Composite Refactorings for Java Programs

Mel Ó Cinnéide[1] and Paddy Nixon[2]

[1] Department of Computer Science, University College Dublin, Dublin 4, Ireland.
mel.ocinneide@ucd.ie
http://www.cs.ucd.ie/staff/meloc/default.htm
[2] Department of Computer Science, Trinity College Dublin, Dublin 2, Ireland.
paddy.nixon@cs.ucd.ie

Abstract. There has been much interest in refactoring recently, but little work has been done on tool support for refactoring or on demonstrating that a refactoring does indeed preserve program behaviour. We propose a method for developing composite refactorings for Java programs in such a way that a rigorous demonstration of behaviour preservation is possible.

1 Introduction

A refactoring is a change made to the internal structure of software that improves it in some way but does not alter its observable behaviour [2]. Refactoring has increased in importance as a technique for improving the design of existing code, especially with the advent of methodologies such as Extreme Programming [1] that involve little up-front design and multiple iterations through the software lifecycle. The earliest significant work on refactoring was the suite of C++ refactorings developed by William Opdyke [4]. This work was hampered by the low-level complexities of the C++ language and was never developed into a practical tool. It did however form the basis for the development of the Smalltalk Refactory Browser [6]. Smalltalk is a much cleaner language than C++ and this refactoring tool has been very successful. Its principal limitation is probably that Smalltalk is not a very widely-used language outside of academia. These experiences suggest that the Java programming language may be a promising language to serve as a domain for refactoring development. In spite of the obvious syntactic similarities, it is a much simpler language than C++ and has become extremely popular in the past number of years.

The ultimate aim of our work is the development of a methodology for the construction of program transformations that introduce design patterns to a Java program [3]. The algorithm that describes a design pattern transformation is expressed as a composition of refactorings using sequencing, iteration and conditional constructs. This type of transformation must not change the behaviour of the progam, and so it is necessary to be able to calculate if a given composition of primitive refactorings is itself behaviour preserving. Each primitive refactoring has a precondition and a postcondition. When applied to a program for which

the precondition holds, the resulting transformed program exhibits the same behaviour as the original, and the postcondition holds for this program. This paper describes a technique that, given a composition of refactorings, computes whether it is a refactoring, and what its pre- and postconditions are.

2 Preliminaries

Is this section we describe the mathematical notation we use and outline the basic elements of our approach.

2.1 Notation

We use the same notation as Roberts in his refactoring work [5]:

- P: This is the program to be refactored.
- $\mathcal{I}_P$: Denotes an interpretation of first-order predicate logic where the domain of discourse is the program elements of P.
- $\models_{\mathcal{I}_P} pre_R$: Denotes the evaluation of the precondition of the refactoring R on the interpretation $\mathcal{I}_P$.
- $post_R(\mathcal{I}_P)$: Denotes the interpretation $\mathcal{I}_P$, rewritten with the postcondition of the refactoring R.
- $f[(x, y)/true]$: Denotes the function f, extended with the new element (x, y). This syntax is used in postconditions to describe the effect of the refactoring on the analysis functions.

2.2 Analysis Functions

We do not explicitly build an internal representation of the program; rather the required information is extracted when needed. Analysis functions are used to extract this information. They serve a dual role in that they are used both in specifying the preconditions to the refactorings and as an transformation designer's view of the program being transformed. An example of the specification of an analysis function is as follows:

> *Boolean* **contains**(*c*:*Class*, *m*:*Method*): Returns true iff the class *c* contains the method *m*.

2.3 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. Helper functions are used to perform this type of task. As they are not at the primitive level of the analysis functions, we provide them with pre- and postconditions. Helper functions are proper functions without side-effects on the program, so the postcondition invariably involves the return value of the helper function itself. For example the *makeAbstract* helper function is specified as follows:

Method **makeAbstract**(*c*:*Constructor*, *newName*:*String*): Returns a method called *newName* that, given the same arguments, will create the same object as the constructor *c*.

pre: None.

post: createsSameObject′ = createsSameObject[(*c*,*m*)/true] ∧
nameOf′ = nameOf[(*m*,*newName*)/true], where *m* is the returned method.

2.4 Primitive Refactorings

Composite refactorings are built upon a layer of primitive refactorings. Each primitive refactoring is given a precondition written in first-order predicate logic and a postcondition that describes the effect of applying this refactoring in terms of changes to the relevant analysis functions. An argument that behaviour is preserved by this transformation is also provided. This is not formal, but it is at least as strong as the argument a programmer would make internally were they to perform the refactoring by hand. Also, this argument is only made once by the designer of the primitive refactoring and is effectively reused each time a new composite refactoring uses the primitive refactoring. As an example, the *addMethod* refactoring is specified as follows:

addMethod(*c*:*Class*, *m*:*Method*): Adds the method *m* to the class *c*. A method with this signature must not already exist in this class or its superclasses. This refactoring extends the external interface of the class.

pre: isClass(*c*) ∧ ¬defines(*c*, *nameOf(m)*, *sigOf(m)*)

post: contains′ = contains[(*c*,*m*)/true] ∧
∀ a:Class, a≠c, if equalInterface(*a*,*c*) then
equalInterface′ = equalInterface[(*a*,*c*)/false].

behaviour preservation: Since a method with the same name and signature as the method being added is not defined in the class, there can be no name clashes and no existing invocations of this method.

3 Composite Refactorings

In this section we describe the way in which refactorings are composed, and present a technique for deriving the pre- and postconditions of a composite refactoring. The importance of this technique lies in the fact that it allows us to build complex transformations as a composition of primitive refactorings and then to check the legality of the composition and calculate its pre- and postconditions. Here we consider two ways in which refactorings are composed, namely *chaining* and *set iteration*.

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

```
addMethod(c,foo)
addMethod(c,foobar)
```

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)
}
```

A selection statement can also be used. For space reasons we omit it here.

3.1 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 repeatedly applied to the remaining chain until the full pre- and postconditions have been computed.

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. Figure 1 presents a graphical depiction of this. The

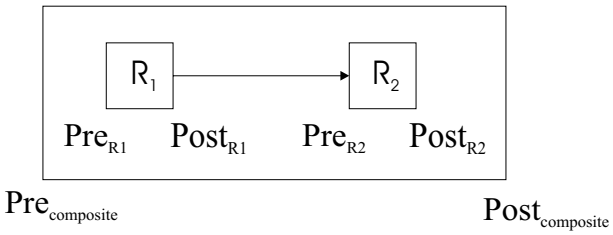

Fig. 1. A Refactoring Chain

precondition of this chain is not simply the conjunction of pre_{R_1} and pre_{R_2}. Firstly, $post_{R_1}$ may guarantee pre_{R_2} which means that an unnecessarily strong precondition would result. Secondly, although the precondition for R_2 may be made part of the precondition for the chain, the refactoring R_1 may break it meaning that this composition of refactorings can never be legal.

The technique we present first attempts to compute the precondition of the chain. During this computation it may emerge that the chain is illegal. Assuming the chain is indeed legal, its postcondition is computed.

1. *Legality test and precondition computation*: First we compute the parts of pre_{R_2} that are not guaranteed by $post_{R_1}$:
   $\models_{post_{R_1}(\mathcal{I}_P)} pre_{R_2}$
   If a contradiction arises in this evaluation, the chain is illegal. The postcondition of the first refactoring creates 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 demands a certain condition that contradicts the precondition to the second refactoring, and the first refactoring does not change this condition.

2. *Postcondition computation*: The postcondition is obtained by "sequentially ANDing" the postconditions. By this we mean that if $post_{R_1} \wedge post_{R_2}$ leads to a contradiction, the part of $post_{R_1}$ that causes the problem is dropped. So if $post_{R_1}$ contains the mapping:

   $classOf' = classOf[(foo, c)/true]$

   and $post_{R_2}$ contains the mapping:

   $classOf' = classOf[(foo, c)/false]$

   then it is $classOf' = classOf[(foo, c)/false]$ that becomes part of the postcondition of the chain. Denoting this operator as $\wedge_{seq}$ we state the postcondition of the chain to be:

   $post_{R_1} \wedge_{seq} post_{R_2}$

3.2 Computing Pre- and Postconditions for a Set Iteration

A set iteration has the following format:

```
ForAll x:someProgElement, somePredicate(x,...) {
        someRefactoring(x, ...)
}
```

where "..." denotes the program entities that are arguments to the predicate and/or arguments to the refactoring. If the set of x of type *someProgElement* that satisfies $somePredicate(x, \ldots)$ is given as $\{x_1, x_2 \ldots, x_n\}$, then this iteration may be viewed as the following chain:

someRefactoring$(x_1, \ldots)$
someRefactoring$(x_2, \ldots)$
...
someRefactoring$(x_n, \ldots)$

However this is a *set* iteration, so the refactorings can take place in any order. In particular any of them can be first 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.
2. *Precondition computation*: The precondition of the first refactoring of a chain must form part of the precondition for the whole chain, so 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 precondition for the above chain can be expressed as:

$$\bigwedge_{i=1}^{i=n} pre_{someRefactoring(x_i,...)}$$

3. *Postcondition computation*: By a similar argument, the postcondition for the above chain can be expressed as:
$$\bigwedge_{i=1}^{i=n} post_{someRefactoring(x_i,...)}$$

The legality test for a set iteration is not as prescriptive as in the chaining case. It is usually necessary to study the general postcondition carefully to ensure that it has no impact on the precondition on another iteration. It has nevertheless proved to be useful in the cases we have examined.

4 Conclusions

We have described a method for building composite refactorings based on a set of primitive refactorings. Our layer of primitive refactorings are Java-specific, though in principle they could be defined for another language as well. The method of computing the pre- and postconditions of a composite refactoring is similar to Roberts' approach [5], but our work improves on this in several ways. We explicitly calculate whether or not a composite refactoring is legal and we also compute its postcondition as we want to be able to use this composite as a component in future composite refactorings. Roberts also only permits chains of refactorings and does not consider any type of iteration.

We have successfully designed and implemented several composite refactorings that introduce design patterns to a Java program. The techniques described here improved our confidence greatly that the transformations are behaviour preserving. The computation of pre- and postconditions is currently performed by hand by the designer of the design pattern transformation, but our aim is to automate much of this work in the future. We believe that the layer of primitive refactorings we have built coupled with the composition method provides a useful support for further work in the area of refactoring of Java programs.

References

1. Kent Beck. *Extreme Programming.* Addison Wesley Longman, Reading, Massachusetts, first edition, 2000.
2. Martin Fowler. *Refactoring: improving the design of existing code.* Object Technology Series. Addison-Wesley Longman, Reading, Massachusetts, first edition, 1999.
3. Mel Ó Cinnéide and Paddy Nixon. A methodology for the automated introduction of design patterns. In Hongji Yang and Lee White, editors, *Proceedings of the International Conference on Software Maintenence*, pages 463–472, Oxford, September 1999. IEEE Press.
4. William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.
5. Donald Roberts. *Eliminating Analysis in Refactoring.* PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 1999.
6. Donald Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.