Program Restructuring to Introduce Design Patterns

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

Paddy Nixon
Department of Computer Science
Trinity College Dublin
Ireland.
Paddy.Nixon@cs.tcd.ie

Abstract

In restructuring legacy code it may be useful to introduce a design pattern in order to add clarity to the system and thus facilitate further program evolution. We show that aspects of this transformation can be automated and present a nascent high-level language for describing transformations that introduce design patterns. The role of preconditions in preserving program behaviour during this process is also discussed. We conclude by considering the value of this approach in dealing with legacy code.

1. Introduction

Legacy systems frequently require restructuring in order to make them more amenable to future changes in requirements. This restructuring is normally performed by hand, but it is interesting to consider the potential for automated support. Much of the work in this area has suffered from a lack of a suitable target model for these restructurings, i.e., what exactly are we restructuring the program towards? One interesting class of transformations comprises those that introduce *design patterns* [Gam95]. Design patterns loosen the binding between program components thus enabling certain types of program evolution to occur with minimal change to the program itself. For example, the creation of a Product object within a Creator class could be replaced by an application of the Factory Method pattern. This enables the Creator class to be extended to use another type of Product object without much reworking of the existing code. We are investigating the possibilities of providing a rigorous foundation, methodology and tool support for the introduction of design patterns into existing object-oriented programs.

Our starting point is to assume that we are dealing with a program that has been developed without the use or knowledge of design patterns. This may be a legacy system that a designer is maintaining, or it may be a program that is being developed from scratch and is now being revised or *consolidated* [Foo94]. At some point the designer decides that a certain set of program components are too tightly coupled and that it would be useful to apply some particular design pattern to them. It is this type of transformation that we are interested in automating. Also, since the program already exhibits useful behaviour, this transformation should ideally be behaviour-preserving. It may however prove infeasible to apply the desired pattern because there are elements in the program that make it unsuitable or inapplicable, and the designer should be informed of this.

What we are considering therefore is transforming a given design structure into another design structure, the latter being the implementation structure of some design pattern. By implication, the initial structure has some poor design quality that is resolved by the application of the chosen design pattern. For this reason we call the initial design structure that is being transformed an *antipattern* [Koe95]. By this term we mean simply a micro-architecture that has some inherent weaknesses or inflexibilities and that suggests that a suitable design pattern might be applied[1].

There are several desirable features that we would expect a tool that performs these transformations to exhibit:

(1) It should be possible to describe the transformations in a way that is abstracted from the details of the programming language being used.

(2) The applicability of the chosen transformation to the program being operated on should be assessed in some way[2].

(3) For any design pattern there are several sets of similar but slightly different antipatterns. It should be possible to write a single transformation that covers as many of these as possible.

(4) The transformation should preserve the behaviour of the program.

In section 2 we look at a solution to the first problem, namely the development of a high-level language that supports the specification of these design pattern transformations. In section 3 we consider an approach that deals with the remaining issues by allowing each transformation to be given a set of preconditions. Finally in section 4 we present some conclusions.

2. Design Pattern Transformation Language (DPTL)

We have taken a set of design patterns, the so-called "Gang of Four" [Gam95] creational patterns (Abstract Factory, Factory Method, Builder, Singleton and Prototype) and for each pattern we have identified a typical antipattern and designed a transformation that converts this antipattern to the corresponding pattern. These transformations have been implemented in a tool called Design Pattern Tool (DPT) that operates on Java[3] programs.

[1] The term *antipattern* has caused an enormous amount of controversy judging by the amount of traffic recently generated on this topic on the pattern mailing lists. In our opinion this controversy is due to the proposal to treat antipatterns as first-class objects and to document them with the same rigour as is given to design patterns themselves. Our intention in using this term is considerably weaker than this.

[2] The scheme we describe in this paper merely assesses whether or not the present program is suited to the required design pattern transformation. A more complicated scheme involves deriving metrics from the design structures of the program and using these metrics as a basis to decide which design pattern transformation to recommend to the designer.

[3] We conjecture that there is already a significant amount of *legacy* Java code that has been designed without the use of design patterns.

The transformations themselves are also implemented in Java and perform surgery directly on the parse trees of the Java program being operated on.

Using our existing Design Pattern Tool as a basis, we are developing a high-level language (DPTL) for describing transformations that introduce design patterns. Some of the properties of this language are as follows:

- It provides the basic types that represent the common elements of an object-oriented program, for example Statement, Method, Constructor, Class, Interface etc.

- The low-level primitives are operations like addClass, addMethod, addInterfaceLink, addSuperclassLink and so on.

- High-level operations are provided that have proven useful in writing DPT so far, e.g., abstractClass, replaceObjectCreations etc.

- Set operations are supported that obviate the need for any form of iteration statement. For example:
  ```
  for all Methods m in class C do
   m.addInitialStatement(s);
  ```

- A decision statement is provided such as (for method `m` and Class `c`):
  ```
  if (m ε c)
   …
  ```

To give an example of how DPTL appears, the algorithm that describes the application of the Factory Method pattern is expressed as follows.

```
ApplyFactoryMethod(Class creator, Class product){

  addInterface(product, "abs"+product.name());

  creator.renameType(product.name(), "abs"+product.name());

  for all Constructor c in Class product do{
     Method newMethod("create"+product.name());
     newMethod.returnType="abs"+product.name();
     newMethod.paramList=c.paramList;
     newMethod.body=("return new P(" + c.paramList + ");");
     creator.addMethod(newMethod);
  }

  creator.replaceObjectCreations(product, "create"+product.name());
}
```

This algorithm works as follows. The public methods of the Product class are abstracted into an interface and an "implements" link is added from the Product class to this interface. All references to the Product class in the Creator class are then updated to refer to this interface. Now for every constructor in the Product class, a similar method is added to the Creator class that returns an instance of the Product class. Finally all creations of Product objects in the Creator class are updated to use these new methods.

At present the syntax of DPTL is described by a formal grammar while the semantics are stated informally in English. We intend to formalise the object model upon which DPTL is based and then specify more precisely the semantics of DPTL in terms of this model. We require this model to support notions such as classes, methods, fields, inheritance, interfaces, and object creation. Two possible models under consideration to base this work on are described in [Cas92] and [Aba96].

3. Behaviour-Preservation and Preconditions

The application of a transformation to introduce a design pattern should not change the behaviour of the program being transformed. In general this can only be guaranteed if the initial structure of the program fulfils certain requirements. These can be specified as a set of preconditions that must hold before the transformation is applied. The resultant transformation is then amenable to mathematical proof, although this is clearly not tractable in all cases. This approach is also taken in other work on refactoring object-oriented programs, e.g. [Opd92], but to our knowledge it has not as yet been applied to design pattern transformations.

Consider the application of the transformation described above that introduces the Factory Method pattern. An obvious requirement is that the Creator class does actually create an instance of the Product class. Also, if the Product class has a public data field this transformation cannot be applied, as such a field cannot be accessed through an interface. A similar problem occurs if the Product class contains a static method.

We can specify these preconditions as follows:

```
applyFactoryMethod Preconditions:

  creator.creates(product)          // antipattern precondition
  not product.hasPublicField()      // refactoring precondition
  not product.hasStaticMethod()     // contraindication
```

The three conjuncts of this precondition illustrate three quite distinct types of requirement. In the following paragraphs we describe each one.

Antipattern precondition: The first conjunct describes the antipattern that this transformation resolves. To state that “class A instantiates class B” is an antipattern may appear extreme, but in fact this does reflect a tight coupling between the two classes. It is reasonable to inspect this relationship and determine what future program evolutions would be facilitated were it to be replaced by a suitable design pattern.

Refactoring precondition: The second conjunct is an example of a minor problem that prevents this transformation from being applied. The class has public data, which is a well-established example of poor class design. This class can be refactored automatically to make this data private and instead provide access to the fields via public accessor and mutator methods[4]. This then removes this obstacle to the application of the transformation.

Contraindication: The final conjunct suggests 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 concrete class of the Product it uses and this cannot be replaced by access via an abstract interface. This is an *inherent* problem in the program design that prevents the application of the required pattern. In this case the design must be revisited to determine if it is possible to resolve this issue.

Note that the antipatterns that can be resolved by any given pattern are impossible to completely specify. For a description of the preconditions for a more complicated transformation (one that introduces the Decorator pattern) see [OCi96].

4. Assessments and Conclusions

We have described a scheme that enables the semi-automatic introduction of design patterns to legacy programs. This has the potential to remove an amount of the repetitive and error-prone work involved in restructuring legacy code. Other similar approaches ([DMR97], [EGY97], [FMW97]) address these issues as well; our approach differs primarily in the use of preconditions to:

- assess the suitability of the pattern the designer wishes to apply;
- perform some pre-transformation refactorings;
- assess if the application of this transformation is contraindicated by other properties of the program.

We chose to describe our transformations as algorithms. Another approach is to simply state the postconditions for the transformation in a declarative style. This opens the interesting possibility of automatically deriving an algorithm to achieve the transformation. Postconditions are used in [FMW97] to ensure that the intent of a design pattern is maintained in the program during later evolution. At a later stage in this project we plan to evaluate the merits of algorithmic versus postcondition approaches.

Our work from here involves developing DPTL further, working on the model within which to rigorously define its meaning and gaining more practical experience with the expression of preconditions for design pattern transformations. We hope to demonstrate that these techniques make a useful contribution to the problem of restructuring legacy code to enhance extendibility.

5. References

[Aba96] Abadí, M. and Cardelli, L., *A Theory of Objects*, Springer-Verlag, 1996.

[4] The techniques described in [Opd92] can be used effectively here.

[Cas92] Casais, E., *An Incremental Class Reorganisation Approach*, ECOOP, June 1992.

[DMR97] Demeyer, S., Meijler, T.D. and Rieger, M., *Towards Design Pattern Transformations*, Proceedings of the Workshop on Object-Oriented Software Evolution and Re-Engineering, ECOOP, June 1997.

[EGY97] Eden A.H., Gil J., Yehudai, A., *Precise Specification and Automatic Application of Design Patterns*, Twelfth IEEE International Automated Software Engineering Conference, 1997.

[FMW97] Florijn, G., Meijers, M. and van Winsen, P., *Tool Support in Design Patterns*, ECOOP, June 1997.

[Foo94] Foote, B. and Opdyke W. F., *Lifecycle and Refactoring Patterns that Support Evolutions and Reuse*, PLoP, 1994.

[Gam95] Gamma, E. et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Koe95] Koenig, A., *Patterns and Antipatterns*, Journal of Object-Oriented Programming, April, 1995.

[OCi96] Ó Cinnéide, M., *Towards Automating the Introduction of the Decorator Pattern to Avoid Subclass Explosion*, Technical Report TR-97-7, Department of Computer Science, University College Dublin, Ireland (also accepted for the Object-Oriented Evolution and Re-engineering Workshop, OOPSLA, San José, October 1996).

[Opd92] Opdyke, W. F., *Refactoring Object-Oriented Frameworks*, PhD thesis, University of Illinois, 1992.