Chapter 3

Foundations of Refactoring: Behaviour Preservation

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

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

3.1 Primitive Refactorings and Behaviour Preservation

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

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

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

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

3.1.1 Mathematical Preliminaries

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

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

3.1.2 Analysis Functions

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

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

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

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

condition, but this is a straightforward task.

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

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

Program Entities

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

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

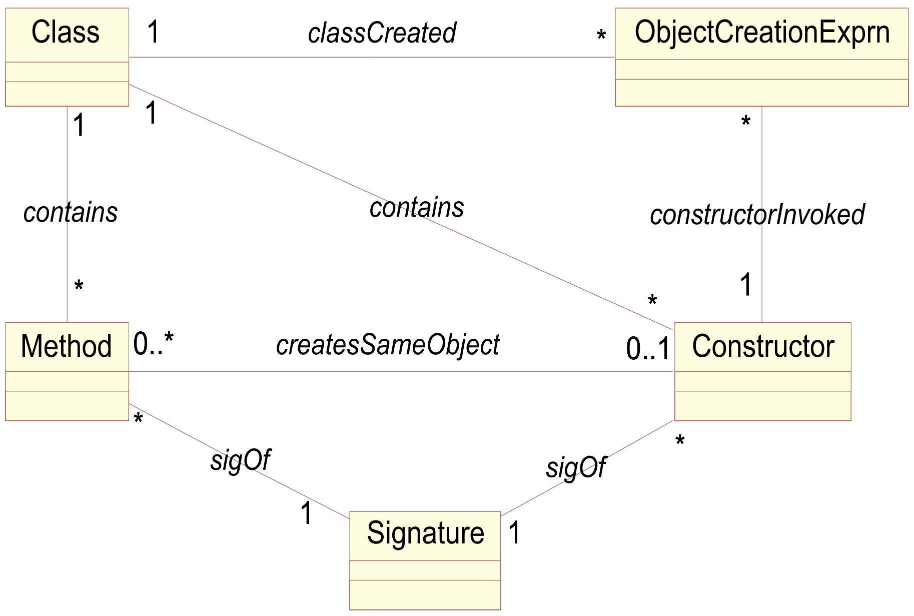

Figure 3.1: Principal Program Entities and their Relationships

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

3.1.3 Helper Functions

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

3.1.4 Primitive Refactorings

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

Each primitive refactoring is described in the following way:

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

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

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

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

3.1.5 Assumptions and Limitations

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

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

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

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

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

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

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

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

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

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

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

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

3.2 Composite Refactorings

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

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

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

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

- If the composite refactoring is legal, testing its precondition will normally be faster, and never slower, than testing the precondition of each component refactoring separately.
- If the composite refactoring is illegal, testing its precondition will be considerably faster than applying several of the component refactorings and then being obliged to rollback to the starting point. Note that some refactorings are not undoable, so supporting rollback would involve checkpointing.

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

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

1. Chaining.
2. Set iteration.

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

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

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

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

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

3.2.1 Computing Pre- and Postconditions for a Chain of Refactorings

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

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

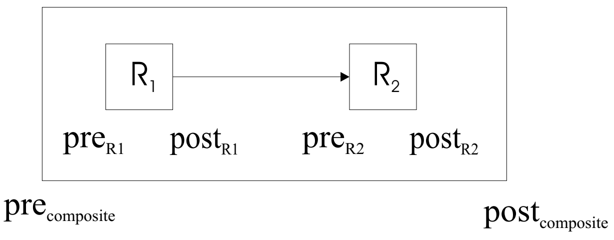

Figure 3.2: A Composite Refactoring with Pre- and Postconditions

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

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

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

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

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

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

addMethod(c,m)

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

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

Legality test and precondition computation

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

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

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

The precondition of the complete chain is obtained by evaluating:

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

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

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

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

Postcondition computation

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

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

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

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

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

and $post_{R_2}$ contains the mapping:

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

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

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

Table 3.1 describes how this operator works in general.

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

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

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

Table 3.1: Concatenation of Postconditions

3.2.2 Computing Pre- and postconditions for a Set Iteration

A set iteration has the following format:

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

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

R_1, R_2, ..., R_n

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

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

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

$$\forall R_i, i \in \{1..n\}, \models_{\mathcal{I}_P} pre_{R_i} = \models_{\mathcal{I}_{P'}} pre_{R_i}$$
$$\text{where } P' = post_{R_{j_m}}(\ldots post_{R_{j_2}}(post_{R_{j_1}}(\mathcal{I}_P))),$$
$$j_m \in \{1..n\} - \{i\},\ j_x = j_y \Rightarrow x = y$$

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

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

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

precondition for the above chain can be expressed as:

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

or in a more useful form as:

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

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

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

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

3.2.3 A Worked Example

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

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

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

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

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

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

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

1. Compute *pre* and *post* for the chain in the first set iteration body
2. Compute *pre* and *post* for the first set iteration
3. Compute *pre* and *post* for the second set iteration
4. Compute *pre* and *post* for the overall chain

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

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

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

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

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

   No contradiction occurred so the chain is legal.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   replaceObjCreationWithMethInvocation(e,createP)

   is that e is deleted, i.e.,

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

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

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

   $containingMethod(e) \neq createP$

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

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

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

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

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

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

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

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

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

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

   Postcondition 3.4 may therefore be rewritten thus[8]:

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

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

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

give:

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

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

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

Simplifying this out gives:

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

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

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

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

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

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

3.2.4 Commentary

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

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

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

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

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

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

3.3 Related Work

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

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

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

[10]An unfortunate redefinition of an existing term.

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

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

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

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

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

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

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

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

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

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

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

3.4 Summary

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

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