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

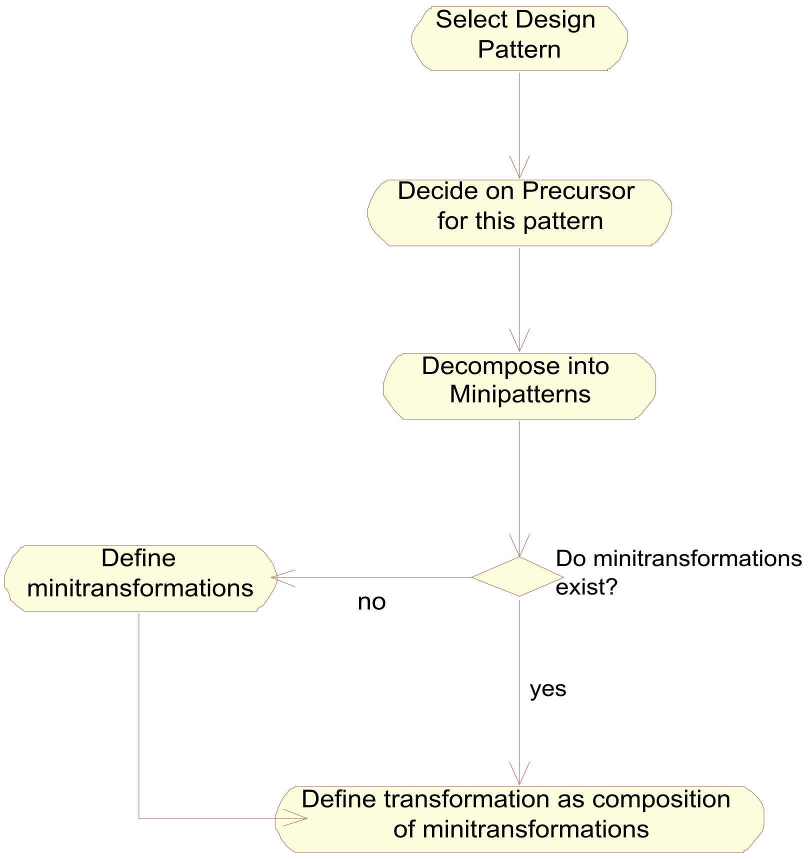
Select Design Pattern
Decide on Precursor for this pattern
Decompose into Minipatterns
Do minitransformations exist?
no
Define minitransformations
yes
Define transformation as composition of minitransformations

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

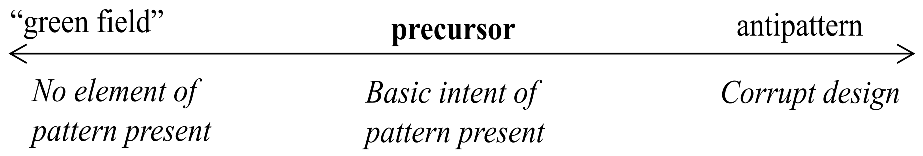
"green field"
precursor
antipattern
No element of pattern present
Basic intent of pattern present
Corrupt design

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].

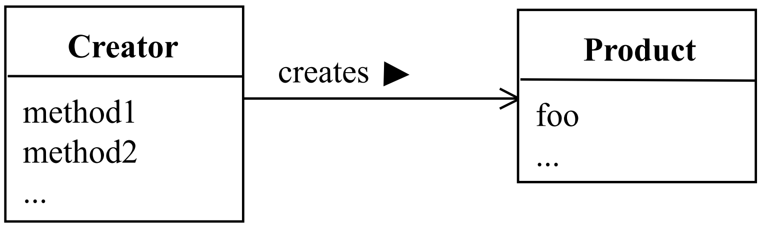

Figure 4.3: Precursor for the Factory Method Transformation

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:

$\neg\text{isClass}(\mathit{newName}) \land \neg\text{isInterface}(\mathit{newName})$

postcondition:

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

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

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

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

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

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

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

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.

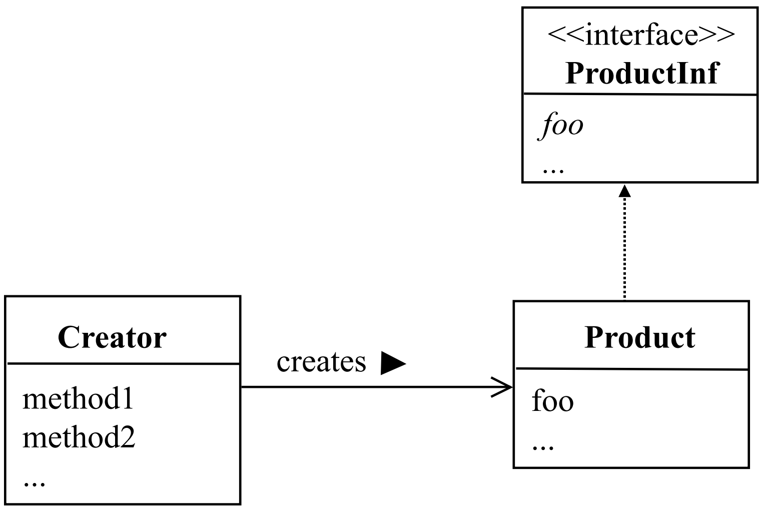

Figure 4.4: Application of the Abstraction Minitransformation

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*:

∀ c:Constructor, c ∈ *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-

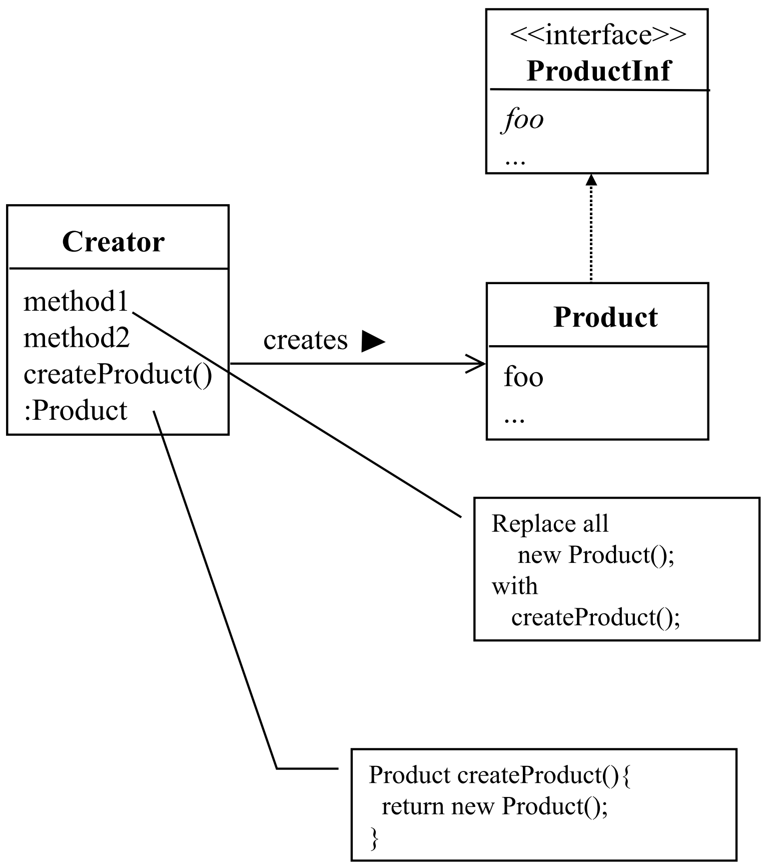

Figure 4.5: Application of the ENCAPSULATECONSTRUCTION Minitransformation

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:

isInterface(*inf*) $\wedge$ isClass(*context*) $\wedge$ isClass(*concrete*)

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

implementsInterface(*concrete*, *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.

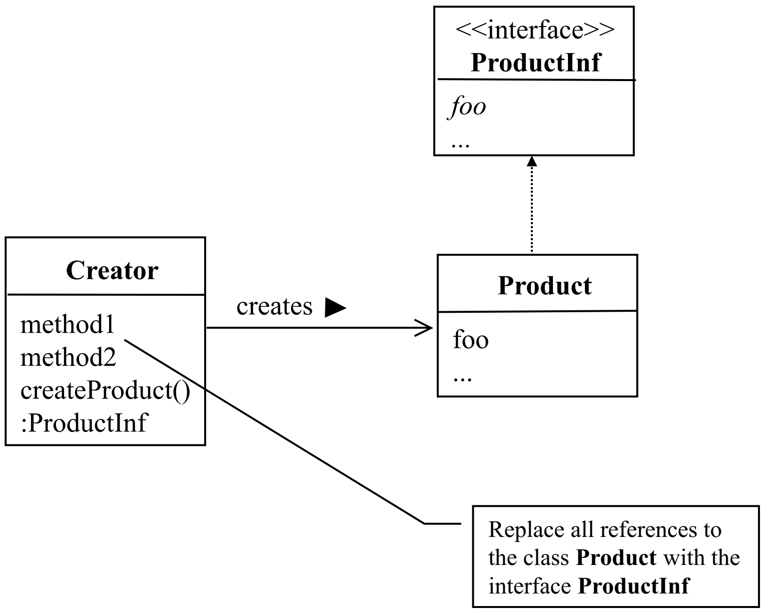

Figure 4.6: Application of the AbstractAccess Minitransformation

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 ∈ concrete, nameOf(m) ∈ 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

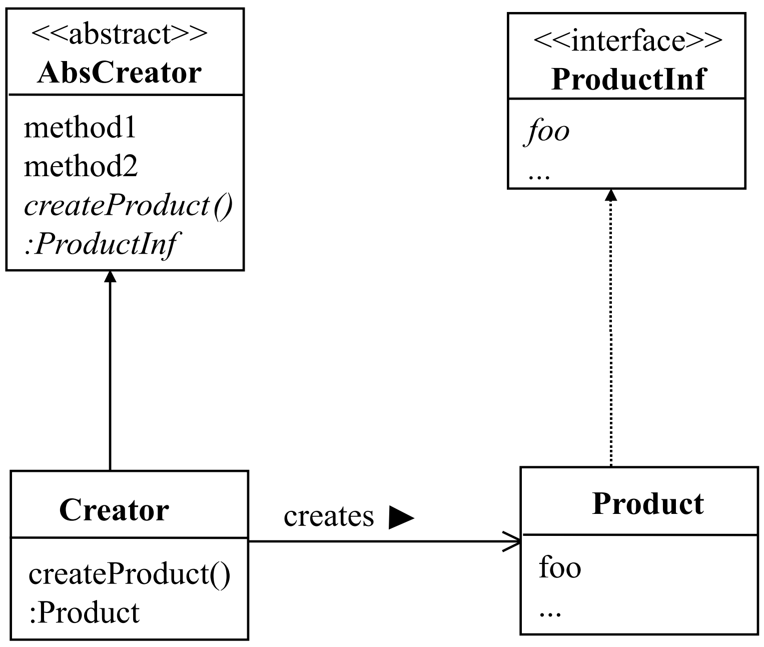

Figure 4.7: Application of the PARTIALABSTRACTION Minitransformation

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, micropattern (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.