Automated Software Evolution Towards Design Patterns

Mel Ó Cinnéide
Department of Computer Science
University College Dublin
Ireland.
mel.ocinneide@ucd.ie

Paddy Nixon
Department of Computer Science
University of Strathclyde
Scotand.
paddy@cs.strath.ac.uk

ABSTRACT

During the evolution of a software system, it may be necessary to refactor the software in order to make it more flexible and amenable to new requirements that are being introduced. A typical aim of such a refactoring is to apply a suitable design pattern to the program in order to enhance its flexibility. Performing such a transformation by hand is an error-prone process, so automated support would be useful.

We have developed a methodology for the creation of automated transformations that can apply a design pattern to an existing program. In this paper we present a brief description of this methodology, and report on the results of using this methodology to develop program transformations that can apply the Gamma *et al* design patterns. It is found that in almost 75% of cases a satisfactory transformation is developed, and that much of the commonality between design patterns can be captured in reusable minitransformations.

Keywords

Design patterns, refactoring, automated program transformations, design quality.

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 acceptance of the notion that the process of building a software system should be an evolutionary one [1, 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. At each stage in this process, 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 [2, 6]:

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.

Our work has focused on developing a novel approach to providing sophisticated automated support for the restructuring step.

We now 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* [7]. 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. This enables the Creator class to be extended to instantiate a subclass of the Product class without significant reworking of the existing code.

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

A key aspect of our 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 paper we will present the results of developing a suite of automated design pattern transformations.

2. METHODOLOGY

The complete methodology is depicted in summary form as a UML activity chart in figure 1. An earlier version of this methodology is described in [9], while the full details of our current methodology are described in [8, chapter 4]. For space reasons, we confine ourselves in this section to providing a coarse outline of the methodology.

Initially a design pattern is chosen that will serve as a target for the design pattern transformation under development. We then consider what 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*. 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 resolved the problem in an inadequate way.

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 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. In terms of Foote and Opdyke's lifecycle model [5], precursors are structures that are likely to be built during the *prototyping* phase. They are simple structures that are adequate for the purposes of building a working system rapidly, but inadequate in terms of supporting future evolution and reuse.

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. Examining the design pattern catalogues [4, 7], 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.

For every minipattern discovered a corresponding *minitransformation* that can apply this minipattern must also be developed. A minitransformation comprises a precondition, an algorithmic description of the transformation, and a postcondition. The program that is being transformed must satisfy this precondition in order for the minitransformation to maintain the behaviour of the program. The transformation itself is expressed as a composition of *primitive refactorings*, which have already been demonstrated to be behaviour preserving. The pre- and postconditions of the entire minitransformation are computed by applying a semi-formal technique, described in full detail in [8, chapter 3].

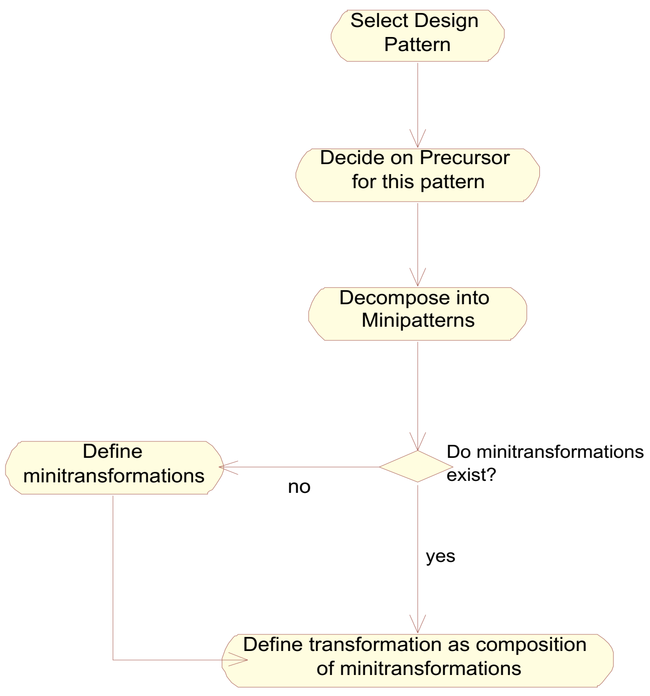
Select Design Pattern
Decide on Precursor for this pattern
Decompose into Minipatterns
Do minitransformations exist?
no
Define minitransformations
yes
Define transformation as composition of minitransformations

Figure 1: The Design Pattern Methodology

Minitransformations are our primary 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. 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 final design pattern transformation can now be defined as a composition of minitransformations and possibly some primitive refactorings. The pre- and postconditions for this design pattern transformation are computed in the same way as they are computed for a minitransformation.

3. RESULTS

We applied this methodology to a set of seven patterns from the Gamma *et al* catalogue [7], and prototyped the resulting transformations. Using the experience gained from this process, we analysed the remaining Gamma *et al* patterns with a view to finding a suitable precursor, assessing if a workable transformation can be built, and determining the minitransformations that are likely to be used. In each case we assessed the result we achieved and placed it in one of the following categories:

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 1: Assessment of Design Pattern Transformations

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

Table 2: Summary of Assessments

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 that means a usable transformation can be developed, but it is not complete. Typically the designer is left with some work to do by hand in order to complete the transformation.

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

The results are presented in complete form in table 1, and in summary form in table 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.

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 Strategy, we used a precursor where 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.

The following minipatterns were identified during the process of building these transformations:

1. Abstraction adds an interface to a class that reflects how the class is used in some context. This enables another class to take a more abstract view of this class by accessing it via this interface.

2. EncapsulateConstruction is applied 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.

3. AbstractAccess is applied 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.

4. PartialAbstraction constructs an abstract class from an existing class and creates an inheritance relationship between the two classes.

5. Wrapper is applied 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.

6. Delegation moves part of an existing class to a component class, and sets up a delegation relationship from the existing class to its component.

Reuse of minipatterns is an 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 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 the development of these design pattern transformations. The actual reuse achieved is even

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 3: Reuse of Minitransformations
The abbreviations in the table are as follows. **Abs**:Abstraction, **AbsAcc**:AbstractAccess, **Encap**:EncapsulateConstruction, **Partial**:PartialAbstraction, **Wrap**:Wrapper, **Deleg**:Delegation.

stronger, as this table only depicts minitransformation reuse and ignores the reuse of primitive refactorings.

4. CONCLUSIONS

We have developed a methodology for the creation of behaviour-preserving design pattern transformations, and applied this methodology to the Gamma *et al* design patterns. 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.

Further empirical studies are necessary to assess if our approach can contribute to the practice of software evolution. Our prototype transformation tool 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 [10], and arguing abstractly that it is nevertheless useful is futile. Our 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 notion of automated design pattern transformations.

5. REFERENCES

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

[2] K. Bennett and V. Rajlich. Software maintenance and evolution: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, New York, 2000. ACM Press.

[3] G. Booch. *Object-oriented analysis and design with applications.* Benjamin/Cummings, Redwood City, California, second edition, 1994.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, Chicester, first edition, 1996.

[5] B. Foote and W. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In J. Coplien and D. Schmidt, editors, *Pattern Languages of Programming*, Monticello, Illinois, 1995.

[6] M. Fowler. *Refactoring: improving the design of existing code.* Object Technology Series. Addison-Wesley Longman, Reading, Massachusetts, first edition, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, first edition, 1995.

[8] M. Ó Cinnéide. *Automated Application of Design Patterns: a Refactoring Approach.* PhD dissertation, University of Dublin, Trinity College, Department of Computer Science, 2000. Available from: http://www.cs.ucd.ie/staff/meloc/home/papers/thesis/thesis.htm.

[9] M. Ó Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 463–472, Oxford, Sept. 1999. IEEE Press.

[10] D. Roberts and J. Brant. “Good enough” analysis for refactoring. In S. Ducasse and J. Weisbrod, editors, *ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, Brussels, July 1998. FZI Karlsruhe report.