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 $\notin$ *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

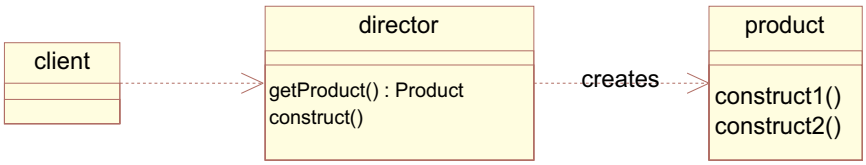

Figure 5.1: The Precursor for the Builder Design Pattern

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 con-

struction 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.

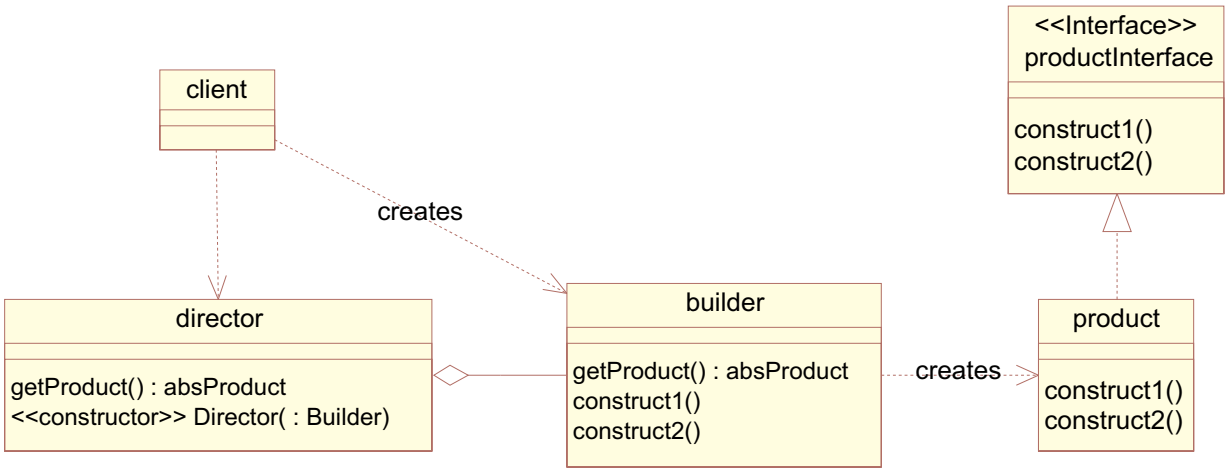

Figure 5.2: The Builder Design Pattern

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:

$$\text{isClass}(director) \wedge \text{isClass}(product) \wedge \neg\text{isClass}(builderName) \wedge \neg\text{isInterface}(builderName)$$

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

$$\forall\ \text{m:Method},\ \text{m} \in product \bullet \neg\text{isStatic(m)}$$

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

$$\forall\ \text{f:field},\ \text{f} \in product \bullet \neg\text{isPublic(f)}$$

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

$$\forall\ \text{c:Constructor},\ \text{c} \in product \bullet \text{noOfParameters(c)}=0$$

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 ∈ 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:

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

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

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

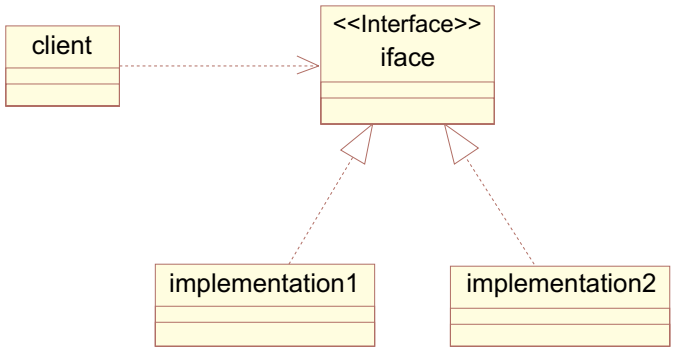

Figure 5.3: The Precursor for the Bridge Design Pattern

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:

isInterface(*iface*)

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

$\neg$ isClass(*wrapperName*) $\wedge$ $\neg$ isInterface(*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:

isClass′ = isClass[*wrapper*/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*:

∀ o:ObjectRef, containingClass(o) ∈ *clients*,

implementsInterface(typeOf(o), *iface*) •

typeOf′=typeOf[o/*wrapper*]

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

∀ e:ObjectCreationExprn, implementsInterface(classCreated(e), *iface*), containingClass(e) ∈ *clients* •

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:

∀ c:Class, implementsInterface(c,*iface*) •

∀ e:ObjectCreationExprn, classCreated(e)=c •

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*);

}

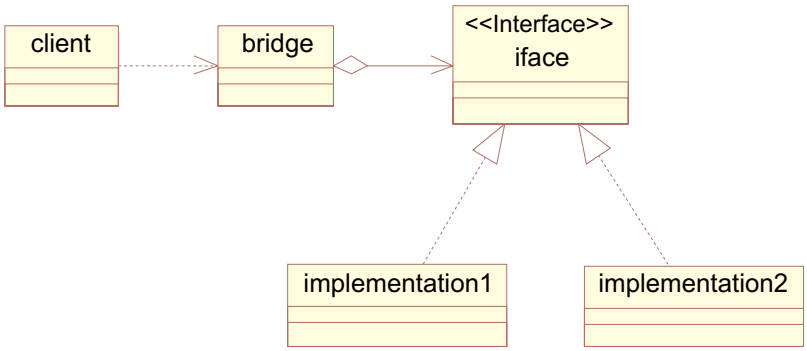

Figure 5.4: The Bridge Design Pattern

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:

isClass(*context*)

The name for the delegation class must not be use:

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

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

$\forall$ m $\in$ *moveMethods* $\bullet$ m $\in$ *context*

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

$\forall$ f:Field, f$\in$*context* $\bullet$ nameOf(f) $\neq$ "delegation"

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

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

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

postcondition:

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

isClass$'$ = isClass[*delegationName*/true]

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

$\exists$ f:Field, f $\in$ *context* such that

typeOf$'$=typeOf[f/*delegationName*]

nameOf$'$=nameOf[f/"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 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.