knowledge of, another class (*concrete*) and we want the relation-



```
Replace all
   new Product();
with
  createProduct();
```

```
Product createProduct(){
  return new Product();
}
```

Figure 4.5: Application of the ENCAPSULATECONSTRUCTION Minitransformation

ship between the classes to operate in a more abstract fashion via an interface. It may well happen that there are methods in the *context* class that need to access the *concrete* class directly, for example, they may instantiate the *concrete* class, and these methods should be excluded from the transformation. This minitransformation is implemented in the following way as a set iteration:

```
AbstractAccess(Class context, Class concrete, Interface inf,
                    SetOfString skipMethods){
    ForAll o:ObjectRef, typeOf(o)=concrete, containingClass(o)=context,
    nameOf(containingMethod(o)) $\notin$ skipMethods {
        replaceClassWithInterface(o,inf);
    }
}
```

This minitransformation takes each object reference in the class *context* that is of the type *concrete*, excluding any references that are contained in any method called *skipMethods*, and changes their existing type from the class *concrete* to the interface *inf*. Applying the method described in section 3.2.2, the pre- and postconditions are computed to be[5]:

precondition:

The interface *inf* and the classes *context* and *concrete* exist:

$$\text{isInterface}(\mathit{inf}) \wedge \text{isClass}(\mathit{context}) \wedge \text{isClass}(\mathit{concrete})$$

An implements link exists from the class *concrete* to the interface *inf*:

$$\text{implementsInterface}(\mathit{concrete}, \mathit{inf})$$

Any static methods in the *concrete* class are not referenced through any of the object references to be updated:

[5]The *isClass*(*context*) part of the precondition is added to avoid the transformation reducing to the null transformation, as described on page 56.

$\forall$ m:Method, m $\in$ *concrete*, isStatic(m) $\bullet$

$\forall$ o:ObjectRef, typeOf(o)=*concrete*, containingClass(o)=*context* $\bullet$

$\neg$ uses(o,m)

Any public fields in the *concrete* class are not referenced through any of the object references to be updated:

$\forall$ f:field, f $\in$ *concrete*, isPublic(f) $\bullet$

$\forall$ o:ObjectRef, typeOf(o)=*concrete*, containingClass(o)=*context* $\bullet$

$\neg$ uses(o,f)

postcondition:

All references to the *concrete* class in the *context* class not in *skipMethods* have been changed to refer instead to the interface *inf*:

$\forall$ o:ObjectRef, typeOf(o)=*concrete*, containingClass(o)=*context*, nameOf(containingMethod(o)) $\notin$ *skipMethods* $\bullet$

typeOf$'$ = typeOf$[o/inf]$

The initial conjuncts of the precondition simply ensure that referenced classes and interface exist and have the proper relationship. The last two conjuncts ensure that if the *concrete* class has public fields or static methods, these are not used by any of the object references to be updated. We present a complete categorisation of preconditions in section 4.4.1.

Applying this minitransformation to the structure depicted in figure 4.5 results in the structure depicted in figure 4.6. In the Creator class all references to the Product class have been replaced by references to the Product interface.

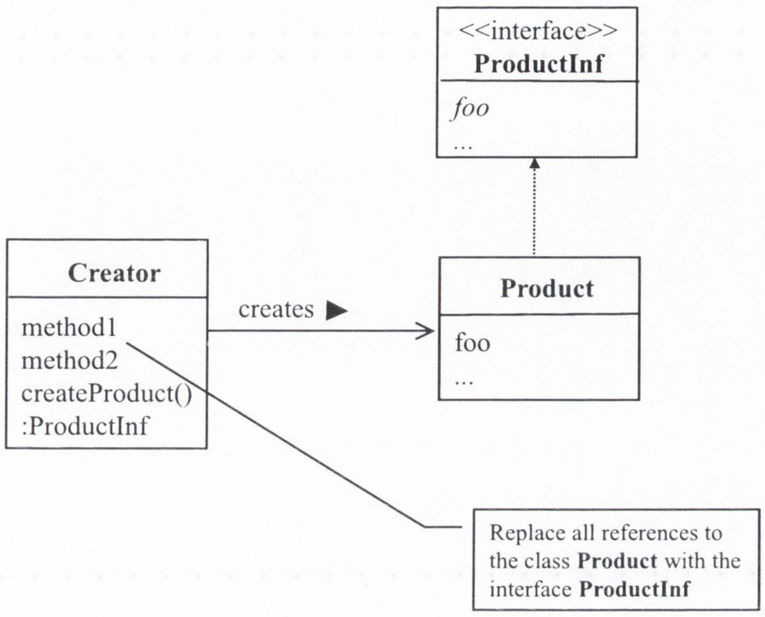


Figure 4.6: Application of the AbstractAccess Minitransformation

4.3.4 The PartialAbstraction Minitransformation

The PartialAbstraction minitransformation is used to construct an abstract class from an existing class and to create an extends relationship between the two classes. It is related to the Abstraction minitransformation of section 4.3.1, but rather than building a completely abstract interface from the class, it builds an abstract class where only certain specified methods are declared abstractly. This minitransformation is implemented in the following way:

```
PartialAbstraction(Class concrete, String newName,
                   SetOfString abstractMethods){
    Class abstract = createEmptyClass(newName);
    addClass(abstract, superclass(concrete), concrete);
    ForAll m:Method, m $\in$ concrete, nameOf(m) $\in$ abstractMethods{
            Method absMethod = abstractMethod(m);
            addMethod(abstract, absMethod);
        }
        ForAll m:Method, m $\in$ $concrete$, nameOf(m) $\notin$ $abstractMethods${
            pullUpMethod(m);
        }
    }
```

This minitransformation creates an empty class called *newName* and inserts it into the inheritance hierarchy just above the class *concrete*. For each method in *abstractMethods*, an abstract method is created and added to this new class. Any methods not in *abstractMethods* are moved from the class *concrete* to this new class. By inspection we see that although the preconditions for the addClass refactoring and the second set iteration are quite complicated, most of the conjuncts are guaranteed by the fact that the new superclass of *concrete* is the empty class that has just been added. Note also that since every method being pulled up into this new class comes from the same class, there can be no name clashes between these methods. The same argument applies to the abstract methods that are added to the superclass. The pre- and postconditions are thus computed to be:

precondition:

No class or interface with the name *newName* may exist:

$\neg$ isClass($newName$) $\wedge$ $\neg$ isInterface($newName$)

The *concrete* class must exist:

isClass($concrete$)

Any fields used by methods that are to be pulled up must not be public:

$\forall$ f:Field, m:Method, f $\in$ $concrete$, m $\in$ $concrete$, m $\notin$ $abstractMethods$ $\bullet$

if uses(m,f) then $\neg$ isPublic(f)

postcondition:

A new class called *newName* exists:

$\text{isClass}' = \text{isClass}[\textit{newName}/\text{true}]$

An extends link exists from the class *concrete* to the class called *newName*, and from the class called *newName* to the former superclass of *concrete*:

$\text{superclass}' = \text{superclass}[\textit{concrete}/\textit{newName}][\textit{newName}/\text{superclass}(\textit{concrete})]$

The class *concrete* and its new superclass define precisely the same type:

$\text{equalInterface}' = \text{equalInterface}[(\textit{concrete}, \text{superclass}'(\textit{concrete}))/\text{true}]$

All methods in *concrete* not in *abstractMethods* are moved to the superclass:

$\forall$ m:Method, m $\in$ *concrete*, m $\notin$ *abstractMethods* $\bullet$

$\text{classOf}' = \text{classOf}[m/\text{superclass}'(\textit{concrete})]$

Any method in *abstractMethods* will have an abstract method declared in the class called *newName*:

$\forall$ m:Method, m $\in$ *abstractMethods* $\bullet$

$\text{declares}' = \text{declares}[(\text{superclass}(\textit{concrete}), \text{m}, \textit{direct})/\text{true}]$

Any fields used by the moved methods are also moved to the superclass:

$\forall$ m:Method, m $\in$ *concrete*, m $\notin$ *abstractMethods* $\bullet$

$\forall$ f:Field, f $\in$ *concrete*, uses(m,f) $\bullet$

$\text{classOf}' = \text{classOf}[\text{f}/\text{superclass}(\textit{concrete})]$

Applying this minitransformation to the structure depicted in figure 4.6 results finally in the Factory Method structure depicted in figure 4.7. An abstract Creator class has been added that defers the definition of the construction methods to its subclasses. The original Creator class simply inherits this class and provides definitions for the construction methods.

We have considered four minitransformations and shown how they can be applied in sequence to produce the Factory Method design pattern structure. We examine this complete design pattern transformation in more detail in

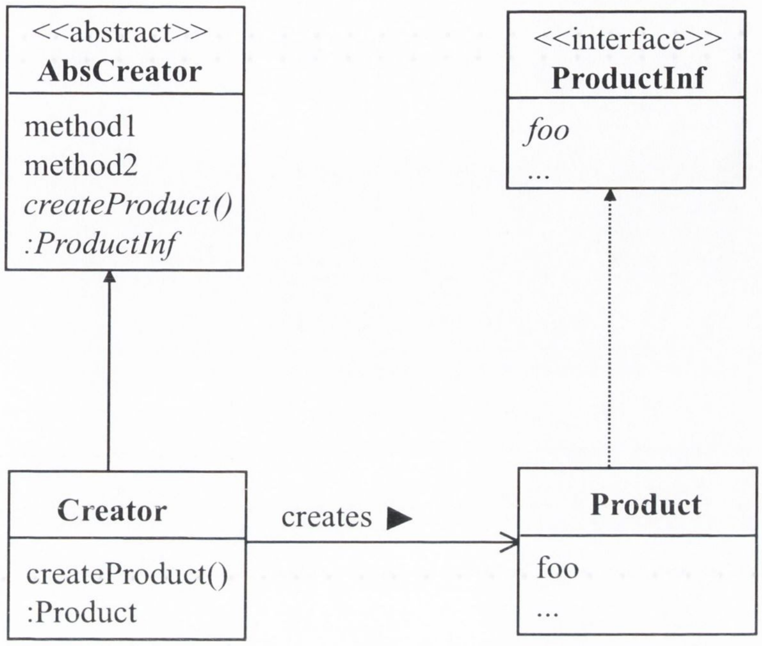


Figure 4.7: Application of the PARTIALABSTRACTION Minitransformation

the next section.

4.4 The Factory Method Transformation

The transformation that introduces the Factory Method pattern is defined simply as the sequential application of the minitransformations defined in the preceding sections:

```
applyFactoryMethod(Class creator, Class product, String productInf,
                   String absCreator, String createProduct){
    ABSTRACTION(product, productInf);
    ENCAPSULATECONSTRUCTION(creator, product, createProduct);
    ABSTRACTACCESS(creator, product, productInf, createProduct);
    PARTIALABSTRACTION(creator, absCreator, createProduct);
```

}

Applying the method described in section 3.2.1, we compute the precondition of this transformation to be:

precondition:

1. The classes *creator* and *product* exist:

isClass(*creator*) $\wedge$ isClass(*product*)

2. No class or interface called *absCreator* or *productInf* exists:

$\neg$isClass(*absCreator*) $\wedge$ $\neg$isInterface(*absCreator*) $\wedge$

$\neg$isClass(*productInf*) $\wedge$ $\neg$isInterface(*productInf*)

3. In the *creator* class there are no methods called *createProduct* that have the same signature as a constructor in the class *product*:

$\forall$ c:Constructor, c $\in$ *product* $\bullet$

$\neg$ defines(*creator*, *createProduct*, sigOf(c))

4. The *creator* class can create instances of the *product* class:

creates(*creator*, *product*)

5. Public fields in the *product* class are not referenced through any of the *product* object references in the *creator* class:

$\forall$ f:field, f $\in$ *product* $\bullet$ $\forall$ o:ObjectRef, typeOf(o)=*product*,

containingClass(o)=*creator* $\bullet$ $\neg$ uses(o,f)

6. Any fields in the *product* class used by methods in that class must not be public:

$\forall$ f:Field, m:Method, f $\in$ *concrete*, m $\in$ *concrete*, uses(m,f) $\bullet$

$\neg$ isPublic(f)

7. Any static methods in the *product* class are not referenced through any of the *product* object references in the *creator* class:

$\forall$ m:Method, m $\in$ *product*, isStatic(m) $\bullet$

$\forall$ o:ObjectRef, typeOf(o)=*product*, containingClass(o)=*creator* $\bullet$

$\neg$ uses(o,m)

Note that we do not compute the postcondition for a design pattern transformation itself. This may appear to be a useful task, as the result could help to capture the essence of the pattern in a formal way. However, recall that we are using pre- and postconditions only as an aid to demonstrating behaviour preservation. While the postcondition for a design pattern transformation would provide a notion of what is true after a pattern is applied, it would not be strong enough to provide real insight into the essential nature of the pattern itself. It would however be very useful as part of a tool that maintains facts and constraints about the program, and this is discussed further in section 6.2.

In appendix D, we present an example of the Factory Method transformation being applied to a sample Java program.

4.4.1 A Categorisation of the Preconditions

The preconditions for the Factory Method transformation can be divided into four categories. The first three preconditions simply ensure that the classes referred to in the parameters to this transformation exist and that the names for the new program entities to be introduced by this transformation do not clash with any existing names. These preconditions are *trivial* but are necessary to ensure that the transformation operates correctly. If one of them fails the programmer need only be requested to choose a different name to replace the offending choice.

The fourth precondition is the key *precursor* precondition. This describes the essence of the starting point for the transformation, as depicted in figure 4.3. It implies that there is a tight binding between the Creator class and the

Product class and this is what the application this pattern is to ameliorate. In general, if a precursor precondition fails, it is of questionable value to continue with the transformation. In the Factory Method example, the transformation can continue, but it is effectively a green field beginning then, and some of the transformations performed will be needless. Note that this precursor precondition was added by hand rather than being the result of computing the precondition of the chain of minitransformations.

The fifth and sixth preconditions are examples of *refactoring* preconditions. Failure of one of these indicates that there are minor problems that prevent the transformation from being applied. The Product class has public data fields, which are a well-established example of poor class design [82]. This prevents the transformation from being performed as public fields cannot be accessed through an interface. If the programmer agrees, this class can be refactored automatically[6] to make this data private or protected and instead to provide access to the offending fields via public accessor and mutator methods. This then removes this obstacle to the application of the transformation. See section 6.2 for further consideration of the possibility of such pre-transformation refactorings.

The final precondition is termed a *contraindication* and failure here indicates that there is a more serious problem in applying the Factory Method pattern. The Product class has a static method that is used by the Creator class. This implies that the Creator class depends on the actual class of the Product it uses and this cannot be replaced by access via an abstract interface. This is an inherent problem in the design of the program that prevents the application of the pattern transformation. In this case the design must

[6]The refactoring used here would be an automated version of the EncapsulateField refactoring described in [38, p.206].

be revisited by the programmer to determine if it is possible to resolve this issue.

4.4.2 Assessing the Factory Method Transformation

We already stated that we regard the transformation for the Factory Method pattern as valuable. In this section we highlight why it is good, that is, what criteria we used in making this assessment:

1. The precursor is *plausible*. By this we mean that it is likely to occur in practice. It is not a bizarre structure, but is one that a programmer would typically use in developing an initial prototype, when their focus is more on correct operation than reuse.

2. The precursor is *strong* in that it captures the essence of where this transformation should be applied. The transformation also made good use of the precursor in terms of providing a behaviour-preserving transformation. The precursor states that one class instantiates another and the transformation made the nature of the instantiation more flexible while not affecting its behaviour.

3. There was significant reuse of minitransformations. This transformation simply used four minitransformations and required no other intervening refactorings. We will see in chapter 5 that this is an unusually simple result. In the general case we can expect to have to add some "glue" refactorings between the minitransformations, in order to ensure that the preconditions for each minitransformation are valid.

4. The transformation is elegant and compelling. This is a matter of judgement of course, but the transformation is certainly straightfor-

ward and it is not difficult to see that its effect is indeed to apply the Factory Method pattern.

4.5 Related Work

In the previous chapter related work in the area of behaviour preservation was evaluated. In this section we consider other work specifically in the area of the automated application of design patterns.

Florijn, Meijers and van Winsen have developed a patterns tool that provides a broad range of support for a programmer working with patterns [36, 64]. Their focus is on the representation of design patterns within the tool itself, and the maintenance of the constraints associated with a design pattern, i.e., checking that changes to the program do not violate any of the design patterns present in the code. Their work also deals with pattern application, but the starting point of their transformations is the green field situation, so the issues of behaviour preservation and reorganisation of existing relationships as part of the transformation process do not arise.

Recent work by Tokuda and Batory has shown how design patterns can be automatically applied to a C++ program [96, 97]. They use a set of refactorings similar to Opdyke's set and show how they can be used to construct design pattern transformations. Whereas we build static composite refactorings and compute the full precondition for the composition, their approach assumes that the programmer is inspecting the code and applying each refactoring in turn. Minitransformations are not used in their work and a green field starting point is assumed. As in the previous work cited, this latter point means that behaviour preservation is not a significant issue in their work, and their transformations have quite a different flavour from ours.

Yehudai, Gil and Eden [30] have developed a prototype tool called the patterns wizard that can apply a given design pattern to an Eiffel program. This work is related to ours in that it takes a metaprogramming approach and organises the transformations into four levels: design pattern, micro-pattern (similar our minipatterns), idioms (our refactorings) and abstract syntax tree. The starting point they use is the green field situation, rather than attempting to deal with a precursor as we do. This makes the patterns wizard unsuitable for reengineering certain types of program that our approach can handle. If the programmer has already partially introduced the intent of the pattern to the code, using the patterns wizard to apply this pattern will leave an amount of manual work for the programmer to do in order to bring the program to a consistent state. As a consequence of taking a green field approach, behaviour preservation is not so important and is more or less ignored in their work. The micropatterns developed in their work are used in the specification of several design pattern transformations. However, they are at a lower-level that the ones we have identified; for example, of the four minipatterns we used to define the Factory Method transformation, only one, ABSTRACTION, appears in Eden's catalogue [34]. This is partly a consequence of our taking a precursor as the starting point for our transformations: certain minipatterns are necessary in our approach that would not be needed otherwise.

Yehudai, Gil and Eden have also developed a declarative language called LePUS for formally specifying the structural and behavioural aspects of design patterns [33]. They propose that this can be developed into a tool that applies a design pattern by adding the required LePUS pattern definition to the program specification. This is true in the abstract LePUS domain, but there are many issues to be resolved in transforming this abstract specification into executable code. At the time of writing practical results in this area are not evident in their published work.

Automatically applying design patterns to a UML model has been explored by Sunyé, Le Guennec and Jézéquel [94]. The approach described here takes a metaprogramming approach as we do, and also argues that it is the programmer that should decide on the application of a pattern while a software tool is best used to help in performing the actual transformation. This work naturally focuses on the design level, so issues of code transformation do not occur and behaviour preservation is not emphasized. The paper mentions the notion of a composite refactoring, but describes neither how composition can take place, nor a method for computing the pre- and postconditions for a composite refactoring.

The work of Schultz and Zimmer is also related to what we have presented here [89, 101]. They merge Opdyke's refactoring work with so-called design pattern operators to produce behaviour-preserving transformations that introduce design patterns. Their published work to date presents only their initial ideas.

Jahnke and Zündorf describe an approach to detecting poor design patterns and transforming them to good design patterns [49]. The detection aspect of their work is discussed in chapter 2, so here we focus on the pattern application part. They also use a notion similar to our precursor (a "naïve solution" they term it) as a starting point, based on the suggested naïve solutions in the Gamma *et al* catalogue [41]. They only present one example, the Singleton pattern, and choose the same starting point as we do on page 128, namely a collection of global variables[7]. In their work the

[7]We also present a Singleton transformation that uses a different precursor in section 5.3.1

design pattern structure is stored at a conceptual level, together with a prototypical implementation of the pattern, a scheme that is similar to that used by Florijn [36]. This scheme is more flexible than ours, in that the transformation tool can be easily configured with a new pattern. However our approach, by developing a collection of minitransformations, effectively builds a high-level language for describing design pattern transformations. This allows a pattern transformation to be described abstractly, without having to explicitly store its structure. Pattern application in their approach is achieved using a rewriting scheme, where, for example, there is a rule that shows how a naïve Singleton structure should be replaced with the Singleton pattern structure. Each rule can have subrules that deal with various aspects of the transformation. The essential difference between this work and ours is the use of a rule-based approach versus a metaprogramming approach. One can regard a minitransformation as a rule, and view the precondition as the predicate that fires this rule. The difference then is that in their work the rule is automatically fired when part of the program matches the predicate, whereas in ours the programmer defines the program components to which the rule is to be applied. The notion of a rule containing subrules is similar to how a design pattern transformation uses other minitransformations and refactorings in its transformation logic. One can certainly imagine a complicated design pattern transformation that could be more easily described as a set of rules than as a complex algorithm with many conditionals and iterations. We conclude that this approach is certainly of interest, though it does not appear to have been taken further than this original paper

The FAMOOS project (Framework-based Approach for Mastering Object-Oriented Software Evolution) also made a contribution in this area, though their single publication that deals explicitly with design pattern transformations only presents their initial ideas [25]. They contrast the notion of a *generic* model of the program being transformed with a *specific* model of the program. A generic model is one that can be abstracted directly from the code, while a specific model requires that the user add some domain-specific information to the model. They argue strongly that while a specific model is of course harder to build, the extra information it provides is essential in performing interesting program transformations. Although we use a generic model of the program (see appendix D) in our work, it is left up to the user to decide what design pattern to apply and what program components are to be transformed, and this in effect brings domain-specific knowledge to bear upon the transformation. In this way we achieve the benefits of both methods: an automatically-extracted model and rich transformation possibilities.

In the paper under discussion [25], the starting point used for the transformations is an antipattern. The Abstract Factory pattern is given as an example, and the starting point is where case analysis has been used to determine what type of widget to create. In section 4.2 we have presented our arguments against allowing for antipatterns in general, though in this case the problem seems to be such a common one that it is worth providing an automated solution.

Lauder and Kent describe a pattern-based approach to legacy system reengineering that also deals with antipatterns [57]. Their work focuses on the concrete antipatterns that occur in legacy systems and the positive patterns that can be applied to replace them. Six antipatterns and their positive resolving patterns are described. The patterns they consider are at an architectural level rather than a design level and so are too abstract to be considered as candidates for the automated approach we have described.

There is a stronger argument in favour of transforming architectural antipatterns than design-level antipatterns. Antipatterns at an architectural level can occur, for example, when many new features are added to a system without the system being given an architectural overhaul. While this is not desirable, it can easily occur on a project given the deadline-driven nature of the software industry. It is considerably less acceptable that a programmer, working on their own, should introduce an antipattern at the design level. Note that we did not argue that an antipattern starting point is a bad idea, rather that the precursor starting point is more logical and valuable in the context of program evolution.

Budinsky *et al* describe a tool built in IBM that can generate code automatically for a given design pattern [14]. The focus of this work is quite different from ours in that it ignores the problem of integrating the pattern with components already existing in the program. The starting point for them is therefore the green field situation so, as elaborated in section 4.2, their transformations can be much simpler and behaviour preservation is not an issue. A similar comment applies to existing industrial software tools that claim to provide support for design patterns, for example [9].

4.6 Summary

In this chapter we presented our approach to developing design pattern transformations by taking one pattern, the Factory Method pattern, decomposing into its constituent minipatterns, developing a minitransformation for each minipattern, and finally specifying the complete transformation as a sequential composition of these minitransformations. In the next chapter we apply this methodology to several other patterns and assess its applicability to the entire Gamma *et al* pattern catalogue.

Chapter 5

Applying the Methodology to the Gamma *et al* Catalogue

To fully apply and evaluate this methodology would involve designing transformations for a large number of design patterns, building a tool that implements these transformations, and evaluating the tool in a practical context. Such a route however would move this project from proof of concept validation to serious industrial software development. We apply the methodology in a more limited way therefore, but one that nonetheless demonstrates the validity of our approach and the range of its application[1].

In section 5.1 we discuss the criteria we use in choosing a precursor for a design pattern. This is an important process, as a transformation will not be useful if its starting point does not occur in practice. Section 5.2 contains some more detail on the notation we use to describe the transformations. In section 5.3 transformations are developed for a collection of creational pat-

[1]Our approach to validation is in keeping with other approaches in this area. Lauder and Kent, for example, in validating their work on pattern formalisation, satisfied themselves by applying their technique to three sample design patterns [56].

terns from the Gamma *et al* catalogue [41]. This illustrates the applicability of the methodology, and shows that the minitransformations identified in the development of one design pattern transformation are indeed reusable in other transformation developments. The leaves in question the applicability of this approach to structural patterns, and especially to behavioural patterns where the structure of the pattern is less important than its dynamic aspects. This question is addressed in sections 5.4 and 5.5 where transformations for a structural pattern and a behavioural pattern are developed.

In section 5.6 we take the remaining patterns in the Gamma *et al* catalogue and assess the applicability of our approach to each design pattern. We attempt to find a compelling precursor for each pattern and sketch a transformation for that design pattern. The results of this work are analysed in section 5.7. In section 5.8 we point to where related work on the topic of design pattern application is considered, and finally, in section 5.9, a summary of this chapter is presented.

All the pattern transformations listed in sections 5.3, 5.4 and 5.5 have been fully prototyped so we are very confident of the value of the results presented there. For details of the prototype tool we have developed, see appendix D. The precursors and transformations proposed in section 5.6 have not been prototyped, but are based on a study of the pattern itself coupled with the experience we have gained from prototyping the other pattern transformations.

The reader is advised that the material of this chapter is very detailed in places, and assumes a working knowledge of the design patterns in the Gamma *et al* catalogue [41].

5.1 Criteria for Selecting a Precursor

The notion of precursor described in the last chapter is supported by the work of Foote and Opdyke [37]. They break the software lifecycle into three phases: *prototyping*, *expansionary* and *consolidatory*. At the end of the prototyping phase a working system has been built that matches the initial set of requirements. As new requirements appear, the system will have to be expanded. However, it will inevitably transpire that the existing design is not flexible enough to support the new requirements that appear. In this case a consolidation must take place, where the software is reorganised and refactored to enhance its flexibility in preparation for accommodating the new requirements.

Our work clearly aims to help in the consolidation phase. Thus the precursors we use as starting points for the design pattern transformations are structures that are likely to be built during the prototyping phase. We expect them to be simple structures that are adequate for the purposes of building a working system rapidly, but inadequate in terms of supporting future evolution and reuse.

We arrive at a precursor for a design pattern by studying the description of the design pattern and attempting to find the structure that a programmer would be likely to have used during the prototyping phase, when the flexibility and power of the pattern were not yet required. This is naturally a matter of judgement. In some cases we are able to find a very likely and compelling precursor, in other cases it less clear how useful the precursor will be. In section 5.6 we provide an assessment of the value of each precursor and the transformation it gives rise to, and in section 5.7 these results are summarised and analysed.

5.2 Transformation Notation

We have already used our simple notation for describing composite refactorings in chapters 3 and 4. In this chapter the same notation is used, but some shortcuts are taken which we describe here:

- In some cases we do not give the full parameter list for a transformation as it may simply be too long. For example, the Builder transformation creates more than a dozen new program entities (classes, variables etc.) and it would be confusing to parameterise the transformation to this extent. Rather, we simply choose suitable names for the newly-created program elements within the transformation algorithm itself.

- In some transformations (for example, Abstract Factory) a set iteration creates a number of program elements that must be referred to later on, so we make some assumptions about names: for a class named Widget, WidgetInterface is a new interface created from this class, absWidget is a new abstract class created from this class, and createWidget is a new method that creates and returns an instance of this class. Where need be, these names are referred to as interfaceName(c), abstractClassName(c) and constructionMethodName(c) respectively[2].

- allClasses is used to denote all the classes of the program.

[2]Being precise about these issues is not a technical challenge, but the verbosity it would add to the transformations would only serve to obfuscate the important issues in the transformation.

5.3 Transformations for the Gamma *et al* Creational Patterns

In the previous chapter the transformation for the Factory Method pattern was presented in detail. In the following subsections transformations are developed for the remaining Gamma *et al* creational patterns, namely Singleton, Abstract Factory, Builder and Prototype.

5.3.1 The Singleton Transformation

The intent of the Singleton pattern [41, p.127] is to constrain a class to having only a single instance, and to provide a global point of access to this instance. The Singleton pattern prevents multiple instaniations of a class by making the constructor of the class protected, and making the class itself responsible for its own instantiation. Access to this instance is then provided using a static method, the getInstance method, that creates the instance only when required to do so.

As explained in [41], the constructor is made protected rather than private, in order to allow the class to be subclassed. There are, however, problems with this approach that are not resolved in that text. The singleton class must be able to instantiate any of its subclasses, and this requires the constructors of the subclasses to be public[3]. This means however that a client is not prevented from creating multiple instances, so the principle aim of the pattern is not enforced. There are several possible ways of resolving this issue:

1. The singleton subclass is made an *inner class* of the singleton class

[3]Overridding the getInstance method in the subclass to create and return an instance of the subclass is not possible as static methods cannot be overridden in Java.

itself[4]. External instantiation is thus not possible, and the singleton constructor can in fact be made private. However the singleton class has explicit knowledge of its subclasses, and switching to a new singleton subclass dynamically is not possible.

2. Each subclass is given a static *register* method that instantiates the class itself and registers this instance with the singleton superclass. The singleton superclass has no knowledge of its subclasses, and a client can install a new singleton dynamically by invoking the *register* method on the required class.

The second solution is more flexible and therefore preferable. In our work however we have used the original, imperfect solution presented in [41, p.133], where the constructor of each subclass is required to be public.

Precursor for the Singleton Transformation

There are two compelling starting points to use for this transformation:

1. A class exists that is only instantiated once, or is instantiated many times but each instance is identical and does not subsequently change state. Applying the Singleton pattern here has the benefit of enforcing the implicit "single instance" constraint, and of improving the clarity of the program.

2. A collection of global variables is used in the program. By collecting these into a singleton class, access to these variables is granted in a

[4]An inner class is known only to its enclosing class, but has access to this class and its superclasses. They are commonly used when one object needs to send another object a chunk of code that can access the first object's methods and fields. The manner in which they are used here, where the inner class is also a subclass of its enclosing class, can be conceptually confusing [46].

disciplined way through method invocation, rather than *ad hoc* variable accesses spread across the program.

Both of these possibilities are useful. The second one has a very clear application in tidying up code that was written without full attention being paid to quality guidelines. We work with the first one here, because, as will become apparent in section 5.3.2, it is also used in applying the Abstract Factory pattern. Later in this chapter (page 128) we develop a transformation that deals with the second case.

Specification of the Singleton Transformation

The transformation that introduces the Singleton pattern is defined as follows:

```
applySingleton(Class concreteSingleton, String newAbstractSingleton){
    PARTIALABSTRACTION(concreteSingleton, newAbstractSingleton);
    addSingletonMethod(newAbstractSingleton, concreteSingleton);
    ForAll e:ObjCreationExprn, classCreated(e)=concreteSingleton,
    e ∉ newAbstractSingleton {
        replaceObjCreationWithMethInvocation(e,
        newAbstractSingleton.getInstance());
    }
    makeConstructorProtected(newAbstractSingleton);
}
```

Initially PARTIALABSTRACTION is applied to make a new abstract class that provides the same interface as the class to be singletonised. The singleton method and field are then added to this abstract class. The object returned by the singleton method getInstance is an instance of the concrete singleton class. All object creation expressions that create an instance of this class are then updated to invoke the singleton method instead. At this point, the constructors of the abstract singleton class are made protected. As explained earlier, the constructors of the concrete singleton class must remain public.

Applying the algorithms of section 3.2, we compute the precondition of this transformation to be:

precondition:

1. No class or interface may have the name *newAbstractSingleton*:

$\neg$ isClass(*newAbstractSingleton*) $\wedge$

$\neg$ isInterface(*newAbstractSingleton*)

2. The *concreteSingleton* class must exist:

isClass(*concreteSingleton*)

3. *concreteSingleton* cannot define a method called "getInstance":

$\neg$defines(*concreteSingleton*, "getInstance")

4. *concreteSingleton* cannot contain a field called "instance":

$\forall$ f:Field, f$\in$*concreteSingleton* $\bullet$ nameOf(f) $\neq$ "instance"

5. A non-private field called "instance" cannot be defined in any superclass of *concreteSingleton*:

if f:Field $\in$ cls, cls $\in$ superclasses(*concreteSingleton*), nameOf(f)="instance" **then** isPrivate(f)

6. *concreteSingleton* must have only one constructor and it must require no parameters:

$\forall$ c:Constructor $\in$ *concreteSingleton* $\bullet$ noOfParameters(c)=0

7. Only a single instance of *concreteSingleton* is ever created:

hasSingleInstance(*concreteSingleton*)

The first two preconditions are trivial, simply ensuring that the *concreteSingleton* class exists, and that the name *newAbstractSingleton* does not clash with any existing name.

The next three preconditions are refactoring preconditions. For simplicity, we have reserved the names "getInstance" and "instance" for use in the Singleton pattern. If they are already in use in the class to be singletonised, a renaming refactoring should be applied. A field named "instance" may be defined in a superclass of the concrete singleton class, but it must be private, otherwise it could be accessed by a subclass of the concrete singleton class and this link would be broken by the addition of a field of the same name to the concrete singleton class.

Precondition 6 is a contraindication. If a class has more than one constructor, we can expect that it is instantiated in different places to different initial states, and this makes it unsuitable for the application of the Singleton pattern. Also, its constructor should be the no-arg constructor, since the class instantiates itself only once and later invocations of the getInstance method merely return this instance, but do not recreate it.

The final precondition is both a contraindication and the precursor. If the singleton class has multiple instances, applying this pattern will surely have a disastrous effect on program behaviour, and this is an inherent property of the program. The notion of a single-instance class also represents the precursor we have used for the Singleton pattern.

5.3.2 The Abstract Factory Transformation

The intent of Abstract Factory pattern [41, p.87] is to allow a program that works with a family of classes (e.g., an interface toolkit) to be easily extended to work with a different, but related, family of classes. It is clearly closely related to the Factory Method pattern, even though the implementation structures of these two patterns are quite different [41]. It is therefore very satisfying that the transformations we develop for these two patterns transpire to be quite similar. Interestingly, Amnon Eden *et al* reported a similar result in their formalisation of these two patterns using the declarative language LePUS [33].

In the following sections the precursor for this transformation is described followed by the specification of the transformation and its preconditions.

Precursor for the Abstract Factory Transformation

We can extend the precursor for the Factory Method pattern to produce a related precursor for the Abstract Factory transformation. We assume that the program being transformed creates and uses concrete instances of a family of Product classes. Again, this is not a poor structure of itself, but if a requirement arises for the program to work with a different family of Product classes, this structure will prove to be too inflexible. Applying the Abstract Factory pattern in this case results in a system where a new family of classes can be plugged in with a minimum of difficulty.

Specification of the Abstract Factory Transformation

The transformation that introduces the Abstract Factory pattern is defined as follows:

```
applyAbstractFactory(SetOfClass products, String newFactoryName,
                     String newAbsFactoryName){
    addClass(createEmptyClass(newFactoryName));
    ForAll c:Class, c ∈ products {
        ABSTRACTION(nameOf(c));[5]
        ABSTRACTACCESS(allClasses, nameOf(c));
        ENCAPSULATECONSTRUCTION(newFactoryName, nameOf(c));
        }
        APPLYSINGLETON(newFactoryName, newAbsFactoryName);
        ForAll e:ObjCreationExprn, classCreated(e) ∈ products {
                replaceObjCreationWithMethInvocation(e,newAbsFactoryName+
                "getInstance().create"+classCreated(e));
        }
}
```

First the empty concrete factory class is added to the program. Then the product classes are processed by adding an interface to each one, redirecting all accesses to the product classes to go via the corresponding interface, and adding construction methods for each product class to the concrete factory class.

The Singleton pattern is applied at this stage to produce the abstract factory class, and to impose the single-instance constraint on the concrete factory class. Finally, the existing object creation expressions that create instances of the product classes are updated to use the corresponding construction method in the abstract factory class.

We apply the algorithms of section 3.2 to compute the following preconditions for this transformation:

precondition:

1. All the classes in *products* must exist, and for each class its interface name must not be in use:

$\forall$ c $\in$ *products* $\bullet$ isClass(c) $\wedge$

$\neg$isClass(interfaceName(c)) $\wedge$ $\neg$isInterface(interfaceName(c))

2. No class or interface may have the name *newFactoryName* or

[5]For simplicity, the full argument lists for the minitransformations in the body of this loop are not given. See section 5.2 for an explanation of this.

newAbsFactoryName:

$\neg$ isClass(*newFactoryName*) $\wedge$ $\neg$ isInterface(*newFactoryName*) $\wedge$

$\neg$ isClass(*newAbsFactoryName*) $\wedge$ $\neg$ isInterface(*newAbsFactoryName*)

3. The classes in *products* have no public fields:

$\forall$ f:field, $\forall$ c:Class, f $\in$ c, c $\in$ *products* $\bullet$ $\neg$ isPublic(f)

4. The classes in *products* have no static methods:

$\forall$ m:Method, $\forall$ c:Class, m $\in$ c, c $\in$ *products* $\bullet$ $\neg$ isStatic(m)

Given that this is a more complex transformation than the related Factory Method transformation, it is at first sight curious that the preconditions transpire to be considerably simpler. This is because we create completely new abstract and concrete factory classes, rather than adding methods to existing classes. For example, using APPLYSINGLETON would normally add a number of new preconditions to a refactoring chain, but in this case it is applied to a class that just been created, and from this we were able to show that all the Singleton transformation preconditions were satisfied.

The categorisation of these preconditions is similar to Factory Method. The first two are trivial, the third is a refactoring precondition and the last one is a contraindication. Note that there is no precursor precondition for this pattern: it may be applied to any set of classes in the program. However, if the set of product classes chosen does not form a logical family, the resulting program will naturally be more complicated than the original program, for absolutely no benefit.

5.3.3 The Builder Transformation

The intent of the Builder pattern [41, p.97] is to separate the construction of a complex object from its representation, so that the same construction process

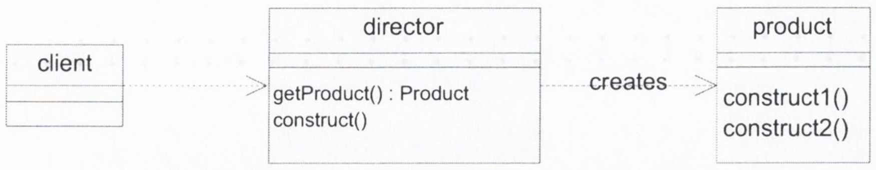


Figure 5.1: The Precursor for the Builder Design Pattern

can create different representations. This pattern is therefore useful when a product object has a complex, step-by-step construction process and it is desirable that the object that directs the construction be able to construct other, related product objects as well. By adding a builder object between the director and the product, it becomes easy to configure the director with another type of builder that will construct the desired product object.

Precursor for the Builder Transformation

The precursor for the Builder transformation is depicted as a UML diagram in figure 5.1. The director class instantiates the product class and then invokes a series of methods on this product object (construct1 and construct2 in the figure) to bring it to its fully-constructed state. The client can then obtain the product object by invoking getProduct on the director. An example of this structure is where a parser object (director) creates an empty parse tree object (product) and then invokes a series of addNode operations on the parse tree to bring it to a state where it represents the input being parsed. When parsing is complete, a client of the parser object may request it to return the parse tree that has just been constructed.

This structure, where the director class communicates with the product class directly, will prove inadequate if the director class has to be extended to construct another type of product object, one that has a different construction process. The addition of a level of indirection through a builder class makes this type of extension easy. We assume for now that the interface to the new builder class is the same as that of the current product class, so all the builder class does is to delegate directly to the product class. In the general case the builder receives construction requests from the director class and translates them into the appropriate requests to the product class. Our transformation assumes this translation to be simply the identity translation, as this produces the desired behaviour-preserving result. The programmer may of course later update this translation to perform something more sophisticated.

Specification of the Builder Transformation

In considering this transformation it is clear that there is a theme involved that has not been encountered thus far, namely that of *delegation*. A new builder class is to be added between the existing director class and the product class, and the duty of the builder is to delegate the requests it receives from the director to the product object. It is tempting to develop a mini-transformation that takes an existing class and delegates its responsibilities to another class. However, arguing behaviour preservation for such a mini-transformation is clumsy, so we choose another perspective where we wrap an existing class with a delegation/wrapper class. This wrapper class delegates its responsibilities to the wrapped class, so program behaviour is preserved. This minipattern is called WRAPPER and is described in full detail in section 5.4.2 in the context of the Bridge pattern.

The transformation that introduces the Builder pattern can now be defined as follows:

```
applyBuilder(Class director, Class product, String builderName){
        Abstraction(product, productInterface);
        Wrapper(director, productInterface, builderName);
        AbstractAccess(allClasses, product, productInterface);
        ForAll c:Constructor, c ∈ builderName{
            absorbParameter(c, 1);
        }
        parameteriseField(director, builderName);
    }
```

First Abstraction is used to add to the program an interface to the given product class. This enables the Wrapper minitransformation (see section 5.4.2) to be used to create the builder class and to set it up to delegate to the product class the construction requests it receives from the director class. AbstractAccess is now used to dissolve the dependency of the program on the concrete product class[6]. At this point the essential structure of the Builder pattern has been introduced, but there is still some work to be done. The builder class currently takes the product it is to construct as a parameter, so the absorbParameter refactoring is used to push the creation of the product object into the builder class where it belongs. The opposite problem exists between director and builder, in that the director object creates the builder it is to use and this does not fit the normal pattern solution. The parameteriseField refactoring is thus applied to enable the clients of the director class to pass it the builder object that it is to use. This completes the application of the Builder pattern[7]. The effect of applying this transformation

[6]Gamma *et al* suggest that this is normally not useful as the products produced by concrete builders tend to differ considerably [41, p.101]. We choose to follow the solution described by Grand [43, p.111], and provide an interface for the product class.

[7]Further refactorings could be applied now, so that clients would get the constructed product object from the builder object, rather than from the director object.

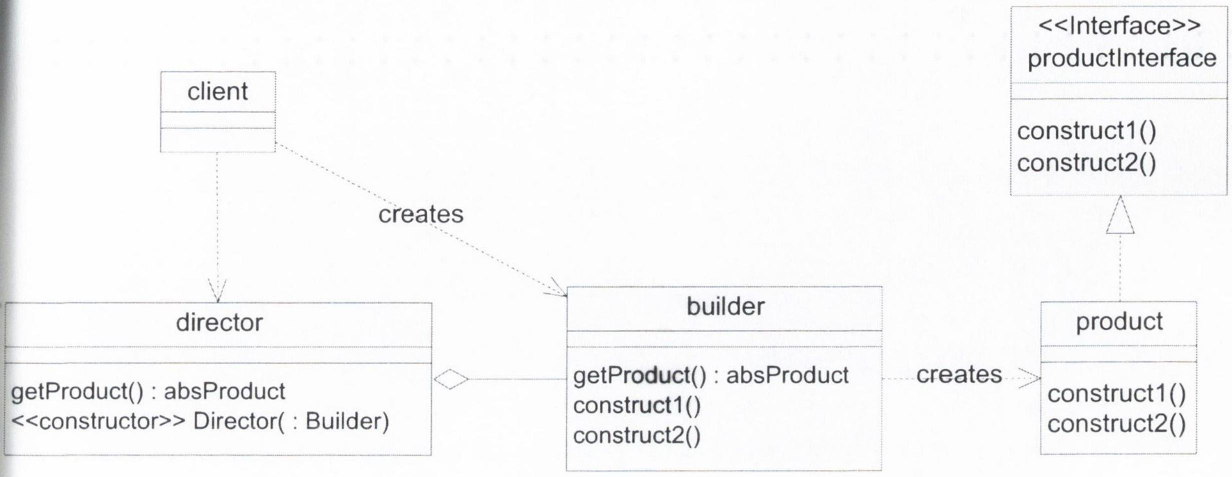


Figure 5.2: The Builder Design Pattern

to the Builder precursor (figure 5.1) is depicted as a UML diagram in figure 5.2.

Applying the algorithms of section 3.2, the preconditions of this transformation are computed as follows:

precondition:

1. The *director* and *product* classes must exist and the name *builderName* must not be in use:

isClass(*director*) $\wedge$ isClass(*product*) $\wedge$

$\neg$isClass(*builderName*) $\wedge$ $\neg$isInterface(*builderName*)

2. The *product* class must not have static methods:

$\forall$ m:Method, m $\in$ *product* $\bullet$ $\neg$isStatic(m)

3. The *product* class must not have public fields:

$\forall$ f:field, f $\in$ *product* $\bullet$ $\neg$isPublic(f)

4. The *product* class must have only one constructor and this constructor must require no parameters:

$\forall$ c:Constructor, c $\in$ *product* $\bullet$ noOfParameters(c)=0

The transformation for the Builder pattern is one of the most complex of all the transformations developed in this work, though much of the complexity is hidden inside its constituent minitransformations and refactorings. The preconditions for the transformation are quite simple, again because most of the preconditions of its constituent minitransformations and refactorings are guaranteed by earlier parts of the transformation. For example, the WRAPPER minitransformation can only be applied if the director class only uses methods of the product class that are declared in the interface productInterface. This condition does not appear in the precondition to the transformation above, since it has already been set up by the application of the ABSTRACTION minipattern.

The first precondition is trivial. The second is a contraindication, though as pointed out in the footnote on page 107, the ABSTRACTACCESS minitransformation that gave rise to this precondition could be omitted from the transformation. The third condition is a straightforward refactoring precondition, while the final condition is also a refactoring precondition but is of more interest. In absorbParameter the construction of product objects is moved from the director class to the builder class. Each such object creation expression must be the same and not be dependent on its context. The only likely way for this to happen is if the product class only admits no-arg construction. Bearing in mind that this pattern is applicable where product objects are constructed in a step-by-step fashion, it is not unreasonable to require that the constructor for the product class itself takes no parameters.

5.3.4 The Prototype Transformation

The intent of the Prototype pattern [41, p.117] is to specify the kind of objects to create by using a prototypical instance, and to create new objects by cloning this instance. The applicability section for this pattern proposes three situations where it may be applied:

1. to achieve dynamic loading of classes, or
2. to avoid building a hierarchy of factory classes, one for each product class, or
3. when instances of a class can have an initial state that is one of only a few possible combinations.

Although these criteria are stated to be disjunct, in fact the Prototype pattern could not be applied if only the second were true and not the third. If objects of the product class can be constructed in a wide range of initial states, applying the Prototype pattern is not possible. Note that a precursor for the first criterion is very likely to be an antipattern, so we do not look further at this possibility.

Precursor for the Prototype Transformation

The precursor we consider is therefore where the programmer has explicitly instantiated the product class at several points in the client class before realising that all these instances are identical[8]. The updating of the object creation statements to use a cloned prototype object is possible only if the arguments to the object creation statements have the same values in every case. This is highly unlikely to occur unless the client class only instantiates the product class using its no-arg constructor. For this practical reason we limit the precursor for this pattern by enforcing this precondition.

[8]A more general solution is also possible, where the initial state of the objects created fits into one of several categories.

Specification of the Prototype Transformation

The transformation that introduces the Prototype pattern is defined as follows:

```
applyPrototype(Class client, Class product, String productInterface){
    createExclusiveComponent(client, product, "prototype");
    ABSTRACTION(product, productInterface);
    ABSTRACTACCESS(client, product, productInterface);
    ForAll e:ObjCreationExprn, classCreated(e)=product, e $\in$ client {
        replaceObjCreationWithMethInvocation(e, "prototype.clone()");
    }
}
```

Using createExclusiveComponent a field called "prototype" is added to the client class to store the prototypical object of the product class. ABSTRACTION is now applied to the product class and ABSTRACTACCESS to abstract the client class from the product class. Finally replaceObjCreationWithMethInvocation is applied to change all creations of product objects to invoke the clone method on the prototypical product object instead. The invocation of the clone method on the product class assumes that this class is indeed clonable; see the definition of isClonable on page 191 for more detail.

A minimalist approach was taken in building this transformation. A more sophisticated approach was also possible, by building a prototype manager that would handle prototypes for a collection of classes and allow the collection to grow and contract dynamically.

We apply the algorithms of section 3.2 to compute the following preconditions for the above transformation:

precondition:

1. The given classes must exist:

$\text{isClass}(\textit{client}) \wedge \text{isClass}(\textit{product})$

2. No class or interface with the name *productInterface* exists:

$\neg\text{isClass}(\textit{productInterface}) \wedge \neg\text{isInterface}(\textit{productInterface})$

3. The *client* class cannot contain a field called “prototype”:

$\forall$ f:Field, f$\in$*client* $\bullet$ nameOf(f) $\neq$ “prototype”

4. A non-private field called “prototype” cannot be defined in any superclass of *client*:

if f:Field $\in$ cls, cls $\in$ superclasses(*client*),

nameOf(f)=“prototype” **then** isPrivate(f)

5. The *product* class must not have public fields:

$\forall$ f:field, f $\in$ *product* $\bullet$ $\neg$isPublic(f)

6. The *product* class must not have static methods:

$\forall$ m:Method, m $\in$ *product* $\bullet$ $\neg$isStatic(m)

7. The *product* class must be clonable:

isClonable(*product*)

8. The *client* class creates *product* objects only using the no-arg constructor:

$\forall$ e:ObjectCreationExprn, e $\in$ *client*, classCreated(e)=*product* $\bullet$

noOfArguments(e)=0

The categorisation of these preconditions is as follows. The first two are trivial, the third, fourth and fifth are refactoring preconditions, the sixth and seventh are contraindications, while the final one is a precursor precondition that we assumed in order to ease the specification of the transformation.

This completes the application of our methodology to the Gamma *et al* creational patterns. We postpone analysing the results until section 5.7 after

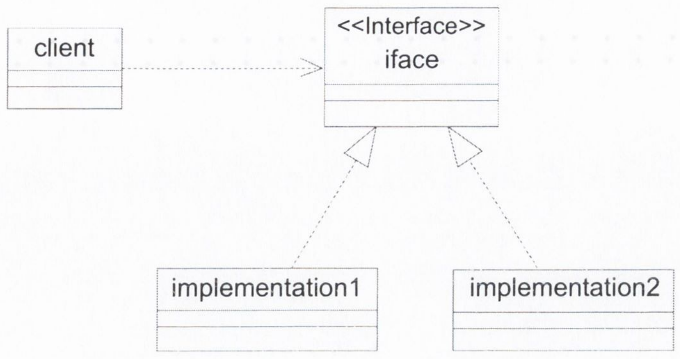


Figure 5.3: The Precursor for the Bridge Design Pattern

the entire catalogue has been considered. In the following sections 5.4 and 5.5, we demonstrate the broader application of this methodology by applying it to a structural pattern and a behavioural pattern.

5.4 Transformation for a Structural Pattern: Bridge

The intent of the Bridge pattern [41, p.151] is to decouple an abstraction from its implementation so that the two can vary independently. It is useful when an abstraction needs to be implemented in several ways, and also needs to be open to extension using inheritance.

5.4.1 Precursor for the Bridge Transformation

The precursor for this pattern follows naturally from the description of the pattern given in [41]. It is depicted graphically as a UML diagram in figure 5.3. We see that there is a client class that makes use of an interface that has been implemented in several different implementation classes. The weakness of this structure becomes apparent if the programmer later wants to extend the interface in some way: for each existing implementation class, a new class will have to be added. For example, a client class might use a queue interface that is implemented in one subclass as a static array and in another as a dynamic linked-list structure. If we need to extend the client to work with a dequeue[9] as well, it is natural to add this as a subinterface of the queue interface. However, now the dequeue interface must itself be provided with two subclasses to provide a static and a dynamic implementation. The application of the Bridge pattern to this situation will enable the queue interface to be extended separately from its implementation.

In considering this transformation it is clear that the theme of *delegation* is involved again. A new bridging class is to be added between the existing client classes and the implementation classes. The duty of this class is to delegate all the requests it receives from the client to the appropriate implementation object. In the following section we describe this minipattern in detail, and in section 5.4.3 the Bridge transformation itself is dealt with.

5.4.2 The Wrapper Minitransformation

The WRAPPER minitransformation is used to "wrap" an existing receiver class with another class, in such a way that all requests to an object of the wrapper class are passed to the receiver object it wraps, and similarly any results of such requests are passed back by the wrapper object. This requires that all existing instantiations of the receiver class be also wrapped with an instantiation of the wrapper class itself. The overall effect of this minitransformation is to add a certain flexibility to the relationship between a client object and the receiver object it uses. All communication now goes

[9]A double-ended queue.

via the wrapper object, which means that run-time replacement of the receiver object becomes possible without the client object being aware of the change. In a certain regard, this minipattern is the dynamic equivalent of the AbstractAccess minitransformation.

An issue that must be dealt with is where one or more of the client classes provide a "getter" method that returns an instance of a receiver object. If the receiver classes are to be wrapped from all other classes in the program, it makes sense to return the wrapped receiver object. However, it is only the client classes that should see the wrapped receiver class; other classes in the program should deal directly with the receiver classes as before. Therefore, to allow for a client that provides direct access to its receiver object, createWrapperClass adds a getter method to the wrapper class to return this object, while useWrapperClass updates the getter method in the client class to delegate to the getter method to the wrapper class[10].

We have assumed in the description of this minitransformation that there is a single receiver class to be wrapped. In the more general case there will be a set of receiver classes to be wrapped. In this case, the set of receiver classes is given by an interface that reflects how the receivers are used in the client classes. For our current purposes of building a transformation for the Bridge pattern, it is the latter version that is of interest, so it is the one we specify here.

This minitransformation is implemented in terms of refactorings in the following way:

[10]This issue resulted in a lot of complexity in the detailed design and implementation of this minitransformation. It is interesting therefore to note that this could be avoided were the assumption to be made that the initial program complies with the Law of Demeter [60]. In a program that observes this law, an object would not extract a subobject from another object, and send a message to it.

```
WRAPPER(SetOfClass clients, Interface iface, String wrapperName){
    Class wrapper = createWrapperClass(iface, wrapperName, "receiver");
    addClass(wrapper);
    ForAll c:Class, implementsInterface(c, iface) {
        useWrapperClass(clients, wrapper, c, "getReceiver");
    }
}
```

Initially the wrapper class is created and added to the program. Then it is used to wrap each of the receiver classes and, correspondingly, any clients that use these receiver classes are updated to wrap each construction of a receiver class with an instance of the wrapper class.

To demonstrate legality of this chain and to compute its pre- and postconditions, we apply the algorithms of section 3.2. The computation is straightforward, especially since most of the preconditions for useWrapperClass are provided by createWrapperClass. The following pre- and postconditions are produced:

precondition:

The given interface must exist:

$\text{isInterface}(\mathit{iface})$

The name for the new wrapper class is not in use:

$\neg\ \text{isClass}(\mathit{wrapperName}) \wedge \neg\ \text{isInterface}(\mathit{wrapperName})$

The client classes only use methods of the receiver classes that are declared in the interface *iface*:

$\forall$ o:ObjectRef, containingClass(o) $\in$ *clients*,

implementsInterface(typeOf(o), *iface*) $\bullet$

$\forall$ m:Method, uses(o,m) $\bullet$ declares(*iface*, m)

postcondition:

The wrapper class has been added to the program:

$\text{isClass}' = \text{isClass}[\mathit{wrapper}/\text{true}]$

(Further properties of the *wrapper* class are given on page 198.)

All object references to receiver classes in *clients* have been changed to *wrapper*:

$\forall$ o:ObjectRef, containingClass(o) $\in$ *clients*, implementsInterface(typeOf(o), *iface*) $\bullet$

  typeOf′=typeOf[o/*wrapper*]

All creations of receiver objects in the *clients* have been updated:

$\forall$ e:ObjectCreationExprn, implementsInterface(classCreated(e), *iface*), containingClass(e) $\in$ *clients* $\bullet$

  classCreated′=classCreated[e/*wrapper*]

Any receiver object will exhibit the same behaviour as an instance of the class called *wrapperName* that has been given this object as its construction argument:

$\forall$ c:Class, implementsInterface(c,*iface*) $\bullet$

$\forall$ e:ObjectCreationExprn, classCreated(e)=c $\bullet$

  exhibitSameBehaviour′ =

  exhibitSameBehaviour[(e, new *wrapperName*(e))/true]

5.4.3 Specification of the Bridge Transformation

The transformation that introduces the Bridge pattern can now be defined very simply as follows:

```
applyBridge(SetOfClass clients, Interface iface, String bridgeName){
    WRAPPER(clients, iface, bridgeName);
}
```

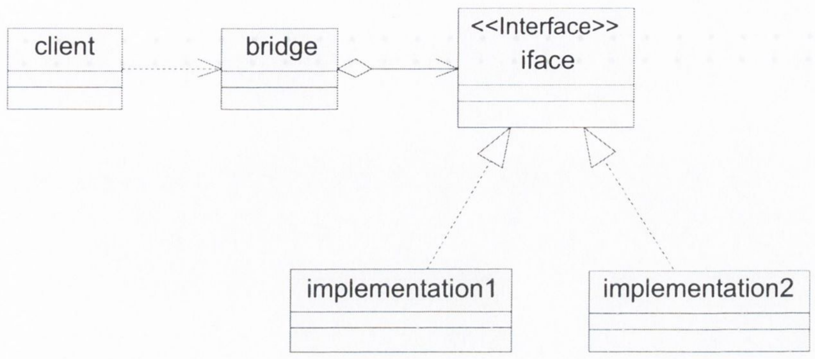


Figure 5.4: The Bridge Design Pattern

The WRAPPER minitransformation does all of the work here, setting up the bridge class and ensuring that it delegates requests from the client classes to the classes that implement the given interface. The effect of applying this transformation to the Bridge precursor (figure 5.3) is depicted as a UML diagram in figure 5.4.

Once the structure of the Bridge pattern has been reified in the program code, the programmer can exploit this. The bridge class can be subclassed and new methods added. If need be, the implementation of methods in the bridge class can be changed to do more than simply delegate to the implementation classes. These changes are facilitated by the introduction of the Bridge pattern, but cannot be made part of the transformation itself, as they are dependent of the intention of the programmer and are not in general behaviour-preserving.

The precondition for this transformation is naturally just the precondition for the WRAPPER minitransformation given in section 5.4.2 above, so it is not restated here.

5.5 Transformation for a Behavioural Pattern: Strategy

Behavioural patterns have the possibility of challenging our approach very strongly. Since we transform one type of program structure (a precursor) into another one (the desired design pattern structure), it is unclear how a pattern that has little structure will be handled. In this section we address this question by applying the proposed methodology to a behavioural pattern and assessing the result.

The intent of the Strategy pattern [41, p.315] is to enable several related algorithms to be encapsulated into their own respective classes, so that a client can be dynamically configured with an object of one of these classes. For example, a `tree` class might incorporate a traversal algorithm that returns the nodes of the tree in some order. Rather than hardcoding one particular traversal algorithm into the `tree` class itself, the Strategy pattern encapsulates the traversal algorithm into its own class and allows a `tree` object to be configured with different traversal algorithms. This makes it easy to achieve in-, pre- and post-order traversals of the same `tree` object.

5.5.1 Precursor for the Strategy Transformation

The natural precursor for this pattern is where a class incorporates a number of methods and fields that are all related to some particular algorithm. While this cannot be regarded as a bad structure, its inadequacies become apparent if a requirement arises that the class be configurable to use a one of a number of related algorithms.

As with the Bridge and Builder patterns, there is a form of delegation taking place here as well. The strategy methods will be moved to their own class and the original class will delegate to them. The WRAPPER minipattern that was used earlier is not so suitable here: nothing is being "wrapped," rather part of the original class is being split off into a new class and behaviour is being preserved by the original class delegating to the new one. In the following section we describe this minipattern in detail, and in section 5.5.3 the Strategy transformation itself is dealt with.

5.5.2 The Delegation Minitransformation

The DELEGATION minitransformation is used to move part of an existing class to a component class, and to set up a delegation relationship from the existing class to its component.

This minitransformation is defined as follows[11]:

```
Delegation(Class context, SetOfMethod moveMethods,
        String delegationName){
    addClass(createEmptyClass(delegationName));
    createExclusiveComponent(context, delegationName, "delegation");
    ForAll m:Method, m ∈ moveMethods {
        abstractMethodFromClass(m);
        moveMethod(context, "delegation", m);
    }
}
```

The empty delegation class is first added to the program and an exclusive

[11]Roberts deals with this transformation as well [84, p.40]. Since he does not use a precursor, he can ignore the problem of initialising the component object that is being delegated to. Also, he does not abstract the method to be moved from its class, so he only permits the moving of a method that does not access any fields or methods in its own class or any of its superclasses.

component of this class is added to the context class. Now each of the methods to be moved can be processed. A method to be moved must first be "abstracted" from its class, that is, everything it refers to in the class must be made public. At this point, the moveMethod refactoring may be invoked to move the method to the delegation class.

Using the algorithms of section 3.2, the pre- and postconditions for this minitransformation are computed as follows:

precondition:

The given context class must exist:

$\text{isClass}(\textit{context})$

The name for the delegation class must not be use:

$\neg\text{isClass}(\textit{delegationName}) \wedge \neg\text{isInterface}(\textit{delegationName})$

The methods to be moved must belong to the context class:

$\forall\ \text{m} \in \textit{moveMethods} \bullet \text{m} \in \textit{context}$

The *context* class cannot contain a field called "delegation":

$\forall\ \text{f:Field, f} \in \textit{context} \bullet \text{nameOf(f)} \neq \text{"delegation"}$

A non-private field called "delegation" cannot be defined in any superclass of *context*:

$\textbf{if}\ \text{f:Field} \in \text{cls, cls} \in \text{superclasses}(\textit{context}),$

$\text{nameOf(f)=``delegation''}\ \textbf{then}\ \text{isPrivate(f)}$

postcondition:

A new class called *delegationName* has been added to the program:

$\text{isClass}' = \text{isClass}[\textit{delegationName}/\text{true}]$

The class *context* has a field called "delegation" of type *delegationName*:

$\exists\ \text{f:Field, f} \in \textit{context}\ \text{such that}$

$\text{typeOf}'=\text{typeOf}[\text{f}/\textit{delegationName}]$

$\text{nameOf}'=\text{nameOf}[\text{f}/\text{``delegation''}]$

"delegation" refers to an exclusive component of *context*:

isExclusiveComponent′=isExclusiveComponent[(*context*, "delegation")/true]

All methods/fields defined directly or indirectly in *context* that are used by a method in *moveMethods* are now public:

```
∀ m:Method ∈ moveMethods •
∀ x:Field/Method, defines(context, x), uses(m,x) •
    isPublic′=isPublic[x/true]
```

The given methods have been moved to the delegation class:

```
∀ m:Method ∈ moveMethods •
    classOf′=classOf[m/delegationName]
```

The class *context* delegates invocations of moved methods to methods that exhibit the same behaviour in the delegation class:

```
∀ m:Method ∈ moveMethods • ∃ n:Method, classOf′(n)=context,
nameOf′(n)=nameOf(m), sigOf′(n)=sigOf(m) such that
    uses′=uses[(n,m)/true]
    exhibitSameBehaviour′=exhibitSameBehaviour[n/m]
```

5.5.3 Specification of the Strategy Transformation

The transformation that introduces the Strategy pattern can now be defined very simply as follows:

```
applyStrategy(Class context, SetOfMethod strategyMethods,
              String strategyName){
    DELEGATION(context, strategyMethods, strategyName)
    ABSTRACTION(strategyName, strategyInterface);
    ABSTRACTACCESS(context, strategyName, strategyInterface);
}
```

At the completion of this transformation, the strategy methods have all been moved to the strategy class. Each one takes its context object as an argument, and refers back to this context for any fields it needs access to. If the programmer has chosen a cohesive set of strategy methods, it is to be expected that most of these fields can be moved to the strategy class as well, and then some or all of the strategy methods will not need the context argument anymore. This part of the transformation can be automated quite straightforwardly, but for clarity we have omitted it.

Applying the algorithms of section 3.2, the preconditions for this transformation are computed as follows:

precondition:

1. The given context class must exist:

isClass(*context*)

2. The name for the strategy class must not be use:

¬isClass(*strategyName*) ∧ ¬isInterface(*strategyName*)

3. The name for the strategy interface must not be use:

¬isClass(*strategyInterface*) ∧ ¬isInterface(*strategyInterface*)

4. The strategy methods must belong to the context class:

∀ m ∈ *strategyMethods* • m ∈ *context*

5. The *context* class cannot contain a field called "delegation":

∀ f:Field, f∈*context* • nameOf(f) ≠ "delegation"

6. A non-private field called "delegation" cannot be defined in any superclass of *context*:

if f:Field ∈ cls, cls ∈ superclasses(*context*),

nameOf(f)="delegation" **then** isPrivate(f)

7. No strategy method may be static:

∀ m ∈ *strategyMethods* • ¬isStatic(m)

The first four preconditions are trivial, the next two are refactoring preconditions while the last one is a contraindication. The contraindication is derived from the use of the ABSTRACTACCESS minipattern, since moving a static method to the strategy class would make it subsequently inaccessible when the strategy interface is added.

In spite of initial concerns that our approach would have problems dealing with a behavioural pattern, a compelling precursor was found for the Strategy pattern and the transformation to apply this pattern did not prove to be particularly difficult to work out. In section 5.7 we provide an explanation for this phenomenon.

5.6 Precursors and Transformations for the Gamma *et al* Patterns Catalogue

In this section the remaining patterns of the Gamma *et al* catalogue [41] are analysed with a view to finding a suitable precursor, assessing if the transformation is workable, and determining the minitransformations that are likely to be used. Please note that the transformations offered in this section have not been prototyped and worked out in as much detail as those in previous examples. Our aim here is to make a global assessment of the applicability of the methodology, without applying the full rigour of the approach to every example. In each case we assess the result we achieve and place it in one of the following categories:

1. *Excellent*: The methodology worked very well. A plausible precursor was found and a compelling transformation was built, making use of some of the minitransformations already identified.

2. *Partial*: There is some problem with the result (see list below) that means a usable transformation can be developed, but it is not complete.

3. *Impractical*: There is a serious problem with the result (see list below) that makes it impossible to build a transformation, or produces one that is so constrained that it is of no practical value.

There are a number of ways in which a design pattern can be found to be less suitable for the application of our methodology. We describe them below.

- A convincing and useful precursor cannot be found. Sometimes there is no compelling way a programmer might have partially implemented the intent of the pattern without either using a poor design (an antipattern), or going the whole way and implementing the full pattern structure. We may in this case be able to work with a weak precursor that is very close to the green field starting point. This is a workable solution, but not very satisfactory, as there is little need for behaviour-preservation proofs in this case. Examples: Decorator and Observer.

- There is a compelling precursor, but it is not a structure that can easily be pointed to and identified in code, even by a programmer who knows the code well. It may, for example, contain behavioural elements that are dispersed around the code. The problem here is that this type of precursor is too inexact to be used to drive a behaviour-preserving transformation, and so is useless as a starting point for an automated approach. In some cases dynamic analysis or sophisticated pattern recognition might provide a solution, but this is beyond the scope of this work. Examples: Facade, Mediator, Interpreter and Flyweight.

- Even if a compelling and easily identifiable precursor can be found, it may be that the resulting transformation still leaves a certain amount of work for the programmer to do in order to complete the application of the pattern. Note that if the amount of work to be done is small, we may still categorise the result as excellent. Examples: Adapter, Builder, Bridge, Chain of Responsibility, Proxy and State.

A word on the precision of the specification of the precursor is useful here. If we were searching for the precursor in the code, its specification would have to be completely precise. However, in our approach, the programmer identifies exactly where the design pattern is to be applied. This means that the automated tool need only identify the aspects of the existing structure that need to be restructured, and this is the purpose of the precursor. For example, in the case of the Factory Method pattern, the tool only has to identify the places in the class where a product object is created. The"extra" part of the precursor, the fact that this is a good spot to apply the Factory Method pattern, has been provided by the programmer.

In general the applicability section of a design pattern description suggests the precursor [41]. If there are several distinct applicability clauses (i.e., if they are disjunctives) this may give rise to several precursors. In the case of the Prototype pattern, for example, there are three applicability clauses, but we find that one of them is natural to choose as the basis for the precursor.

5.6.1 The Gamma *et al* Creational Patterns

In this section we consider the application of our methodology to each of the creational patterns of the Gamma *et al* catalogue [41]. Since we have dealt with these patterns already, we simply place the precursor and resulting transformation into one of the three categories listed on page 124.

Abstract Factory

This pattern has been fully dealt with in section 5.3.2. The precursor is a structure that is likely to occur during the evolution of a software system and the transformation is compelling.
Overall Assessment: Excellent.

Builder

This pattern has been fully dealt with in section 5.3.3. The precursor and transformation are compelling, though they lack the simplicity and elegance of, for example, the Factory Method transformation. We explained on page 106 that a small amount of work is left to the programmer at the end of the transformation, but it is nevertheless a very valuable result.
Overall Assessment: Excellent.

Factory Method

Due to the elegance of its solution, this pattern was chosen as our flagship example and was presented in detail in chapter 4.
Overall Assessment: Excellent.

Prototype

This pattern has been fully dealt with in section 5.3.4. This solution has weaknesses in that the precursor is somewhat more constrained than that for the other creational patterns, and the construction of the `clone` method is not automatable in every case. However, the transformation is generally straightforward, and constructing the `clone` method could be a problem even for a programmer applying this pattern by hand.
Overall Assessment: Excellent.

Singleton

This pattern has been fully dealt with in section 5.3.1. The precursor used there (a single-instance class) is not a very compelling one and cannot be verified automatically. However, we chose this precursor because it made it possible to reuse the entire transformation in developing the Abstract Factory transformation. As already stated, a more generally applicable precursor is where there is a set of global variables to be packaged into a singleton class[12]. This gives rise to the following transformation:

1. Add an empty class to the program and use APPLYSINGLETON from section 5.3.1 to make it a singleton class.

2. For each global variable to be encapsulated, add a field of this type to the singleton class, along with "getter" and "setter" methods for this field.

3. Replace every reading of a global variable with an invocation of the corresponding "getter" method, and every writing of a global variable with an invocation of the corresponding "setter" method.

4. Delete all the (now unused) global variables.

This is both a practical precursor and a straightforward transformation.

Overall Assessment: Excellent.

[12]This was also the precursor used by Jahnke and Zündorf [49], the only other approach to design pattern transformations that uses a similar notion to that of a precursor. See page 89 for a more detailed description of this work.

5.6.2 The Gamma *et al* Structural Patterns

In this section we consider the application of our methodology to each of the structural patterns of the Gamma *et al* catalogue [41]. If the pattern has been dealt with before, we simply place the precursor and resulting transformation into one of the three categories listed on page 124.

Adapter

The intent of the Adapter pattern [41, p.139] is to convert the interface of an existing class in order to make it compatible with the interface that its clients expect. This allows classes to work together that could not otherwise do so, due to minor incompatabilities in the interface provided and the interface expected. Fully automating the application of this pattern poses a problem in that the mapping from the new adapter interface to the existing adaptee class should be specified by the programmer. Developing a language to specify this mapping is non-trivial and beyond the scope of this work.

We take a simple approach and assume this mapping to be the identity mapping. This allows the construction of a transformation that applies the Adapter structure in a behaviour-preserving fashion, but leaves an amount of work for the programmer to do. The precursor is simply where a client class uses a supplier (adaptee) class and a requirement is introduced that the client be able to work with any one of a family of supplier classes, each one providing essentially the same functionality as the existing one, but with a different interface.

The transformation then becomes:

1. Apply WRAPPER to the supplier class to produce the concrete adapter class.

2. Apply Abstraction to the concrete adapter class and AbstractAccess to abstract the client class from the concrete adapter class.

This application of three minitransformations produces the Adapter structure, where the adapter class simply delegates each request to the existing supplier class. The programmer may now update the adapter class to perform a more sophisticated adaptation, and add new supplier classes.
Overall Assessment: Excellent.

Bridge

This pattern has been fully dealt with in section 5.4.3. As explained on page 118, a small amount of work may be left to the programmer at the end of the transformation, but overall the precursor and transformation are compelling.
Overall Assessment: Excellent.

Composite

The intent of the Composite pattern [41, p.163] is to enable a client class to treat a single component object and a composition of component objects in a uniform fashion. The most natural precursor here is where the programmer has identified a 1:1 relationship between a client class and a component class, and has implemented this by giving the client class a field of type component. If it later transpires that the cardinality of this relationship must be extended to 1:N, it may be natural to apply the Composite pattern. This will involve replacing the component field with a field of a type that represents both the interface to the component class itself, and the "composite" interface (addComponent, removeComponent etc.).

One issue is the actual composite data structure that is to be used. This could be any type of generic container structure, but is more usually a type of list. Let us parameterise the transformation with the container class that is to be used for the composite implementation, and demand that this container class contains an iteration interface. The resulting transformation is:

1. Apply ABSTRACTION to the component class to produce the component interface.

2. Extend the component interface with the supplied composite interface.

3. Provide implementations for the composite operations in the component class[13].

4. Add the composite class and provide it with an implements link to the component interface. It will contain a private field of type container. The composite methods will be implemented by delegating them to the container field, while the component methods will be implemented by iterating through the elements of the container and applying the method to each one.

5. Apply ABSTRACTACCESS to abstract the client class from the component class, so that it now uses the component interface instead.

The result of this transformation is that the client class now uses the component class through its interface. It is also easy to extend the client so that it uses compositions of components in place of the single component instances it was dealing with originally.

Overall Assessment: Excellent.

[13]Operations like addComponent are unintuitive for the component class and must be implemented to do nothing. However, even if the pattern is being applied by hand, this is necessary to achieve a transparent interface to both leaves and composites.

Decorator

The intent of the Decorator pattern [41, p.175] is to enable the dynamic addition and removal of responsibilities to/from an object. It allows the functionality of an object to be transparently extended at runtime, by wrapping the object with the appropriate decorator objects.

A transformation that introduces this pattern to a C++ program was built as part of our earlier work [71]. The starting point for the transformation was taken to be where multiple inheritance had been used to provide the multiply-decorated component class. This tends to lead to an explosion of subclasses, where each subclass represents a certain combination of decorator classes. The application of the Decorator pattern is valuable here to reduce the number of classes in the program and to enable the dynamic creation of new combinations of decorators.

Java does not support multiple inheritance, so this is not a possible precursor. The alternative precursor is where the component class achieves its decoration by storing a list of decorator objects and iterating through them whenever it receives a message. This is an implausible precursor, so we do not consider it further.

The most suitable starting point for this transformation is close to the green field situation. There are a number of client classes that use a component class, and there are a number of decorator classes. There is as yet no relationship between the component class and the decorator classes, but there is commonality between the interfaces they present. Application of the Decorator pattern means that this commonality can be exploited to allow component objects be dynamically extended with new behaviour, by wrapping them with the appropriate decorator objects.

The transformation to apply the Decorator pattern structure is then as follows:

1. Apply Abstraction to the component class to produce the component interface and AbstractAccess to the client classes to abstract them from the concrete component class.

2. Apply Wrapper to a decorator class to create the abstract decorator class that delegates all messages it receives to its component object.

3. Make each concrete decorator a subclass of the abstract decorator class and update each method that is declared in the component interface so that it first invokes the operation of the same name in its superclass.

The clients continue to use the same component objects as before, but access them through the component interface; behaviour preservation is thus simple to demonstrate. The Decorator structure is now present, so the client may be easily updated to decorate these components as need be.
Overall Assessment: Partial.

Facade

The intent of the Facade pattern [41, p.185] is to provide a unified interface to an existing set of classes in a subsystem. The natural precursor for this pattern is stated clearly in the description of this pattern. A set of classes (clients) use another set of classes (subsystem classes), and this interaction should be encapsulated and directed through a single facade class.

However, apart from adding an empty facade class it is very difficult to further automate this transformation in the general case. A client class may create multiple instances of a subsystem class and interact with them in different ways. The key aspect of the Facade pattern is that these interactions must be understood in some way, grouped into cohesive units and encapsulated in the interface to the facade class. Finding these groupings involves sophisticated pattern recognition that is poorly supported by automated approaches. Packaging these groupings into cohesive methods in the facade interface is likely to involve method splitting and low-level analysis that other transformations do not need[14].

So while a compelling precursor can be identified, our methodology can achieve little by way of automating this transformation.
Overall Assessment: Impractical.

Flyweight

The intent of the Flyweight pattern [41, p.195] is to use sharing to support a large number of fine-grained objects efficiently. The precursor for this pattern is quite clear from the pattern description. A class exists that has a large number instances and part of the state of these instances never changes after construction. The immutable part of the state can be made intrinsic to the flyweight and the mutable part stored in the context of the flyweight.

Not much of this transformation can be automated using the techniques we have proposed. The structure of the Flyweight pattern can be built but "populating" it and transforming the existing class into this structure has to be done by the programmer. The number of flyweight objects, their initial state, and a key for accessing them are all crucial aspects of this pattern that cannot be determined from the program code using our techniques. Also, determining how to integrate the extrinsic state into the context of the flyweight is an issue requiring considerable design judgement.

[14]Bengtsson and Bosch describe an experience of reengineering the software system for a dialysis machine [4]. They report applying the Facade pattern with enthusiasm and finding that it resulted in unnecessary complexity. This suggests that even applying this pattern by hand is not an easy task.

Overall Assessment: Impractical.

Proxy

The intent of the Proxy pattern [41, p.207] is allow one object to "stand in" or act as a surrogate for another object. There are many reasons why it may be desirable to proxy an object: the real object may reside on a remote machine (remote proxy), or it may be necessary to restrict access to certain operations (protection proxy), or constructing the entire object may be expensive and a light proxy can be used in its place until full construction becomes necessary (virtual proxy).

Regardless of the type of proxy, its essential structure can be achieved by the application of the Wrapper minitransformation, to wrap the original object with its proxy object. We will consider the transformation for the virtual proxy further. The natural precursor is where a class has been developed but the programmer realises that the construction of objects of this class is time-consuming (e.g., they may access an image across a network). It may therefore be beneficial to postpone construction of the expensive parts of this class until they are actually needed.

The parameter to this transformation is just the class to be proxied. The transformation to apply a virtual proxy is as follows:

1. Apply Wrapper to the given class to create the proxy class.
2. Apply Abstraction to the given class and add an implements link from the proxy class to the new interface.
3. Apply AbstractAccess so clients of the given class now access it through the interface.

Now the essential pattern structure is available, the programmer can develop the program further to achieve the relevant type of proxying. In the case of the virtual proxy, the "cheap" fields of the class may be stored in the proxy enabling certain requests to be met by the proxy alone. Other requests will result in the creation of the proxied object and the delegation of the requests to this object.
Overall Assessment: Partial.

5.6.3 The Gamma *et al* Behavioural Patterns

In this section we consider the application of our methodology to each of the behavioural patterns of the Gamma *et al* catalogue [41]. If the pattern has been dealt with before, we simply place the precursor and resulting transformation into one of the three categories listed on page 124. Before considering the patterns themselves, we first deal with a difficult problem that arises in several of the transformations for behavioural patterns.

Issues in Class-splitting Transformations

Many of the transformations in this section involve splitting an existing class. In the simple case, e.g., Strategy, after the class is split one part retains a reference to the other part. The relationship is reflected in the object structure in that what was originally a single object before the transformation, will now become two objects, one with a reference to the other. This does not present any particular problem to our approach. Given a reference to an object, the part that has been split off can be accessed by traversing the link to that object.

A much more serious issue arises when a class is split and the cardinality of the relationship between the parts is made 1:N, but the traversal of this relationship must only be available from the N side to the 1 side. In this case it is up to the programmer to keep track of which object is related to which, i.e., there is no explicit link between the objects. This occurs in a number of design pattern transformations:

- Iterator. In the precursor the iteration is part of the composite class, while in the design pattern structure it is moved to an object on its own. A composite object may have many active iterations, but should not know about them.

- Memento. In the precursor the originator class itself stores the memento object, while in the design pattern structure it is stored in an object on its own. An originator object may have many mementos, but should not know about them.

Here is a concrete example of the problem, based on the design pattern transformation for the Iterator pattern (see page 142):

```
Composite x = new Composite();
Composite y = new Composite();
Composite z;
x.startIteration();
y.startIteration();
...
if (someCondition)
      z=x;
else
      z=y;
...
return(z.getNextElement());
```

The Composite class provides the usual methods to add and remove elements, as well as methods to iterate through the elements of the composition. The object reference z is assigned one of the two Composite objects that have been created at the start of the block.

In applying the Iterator pattern to this program, the iteration part of Composite will be split off into a class on its own. At points in the code where an iteration is started, a new iteration object will be created, parameterised with the composite object. In the above code, two new iteration objects will be created, one for the iteration over x, and one for the iteration over y. The problem faced here is how to work out which iterator object should be used in the return statement.

In the original program, the fact that we had a reference to the object meant that we knew which iterator it was connected to, since the iterator was part of the object itself. In the transformed program, the iterator object holds a reference to its composite object, but not vice versa. This means that code in the original program that accesses the iteration interface of a composite object cannot be easily transformed to use the appropriate iteration object. In fact, this problem is not decidable in general, and could be a problem for a programmer performing the task by hand.

If an iterator is initialised and used on a named object, not passed to another context and not aliased, it will not be a problem to transform. Such cases can be transformed automatically. More complicated cases cannot be dealt with using our approach.

Chain of Responsibility

The intent of the Chain of Responsibility pattern [41, p.223] is to decouple the sender of a request from the ultimate receiver of the request. The request is passed along a chain of objects until one object finally handles it.

A starting point for this transformation that involves an object sending a request to various other objects, and testing if they have handled it, is likely to be an antipattern. A more suitable precursor starting point is where the receiver object is known to the sender, but a requirement has emerged to make this relationship more flexible. For example, in developing an application a programmer may start with a simple interface where any help request from the user is always handled by the same object. As the interface becomes more complicated, and a full graphical user interface is used, it will be necessary to introduce context-sensitive help. In this case, a user help request may be passed through several user interface objects until it reaches the appropriate one that can handle it.

The input to this transformation is the sender class and the receiver class. It proceeds then as follows:

1. Apply Wrapper to the receiver class to produce the chaining class.

2. Make the receiver class a subclass of the chaining class. This has the effect of making the default behaviour for any undefined method in the receiver class be delegation to the next object in the chain[15].

3. Apply AbstractAccess to the sender class so it uses the chaining class rather than the receiver class.

Any receiver object has now been made part of a null-terminated chain of objects of length 1. To add a new receiver class that handles any `foo` requests, the new receiver class should be made a subclass of the chaining class, the `foo` method should be removed from the existing receiver class (thus causing the

[15]This is a surprising and valuable reuse of the class produced by the Wrapper mini-transformation.

default delegation behaviour to come into play), and the required receiver object should be constructed and added to the end of the current chain of objects.

After application of this pattern, the programmer is left with some work to do to exploit the flexibility of the pattern structure. The precursor for this pattern is nevertheless plausible and the transformation does not present any serious problems[16].

Overall Assessment: Excellent.

Command

The intent of the Command pattern [41, p.233] is to encapsulate a request as an object. This enables a client to be parameterised with different requests, and supports queuing and logging of requests.

This pattern aims to loosen the coupling between the originator of a request and the receiver of the request. The originator is initialised with a command object that simply supports the operation execute. At some point the originator invokes execute on its command object and this sends the request to the receiver object.

The precursor is as follows. An instance of the originator class invokes the parameterless operation foo on its receiver object. The receiver object is passed to the originator class as an argument to its constructor (if it is created within the constructor we can use the parameteriseField refactoring to extract its construction). The only use the originator class makes of its

[16]Tokuda and Batory state of this pattern [96]: "there is no refactoring-enabled evolutionary path which leads to [its] use." We have nevertheless presented a successful transformation for this pattern. The reason is that the precursor actually simplifies matters by ensuring that the key behavioural abstractions are already packaged into methods so what remains is a mainly structural transformation.

receiver field is to invoke foo on it. The transformation proceeds as follows:

1. The createWrapperClass refactoring is used to partially wrap the receiver class. This creates the concrete command class that stores a reference to a receiver object and delegates the foo request to it.
2. Rename the foo method in the command class to execute.
3. Apply the ABSTRACTION minitransformation to the command class to produce the command interface.
4. The useWrapperClass refactoring is used to update all creations of originator objects to wrap the receiver parameter in a concrete command object. This concrete command object is stored in the originator class and any previous invocations of receiver.foo() are changed to command.execute().
5. Delete the receiver field from the originator class.

The precursor appears valuable, though quite constrained, and the transformation is satisfactory.
Overall Assessment: Partial.

Interpreter

The intent of the Interpreter pattern [41, p.243] is to enable the definition of the representation of a grammar, along with an interpreter that uses this representation to interpret sentences in the language defined by the grammar. This pattern is useful when the program being developed has to interpret a simple language that can be stored as an abstract syntax tree. Each grammar rule in the language is represented as a class and an interpret method is added to each class that defines how this part of the sentence is to be interpreted and processed.

The natural precursor is where a problem is represented and solved in some particular way, but it becomes necessary to deal with a more general problem, one that can be usefully specified as a simple language. For example, a program may allow the user to search for a string in a text file. A natural evolution of this facility would be to allow the user to specify a more general pattern to search for, and in this case it would be useful to specify the problem using a regular expression grammar.

Although the precursor is plausible, it is too vague to serve as a concrete starting point for an automated transformation. Nor does there appear to be any obvious precursor that could serve as a starting point for a transformation for this pattern[17].

Overall Assessment: Impractical.

Iterator

The intent of the Iterator pattern [41, p.257] is to enable sequential access to the elements of an aggregate object without exposing the underlying representation of the object. It allows multiple concurrent iterations over the aggregate object and does not expose the underlying structure of the aggregation.

The ideal starting point for this transformation would be simply an aggregate class that does not have any iterator yet. However, automatically extracting the structure of the aggregate and how to iterate through it is not feasible, so we seek a simpler precursor. A natural one is where the iterator

[17]In his work on automated pattern detection, Kyle Brown also classifies this pattern as too general to be detectable by an automated tool [13].

has been built into the aggregate class itself through the use of a cursor. This is common practice when prototyping an aggregate class initially, and will allow a single iteration to be active at any one time[18]. If the aggregate class becomes more widely used, the requirement for multiple concurrent iterations will surely arise, and this will require the application of the Iterator pattern.

The parameters to this transformation are the aggregate class itself and the iteration methods and fields that are part of this class. The iteration fields should only be accessed by the iteration methods. The transformation works as follows:

1. Copy the iteration methods and fields to the new iteration class, which is parameterised with an instance of the aggregate class and delegates any internally-generated, non-iterator requests to this instance. A form of the DELEGATION minitransformation can be used here, but the original aggregation class should remain unchanged for now.

2. Apply ABSTRACTION to the iterator class to produce an iterator interface. Apply ENCAPSULATECONSTRUCTION to the aggregate class with the iterator class as createe. This will add a construction method for the iterator class to the aggregate class that returns an iterator instance initialised with `this`.

3. Wherever in the program an instance of the aggregate class is iterated over, replace this with access via an iterator object.

4. Delete the iteration methods from the aggregate class.

[18]It is also the solution used by Bertrand Meyer to enable iteration though the elements of a list [66, p.192].

Step (3) may produce a clumsy result. If an aggregate object is partially iterated over and then passed as an argument to another method, the iterator will have to be passed in as well, and possibly then the aggregate object need not be passed. This is an example of the class-splitting problem discussed on page 136. Apart from this, the precursor for this transformation is plausible and the transformation generally compelling.
Overall Assessment: Partial.

Mediator

The intent of the Mediator pattern [41, p.273] is to define an object that encapsulates how a set of objects communicate. By centralising communication in the mediator object, coupling between the colleague objects is reduced, and knowledge of how they communicate is defined in one place rather than distributed across the colleague objects. This pattern works best when the colleague objects communicate in a well-defined way.

This pattern is similar to Facade [41, p.185], except that it allows for multidirectional communication between the colleague objects, rather than the unidirectional communication that Facade supports. As with Facade, there is little that can be done here by way of providing automated support. A mediator class can be introduced, but the analysis of the inter-object communication, so that it can be abstracted and centralised in the mediator, is a task that has to be performed by hand.
Overall Assessment: Impractical.

Memento

The intent of the Memento pattern [41, p.283] is to make it possible to capture and externalise the state of an object, and to restore the object to this state at a later time. This must occur without violating the encapsulation of the object.

A suitable precursor is as follows. The originator class supports two operations, say store and reset. Store requests the originator to make a copy of its state and store this internally in a field called state, while reset restores the originator to its earlier state. This reflects the intent of the Memento pattern, but not the flexibility. For example, a client (caretaker) cannot store multiple mementos; the originator can only store one. The transformation replaces the store and reset methods with createMemento and setMemento, and updates the caretaker classes to use these methods. A green field starting point for this design pattern transformation is possible as well, and would also be a practical starting point.

The input to this transformation is the originator class, the memento class, the store and reset methods and the state field.

1. The store and reset methods are copied to methods called createMemento and setMemento in the originator class.

2. The createMemento method is updated to create a local object of the class memento and to access this instead of the state field of the originator class. It returns this object at completion of the method's execution.

3. The setMemento method is similarly updated to take an argument of the class memento and to access this instead of the state field of the originator class.

4. The memento class is given an empty interface and AbstractAccess is used to update the createMemento and setMemento methods to use this interface rather than the memento class.

5. All caretaker classes that use the store and reset methods are updated to use createMemento and setMemento and to store the memento object locally[19].

6. The store and reset methods, and the state field are deleted from the originator class.

The precursor is not that useful in that it assumes that the essential memento aspects are present. The transformation then moves from a "one memento per originator object" situation to a more flexible"many mementos per originator object" situation. We used a similar precursor for the Iterator pattern, but it more likely that an aggregation class will provide an interface for iteration than that a given class will provide a store/reset interface as we have assumed here.
Overall Assessment: Partial.

Observer

The intent of the Observer pattern [41, p.293] is to define a dependency between a subject and a number of observer objects such that whenever the subject changes state, all the observers are notified of the change and can take appropriate action. A reasonable precursor would be where the relationship is one-to-one, i.e., there is a single observer object and the dependency between the subject and observer has been implemented in an *ad hoc* fashion. This is a reasonable design, though in the presence of a requirement to add

[19]There is an issue here in that we must know which reset matches which store. An invocation of reset will match the previous invocation of store, and while this is easy to work out in many cases, it is not decidable in general. This is an example of the class-splitting problem discussed on page 136.

more observers, it will be necessary to make the relationship more flexible by applying the Observer pattern.

Automating this transformation is a problem as the precursor described is too vague. The dependency between subject and observer could be implemented in many different ways. We could make progress by assuming that there is a single observer that uses the "attach/notify" protocol provided by the subject, and build a transformation that allows multiple observers to attach to the subject. We assess that this precursor is not a very likely structure to occur in practice. It is possible to provide the basic Observer structure for the programmer to work with, but we have not found a convincing precursor and transformation for this pattern.
Overall Assessment: Impractical.

State

The intent of the State pattern [41, p.305] is to enable an object to undergo a qualitative change in behaviour when its internal state changes. Rather than expressing this as extensive and similar case analysis in each method, this pattern defines a class to represent each possible state the object may be in. For example, a stream object will behave very differently depending on whether or not the file it is connected to is open or not. Rather than having a single stream class whose methods test whether or not the file is open, the State pattern would model this situation as two separate classes, one representing an open file, the other a closed file.

There is a very compelling precursor for this pattern. A class defines objects that can be in any one of a number of distinct states, and which state an object is in has a qualitative effect on behaviour. This will be evident because the methods of the class will contain a similar case analysis structure, e.g.,

```
if (someCondition){
        ...
}
else{
        ...
}
```

A class that contains several methods that have this structure can be split into two classes, one where someCondition is true and one where someCondition is false. The if...else statement can then be removed and simply replaced by the appropriate body of code.

The input to this transformation is the context class to be split, the condition that is to be used as a basis for the splitting, and the points in the methods of the class where the value of this condition changes. The transformation proceeds as follows:

1. Apply the DELEGATION minitransformation to the context class, so it now delegates all requests to a component object of the newly-created state class.

2. Apply ABSTRACTION to the state class and ABSTRACTACCESS to the context class, so the context class now only refers to the state class via the state interface.

3. For each interesting value of the given condition, create a subclass of the state interface. Simplify all case analysis in the methods of these classes based on the value the given condition is known to have[20].

[20]Opdyke presents a detailed description of how to simplify conditionals in [77, p.71].

4. Add a setState method to the context class that sets its local state field to the given instance of one of the state subclasses. At each of the points in the methods of the state subclasses where the given condition may change value, add an invocation of the setState method to set the new state object in the context class.

5. Update the creation of context objects to initialise them with the appropriate state object.

6. Delete the original (unsplit) state class that was created in step 1.

The structural aspects of this transformation can be automated, but in general user intervention is needed in assessing where a state change occurs.
Overall Assessment: Partial.

Strategy

This pattern has been fully dealt with in section 5.5.3. The precursor and transformation are compelling, though a small amount of refactoring work is left to the programmer at the end of the transformation.
Overall Assessment: Excellent.

Template Method

The intent of the Template Method pattern [41, p.325] is to enable a method to be expressed as a skeleton algorithm, thus deferring the details of the implementation to subclasses. Each subclass reuses the abstract algorithm defined in its superclass, and supplies the details that are specific to itself. For example, a search routine in an abstract container class could be described as follows:

```
boolean search(Element e){
    initSearch(e);
    while(!exhausted() && !found(e))
        advanceSearch(e);
    return !exhausted();
}
```

This method is in effect a high-level algorithm that describes searching[21]. Each concrete subclass of container will define initSearch, exhausted, found and advanceSearch in its own way.

The natural precursor for this pattern is where a method has been implemented in terms of the other concrete methods defined in its class. This is a normal situation, but in the face of a requirement to reuse the algorithm contained in the method, but not its detailed implementation in terms of the other methods of the class, the weakness of this tight coupling becomes clear. Applying the Template Method pattern in this situation separates the essential algorithm of the method from the methods it invokes, and allows the algorithmic abstraction to be reused.

The input to this transformation is the method to be templated. The transformation proceeds as follows:

1. Apply PARTIALABSTRACTION to the class of the method to produce an abstract class where the methods used by the method to be templated are defined to be abstract.

2. Update clients of the given class to use references to the abstract class instead (uses a form of ABSTRACTACCESS).

[21]Instances of the Template Method pattern are also referred as *hot spots*, as they describe a flexible part of the application that is open to change. An approach for automatically detecting hot spots is described in [87].

The transformation is simple and straightforward. The only weakness is that the precursor assumes that the components of the method to be templated have been encapsulated as methods. If this is not the case, a refactoring similar to Opdyke's convert_code_segment_to_function [77, p.53] could be used to encapsulate these code segments as methods.

Overall Assessment: Excellent.

Visitor

The intent of the Visitor pattern [41, p.331] is to enable an operation over an object structure to be defined separately from the object structure itself. For example, adding a new operation to a parse tree usually involves adding a method to each class of node in the tree, to define how the operation works for that type of node. This distributes a cohesive algorithm over several classes, which is not in general a desirable design. The Visitor pattern enables such an operation to be defined in one class, thus keeping all the details of the operation in one place[22].

The natural precursor for this pattern in where an operation has already been implemented as part of the object structure, and the programmer now wants to switch to a Visitor pattern solution to enable easy addition of other operations. The transformation can easily create the visitor interface and a concrete visitor class for the operation as well as adding the accept method to the classes of the object structure. However, the key step of taking the operation that has been distributed across the classes of the object structure and

[22]This does not come for free of course; the principle disadvantage of the Visitor pattern is that the class that defines the visitor operation must have knowledge of the classes defining the object structure. If these classes change, so too must the visitor class itself. This problem and the use of *subject-oriented programming* to resolve it are discussed in [21].

centralising this in the concrete visitor subclass cannot be fully automated using our techniques. The existing definition of the operation will probably combine operation-related code with traversal code in various ways. Separating out this code requires intervention from the programmer. So while a small part of this transformation may be automated, the precursor is not really being exploited to produce an interesting, behaviour-preserving transformation.
Overall Assessment: Impractical.

5.7 Analysis of Results

The results from the previous sections of this chapter are presented in complete form in table 5.1, and in summary form in table 5.2. These tables indicate a very satisfactory result. An excellent transformation was achieved for close to half the patterns considered, and in a further 26% of cases a workable, though partial, transformation was found.

The methodology worked very well for the creational patterns, but not so successfully for the structural patterns or behavioural patterns. It was to be expected that behavioural patterns would cause problems, but it is surprising that the results for the structural patterns were not better. Our approach is based on static analysis of the program, and so deals more easily with concrete program structure than with dynamic behaviour. The reason for this apparent anomaly is that although a pattern is assigned one of three categories, it may well contain elements from all three. For example, Abstract Factory is a very static, creational pattern but Builder, although also categorised as creational, has a distinct behavioural flavour as the objects in question are created in a dynamic "piecemeal" fashion.

Pattern Name	**Purpose**	**Assessment**
Abstract Factory	creational	Excellent
Builder	creational	Excellent
Factory Method	creational	Excellent
Prototype	creational	Excellent
Singleton	creational	Excellent
Adapter	structural	Excellent
Bridge	structural	Excellent
Composite	structural	Excellent
Decorator	structural	Partial
Facade	structural	Impractical
Flyweight	structural	Impractical
Proxy	structural	Partial
Chain of Responsibility	behavioural	Excellent
Command	behavioural	Partial
Interpreter	behavioural	Impractical
Iterator	behavioural	Partial
Mediator	behavioural	Impractical
Memento	behavioural	Partial
Observer	behavioural	Impractical
State	behavioural	Partial
Strategy	behavioural	Excellent
Template Method	behavioural	Excellent
Visitor	behavioural	Impractical

Table 5.1: Assessment of Design Pattern Transformations

Assessment	**No. of Patterns**	**Percentage**
Excellent	11	48%
Partial	6	26%
Impractical	6	26%

Table 5.2: Summary of Assessments

Other initially surprising results were those for Strategy (a behavioural pattern that worked well) and Facade (a structural pattern that failed). In the case of the precursor for Strategy, the behavioural aspects of the pattern are already encapsulated within methods. The transformation therefore just has to deal with the structure of this pattern, and this proved straightforward to handle. Facade presented the opposite problem. Its structure is easy to deal with, but there is also a behavioural element in how the client classes interact with the subsystem classes that are to be encapsulated, and this behavioural element could not be extracted and transformed.

Reuse of minipatterns is another important issue to consider. We hoped that the minipatterns uncovered during the development of the earlier design pattern transformations would prove useful in later developments. In table 5.3 we depict the reuse of minipatterns across the design pattern transformations. Note that for simplicity, when one transformation reuses another in its entirety (e.g., Abstract Factory uses Singleton), we depict this as reuse of the component minitransformations. Also, we omit from the table design patterns for which no satisfactory transformation was found.

It is clear from this table that we have achieved considerable reuse of the set of six minitransformations that were uncovered during development of transformations for the creational patterns and the sample structural and behavioural pattern. The actual reuse achieved is even stronger, as this table only depicts minitransformation reuse and ignores the reuse of refactorings such as createExclusiveComponent.

Pattern	**Abs**	**AbsAcc**	**Encap**	**Partial**	**Wrap**	**Deleg**
Abstract Factory	x	x	x	x		
Builder	x	x			x	
Factory Method	x	x	x	x		
Prototype	x	x				
Singleton				x		
Adapter	x	x			x	
Bridge					x	
Composite	x	x				
Decorator	x	x			x	
Proxy	x	x			x	
Chain of Responsibility		x			x	
Command	x				x	
Iterator	x		x			x
Memento		x				
State	x	x				x
Strategy	x	x				x
Template Method		x		x		

Table 5.3: Reuse of Minitransformations

The abbreviations in the table are as follows. **Abs**:ABSTRACTION, **AbsAcc**:ABSTRACTACCESS, **Encap**:ENCAPSULATECONSTRUCTION, **Partial**:PARTIALABSTRACTION, **Wrap**:WRAPPER, **Deleg**:DELEGATION.

5.7.1 Comments on the Development of the Transformations

Developing a transformation for a design pattern is not a trivial task. Insight and experience are necessary, and, as with any design task, many iterations were usually required before a satisfactory solution was reached. Our approach to demonstrating behaviour preservation demands that program behaviour be maintained at every step. This constrains the type of transformations we can use, in that the following structure is not permitted:

```
transformation_i // program behaviour is changed
...
transformation_j // program behaviour is reinstated
```

Although this overall chain is a refactoring and could be permitted, it will be disallowed because the application of $\mathsf{transformation}_i$ will be deemed to have changed program behaviour. It would be desirable to allow this type of chaining, but it would be extremely difficult to extend our approach to behaviour preservation so as to be able argue that a program has changed behaviour, and then changed back to its earlier behaviour[23]. The reason why we are able to reason about program behaviour so easily is that we need never be concerned with what the behaviour actually is, only that it has not been changed. To weaken this criterion would lose the relative simplicity of the approach that we have used.

That this type of erroneous composition is tempting is evidenced in Roberts's work. In [84, p.40] he presents a composite refactoring chain that

[23]Tokuda and Batory call this type of refactoring a *transactional* refactoring [96]. They propose allowing this type of refactoring but demanding that it operates in atomic mode, thus ensuring behaviour preservation. However, producing a semi-formal argument of behaviour preservation remains a problem.

creates a strategy object. Part of the composition involves the application of the moveMethod and moveField refactorings to move the strategy methods and fields from the context class to the strategy class. However, a precondition of his moveMethod refactoring is that the method must not access any fields of its current class. Clearly then, the program will be in an illegal state after the application of the moveMethod refactoring, and will only be returned to a legal state when the moveField refactoring has been applied[24]. We dealt with this problem by first "abstracting" the method from its class so it could be moved away and still access fields in that class. See section 5.5 for more details.

Scanning our catalogue of design pattern transformations, we observe that a transformation generally has three phases:

1. *Applying the design pattern structure.* This involves adding new classes, interfaces, methods etc. to the program. They are just added, not used, accessed or invoked, so arguing behaviour preservation for this stage is quite trivial. The changes made by this stage typically set the scene for the pattern, and would not make sense to perform on their own, unless the following step was performed as well.

2. *The operation-affecting step.* This is the "big step" that switches the program from its old inflexible structure to the more flexible pattern structure set up in the previous step. The precondition for this step is usually quite sophisticated, but has been largely set up by the previous step if all has gone well. It is therefore common that the precondition for this step does not contribute much to the precondition of the overall transformation.

[24]At the point in the derivation of the preconditions for the chain [84, p.41] where this should become apparent, the conflicting condition is omitted.

3. *Tidying up.* In this step any program elements that are no longer needed are deleted. The postcondition of the previous step must make it clear that they are no longer required. In many transformations, there is no need for this step, as no program elements are made redundant by the transformation.

There is a fractal element in this structure, in that a design pattern transformation may use a minitransformation that itself has this three-part structure. The actual low-level refactorings that are the foundation of this work do not have this structure however. They typically fit into one of the above three categories. For example, addClass clearly belongs to the first, replaceObjCreationWithMethInvocation to the second, and deleteClass to the third. Green field approaches to design pattern application need only to use the first step, that of setting up the pattern structure. The second and third steps are required in our approach as a direct result of our using a precursor as a starting point for the transformation, and demanding that the transformation be behaviour preserving.

5.7.2 Comments on Precondition Computation

In this section we make some general observations about the process of precondition computation.

- It is not a simple task.
- It can be applied rapidly with experience, though doing it step-by-step as in chapter 3 is very tedious.
- Usually earlier refactorings set up the preconditions for later ones, so even though the overall transformation can be quite complicated, the precondition is usually not too extensive.

- Computing preconditions was a very useful process. Frequently it uncovered aspects of the transformation that might otherwise have been missed. For example, the fact that the Factory Method transformation cannot be applied if the Creator class uses a static method of the Product class is not obvious in itself. However the process of computing the precondition for this transformation brought this aspect to the foreground (see section 4.4.1).

5.8 Related Work

In chapter 4 we discussed related work in the general area of automated design pattern transformations. Specific details of how other approaches deal with particular patterns were considered in this chapter as part of the analysis of the relevant pattern.

5.9 Summary

We have rigorously applied our proposed methodology to the entire set of Gamma *et al* creational patterns, and to a sample structural and behavioural pattern. For the remaining Gamma *et al* patterns, we assessed if they were amenable to our approach and, where possible, proposed a precursor and sketched a transformation. Our results were promising in that for most patterns a workable solution could be found, and there proved to be extensive reuse of the minitransformations that were developed during this work.

Chapter 6

Conclusions

This chapter concludes the thesis. In section 6.1 we state again the contributions that have been made by this research. In section 6.2 we present a number of proposals for future work that would extend this research, and finally, in section 6.3, we make some concluding remarks.

6.1 Contributions

The principle contributions of this thesis were stated in chapter 1. Here we restate them:

- *A methodology for developing design pattern transformations.* This is the essential contribution of this work. The methodology we have developed has been applied with full rigour to seven common design patterns[1], and a prototype software tool has been built that can apply these seven design patterns to Java programs. The methodology has also been applied to the remaining patterns in the Gamma *et al* pat-

[1]The seven design patterns to which the methodology has been fully applied are Abstract Factory, Factory Method, Singleton, Builder, Prototype, Bridge and Strategy [41].

tern catalogue [41], though these pattern transformations have not been prototyped. The essence of our methodology has been published in summary form in [74, 72], and more completely in [75].

- *A minitransformation library.* Design pattern transformations have a strong degree of commonality and this has been captured in a set of six minitransformations. These minitransformations have been implemented and demonstrated to be widely applicable in developing design pattern transformations.

- *A model for behaviour-preservation proofs.* The transformations we develop must be invariant with respect to program behaviour. In order to prove this rigorously for the sophisticated program transformations that we develop, we have extended existing refactoring work by allowing the transformation definition to contain not only simple sequences, but also iteration and conditional statements. This model has been applied in full rigour to several examples, and has been published in [76].

Other contributions are:

- *The notion of Precursor.* We introduced the notion of a precursor for a design pattern, i.e., a design structure that expresses the intent of the design pattern in a simple way, but that would not be regarded as an example of poor design. We demonstrated the usefulness of this notion by developing precursors for the Gamma *et al* design patterns, and using them as starting points for our design pattern transformations. This set of precursors provides an insight into the type of program to which a given pattern can be applied.

- *A refactory for Java.* The lowest layer of transformations is a collection of refactorings that can be applied to a Java program, and this can serve as a basis for other transformation work. An extensive set has been designed and implemented, and these are described in appendix B. Some are naturally similar to existing refactorings, while others are peculiar to the development of design pattern transformations.

- *A Precondition Categorisation.* In section 4.4.1 we described how each clause of the precondition to a design pattern transformation can be put into one of four categories. We also described how this categorisation can be used in practice to decide how to deal with the failure of a precondition clause.

6.2 Future Work

In the following subsections we consider possible future work in the area of this thesis.

Practical Tests of the Design Pattern Tool (DPT)

The software prototype we have built as part of this work, DPT, has been tested on several sample programs to establish a base-level confidence that it operates correctly. Naturally, extensive further testing and updating would be required to bring the quality of this prototype to production level.

A more interesting issue in this context relates to programmer acceptance of the type of transformation DPT performs. DPT makes sweeping changes to a program when it applies a pattern, and it is an open question whether a programmer would be content to allow a large system to be updated in this way. Indeed, a software tool can fail in practice for any number of reasons [83], and arguing abstractly that it is nevertheless useful is futile. The author's position is that a programmer will use a software tool only if they have a very clear mental model of what the tool does. Compilers, debuggers and profilers all fit into this category. As design patterns become more established, we can expect programmers to become more comfortable with the type of transformations DPT applies.

One way to aid the programmer's comprehension of the transformation DPT has applied is to present each of the program changes to them and ensure that they are satisfied with each one. If they are not, the whole program can be rolled back to its pre-transformation state. A more ambitious approach is to try to explain the pattern to the programmer (depending on their pattern expertise), and put the changes in this context. Note that existing work in the area of program comprehension has focused on comprehension as part of software maintenance (e.g., [86]). The problem described here, that of presenting the effects of a large refactoring in a comprehensible manner, is a future challenge for this field.

Further Construction of Pattern Transformations

Our refactorings and minitransformations provide a library of reusable components for design pattern transformation development. As with any such library, many iterations are required to fully comprehend the domain and to provide a stable set of components. With each new design pattern development, our understanding of the minitransformations was refined, and frequently this resulted in the refactoring of the library itself. We do not claim that this process is complete. As more design pattern transformations are developed using this approach we can expect more minitransformations to appear and the existing ones to require further work and refinement.

Automation

At present the construction of the behaviour preservation arguments is fragile, in that any change made to a low-level refactoring or analysis function requires that all proofs that use this refactoring or analysis function be rechecked. This dependency itself is unavoidable, but automated software support would be very useful to help manage it. A repository of refactorings, analysis functions and helper functions could be created and this used in performing syntax checking and typechecking of the behaviour preservation arguments. For example, if testing of DPT reveals that the precondition of a refactoring is not strong enough, the specification of this refactoring would then be updated in the repository. The automated assistance software could then highlight which minitransformations and design pattern transformations have to be revisited.

More ambitiously, an attempt could be made to automate the construction of the behaviour preservation argument. This is a challenging task, as we currently use semantic knowledge in building the behaviour preservation arguments. To completely formalise this would involve working with a formal semantics for Java (e.g., [47, 99]), and this would be likely to run into tractability problems. Partial automation is a more promising approach to take, and it would be interesting to see what contribution such an approach could make to the computation of pre- and postconditions for a design pattern transformation.

Pattern Maintenance

Applying a design pattern changes the program code, and some of these changes must be maintained in order for the pattern to remain intact. This means that certain constraints are put on the possible future evolutions of the program. For example, in a program where the Factory Method pattern has been applied, the addition of a new Product class means that a new method must be added to the Creator class as well.

Developing tool support to manage and check these constraints is a valuable extension to our work. The postcondition for a design pattern transformation provides a basis from which to develop the constraints associated with a design pattern. These constraints can be defined using our analysis functions. This enables a software tool to manage the constraints associated with patterns that have been applied to the program, and to notify the programmer if they are updating code that relates to a pattern. The programmer may be advised that their updates are violating a pattern-related constraint, and informed of what other changes are necessary in order to re-establish the pattern constraints.

Language Independence

In our work we focused on the application of design patterns to Java programs. This raises the question of the extent to which our approach is applicable to other programming languages. Some refactorings and minitransformations are applicable to any class-based, object-oriented language, while others are quite Java-specific, for example, those that deal with interfaces.

One approach would be to use the Template Method pattern to describe abstractly how the design pattern transformation operates, and provide the language specific details in subclasses. This is certainly possible; whether it is actually useful depends on the degree of commonality between a set of design pattern transformations that each apply the same pattern, but to programs written in different languages. All refactoring work to date has been language-specific, so this direction would present an interesting challenge.

Pre-transformation Refactorings

For each design pattern transformation we compute its pre- and postconditions, and add its precursor precondition where necessary. This precondition characterises the type of program to which the design pattern transformation can be applied. In section 4.4.1 we categorised the different types of precondition that a design pattern transformation can have. We stated that if a *refactoring precondition* fails, the program can be automatically refactored to correct the problem, and the transformation then applied.

We can view the design pattern transformation as describing a prototypical transformation. If a refactoring precondition fails, the program has to be massaged into a suitable state so that the prototypical transformation can be performed. This is an area for future investigation, and has the potential to make the transformations we have developed applicable to a much broader range of programs.

Pattern Applicability

Our current preconditions simply ensure that the design pattern transformation can be applied without changing program behaviour. It is left up to the programmer to decide if applying the pattern is a good idea or not. We argued strongly in section 2.2 that there are aspects of patterns that require human insight, and that automated attempts to locate suitable places to apply a pattern are of limited value.

However, a software tool could do more in terms of assessing whether the pattern is applicable or not, by asking the programmer certain questions about their intention. For example, in applying the Visitor pattern, the tool might ask the programmer "Do you expect the classes in the object structure to change often?" The answers from the programmer may cause the tool to suggest that the pattern is not a suitable solution, or to configure the exact manner in which the pattern is applied.

Pattern Removal

An over-zealous programmer might apply a pattern even though it is not required, thus obscuring the program rather than enhancing its clarity [84, p.23]. It might also be useful to optimise a program prior to compilation by removing any unnecessary patterns, as they typically have a detrimental effect on runtime performance. An interesting extension to our work is therefore to develop transformations that remove patterns, rather than apply them. In this case, the design pattern structure is the starting point for the transformation, and the corresponding precursor is the target. The informal statement of the starting point for this type of transformation would be "the design pattern structure is present, but its flexibility is not required."

This is not as simple as defining an inverse for each refactoring, and applying them in reverse order. Many refactorings require extra state to be maintained in order to define their inverse. For example, the inverse of a refactoring that deletes an unused class must have access to the deleted class in order to restore it. Even if this extra state is maintained, any changes to the program between the pattern being applied and it being removed might render the inverse refactorings unusable. This area may be interesting to look at, though it is obviously of less impact than the application of design patterns[2].

[2]Unless of course the current interest in design patterns turns to disdain, and the software industry starts "reengineering to depatternise."

6.3 To Conclude

We stated the fundamental thesis of this work in chapter 1 as follows:

> *Automating the application of design patterns to an existing program in a behaviour preserving way is feasible.*

The research presented in this dissertation has demonstrated the validity of our original thesis. In section 5.7 we found that an excellent transformation was constructed for close to half the patterns considered, and in only 26% of cases could no useful precursor or transformation be found. For seven of the design patterns considered, a rigorous argument of behaviour preservation was also developed. We achieved strong reuse of the minitransformations, as is depicted in figure 5.3 on page 155.

Design patterns have been gaining acceptance in the software engineering community, though the lack of formalisation or automated support has been a weakness of this field. Refactoring has also been gaining support, though again, most of the recent interest has been in non-automated approaches. We have contributed to the formalisation of the refactoring field, and used our contribution to develop a rigorous and practical approach to the automated application of design patterns.

Bibliography

[1] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[2] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, Reading, Massachusetts, first edition, 1996.

[3] Kent Beck. *Extreme Programming Explained.* Addison Wesley Longman, Reading, Massachusetts, first edition, 2000.

[4] PerOlof Bengtsson and Jan Bosch. Haemo dialysis software architecture design experiences. In *Proceedings of the International Conference on Software Engineering*, pages 516–525, Los Angeles, 1999. ACM Press.

[5] Keith Bennett and Vaclaw Rajlich. Software maintenance and evolution: A roadmap. In Anthony Finkelstein, editor, *The Future of Software Engineering*, New York, 2000. ACM Press. Produced as part of ICSE 2000, Limerick, Ireland.

[6] Keith H. Bennett. Do program transformations help reverse engineering? In *Proceedings of the International Conference on Software Maintenance*, Maryland, November 1998. IEEE Press.

[7] Paul Bergstein. Object-preserving class transformations. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 299–313, Phoenix, Arizona, 1991. ACM Press. SIGPLAN Notices, Vol. 26, 11 (November).

[8] Lucy M. Berlin. When objects collide: experiences with reusing multiple class hierarchies. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 181–193, Ottawa, Canada, October 1990. ACM Press.

[9] Blueprint Technologies, McLean, VA. *Framework Studio 1.5*, 2000.

[10] Grady Booch. *Object-oriented analysis and design with applications.* Benjamin/Cummings, Redwood City, California, second edition, 1994.

[11] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In Eric Jul, editor, *Proceedings of the European Conference on Object-Oriented Programming*, Brussels, July 1998. LNCS.

[12] Michael L. Brodie and Michael Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach.* Morgan Kaufman, San Francisco, 1995.

[13] Kyle Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, North Carolina State University, Computer Engineering Department, 1996. available from: http://hometown.aol.com/kgb1001001/Articles/THESIS/thesis.htm.

[14] F. J. Budinsky et al. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.

[15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, Chicester, first edition, 1996.

[16] Eduardo Casais. *Managing Evolution in Object Oriented Environments: an Algorithmic Approach.* PhD dissertation, University of Geneva, Faculty of Economic and Social Sciences, 1991.

[17] Eduardo Casais. An incremental class reorganization approach. In O. Lehrmann Madsen, editor, *Proceedings of the European Conference on Object-Oriented Programming*, pages 114–131, Utrecht, June 1992. LNCS.

[18] Eduardo Casais. Automatic reorganization of object-oriented hierarchies: a case study. *Object Oriented Systems*, 1(2):95–115, December 1994.

[19] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery - a taxonomy. *IEEE Software*, pages 13–17, January 1990.

[20] Franco Civello. Roles for composite objects in object-oriented analysis and design. In *Object-Oriented Programming Systems, Languages and Applications Conference*, Washington, September 1993. ACM Press.

[21] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Object-Oriented Programming Systems, Languages and Applications Conference*, Denver, November 1999. ACM Press.

[22] J. Craig Cleaveland. *An introduction to data types.* Addison-Wesley, Reading, Massachusetts, 1986.

[23] James O. Coplien. Software design patterns: Common questions & answers. In *Proceedings of Object Expo New York*, pages 39–42, New York, June 1994. SIGS Publications.

[24] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. A pattern language for reverse engineering. In *Fifth European Conference on Pattern Languages of Programs (EuroPLoP)*, Irsee, Germany, July 2000.

[25] Serge Demeyer, T.D. Meijler, and Matthias Riegler. Towards design pattern transformations. In *ECOOP Workshop Object-Oriented Software Evolution and Re-Engineering*, Finland, June 1997. Springer-Verlag LNCS 1241.

[26] Rick Dewar et al. Identifying and communicating expertise in systems reengineering: a patterns approach. *IEE Proceedings - Software*, 146(3):145–152, 1999.

[27] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., first edition, 1976.

[28] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 109–118, Oxford, September 1999. IEEE Press.

[29] Stéphane Ducasse, Matthias Rieger, and Georges Golomingi. Tool support for refactoring duplicated oo code. In Ana Moreira and Serge Demeyer, editors, *Object-Oriented Technology: ECOOP'99 Workshop Reader*, number 1743 in LNCS, Lisbon, June 1999. Springer Verlag.

[30] A. H. Eden, J. Gil, and A. Yehudai. Precise specification and automatic application of design patterns. In *Proceedings of the Twelfth IEEE International Automated Software Engineering Conference*, Nevada, November 1997. IEEE.

[31] A. H. Eden, Yossi Gil, Y. Hirshfeld, and A. Yehudai. Motifs in object oriented architecture. Technical report, Uppsala University, Department of Information Technology, 1999. Available from: http://www.cs.concordia.ca/~faculty/eden.

[32] A. H. Eden, Y. Hirshfeld, and K. Lundqvist. LePUS - symbolic logic modeling of object oriented architectures: A case study. In *Proceedings of the Second Nordic Workshop on Software Architecture (NOSA'99)*, Ronneby, Sweden, August 1999.

[33] A. H. Eden, Y. Hirshfeld, and A. Yehudai. Towards a mathematical foundation for design patterns. Technical report 1999-004, Uppsala University, Department of Information Technology, 1999.

[34] A. H. Eden and A. Yehudai. Tricks generate patterns. Technical report 324, Tel Aviv University, Department of Computer Science, 1997.

[35] R. Fanta and V. Rájlich. Removing clones from the code. *Journal of Software Maintenance*, 11(4):113–243, July 1999.

[36] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support in design patterns. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming*, pages 472–495. LNCS vol. 1241, June 1997.

[37] Brian Foote and William Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In J.O. Coplien and D.C. Schmidt, editors, *Pattern Languages of Programming*, Monticello, Illinois, 1995.

[38] Martin Fowler. *Refactoring: improving the design of existing code.* Object Technology Series. Addison-Wesley Longman, Reading, Massachusetts, first edition, 1999.

[39] Richard Gabriel. Pattern definitions. Available from: http://hillside.net/patterns/definition.html, 1995+.

[40] John Gall. *Systemantics: How systems work and especially how they fail.* Quadrangle/New York Times Book Company, New York, first edition, 1977.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, first edition, 1995.

[42] J. Paul Gibson, Thomas F. Dowling, and Brian A. Malloy. The application of correctness preserving transformations to software maintenance. In Lionel Briand and Jeffrey M. Voas, editors, *Proceedings of the International Conference on Software Maintenance*, San José, October 2000. IEEE Press.

[43] Mark Grand. *Patterns in Java*, volume 1. Wiley, New York, 1998.

[44] Mark Grand. *Patterns in Java*, volume 2. Wiley, New York, 1999.

[45] Walter L. Hürsch and Linda M. Seiter. Automating the Evolution of Object-Oriented Systems. In *International Symposium on Object*

Technologies for Advanced Software, pages 2–21, Kanazawa, Japan, March 1996. Springer Verlag, Lecture Notes in Computer Science.

[46] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. In Elisa Bertino, editor, *Proceedings of the European Conference on Object-Oriented Programming*, pages 129–153, Sophia Antoplis and Cannes, June 2000. LNCS.

[47] Bart Jacobs, Joachim van den Berg, Marieke Huisman, and Martijn van Berkum. Reasoning about java classes (prelimary report). In *Object-Oriented Programming Systems, Languages and Applications Conference*, Vancouver, October 1998. ACM Press.

[48] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley, Reading, Massachusetts, 1997.

[49] Jens Jahnke and Albert Zündorf. Rewriting poor design patterns by good design patterns. In Serge Demeyer and Harald Gall, editors, *ESEC/FSE Workshop on Object-Oriented Reengineering*, Zürich, September 1997. University of Vienna technical report.

[50] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[51] Ralph Johnson and William Opdyke. Refactoring and aggregation. In S. Nishio and A. Yonezawa, editors, *Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, November 1993. LNCS vol. 742.

[52] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *Pro-*

ceedings of the International Conference on Software Engineering, Los Angeles, 1999. ACM Press.

[53] Andrew R. Koenig. Patterns and antipatterns. *Journal of Object-Oriented Programming*, April 1995.

[54] Walter F. Korman. Elbereth: Tool support for refactoring java programs. Master's project, University of California, San Diego, Department of Computer Science and Engineering, June 1998.

[55] Alexej Kupin. Design and development of program transformation tool. Master's thesis, University of Munich, Department of Computer Science, August 2000.

[56] Anthony Lauder and Stuart Kent. Precise visual specification of design patterns. In Eric Jul, editor, *Proceedings of the European Conference on Object-Oriented Programming*, Brussels, July 1998. LNCS.

[57] Anthony Lauder and Stuart Kent. Legacy System Anti-Patterns and a Pattern-Oriented Migration Response. In P. Henderson, editor, *Systems Engineering for Business Process Change*. Springer Verlag, January 2000.

[58] H.K.N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance*, pages 290–301, San Diego, November 1990.

[59] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. From objects to classes: algorithms for optimal object-oriented design. *Journal of Software Engineering*, 6(4):205–228, July 1991.

[60] Karl J. Lieberherr, Ian Holland, and Arthur Riel. Object-oriented programming: An objective sense of style. In Norman K. Meyrowitz, editor, *Object-Oriented Programming Systems, Languages and Applications Conference*, San Diego, September 1988. ACM Press.

[61] Karl J. Lieberherr and Cun Xiao. Minimizing depenency on class structures with adaptive programs. In S Nishio and A. Yonezawa, editors, *International Symposium on Object Technologies for Advanced Software*, Kanazawa,Japan, April 1993. Springer Verlag.

[62] Jaques Loeckx and Kurt Sieber. *The Foundations of Program Verification*. Wiley&Sons, 1987.

[63] Katsuhisa Maruyama and Ken ichi Shima. Automatic method refactoring using weighted dependence graphs. In *Proceedings of the International Conference on Software Engineering*, pages 236–245, Los Angeles, 1999. ACM Press.

[64] Marco Meijers. Tool support for object-oriented design patterns. Master's project, Rijksuniversiteit Utrecht, Department of Computer Science, August 1996.

[65] Metamata, Fremont, CA. *JavaCC - The Java Parser Generator*, 2000. Available from: http://www.metamata.com/JavaCC/.

[66] Betrand Meyer. *Object Oriented Software Construction*. Prentice Hall, Hemel Hempstead, first edition, 1988.

[67] Ivan R. Moore. Guru - a tool for automatic restructuring of self inheritance hierarchies. In *TOOLS USA*, pages 267–275. Prentice-Hall, 1995.

[68] Ivan R. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 235–50, San José, October 1996. ACM.

[69] Ivan R. Moore and Tim P. Clement. A simple and efficient algorithm for inferring inheritance hierarchies. In *TOOLS Europe*, pages 173–184, Paris, February 1996. Prentice-Hall.

[70] Thomas Mowbray et al. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* The Art of Computer Programming. John Wiley & Sons, March 1998.

[71] Mel Ó Cinnéide. Towards automating the introduction of the decorator pattern to avoid subclass explosion. In *OOPSLA Object-Oriented Evolution and Re-engineering Workshop*, San José, October 1996. ACM Press. Available as TR-97-7, Department of Computer Science, University College Dublin, Ireland.

[72] Mel Ó Cinnéide. Automated refactoring to introduce design patterns. In Jeff Magee and Mauro Pezzè, editors, *Proceedings of the International Conference on Software Engineering (Doctoral Workshop)*, pages 722–724, Limerick, June 2000. ACM Press.

[73] Mel Ó Cinnéide and Paddy Nixon. Program restructuring to introduce design patterns. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology: ECOOP'98 Workshop Reader*, number 1543 in LNCS, Brussels, July 1998. Springer Verlag.

[74] Mel Ó Cinnéide and Paddy Nixon. Automated application of design patterns to legacy code. In Ana Moreira and Serge Demeyer, edi-

tors, *Object-Oriented Technology: ECOOP'99 Workshop Reader*, number 1743 in LNCS, Lisbon, June 1999. Springer Verlag.

[75] Mel Ó Cinnéide and Paddy Nixon. A methodology for the automated introduction of design patterns. In Hongji Yang and Lee White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 463–472, Oxford, September 1999. IEEE Press.

[76] Mel Ó Cinnéide and Paddy Nixon. Composite refactorings for Java programs. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *ECOOP Workshop on Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[77] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.

[78] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications*, New York, September 1990.

[79] Wolfgang Pree. *Design patterns for object-oriented software development*. ACM Press books. Addison-Wesley, Wokingham, 1995.

[80] Winnie Pun and Russel Winder. Automating class hierarchy graph construction. Research note RN/89/23, University College London, Deptment of Computer Science, March 1989.

[81] Trygve Reenskaug. *Working with Objects: The OOram Software Engineering Method.* Manning Publications Co., Greenwich, Connecticut, 1995.

[82] Arthur J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, Reading, Massachusetts, first edition, 1996.

[83] Don Roberts and John Brant. "good enough" analysis for refactoring. In Stéphane Ducasse and Joachim Weisbrod, editors, *ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, Brussels, July 1998. FZI Karlsruhe report.

[84] Donald Roberts. *Eliminating Analysis in Refactoring.* PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 1999.

[85] Donald Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.

[86] Sébastien Robitaille, Reinhard Schauer, and Rudolf K. Keller. Bridging program comprehension tools by design navigation. In Lionel Briand and Jeffrey M. Voas, editors, *Proceedings of the International Conference on Software Maintenance*, San José, October 2000. IEEE Press.

[87] Reinhard Schauer, Sébastien Robitaille, Rudolf K. Keller, and François Martel. Hot spot recovery in object-oriented software with inheritance and composition template methods. In Hongji Yang and Lee White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 220–229, Oxford, September 1999. IEEE Press.

[88] Benedikt Schulz. Behaviour preserving reorganisation of object-oriented systems and adaptive programming. In Serge Demeyer and

Harald Gall, editors, *ESEC/FSE Workshop on Object-Oriented Reengineering*, Zürich, September 1997. University of Vienna technical report.

[89] Benedikt Schulz. Design patterns as operators implemented with refactorings. In Stéphane Ducasse and Joachim Weisbrod, editors, *ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, Brussels, July 1998. FZI Karlsruhe report.

[90] Benedikt Schulz, Thomas Genssler, Berthold Mohr, and Walter Zimmer. On the computer aided introduction of design patterns into object-oriented systems. In *Proceedings of the 27th TOOLS conference.* IEEE CS Press, 1998.

[91] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, pages 99–110, Lake Buena Vista, Florida, November 1998. ACM Press.

[92] Perdita Stevens and Rob Pooley. Systems reengineering patterns. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, pages 17–23, Lake Buena Vista, Florida, November 1998. ACM Press.

[93] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[94] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design patterns application in UML. In Elisa Bertino, editor, *Proceedings of the European Conference on Object-Oriented Programming*, pages 44–62, Sophia Antoplis and Cannes, June 2000. LNCS.

[95] Peter F. Sweeney and Frank Tip. A study of dead data members in C++ applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 324–332, Montreal, June 1998. ACM Press.

[96] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, Florida, October 1999. IEEE Press. An extended version will appear in the Journal of Automated Software Engineering.

[97] Lance Aiji Tokuda. *Evolving Object-Oriented Designs with Refactorings*. PhD dissertation, Department of Computer Sciences, University of Texas at Austin, September 1999.

[98] Paolo Tonella and Giulio Antoniol. Object oriented design pattern inference. In Hongji Yang and Lee White, editors, *Proceedings of the International Conference on Software Maintenence*, pages 230–238, Oxford, September 1999. IEEE Press.

[99] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer, 1999.

[100] Steven Woods, S. Jeromy Carriere, and Rick Kazman. A semantic foundation for architectural re-engineering and interchange. In Hongji Yang and Lee White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 391–398, Oxford, September 1999. IEEE Press.

[101] Walter Zimmer. *Frameworks und Entwurfsmuster.* PhD dissertation, Forschungszentrum Informatik Karlsruhe, 1997.

Appendix A

The Factory Method Pattern

Design patterns were introduced in section 2.2. In this appendix we provide a more detailed description of the Factory Method pattern, as a transformation that introduces this pattern was developed in detail in chapter 4. For more detail see [41], which is also the source of the example we use here.

The Factory Method pattern is used to loosen the coupling between a class (Creator) and another class that it instantiates (Product). Specifically, it enables the Creator class to defer instantiation to a subclass; in this way it is easy to extend the Creator class to work with a new type of Product class.

For example, consider a framework that can present multiple documents to the user. Two key abstract classes in this domain are Application and Document. The designer has to subclass these classes in order to realise the required functionality. Consider for example using these classes to build a drawing application. The designer would create a subclass of Application, DrawingApplication, and a subclass of Document, DrawingDocument. The Application class is responsible for creating and managing Documents, but it only knows *when* it should create a Document; it does not know *what kind*

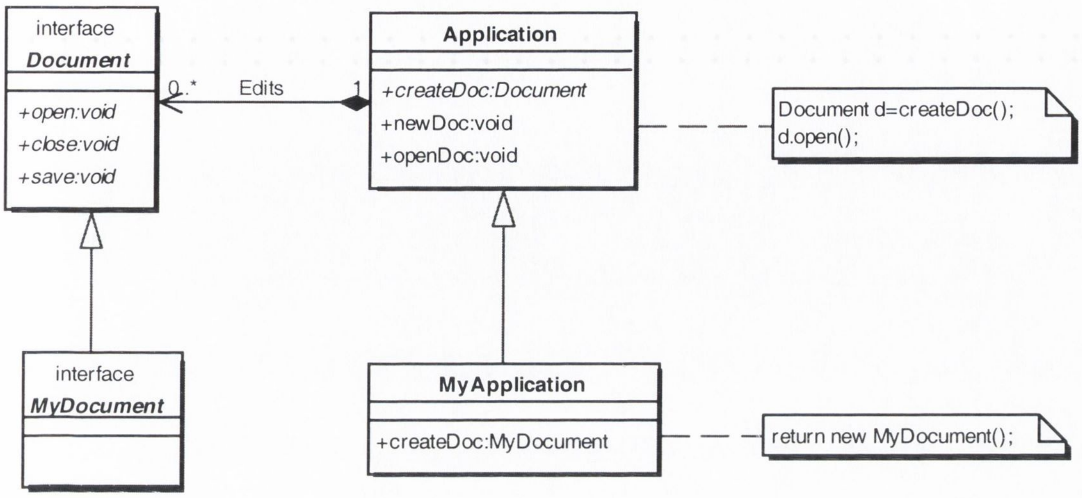


Figure A.1: The Factory Method pattern structure

of Document to create. This is the kernel of the problem: the framework must create instances of Document, but it knows nothing of the concrete Document classes it should instantiate.

The Factory Method pattern offers a solution to this problem (see figure A.1). It encapsulates the knowledge of which Document to create and defers this to a subclass. The abstract Application class invokes an abstract method, createDoc, whenever it needs to create a Document object. Each concrete subclass of Application must now override the createDoc method to create and return an instance of the appropriate type of Document. In figure A.1, the MyApplication class redefines the createDoc method to return an instance of MyDocument. The other methods in Application work with this instance through the Document interface.

Appendix B

Analysis Functions, Helper Functions and Primitive Refactorings

This appendix contains the complete specification of all analysis functions, helper functions and primitive refactorings that are used in this work. These topics were introduced in chapter 3. In section B.1 we detail the analysis functions we have made use of. In section B.2 the helper functions are listed, and finally, in section B.3, the primitive refactorings used in this work are specified. As an aid to the reader, an alphabetical listing of all analysis functions, helper functions and primitive refactorings, together with relevant page numbers, is presented in table B.1 on page 187.

B.1 Analysis Functions

In this section we describe the analysis functions (section 3.1.2) that are used to extract information from the program being transformed. They serve a

Name	Kind	Page	Name	Kind	Page
absorbParameter	AF	188	isClonable	AF	191
abstractClass	HF	197	isExclusiveComponent	AF	192
addClass	PR	204	isInterface	AF	192
addGetMethod	PR	206	isPrivate	AF	192
addImplementsLink	PR	207	isPublic	AF	192
addInterface	PR	208	isStatic	AF	192
addMethod	PR	209	isSubtype	AF	193
addSingletonMethod	PR	210	localVars	AF	193
argument	AF	188	makeAbstract	HF	200
classCreated	AF	188	makeConstructorProtected	PR	213
classOf	AF	188	methodsInvoked	AF	193
constructorInvoked	AF	189	moveMethod	PR	214
containingClass	AF	188	nameOf	AF	193
containingMethod	AF	189	noOfArguments	AF	193
contextFree	AF	189	noOfParameters	AF	193
createEmptyClass	HF	198	parameter	AF	193
createExclusiveComponent	PR	212	parameteriseField	PR	216
createsSameObject	AF	189	pullUpMethod	PR	218
createWrapperClass	HF	198	replaceClassWithInterface	PR	221
declares	AF	189	replaceObjCreationWith...	PR	222
defines	AF	190	returnsObject	AF	193
equalInterface	AF	190	returnsSameObject	AF	194
exhibitSameBehaviour	AF	190	returnType	AF	194
hasSingleInstance	AF	191	sigOf	AF	194
implementsInterface	AF	191	superclass	AF	194
initialises	AF	191	superclasses	AF	194
isAbstract	AF	191	typeOf	AF	194
isClass	AF	191	useWrapperClass	PR	223

Table B.1: Alphebetical Listing of Analysis Functions (AF), Helper Functions (HF) and Primitive Refactorings (PR), with relevant page numbers.

dual role, in that they are used both in specifying the preconditions to the refactorings, and as a transformation programmer's view of the program to which a design pattern transformation is being applied.

Some of the analysis functions are obviously easy to evaluate, while others are more difficult. A number are intractable in the general case, namely contextFree, createsSameObject, hasSingleInstance, isClonable, isExclusiveComponent, returnsObject, returnsSameObject, and uses. In section 3.1.2 we described the possible ways in which intractable analysis functions can be handled.

For each analysis function we specify its name, return type, argument types, and provide a brief textual description of what its purpose is. The listing is in alphabetical order.

Argument **argument**(MethodInvocation/ObjectCreationExprn *invocation*, int i): Returns the i^{th} argument to the given method invocation or object creation expression, or $\perp$ if no such argument exists.

Class **classCreated**(ObjectCreationExprn e): Returns the class of the objects that can be created by the given object creation expression.

Class **classOf**(Constructor/Method/Field a): Returns the class to which the given constructor/method/field belongs. The condition classOf(a)=c is also written as $a \in c$.

Class **containingClass**[1](ObjectRef o): Returns the class that contains the given object reference.

[1]We do not apply the analysis function $classOf$ to an object reference, as this would suggest its type rather than its containing class. The $typeOf$ analysis function is used to determine the type of an object reference.

Method **containingMethod**(ObjectRef/ObjectCreationExprn *e*): Returns the method containing the given object reference or object creation expression.

Constructor **constructorInvoked**(ObjectCreationExprn *e*): Returns the constructor that is invoked by the given object creation expression. In Java this can be determined based on the static types of the arguments to the constructor; a dynamic analysis is not required.

Boolean **contextFree**(Expression *e*): Returns true iff the expression *e* has the same effect regardless of the context in which it is evaluated. It may create a new object, or update a global object, but the effect must be the same regardless of the method in which it is invoked. The impact of this is that if this expression is passed as an argument to a method, we can move the evaluation into the method without changing program behaviour.

Boolean **creates**(Class *c1*, Class *c2*): Returns true iff a method in the class *c1* creates an instance of the class *c2*.

Boolean **createsSameObject**(Constructor *c*, Method *m*): Returns true iff the method *m* creates and returns a new object of the same class and in precisely the same state as would be created by *c*, given the same argument list. *m* must have no other side-effects; in particular it must neither access any variables other than its parameters, nor send a message to another object. See also the weaker condition, returnsSameObject.

Boolean **declares**(Class *c*, String *n*, String *s*): Returns true iff the class *c* contains a method named *n* of signature *s* in its interface. An implementation need not be provided, and the declaration of $n(s)$ may appear as an abstract method in a superclass or as an element of an interface that c (in)directly implements. The parameter *direct* is added if the test is only to refer to the class c itself, and, for simplicity, a method may be provided as parameter instead of the method name and signature. The parameter c may also be an interface, with the natural interpretation.

Boolean **defines**[2](Class c, String n, Signature s): Returns true iff a concrete method called n of signature s is contained in the class c or one of its superclasses. If no signature is provided, it simply tests if a method named n is contained in the class c or one of its superclasses. Again, the parameter *direct* is added if the test is only to refer to the class c itself, and, for simplicity, a method may be provided as parameter instead of the method name and signature.

Boolean **equalInterface**(Class/Interface *c1*, Class/Interface *c2*): Returns true iff *c1* and *c2* declare precisely the same public methods. Public fields and static methods are not included in the comparison.

Boolean **exhibitSameBehaviour**(Method *m1*, Method *m2*): Returns true iff *m1* and *m2* will, if invoked in the same program state, exhibit the same external behaviour and lead to the same resulting program state. Note that this relationship normally exists only when *e1* and *e2* are in a delegation relationship[3].

[2]The differentiation we make between declaration and definition is maintained rigorously in C++ [93], but is not followed so strongly in Java [2]. For example, page 21 of [2] the authors write of an interface *defining* a method. In this work we need clear terminology to distinguish between the two situations.

[3]Assessing if two methods have the same behaviour is undecidable in general. For example, in [67] equivalence is based on the very constrained criterion that the parse trees of the methods must be identical.

Boolean **exhibitSameBehaviour**(ObjectCreationExprn *e1*, ObjectCreation-Exprn *e2*): Returns true iff *e1* and *e2* will, in the same program state, create objects that exhibit the same behaviour. Note that this condition is normally established when *e1* delegates all requests to a contained instance that is identical to *e2*, so the objects need not even be of the same class.

Boolean **hasSingleInstance**(Class *c*): Returns true iff the program only ever creates at most a single instance of the class *c*.

Boolean **implementsInterface**(Class/Interface *e*, Interface *i*): Returns true iff there is an implements link from the class/interface *e* to the interface *i*.

Boolean **initialises**(Method/Constructor *m*, Field/Variable *f*, Exprn *e*): Returns true iff the method/constructor *m* initialises the field/variable *f* to the expression *e*.

Boolean **isAbstract**(Class/Method *x*): Returns true iff the class/method *x* is declared to be abstract.

Boolean **isClass**(Class *c*): Returns true iff *c* is a class. If given a string as argument, it tests if a class of the given name exists in the program.

Boolean **isClonable**(Class *c*): Returns true iff the class *c* can be cloned. All classes in Java inherit a clone method from the Object class which performs a bitwise copy of the object on which it is invoked. This is adequate in some cases, but if objects of the class contains references to other objects, the programmer will probably have to implement a class-specific clone method. If objects of the class have circular references, or are part of a very complicated structure, it may not be feasible to implement a clone method. This can be tested automatically in simple cases, but in general the user must be queried to assess if it is safe to clone a particular class.

Boolean **isExclusiveComponent**(Class c, Field f): Returns true iff f is a field of the class c and the object referred to by f is an exclusive component of c. By this we mean a total form of ownership[4]:

- f is initialised in all constructors of c.
- The object referred to by f is not referred to by any other reference in the program.
- The object reference f may not refer to any other object during its lifetime, nor may it ever be set to null.

Boolean **isInterface**(Interface i): Returns true iff i is an interface in the program. If given a string as argument, it tests if an interface of the given name exists in the program.

Boolean **isPrivate**(Method/Field e): Returns true iff the method/field e is a private member of its class.

Boolean **isPublic**(Method/Field e): Returns true iff the method/field e is a public member of its class.

Boolean **isStatic**(Method/Field e): Returns true iff the method/field e is a static member of its class.

[4]In terms of the sophisticated categorisation of whole-part relationships described by Franco Civello in [20], we are describing a whole-part relationship that is visible, encapsulated, non-shared, part-whole inseparable, whole-part inseparable, immutable, owned and collaborative.

Boolean **isSubtype**(Class/Interface e_1, Class/Interface e_2): Returns true iff the type defined by the class/interface e_1 is a subtype of the type defined by the class/interface e_2. This is based on the normal syntactic notion of subtyping [22], but does not depend on their being an explicit implements/extends relationship between the entities. As a shorthand, isSubtype(e_1, e_2) will normally be written $e_1 \leq e_2$.

SetOfVariable **localVars**(Method/Constructor m): Returns the set of local variables that are defined within the given method/constructor, regardless of the block scope they are in.

SetOfMethod **methodsInvoked**(MethodInvocation i): Returns the set of methods that could be invoked by the method invocation i.

String **nameOf**(Class/Interface/Method/Constructor x): Returns the name of the given class/interface/method/constructor.

int **noOfArguments**(MethodInvocation/ObjectCreationExprn x): Returns the number of arguments to the given method invocation or object creation expression.

int **noOfParameters**(Method/Constructor m): Returns the number of parameters of the given method/constructor.

Parameter **parameter**(Method/Constructor m, int i): Returns the i^{th} parameter of the given method/constructor, or $\perp$ if the given parameter does not exist.

Boolean **returnsObject**(Method m, ObjectRef o): Returns true iff the method m returns the object referred to by the object reference o, and has no other effects.

Boolean **returnsSameObject**(Constructor *c*, Method *m*): Returns true iff the method *m* returns an object of the same class and in precisely the same state as would be created by *c*, given the same argument list. *m* must have no other side-effects; in particular it must neither access any variables other than its parameters, nor send a message to another object. Note that the method *m* need not actually create a new object. See also the stronger condition, createsSameObject.

Class/Interface **returnType**(Method *m*): Returns the class/interface that is the return type of the method *m*.

Signature **sigOf**(Method/Constructor *x*): Returns the signature of the given method or constructor.

Class **superclass**(Class *c*): Returns the direct superclass (based on the extends relationship) of the class *c*, or $\perp$ if none exists. It can be also applied to a constructor, method or field, in which case the superclass of the class of the given element is returned.

SetOfClass **superclasses**(Class *c*): Returns the set of (in)direct superclasses (based on the extends relationship) of the class *c*. Note that the class *c* itself is not a member of the set of classes returned.

Class/Interface **typeOf**(ObjectRef *o*): Returns the Class or Interface of the given object reference (field, parameter or local variable).

Boolean **uses**(Method *m*, Field *f*): Returns true iff the method *m* directly references the field *f*.

Boolean **uses**(Method m_1, Method m_2): Returns true iff the method m_2 may be directly invoked by the method m_1[5].

Boolean **uses**(ObjectRef o, Method m): Returns true iff the method m may be directly invoked through the object reference o.

Boolean **uses**(ObjectRef o, Field f): Returns true iff the field f is directly accessed through the object reference o.

B.1.1 Relationships between Analysis Functions

The analysis functions are not completely orthogonal and this is unavoidable. For example, it is important to know if one class defines a subtype of another class, as this affects what type of refactorings are possible involving these classes. It is also important to be able to determine if one class has an extends link to another class. If we determine that the class B extends the class A, then we know that B must also be a subtype of A. It is important to note these relationships and to use them in proofs as necessary. The relationships between the analysis function we use are as follows:

If a method is in a class, that class defines the method directly, and vice versa:

$\forall$ c:Class, m:Method, classOf(m)=c $\Leftrightarrow$

defines(c, nameOf(m), sigOf(m), direct)

If a class defines a method, it declares it as well:

$\forall$ c:Class, m:Method, defines(c, nameOf(m), sigOf(m)) $\Rightarrow$

declares(c, nameOf(m), sigOf(m))

[5]For this analysis function and the next, some false positives may be returned since a static analysis cannot determine exactly what methods a particular method invocation may bind to.

A class that defines a method of a given name and signature must also define a method of that name:

$\forall$ c:Class, n:String, s:Signature, defines(c, n, s) $\Rightarrow$
defines(c, n)

If a method and constructor return the same object, they must have the same signature:

$\forall$ m:Method, c:Constructor, returnsSameObject(c, m) $\Rightarrow$
sigOf(m)=sigOf(c)

Two classes/interfaces have the same interface iff each one is a subtype of the other:

$\forall$ e_1:Class/Interface, e_2:Class/Interface, equalInterface(e_1, e_2) $\Leftrightarrow$
$e_1 \leq e_2 \land e_2 \leq e_1$

If a class/interface implements another interface, it must be a subtype of that interface:

$\forall$ e:Class/Interface, i:Interface, implementsInterface(e, i) $\Rightarrow$ e $\leq$ i

If a class extends another class, it must be a subtype of that class:

$\forall$ c_1:Class, c_2:Class, superclass(c_1)=c_2 $\Rightarrow$ $c_1 \leq c_2$

If a class is abstract, it must declare a method that it does not define:

$\forall$ c:Class, isAbstract(c) $\Leftrightarrow$ $\exists$ m:Method such that
declares(c, m) $\land$ $\neg$ defines(c, m)

One class *creates* another iff there is an object creation expression contained in the first class (or any superclass) that creates an instance of the second class:

creates(c_1, c_2) $\Leftrightarrow$ $\exists$ o:ObjectCreationExprn, m:Method such that
classCreated(o)=c_2 $\land$ containingMethod(o)=m $\land$
classOf(m) $\in \{c_1\} \cup$ superclasses(c_1)

If a method creates and returns the same object as a constructor, it also just returns it:

$\forall$ c:Constructor, m:Method, createsSameObject(c, m) $\Rightarrow$

returnsSameObject(c, m)

B.2 Helper Functions

In describing a refactoring it may be necessary to extract richer content from the program code than is provided by the analysis functions. Helper functions are used to perform this type of task. As they are not at the primitive level of the analysis functions, we provide them with a pre- and postcondition. Helper functions (3.1.3) are proper functions without side-effects on the program, so the postcondition invariably involves the return value of the helper function itself.

Interface **abstractClass**(Class *c*, String *newName*): Construct and return an interface called *newName* that reflects all the public methods of the given class *c*.
precondition:
The class *c* must exist:

isClass(c)

postcondition:
The returned interface *inf* declares the same public methods as the class *c*:

isInterface$'$ = isInterface[inf/true]

equalInterface$'$ = equalInterface[(c,inf)/true]

The name of the returned interface is *newName*:

nameOf$'$ = nameOf[inf/$newName$]

Method **abstractMethod**(Method *m*): Construct and return an abstract method that has the same name and signature as the given method *m*.
precondition:

The method m must exist:

isMethod(m)

postcondition:

The returned method *meth* is abstract and has the same name and signature as the given method m:

isAbstract′ = isAbstract[*meth*/true]

nameOf′ = nameOf[*meth*/nameOf(m)]

sigOf′ = sigOf[*meth*/sigOf(m)]

Class **createEmptyClass**(String *name*): Construct and return an empty class called *name*.

precondition:

This may be used in any state:

true

postcondition:

An empty class called *name* is returned:

nameOf′ = nameOf[*returned*/*name*]

∀ e:Method/Field/Constructor • ¬ classOf(e)=*returned*

where *returned* is the class returned by this function.

Class **createWrapperClass**(Interface *iface*, String *wrapperName*, String *fieldName*): Creates a class called *wrapperName* that provides the same interface as *iface* and implements all its methods by delegating them to a private field of the type *iface*, called *fieldName*. The class is given a constructor that accepts an object of the type *iface* and initialises the field *fieldName* to this object. A method called "get"+*fieldName* is also added that returns the contents of this field (i.e., returns the wrapped object).

precondition:

The given interface must exist:

isInterface(*iface*)

The name of the class to be added is not in use:

$\neg$ isClass(*wrapperName*) $\wedge$ $\neg$ isInterface(*wrapperName*)

postcondition:

A class called *wrapperName* is returned:

nameOf′ = nameOf[*returned*/*wrapperName*]

The returned class has a field of type *iface* called *fieldName*:

$\exists$ f:Field, such that

classOf′=classOf[f/*returned*]

typeOf′=typeOf[f/*iface*]

nameOf′=nameOf[f/*fieldName*]

The constructors of *returned* initialise this field with the first parameter:

$\forall$ c:Constructor, classOf(c)=*returned* $\bullet$

initialises′=initialises[c, f, parameter(c,1)]

The class *wrapper* has a method called "get"+*fieldName*:

$\exists$ m:Method such that

classOf′=classOf[m/*wrapper*]

nameOf′=nameOf[m/"get"+*fieldName*]

This method returns the contents of the field *fieldName*:

returnsObject′=returnsObject[m/*fieldName*]

Any object of a concrete subclass of *iface* will exhibit the same behaviour as an instance of *returned* that has been given this object as its construction argument:

$\forall$ c:Class, implementsInterface(c,*iface*) $\bullet$

$\forall$ e:ObjectCreationExprn, classCreated(e)=c $\bullet$

exhibitSameBehaviour′ =

exhibitSameBehaviour[(e, new *wrapperName*(e))/true]

Method **makeAbstract**(Constructor c, String *newName*): Returns a method called *newName* that, given the same arguments, will create the same object as the constructor c. The method signature is obtained by copying that of the constructor[6], and the method is given a body that is simply an object creation expression that invokes the given constructor, using the arguments to the method as its own arguments.

precondition:

This may be used in any state:

$$\text{true}$$

postcondition:

A method called *newName* is returned that, given the same argument list, creates the same object as the constructor c:

$$\text{createsSameObject}' = \text{createsSameObject}[(c, \mathit{returned})/\text{true}]$$

$$\text{nameOf}' = \text{nameOf}[\mathit{returned}/\mathit{newName}]$$

where *returned* is the returned method.

B.3 Primitive Refactorings

The primitive refactorings (section 3.1.4) that are used in this work are detailed in this section. As with the helper functions, a pre- and postcondition is given in each case, and these may range over the arguments to the refactoring and the program itself that is being transformed. An argument that the refactoring does not change the behaviour of the program is presented in each case.

[6]This does not actually need to be stated explicitly in the postcondition, as from section B.1.1 we know it can be derived from the first conjunct of the postcondition.

void **absorbParameter**[7](Method/Constructor m, int *paramNumber*): Remove the specified parameter from the method/constructor m (assume method from here on), converting the parameter into a local variable in the method, and initialising it with the expression given for the argument.

precondition:

The parameter exists in the given method:

$\text{noOfParameters}(m) \geq \textit{paramNumber}$

All invocations of m take the same expression (which must be independent of context) as an argument for the specified parameter:

$\exists$ exprn:Exprn, contextFree(exprn) such that

$\forall$ i:MethodInvocation, $m \in$ methodsInvoked(i) $\bullet$

argument(i, *paramNumber*) = exprn

postcondition:

The parameter list for m has been reduced by 1:

$\text{noOfParameters}' = \text{noOfParameters}[m/\text{noOfParameters}(m)\text{-}1]$

m now defines a new local variable of the same name and type as the parameter that has been removed:

$\text{localVars}' = \text{localVars}[m/\text{localVars}(m) \cup \text{v}\ ]$ where

$\text{nameOf}(\text{v}) = \text{nameOf}(\text{parameter}(m, \textit{paramNumber})) \wedge$

$\text{typeOf}(\text{v}) = \text{typeOf}(\text{parameter}(m, \textit{paramNumber}))$

This new local variable is initialised to the expression that was previously passed in as an argument:

$\text{initialises}'=\text{initialises}[(m, \text{v}, \text{exprn})/\text{true}]$

Behaviour preservation:

The expression that was originally passed as an argument is context free,

[7]This is similar to, but more flexible than, the removeParameter refactoring described by Fowler in [38, p.277]. Fowler assumes the parameter is not in use; we allow it to be in use once it is always passed the same argument.

so it evaluates to the same result for each method invocation, and will also evaluate in the same way after being moved into the method itself. The parameter it was originally bound to has been removed, and instead this expression is evaluated and stored in a local variable of the same name and type as the removed parameter. The method thus executes in the same context, except that previous references to the removed parameter now bind to the new variable. Since the new variable has been given the same initial value, program behaviour will remain the same.

void **abstractMethodFromClass**(Method m): Makes public any method or field that is (i) a member of the same class that m belongs to, or a superclass, and (ii) is used by m[8].

precondition:

The class referred to exists and m is a member of this class:

isClass(classOf(m)) $\wedge$ $m \in$ classOf(m)

postcondition:

All methods/fields defined directly or indirectly in classOf(m) that m uses have been made public:

$\forall$ x:Field/Method, defines(classOf(m), x), uses(m, x) $\bullet$

isPublic$'$=isPublic[x/true]

Behaviour preservation:

Making a private or protected field/method public cannot affect compilation or behaviour. It may appear that making a private member of a class public or protected might cause a reference in a subclass to bind to the new public member rather than one defined in a superclass. However in Java an overriding method cannot reduce the access level defined in its superclass, so if a method is private in a class, making it public cannot cause it to override a method in a superclass. Also, a reference to a field that is defined to be protected in a superclass will not compile if there is a private definition of a field of the same name in an intervening superclass, so again making a field public cannot interfere in the binding of references in subclasses.

[8]This refactoring is usually used as a preparation for moving the method m to a component of its current class. Prior to pulling out m, everything it refers to in its current class must be made public. If m is in fact a cohesive member of its class, this refactoring is likely to severely damage the encapsulation of the class and its superclasses.

void **addClass**(Class *c*, Class *super*, SetOfClass *subclasses*): Add the class *c* to the program. If a superclass is given, an extends link is added from the class *c* to this superclass. If subclasses are given, an extends link is added from each one to the class *c*.

precondition:

The name of the class to be added is not in use:

$\neg$ isClass(nameOf(*c*)) $\wedge$ $\neg$ isInterface(nameOf(*c*))

Any given subclasses must exist:

$\forall$ s $\in$ *subclasses* $\bullet$ isClass(s)

If the superclass exists, it must be a superclass of all the subclasses:

if isClass(*super*) then $\forall$ s $\in$ *subclasses* $\bullet$ superclass(s) = *super*

If *c* is a concrete class, then any abstract methods declared in *super* or its superclasses must be defined in *c*:

if $\neg$ isAbstract(*c*) then

$\forall$ m:Method, declares(*super*, m) $\wedge$ $\neg$ defines(*super*, m) $\bullet$

defines(*c*, nameOf(m), sigOf(m))

The class *c* must not contain any method that overrides one declared (in)directly in the superclass:

$\forall$ n:String, s:Signature $\bullet$ if declares(*super*, n, s) then

$\neg$defines(*c*, n, s, direct)

The class *c* must not contain any field that redefines one declared in any of its (in)direct superclasses:

$\forall$ f:Field, f$\in$*c*, $\neg$isPrivate(f) $\bullet$

$\forall$ g:Field, g$\in$sup, where sup$\in$superclasses(*c*), $\neg$isPrivate(g) $\bullet$

nameOf(f) $\neq$ nameOf(g)

postcondition:

c is a class in the program:

$\text{isClass}' = \text{isClass}[c/\text{true}]$

An extends link exists from the class c to the class *super*:

$\text{superclass}' = \text{superclass}[c/\mathit{super}]$

All the given subclasses are now subclasses of c:

$\forall\ \text{s} \in \mathit{subclasses},\ \text{superclass}' = \text{superclass}[\text{s}/c]$

Behaviour preservation:

The class c did not exist, so no references can exist to this class. Consequently the only threat to behaviour preservation is that a subclass may refer to a method or field in a superclass, and this reference is now bound to a method or field of c. The final two conjuncts of the precondition prevent this by disallowing the class c from redefining any field or method that is already defined in any of its superclasses[9].

[9]This refactoring is an example of where our requirement for behaviour preservation forces us to be very strict in defining preconditions. In the work of both Roberts [84, p.103] and Sunyé *et al* [94, p.57], the last two conjuncts of the precondition for this refactoring are omitted. Our approach is nevertheless conservative, since the class c can redefine fields and methods in its superclasses once these are not used in any of the subclasses.

void **addGetMethod**(Class *concrete*, String *fieldName*): Add a "getter" method to the *concrete* class that returns the contents of the field called *fieldName*.

precondition:

The class *concrete* exists and has a field called *fieldName*:

isClass(*concrete*) $\wedge$ classOf(*fieldName*)=*concrete*

The class *concrete* does not declare a method called "get"+*fieldName*:

$\forall$ m:Method, declares(*concrete*, m) $\bullet$ nameOf(m) $\neq$ "get"+*fieldName*

postcondition:

The class *concrete* has a method called "get"+*fieldName*:

$\exists$ m:Method such that

classOf′=classOf[m/*concrete*]

nameOf′=nameOf[m/"get"+*fieldName*]

This method returns the contents of the field *fieldName*:

returnsObject′=returnsObject[m/*fieldName*]

Behaviour preservation:

Since a method with the same name as the method being added does not already exist in the class, there can be no name clashes and no existing invocations of this method.

void **addImplementsLink**(Class *concrete*, Interface *inf*): Add an implements link from the class *concrete* to the interface *inf*. The class *concrete* must not be abstract, i.e., it must implement all the abstract methods that are declared in *inf*.

precondition:

The class *concrete* and the interface *inf* must exist:

$\text{isClass}(concrete) \wedge \text{isInterface}(inf)$

The class *concrete* must be a subtype of the interface *inf*:

$concrete \leq inf$

The class *concrete* must implement all the methods that are declared in *inf*:

$\forall$ m:Method, declares(*inf*, m) $\bullet$ defines(*concrete*, m, direct)

postcondition:

An implements link had been added from the class *concrete*) to the interface *inf*:

$\text{implementsInterface}' = \text{implementsInterface}[(concrete, inf)/\text{true}]$

Behaviour preservation:

Adding a implements link from a class to an interface may affect the legality of the program, but cannot cause it to change its runtime behaviour. From the precondition, we see that the class fully implements the interface[10], so this refactoring must result in a legal program and consequently it is behaviour preserving.

[10]In the case of an abstract class, this part of the precondition could be safely weakened.

void **addInterface**(Interface i): Adds the interface i to the program. A class or interface with this name must not already exist.

precondition:

No class or interface with the name nameOf(i) exists:

$$\neg\text{isClass}(\text{nameOf}(i)) \land \neg\text{isInterface}(\text{nameOf}(i))$$

postcondition:

i is a new interface in the program:

$$\text{isInterface}' = \text{isInterface}[i/\text{true}]$$

Behaviour preservation:

Adding an unreferenced interface to the program cannot affect its behaviour. If a reference to the interface did exist before the refactoring, then the original program would not be legal.

void **addMethod**(Class c, Method m): Adds the method m to the class c. A method with this signature must not already exist in this class or its superclasses. This refactoring extends the external interface of the class.

precondition:

The class c exists and does not define any method with the same name and signature as m:

$\text{isClass}(c) \wedge \neg\text{defines}(c, \textit{nameOf(m)}, \textit{sigOf(m)})$

postcondition:

The method m has been added to the class c:

$\text{classOf}' = \text{classOf}[m/c]$

Any class or interface that previously had the same interface as c does not have the same interface anymore:

$\forall$ a:Class, a$\neq$c, if equalInterface(a,c) then

$\text{equalInterface}' = \text{equalInterface}[(a,c)/\text{false}]$.

Behaviour preservation:

Since a method with the same name and signature as the method being added does not already exist in the class, there can be no name clashes and no existing invocations of this method.

void **addSingletonMethod**(Class *singletonClass*, Class *concreteSingleton*, String *methodName*, String *fieldName*): Adds a static field named *fieldName* of type *singletonClass* to the class *singletonClass*. Also adds a static method named *methodName* that gives access to this field and instantiates it lazily as a *concreteSingleton* object when necessary (See [41, 43], both pp 127-133 for more detail). If the last two parameters are omitted, we assume them to be named "getInstance" and "instance" respectively.

precondition:

The first two parameters must be classes and the class *singletonClass* must be a superclass of *concreteSingleton*:

singletonClass ∈ superclasses(*concreteSingleton*)

The class *singletonClass* can have no field called *fieldName*:

∀ f:Field, f∈*singletonClass* • nameOf(f) ≠ *fieldName*

A non-private field called *fieldName* cannot be defined in any superclass of *singletonClass*:

if f:Field ∈ cls, cls ∈ superclasses(*singletonClass*),
nameOf(f)=*fieldName* **then** isPrivate(f)

A method called *methodName* cannot be defined in the class *singletonClass*:

¬defines(*singletonClass*, *methodName*)

The class *concreteSingleton* must have a no-arg constructor:

∃ c:Constructor ∈ *concreteSingleton* such that noOfParameters(c)=0

postcondition:

A new method m has been added to the class *singletonClass*, with certain properties:

classOf′ = classOf[m/*singletonClass*]

The name of m is *methodName*:

nameOf′ = nameOf[m/*methodName*]

The method m returns an object of the class *concreteSingleton*, in the same state as would be returned by the no-arg constructor:

returnsSameObject′ = returnsSameObject[(c,m)/true]

where c:Constructor $\in$ *concreteSingleton* $\wedge$ noOfParameters(c)=0

Behaviour preservation:

A new method and field are added to the class *singletonClass*. Since neither already exist, nor are they referenced in the existing program, program behaviour cannot be affected.

void **createExclusiveComponent**(Class *context*, Class *component*, String *fieldName*): Add a new component to the class *context*, called *fieldName*, of type *component*. All constructors in *context* are updated to instantiate this field as well.

precondition:

The classes must exist:

$\text{isClass}(\mathit{context}) \land \text{isClass}(\mathit{component})$

Neither the class *context* nor any of its superclasses may have a non-private field called *fieldName*:

$\forall$ f:Field, f$\in$sup, where sup $\in$ superclasses(*context*) $\cup$ *context*,

$\neg$isPrivate(f) $\bullet$ nameOf(f) $\neq$ *fieldName*

postcondition:

The class *context* has a field called *fieldName* of type *component*:

$\exists$ f:Field, f $\in$ *context* such that

typeOf′=typeOf[f/*component*]

nameOf′=nameOf[f/*fieldName*]

All constructors of *context* initialise this field:

$\forall$ c:Constructor, c $\in$ *context* $\bullet$

initialises′=initialises[(c, *fieldName*, "new *component*()")/true]

fieldName refers to an exclusive component of *context*:

isExclusiveComponent′=isExclusiveComponent[(*context*, *fieldName*)/true]

Behaviour preservation:

The name *fieldName* does not clash with any field defined in *context*, or any of its superclasses, so it may be added to *context* safely. *fieldName* is initialised in the constructor of *context* using the no-arg constructor of the *component* class, so this has no observable effect on external program behaviour[11].

[11]We assume that the no-arg constructor of the *component* class only initialises its own internal data fields.

void **makeConstructorProtected**(Class c): Makes all constructors of the class c protected. If the class has no explicit constructors, a no-arg one is added and made protected.

precondition:

The class c exists:

    isClass(c)

Creations of objects of the class c occur only in c and its subclasses:

    $\forall$ e:ObjectCreationExprn, classCreated(e)=c $\bullet$

    e $\in$ c $\vee$ c $\in$ superclasses(containingClass(e))

postcondition:

The method m has been added to the class c:

    classOf$'$ = classOf[m/c]

Any class or interface that previously had the same interface as c does not have the same interface anymore:

    $\forall$ a:Class, a$\neq$c, if equalInterface(a,c) then

    equalInterface$'$ = equalInterface[(a,c)/false].

Behaviour preservation:

The behaviour of the constructors of the class c is not changed, and objects of the class c are only created within c itself or its subclasses. Therefore, making these constructors protected will have no effect on program behaviour.

void **moveMethod**(Class *context*, Field *component*, Method *meth*): Moves the method *meth* from the class *context* to the class of the field *component*. The existing method is replaced by one that delegates the same request to the *component* field. The moved method is given an extra parameter that refers to the *context* object it has been moved from, and any references it makes to this (implicitly or explicitly) are sent back to this *context* object.

precondition:

The classes referred to exist:

$\text{isClass}(\mathit{context}) \wedge \text{isClass}(\text{typeOf}(\mathit{component}))$

meth is a method of the class *context*:

$\mathit{meth} \in \mathit{context}$

Every method/field in *context* that is used by *meth* must be public:

$\forall$ x:Method/Field, x $\in$ *context*, uses(*meth*, x) $\bullet$ isPublic(x)

The field *component* refers to an exclusive component of *context*:

isExclusiveComponent(*component*, *context*)

A similar method to *meth* cannot be defined in the *component* class:

$\forall$ m:Method $\in$ typeOf(*component*), nameOf(*meth*)=nameOf(m) $\bullet$

sigOf(*meth*)$\neq$sigOf(m)

postcondition:

The method *meth* is now a member of the class of the component field:

classOf′=classOf[*meth*/classOf(*component*)]

The class *context* delegates invocations of the moved method to a method that exhibits the same behaviour in the class of the *component* field:

$\exists$ m:Method such that

classOf′ = classOf[m/*context*]

nameOf′ = nameOf[m/nameOf(*meth*)]

sigOf′=sigOf[m/sigOf(*meth*)]

uses′=uses[(m,*meth*)/true]

exhibitSameBehaviour′=exhibitSameBehaviour[m/*meth*]

Behaviour preservation:

The moved method is replaced by one of the same name and signature, so compilation will not be affected. The replacement method delegates to the moved method, and passes the context object as an extra argument. Any references to the context object itself in the moved method are invoked on the context object (from the precondition they must be public) and so will bind in the same way as before.

void **parameteriseField**(Class *client*, Class/Interface *product*): Moves the initialisation of the field of type *product* in the class *client* outside the constructor of the class, so the initial value for this field is now passed as an argument to the *client* class constructor.

precondition:

The given interface/classes exist:

$\text{isClass}(client) \wedge (\text{isClass}(product) \vee \text{isInterface}(product))$

The *client* class has a single field of type *product*:

$\exists!\ \text{f:Field, f} \in client \text{ such that typeOf(f)}=product$

This field is initialised to a context free expression, *exprn*, in all constructors:

$\exists\ exprn\text{:Exprn, contextFree}(exprn) \text{ such that}$

$\quad \forall \text{ c:Constructor, c} \in client\ \bullet$

$\quad\quad \text{initialises(c, f, } exprn)$

postcondition:

Each *client* constructor has a new parameter of type *product*:

$\forall \text{ c:Constructor, c} \in client\ \bullet$

$\quad \text{noOfParameters}'=\text{noOfParameters}[\text{c/noOfParameters(c)+1}]$

$\quad \text{typeOf}'=\text{typeOf}[\text{parameter(c,noOfParameters(c)+1)}/product]$

The field f is initialised with this parameter rather than *exprn*:

$\text{initialises}'=\text{initialises}[(\text{c, f, } exprn)/\text{false}]$

$\text{initialises}'=\text{initialises}[(\text{c, f, parameter(c,noOfParameters(c)+1)})/\text{true}]$

All creations of *client* objects now take the expression *exprn* as an extra argument:

$\forall \text{ e:ObjectCreationExprn, classCreated(e)}=client\ \bullet$

$\quad \text{noOfArguments}'=\text{noOfArguments}[\text{e/noOfArguments(e)+1}]$

$\quad \text{argument}'=\text{argument}[\text{(e,noOfArguments(e)+1)}/exprn]$

Behaviour preservation:

Initially the *product* field in the *client* class was set to the expression *exprn* in the constructor of the *client* class. After applying this transformation, the expression *exprn* is evaluated outside the *client* class and passed in as a parameter to the constructor. Within the constructor it is used to initialise the field as before. Since *exprn* is context-free, it will evaluate the same way in both cases, so the *product* field in the *client* class gets initialised to the same value and program behaviour is therefore maintained.

void **pullUpMethod**(Method m): Move the method m from its current class to its superclass[12]. All fields directly referenced by m are moved to the superclass as well. An abstract method declaration is added to the superclass for any method referenced by m that is not (in)directly declared in the superclass.

precondition:

The method m must exist:

$\text{isMethod}(m)$

The class must have a superclass to which to move the method:

$\text{superclass}(m) \neq \perp$

m must not be defined in the superclass:

$\neg\ \text{defines}(\text{superclass}(m), \text{nameOf}(m), \text{sigOf}(m))$

Any fields m uses must not be public and must not clash with fields in the superclass:

$\forall$ f:Field, f $\in$ classOf(m), if uses(m,f) then

$(\neg\ \text{isPublic(f)} \wedge \forall$ g:Field, g $\in$ superclass(m), nameOf(f) $\neq$ nameOf(g))

postcondition:

m is moved from its existing class to its superclass:

$\text{classOf}' = \text{classOf}[m/\text{superclass}(m)]$

Any methods m uses that are not declared in its superclass are declared there now:

$\forall$ n:Method, n $\in$ classOf(m), m$\neq$n,

if uses(m,n) $\wedge\ \neg$ declares(superclass(m),n) then

$\text{declares}' = \text{declares}[(\text{superclass}(m), \text{n}, \mathit{direct})/\text{true}]$

Any fields m uses are moved to the superclass:

$\forall$ f:Field, f $\in$ classOf(m) if uses(m,f) then

[12]Although it appears natural to decompose this refactoring into a chain of refactorings, this is not useful for our present purposes. Opdyke [77] provides a partial solution, but does not deal with the details of moving the referenced fields up to the superclass and adding new abstract method declarations to the superclass for each referenced method.

classOf′ = classOf[f/superclass(m)]

Behaviour preservation:

The method m and the fields it uses have been moved, but not changed, so the behaviour of the program could change in three possible ways:

1. *An existing invocation of this method fails or invokes another method*: The existing method m could only be invoked on an object of the class $classOf(m)$ or a subclass of this class. In either case the search for the method attempts to find it in the class $classOf(m)$ and then moves to the class $superclass(m)$. The method m has been moved to this class so it is found here.

2. *An existing access to a moved field fails or finds another field*: The argument is similar to the previous case. The method m will find the field in its own class as normal. Other references to a moved field can only come from the class $classOf(m)$ or its subclasses, and these will bind correctly to the field in the superclass. Note that this argument would fail if a field accessed by m was public.

3. *An method invocation or field access in this method is bound to a different method/field*: The existing method m was not defined in any superclass of $classOf(m)$, so it may only be invoked on an object of $classOf(m)$ or one of its subclasses. The search for a method invoked in m will therefore commence at the same class as before, and will find the same method (if the search somehow began at $superclass(m)$, a failure could occur). To highlight this, consider the following code sketch:

```
class A{
        public void foobar(){...}
}
class B extends A{
        public void foo(){
                ...
                foobar();
                ...
        }
        public void foobar(){...}
}
```

It may appear that moving foo from B to A will cause a problem in that the invocation of foobar will now bind to the implementation of foobar in A rather than that in B. However, since we have disallowed situations where foo is defined in A or a superclass, invocations of foo on an object of class A cannot exist. Invocations of foo on objects of class B will now result in the foo in A being executed, but by dynamic binding the subsequent invocation of foobar will bind correctly to the implementation in B.

void **replaceClassWithInterface**(ObjectRef o, Interface inf): Change the type of the object reference o to the interface inf.

precondition:

The interface inf exists:

isInterface(inf)

The class of the object reference o must have an implements link to the interface inf:

implementsInterface(typeOf(o), inf)

Any static methods or fields in the class of the object reference o are not accessed through the object reference o:

$\forall$ m:Method, classOf(m)=typeOf(o), if isStatic(m) then $\neg$ uses(o,m) $\wedge$

$\forall$ f:field, classOf(f)=typeOf(o), $\neg$ uses(o,f)

postcondition:

The type of the object reference o is inf:

typeOf′ = typeOf[o/inf]

Behaviour preservation:

Changing the type of an object reference from a class to an interface may affect the legality of the program, but cannot cause it to change its runtime behaviour. From the precondition we see that the class of the object reference implements the interface, and that no static methods or fields are accessed through this reference, so this refactoring must result in a legal program and consequently it is behaviour preserving.

void **replaceObjCreationWithMethInvocation**(ObjectCreationExprn e, Method m): Replace the given object creation expression e with an invocation of the method m using the same argument list.

precondition:

The object creation expression e and the method m must both, given the same argument list, create and return the same object, OR they must both simply return the same object, and this must be the only instance of the class:

$$\text{createsSameObject}(\text{constructorInvoked}(e), m) \lor$$
$$(\text{returnsSameObject}(\text{constructorInvoked}(e), m) \land$$
$$\text{hasSingleInstance}(\text{classCreated}(e)))$$

The object creation expression e must not be in the method m:

$$\text{containingMethod}(e) \neq m$$

postcondition:

The object creation expression e has been removed:

$$\text{containingMethod}' = \text{containingMethod}[e/\bot]$$

Behaviour preservation:

The new method invocation returns the identical object to the same point in the program as was returned by the original object creation expression (the method either creates a new object or returns the only instance of the class). The only risk to behaviour preservation therefore is that of an infinite recursion occurring. The expression e is not contained in m so a direct recursion cannot take place. The *createsSameObject* precondition demands that m has no side-effects; in particular it cannot send any messages itself, so an indirect recursion is also impossible.

void **useWrapperClass**(Class *client*, Class *wrapper*, Class *receiver*, String *getterMethod*): Updates the *client* class so that any construction of the *receiver* class is replaced by a construction of the *wrapper* class, taking the corresponding *receiver* object as an argument. All variables of type *receiver* in the client classes are also renamed to *wrapper*. Any methods in *client* whose return type is *receiver* are updated to return the wrapped receiver object by delegating to the *getterMethod* in *wrapper*.

precondition:

The specified classes exist:

$\text{isClass}(\mathit{client}) \wedge \text{isClass}(\mathit{wrapper}) \wedge \text{isClass}(\mathit{receiver})$

The classes *wrapper* and *receiver* support the same interface:

$\text{equalInterface}(\mathit{wrapper}, \mathit{receiver})$

Any object of the class *receiver* will exhibit the same behaviour as an instance of *wrapper* that has been given the corresponding *receiver* object as its construction argument:

$\forall$ e:ObjectCreationExprn, classCreated(e)=*receiver* $\bullet$

exhibitSameBehaviour(e, new *wrapper*(e))

The method *getterMethod* in *wrapper* returns the wrapped receiver object:

returnsObject((new *wrapper*(e)).*getterMethod*(), e)

postcondition:

All object references to *receiver* in *client* have been changed to *wrapper*:

$\forall$ o:ObjectRef $\in$ *client*, typeOf(o)=*receiver* $\bullet$

typeOf$'$=typeOf(o/*wrapper*)

Methods in *client* that return a *receiver* object are updated to return the wrapped receiver object by delegating to the *getterMethod*:

$\forall$ m:Method, m$\in$*client*, returnType(m)=*receiver* $\bullet$

uses$'$=uses[m/*getterMethod*]

Behaviour preservation:

The *wrapper* class has the special property that when it is instantiated with an instance of *receiver*, it stores this receiver object and delegates all the requests it receives to this object. Thus the updating of the object creation expressions does not affect behaviour[13]. The object references that store these new objects are also updated to be of type *wrapper*, so they match the type of the updated object creation expressions. Finally, since the *receiver* and *wrapper* classes support the same interface, no type mismatch errors can occur.

[13]The new objects are of a different type however, so any existing downcasts will fail.

Appendix C

Listing of Minitransformations

Six minitransformations were identified in the development of the design pattern transformations. Each one has been analysed in detail in the body of this thesis. In this appendix we describe each minitransformation briefly, and provide a reference to the more detailed description in the main text.

1. The ABSTRACTION minitransformation is used to add an interface to a class. This enables another class to take a more abstract view of this class by accessing it via this interface. See section 4.3.1.

2. The ENCAPSULATECONSTRUCTION minitransformation is used when one class creates instances of another, and it is required to weaken the binding between the two classes by packaging the object creation statements into dedicated methods. See section 4.3.2.

3. The ABSTRACTACCESS minitransformation is used when one class uses, or has knowledge of, another class, and we want the relationship between the classes to operate in a more abstract fashion via an interface. See section 4.3.3.

4. The PARTIALABSTRACTION minitransformation is used to construct an abstract class from an existing class and to create an extends relationship between the two classes. See section 4.3.4.

5. The WRAPPER minitransformation is used to "wrap" an existing receiver class with another class, in such a way that all requests to an object of the wrapper class are passed to the receiver object it wraps, and similarly any results of such requests are passed back by the wrapper object. See section 5.4.2.

6. The DELEGATION minitransformation is used to move part of an existing class to a component class, and to set up a delegation relationship from the existing class to its component. See section 5.5.2.

Appendix D

Architecture of the Software Prototype

We have constructed a prototype software tool, DPT (Design Pattern Tool), that implements seven of the design pattern transformations that have been discussed in this thesis. In section D.1 we describe the architecture of this prototype, while in section D.2 an example of the application of the prototype to a Java program is presented.

D.1 Tool Architecture

DPT has a 4-tier architecture (see figure D.1) that matches the layers defined in the structure of the behaviour preservation arguments:

1. Design Pattern transformations.

2. Minitransformations.

3. Analysis functions, helper functions and primitive refactorings.

4. AST operations.

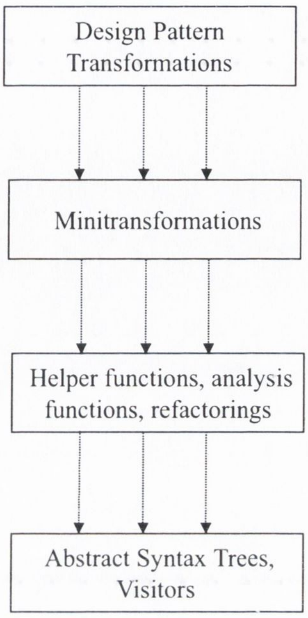

Design Pattern Transformations
Minitransformations
Helper functions, analysis functions, refactorings
Abstract Syntax Trees, Visitors

Figure D.1: Architecture of the Design Pattern Tool

The top layer implements the design pattern transformations we have discussed[1]. The next layer comprises the implementations of the six minitransformations that emerged during the development of the design pattern transformations. The third layer is the implementation of the supporting analysis functions, helper functions and primitive refactorings described in appendix B.

The bottom layer implements the actual changes to the program code by performing surgery directly on the parse trees generated from the Java source files. Visitors [41] are frequently used at this level to perform operations that involve an entire parse tree. The parsing of the source files and the construction of the parse trees were implemented using the parser generator JavaCC [65].

[1]Seven design pattern transformations have been prototyped, namely, Abstract Factory, Factory Method, Singleton, Builder, Prototype, Bridge and Strategy.

DPT does not extract an abstract model from the Java source code. This would have made the high-level transformations such as addClass much easier, but would have made the subsequent code regeneration much more difficult. The program being transformed is stored internally as a set of parse trees, and it is the operations provided in the top three layers of the architecture that provide an abstract view of this program. A programmer building a design pattern transformation need only be concerned with the minitransformation layer, and some refactorings and helper functions, in order to complete their task.

D.2 Sample Operation of DPT

We provide an example of the application of the Factory Method transformation to a generic program:

```
class Creator {
        public void doIt() {
                Product p = new Product("some text");
                Product q = new Product(1234);
                p.foo();
                q.foo();
        }
}
class Product {
        public Product(int x){...}
        public Product(String s){...}
        public void foo() {...}
}
```

```
}
```

The Factory Method transformation (section 4.4) is now applied to the above program as follows:

```
applyFactoryMethod("Creator", "Product", "absProduct",
                        "absCreator", "createProduct")
```

DPT applies the transformation and outputs the following code:

```
abstract class absCreator {
    public void doIt (){
        absProduct p = createProduct("some text");
        absProduct q = createProduct(1234);
        p.foo();
        q.foo();
    }
    public abstract absProduct createProduct (int x);
    public abstract absProduct createProduct (String s);
}
class Creator extends absCreator {
    public absProduct createProduct (int x) {
        return new Product(x);
    }
    public absProduct createProduct (String s) {
        return new Product(s);
    }
}
interface absProduct {
    public void foo ();
}
```