Refactoring Composite to Visitor and Inverse Transformation in Java*

Akram Ajouli$^1$ & Julien Cohen$^2$

1: INRIA – ASCOLA team (EMN - INRIA - LINA)
2: Université de Nantes – LINA (UMR 6241, CNRS, Univ. Nantes, EMN)

We describe how to use refactoring tools to transform a Java program conforming to the Composite design pattern into a program conforming to the Visitor design pattern with the same external behavior. We also describe the inverse transformation. We use the refactoring tools provided by IntelliJ IDEA and Eclipse.

Contents

1 Introduction 2
2 General Approach 3
   2.1 Guidelines in the Literature 3
   2.2 Automation 4
3 Composite ↔ Visitor Transformation Scheme 7
   3.1 Composite → Visitor Transformation 7
   3.2 Visitor → Composite Transformation 9
   3.3 Result after Round Trip Transformation 13
4 Variants of Transformations for Various Pattern Instances 16
   4.1 Methods with Parameters 16
      4.1.1 Composite → Visitor Transformation 16
      4.1.2 Visitor Program 17
      4.1.3 Visitor → Composite Transformation 17
   4.2 Methods Returning Values 17
      4.2.1 Composite → Visitor Transformation 18
      4.2.2 Visitor Program 18
      4.2.3 Visitor → Composite Transformation 18
   4.3 Interface instead of Abstract class in the Composite structure 18
      4.3.1 Composite → Visitor Transformation 18
      4.3.2 Visitor Program 22
      4.3.3 Visitor → Composite Transformation 22
   4.4 Class Hierarchies with Several Levels 22
      4.4.1 Composite → Visitor Transformation 22
      4.4.2 Visitor Program 23
      4.4.3 Visitor → Composite Transformation 23
5 Application to JHotDraw 26
6 Related work 26
   6.1 Refactoring to Patterns 26
   6.2 Building Complex Refactoring Operations 27
   6.3 Design Patterns Discovery 27
7 Conclusion 27
References 27

*This is the version 2 of the report. The main difference with the version 1 is the description of the computation of the minimum precondition for the base round-trip transformation (see App. B).
Composite and Visitor patterns have dual properties with respect to modularity: while the Composite pattern (as well as the Interpreter pattern and classic class hierarchies) provides modularity along subtypes and leaves operation definitions crosscut, the Visitor pattern provides modularity along operations and leaves behavior definitions crosscutting with respect to subtypes [GHJV95].

One solution to have modularity along operations and subtypes would be to be able to transform automatically a program conforming to the Composite pattern into a program with the same behavior, but which structure would conform to the Visitor pattern, and vice-versa [CDA12].

Chains of elementary refactorings can be used to make design patterns appear [OCN99, Ker04], for instance to introduce the Visitor pattern [MT04, Ker04], or to replace the Visitor pattern by the Interpreter pattern [HKVDSV11]. However, such transformations are not fully automated yet, and our proposal of navigation between several architectures for a same program [CDA12] is not currently workable.

In this report we do preliminary work before automating refactoring based Composite ↔ Visitor transformations:

1. We give chains of refactoring operations that provide Composite → Visitor and Visitor → Composite transformations for a simple Java program. Each refactoring operation is supported by at least one refactoring tool.
2. We explain how to use the refactoring tools IntelliJ IDEA and Eclipse to perform the needed refactoring operations (composition of several operations of the tools, specific options, applying some operations before being able to perform another one, bugs to overcome, missing operations...).
3. We study variants of the transformations for several variations in the implementation of the patterns.

Our algorithms are validated on a running toy example and on the JHotDraw program [GI].
2 General Approach

We consider the Java program of Fig. 1. It contains a classic class hierarchy: the abstract class Graphic has two subclasses, Square and Ellipse, and two methods, print and prettyprint implemented in the subclasses. We also consider that two classes Printer and PrettyPrinter already exist in the program: they will become visitor subclasses.

```
abstract class Graphic {
    abstract public void print();
    abstract public void prettyprint();
}

class Square extends Graphic {
    int l;
    public void print() {
        System.out.print("Square( + l + ")");
    }
    public void prettyprint(){
        System.out.print("Square.");
    }
}

class Ellipse extends Graphic{
    int l1, l2;
    public void print() {
        System.out.print("Ellipse: ( + l1 +", l2 + ")");
    }
    public void prettyprint(){
        System.out.print("Ellipse.");
    }
}
```

![Diagram](image)

Figure 1: Base Program (classic class hierarchy)

In the following algorithms, we make abstraction of the class and method names and number: let \( \text{LM} \) be the set of traversal functions, \( \text{LC} \) the set of concrete classes in the composite structure, and \( \text{S} \) the superclass of the composite structure.

Here, \( \text{LM} = \{\text{print, prettyprint}\} \), \( \text{LC} = \{\text{Ellipse, Square}\} \) and \( \text{S} = \text{Graphic} \).

We also define a function \( V \) that maps a name of visitor class to a name of method. We consider here \( V(\text{print}) = \text{Printer} \) and \( V(\text{prettyprint}) = \text{PrettyPrinter} \). We also define \( \text{LV} = V(\text{LM}) = \{V(m)\}_{m \in \text{LM}} \).

2.1 Guidelines in the Literature

We start by considering some guidelines given in the literature for introducing an instance of the Visitor pattern into a typical object-oriented class hierarchy. We consider the guidelines of Mens and Tourwé [MT04], rephrased in Fig. 2.
To introduce a visitor pattern, the first obvious step is to move the business code from the class hierarchy to visitor classes (in this section, we consider the target classes for the moved methods already exist in the project). This is done in step 1 (Fig. 2). We move the business code but, in order not to change the interface of the class hierarchy, we keep in the class hierarchy some methods with the same profiles as the original ones, which will be delegators to visitor’s methods (see Move Method in Fowler [Fow99]).

The new methods in visitor classes are named visit so that the visitor classes will all be able to implement the abstract class Visitor, which is added afterward (step 2). In visitor classes, there is one method visit for each concrete class of the class hierarchy LC (with overloading). They are introduced as abstract methods in the Visitor class (step 3).

To introduce the double dispatch which is typical of the visitor pattern without changing the interface of the class hierarchy, another delegation is introduced inside the concrete classes of LC (step 4). The delegate method is named accept.

Since the initial methods are now delegators to accept, the overriding bodies are the same in the concrete classes of LC, and it can be defined once for all in the super class (step 5).

The refactoring results in the program given in Figs. 3 and 4.

2.2 Automation

If we refer to Fowler [Fow99], a refactoring is manual with checks under the responsibility of the programmer. In the same way, the general guidelines given in Fig. 2 must be interpreted by someone which will adapt them to his particular program.

We now consider that the programmer uses a refactoring tool. We consider IntelliJ IDEA but the same is also possible with Eclipse with small variations.

Prepare the move. A first problem occurs with the Move Method operation. The refactoring tool cannot move instance methods to a class if there is no reference of the destination class in that method (in parameters or in body). The reason is that the receiver object cannot be inferred otherwise (we consider instance methods).

Before moving the methods, we have to create delegates for these methods (to keep the initial interface), then add a parameter of the convenient visitor type to the delegates, then move them (see Fig. 5, step 1).

Restore object type after move. In our example, the pretty-print method does not access to any instance variables or methods (see Fig. 1) of the receiver object. In this case, when the prettyprint delegate methods are moved, the tool does not make a parameter of type Ellipse or Square appear in the resulting method.

This is problematic because we want overloaded visit methods (it’s a design choice, here we could also use different method names) but the lack of these parameters introduces a name clash.

To solve this, it is sufficient to apply the Add Parameter refactoring to the methods which have been moved. We do not make this appear into the algorithm of Fig. 5 because we encapsulate this behavior into the Move Method operation. We consider Move Method is an abstract operation, which can be implemented by a refactoring tool with a single operation or with a composition/chain of several basic operations. We make the correspondence between abstract operation and tool operations in App. A (see App. A.6).

We call business code the code that defines the operations, here the bodies of print and prettyprint, which are spread over several classes (by the means of overriding).
abstract class Graphic {
    public void print() {
        accept(new PrintVisitor());
    }
    public void prettyprint() {
        accept(new PrettyPrintVisitor());
    }
    public abstract void accept(Visitor v);
}

class Square extends Graphic {
    int l;
    public void accept(Visitor v) {
        v.visit(this);
    }
}

class Ellipse extends Graphic{
    int l1, l2;
    public void accept(Visitor v) {
        v.visit(this);
    }
}

Figure 3: Program with Visitor (classic class hierarchy)

public abstract class Visitor {
    public abstract void visit(Square square);
    public abstract void visit(Ellipse ellipse);
}

public class PrintVisitor extends Visitor {
    public void visit(Square square) {
        System.out.print("Square(" + square.l + ")");
    }
    public void visit(Ellipse ellipse) {
        System.out.print("Ellipse: (" + ellipse.l1 +".," + ellipse.l2 + ")");
    }
}

public class PrettyPrintVisitor extends Visitor {
    public void visit(Square s)
    {
        System.out.print("Square.");
    }
    public void visit(Ellipse e)
    {
        System.out.print("Ellipse.");
    }
}

Figure 4: Program with Visitor (classic class hierarchy – visitor part)
1. ForAll (m, param) in LM, c in LC do
   Let visitorname = V(m) in
   AddParameterWithDelegate(c, m, param, visitorname)
   MoveMethod(c, m, param+visitorname, visitorname)
   RenameMethod(visitorname, m, param+c, "visit")
   done
2. ExtractSuperClass(LV, "Visitor") // with visit abstract methods
3. ForAll c in LC do
   ExtractGeneralMethod(c, LM, "accept", "Visitor")
4. PullUpAbstract(LC, "accept", "Visitor", S)
5. ForAll m in LM do
   PullUpConcrete(LC, m, S)

Figure 5: Simple Class Hierarchy → Visitor transformation (adapted to IntelliJ IDEA)

**ExtractSuperClass.** Introducing a new superclass and pulling up methods (steps 2 and 3 of Fig. 2) is known as Extract Superclass in Fowler [Fow99]. That composite operations is also available in IntelliJ IDEA and Eclipse. For that reason, we use it in Fig. 5 (step 2).

However, in IntelliJ IDEA, we have had to provide an extension of that operation that applies to several classes simultaneously (it was already possible in Eclipse).

**Extract Method Accept.** In the following code (from Square or Ellipse), the instruction o.visit(this) occurs twice (with a different object o).

```java
public void print() {
    new PrintVisitor().visit(this);
}
public void prettyprint() {
    new PrettyPrintVisitor().visit(this);
}
```

That instruction has to be extracted into a method accept with o as a parameter, and the occurrences of that expression will be replaced by accept(o).

The tool IntelliJ IDEA will accept to extract a same method for the two instances only after we introduce a same type for the receiver objects. In practice, we first introduce a new local variable for new PrintVisitor() (resp. new PrettyPrintVisitor()), then change the type of that variable form PrintVisitor (resp. PrettyPrintVisitor) to Visitor, and then the extraction of the method successes (the two instances are replaced by invocations of that method). The operations used in IntelliJ IDEA are Introduce Variable and Type Migration (as many other refactoring operations Type Migration checks that the change is type safe). One would may also find useful to rename the local variables or the parameter of accept to v or visitor (operation Rename).

The local variables can be inlined afterward (operation Inline).

Note that the task of making accept act on Visitors is left implied in the guidelines of Mens and Tourwé (Fig 2). This task is not explained either by Fowler (Extract Method [Fow99]).

Again, we encapsulate these elementary changes in the ExtractGeneralMethod refactoring operation, defined in App. A.27.

**Pull Up.** Note that when accept is pulled up (step 4 of Fig. 5), IntelliJ IDEA does not add the @Override annotation to all the subclasses, but only in the one the operation is called on.

Also, when print and prettyprint are pulled up (step 5 of Fig. 5), the tool cannot take several classes simultaneously into account, so that the pull up does not verifies that the code are the same in all the concrete classes (in fact they are). Note that for Pull Up, Eclipse can take several classes into account (it allows to remove overriding methods in these classes) but it does not checks that the behavior is preserved by this change.

\[2\] Pull up method refactoring extension plugin: [http://plugins.intellij.net/plugin/?is=6889](http://plugins.intellij.net/plugin/?is=6889)
Visibility. In the example program, instance variables are public (package). If they were private or protected, we would have had to make them public so that the moved methods can access them. This does not depend on the way we implement the transformation, but rather to the nature of the Visitor pattern. Note that Eclipse Move makes the change automatically while with IntelliJ IDEA you have to do it after or before the Move.

Conclusion. We have seen that as soon as we consider a refactoring tool,

1. the guidelines have to be adapted and
2. an algorithm can be defined (and automated).

We have seen also that some steps are implied in the guidelines, and that, on the opposite, some chains of operations of the guidelines can be done with a single tool’s operation.

Finally, we have seen that we also have to adapt the chain of operation to characteristics of the initial program. In the following, after having studied a reverse transformation to get the program back to its initial structure, we will see how the algorithm is adapted to variations in the initial program.

3 Composite↔Visitor Transformation Scheme

We now consider an instance of the Composite pattern as the initial program (Fig. 6). The difference between the classic object structure considered before and the Composite structure is recursion: the data type is recursive (subclasses make references to the superclass) and the operations are recursive (to traverse trees of that datatype which depth in unknown).

In this section, all the business methods we handle take no parameter and do not return any result, and the traversal process is stateless. These constraints are relaxed in Sec. 4.

We also consider that the visitor classes are not part of the project in the Composite state (unlike in previous section).

3.1 Composite→Visitor Transformation

Let us consider this part in the code of the CompositeGraphic class:

```java
public void print() {
    System.out.print("Composite: " + this + " with: (";
    for (Graphic graphic : childGraphics) {
        graphic.print();
    }
    System.out.println(")");
}
```

If we apply the previous transformation algorithm (Fig. 5), after the operation AddParameterWithDelegate (step 1), we get the following (with IntelliJ IDEA):

```java
public void print() {
    print(new PrintVisitor());
}
```

```java
public void print(PrintVisitor v) {
    System.out.print("Composite: " + this + " with: (";
    for (Graphic graphic : childGraphics) {
        graphic.print();
    }
    System.out.println(")");
}
```

We observe that the recursive invocation to graphic.print() in the for loop has been left unchanged. The code is still functionally correct, but it is problematic for the following reason: if we look at the definition of Graphic.print() (in the program at that moment of the transformation, you cannot tell which instance of print() will be invoked because print() is abstract in the class Graphic, but we know that print(), as a delegator, will be pulled up to the class Graphic), we can see that each invocation of print() will result in the construction of a new PrintVisitor object.

Here, if possible, one would choose to use a single PrintVisitor object instead of creating useless new ones. In fact, there is a means to do this with the IntelliJ IDEA refactorer, but, in order to do that, the print() delegator method must be pulled up, which impacts the rest of the algorithm (for instance, the pull-up of step 1 is already done).

3 The trick is to first introduce an indirection (directly in the superclass), then inline the delegator invocation inside the loop, then add the parameter to the delegate, so that the tool is able to insert as new parameter in invocations existing objects instead of using a default value.
```java
abstract class Graphic {
    abstract public void print();
    abstract public void prettyprint();
}

class Ellipse extends Graphic {
    public void print() {
        System.out.println("Ellipse :" + this);
    }
    public void prettyprint() {
        System.out.println("Ellipse corresponding to the object" + this + ".");
    }
}

class CompositeGraphic extends Graphic {
    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void print() {
        System.out.println("Composite :");
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }
    public void prettyprint() {
        System.out.println("Composite " + this + " composed of:");
        for (Graphic graphic : mChildGraphics) {
            graphic.prettyprint();
        }
        System.out.println("(end of composite)");
    }
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}
```

Figure 6: Base Program (class hierarchy)
1. ForAll \( m \) in \( M \) do CreateEmptyClass\((V(m))\)
2. ForAll \( m \) in \( M \) do
   CreateIndirectionInSuperClass\((S, m, aux(m))\)
3. ForAll \( m \) in \( M \), \( c \) in \( C \) do
   InlineMethodInvocations\((c, m, aux(m))\)
4. ForAll \( m \) in \( M \) do
   AddParameterWithReuse\((S, aux(m), V(m))\)
5. ForAll \( m \) in \( M \), \( c \) in \( C \) do
   MoveMethodWithDelegate\((c, aux(m), V(m), \text{"visit"})\)
6. ExtractSuperClass\((V, \text{"Visitor"})\)
7. ForAll \( m \) in \( M \) do
   GeneraliseParameter\((S, \{ aux(m) \}_{m \in M}, \text{"Visitor"})\)
8. MergeDuplicateMethods\((S, \{ aux(m) \}_{m \in M}, \text{"accept"})\)

Figure 7: Base Composite\(\rightarrow\)Visitor transformation

This shows that, as soon as we rely on a refactoring tool, the chain of refactoring operations depends on the characteristics of the tool.

For this reason, here we cannot encapsulate the small change in the transformation into a variation of one of the steps of the algorithm, but we have to adapt the whole algorithm. Our algorithm for basic Composite\(\rightarrow\)Transformation is given in Fig. 7.

In Fig. 7 to generate temporary names, we consider a function \( aux \) that takes a method name and returns a method name. Here, \( aux(print) = \text{printAux} \) and \( aux(\text{prettyprint}) = \text{prettyprintAux} \).

Note that two bugs were encountered with IntelliJ IDEA at the beginning of our work, but were solved by JetBrains, so that no manual intervention is needed now.

The result of this transformation is given in Figs. 8 and 9.

3.2 Visitor\(\rightarrow\)Composite Transformation

Composite\(\rightarrow\)Visitor transformation is based on moving business code from the data-type class hierarchy to the visitor classes. Now we do the opposite (move business code from visitor classes to composite classes). We proceed with three coarse steps (Fig 10):

i. Replace dynamic dispatch with static dispatch.

ii. In-line the business code from the visitor structure to the composite structure.

iii. Make some small changes to get the initial Composite pattern structure back.

Remove Dynamic Dispatch (Fig. 10, steps 1 and 2). We replace the accept(Visitor) method by some overloaded methods accept, one for each subtype of Visitor. This removes all dynamic dispatch in visit method invocations, so that their invocations can be inlined afterward. The visit methods can also be removed from the Visitor class (but not from the concrete visitor classes before they are inlined).

The result of this is given in Figs. 11 and 12.

Move Business Code (Fig. 10, step 3). The business code in the visitor classes is inlined: invocations of the visit methods in the composite classes are replaced by the corresponding body (the business code) and the visit methods are deleted.

The result of this step is given in Fig. 13 (visitor classes are empty).

\[^4\text{Of course, we should ensure that these names are not clashing with other names in the project.}\]
abstract class Graphic {
    public void print() {
        accept(new PrintVisitor());
    }
    public void prettyprint() {
        accept(new PrettyPrintVisitor());
    }
    public abstract void accept(Visitor v);
}

class Ellipse extends Graphic{
    public void accept(Visitor v) {
        v.visit(this);
    }
}

class CompositeGraphic extends Graphic {
    ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void accept(Visitor v) {
        v.visit(this);
    }
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}

Figure 8: Program with Visitor (data classes)
public abstract class Visitor {
    public abstract void visit(Ellipse ellipse);
    public abstract void visit(CompositeGraphic compositeGraphic);
}

public class PrintVisitor extends Visitor {
    public void visit(CompositeGraphic compositeGraphic) {
        System.out.println("Composite:");
        for (Graphic graphic : compositeGraphic.mChildGraphics) {
            graphic.accept(this);
        }
    }
    public void visit(Ellipse ellipse) {
        System.out.println("Ellipse:");
    }
}

public class PrettyPrintVisitor extends Visitor {
    public void visit(CompositeGraphic compositeGraphic) {
        System.out.println("Composite " + compositeGraphic + " composed of:");
        for (Graphic graphic : compositeGraphic.mChildGraphics) {
            graphic.accept(this);
        }
        System.out.println("(end of composite)");
    }
    public void visit(Ellipse ellipse) {
        System.out.println("Ellipse corresponding to the object ":
    }
}

Figure 9: Program with Visitor (visitor classes)

Figure 10: Base Visitor → Composite transformation

1. ForAll v in $V$ do
   addSpecializedMethodInHierarchy(S,"accept","Visitor",v)
   deleteMethodInHierarchy(S,accept,"Visitor")
2. ForAll c in $C$ do pushDownAll("Visitor","visit",c)
3. ForAll v in $V$, c in $C$ do InlineMethod(v,"visit",c)
4. ForAll m in $M$ do
   renameMethod(S,accept,V(m),aux(m))
5. ForAll m in $M$ do removeParameter(S,aux(m),V(m))
6. ForAll m in $M$ do replaceMethodDuplication(S,m)
7. ForAll m in $M$ do pushDownImplementation(S,m)
8. ForAll m in $M$ do pushDownAll(S,aux(m))
9. ForAll m in $M$, c in $C$ do inlineMethod(c,aux(m))
10. ForAll v in $V$ do deleteClass(v)
11. deleteClass("Visitor")

Figure 10: Base Visitor → Composite transformation
abstract class Graphic {
    public void print() {
        accept(new PrintVisitor());
    }

    public void prettyprint() {
        accept(new PrettyPrintVisitor());
    }

    public abstract void accept(PrintVisitor v);
    public abstract void accept(PrettyPrintVisitor v);
}

class Ellipse extends Graphic {
    public void accept(PrettyPrintVisitor v) {
        v.visit(this);
    }

    public void accept(PrintVisitor v) {
        v.visit(this);
    }
}

class CompositeGraphic extends Graphic {
    ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();

    public void accept(PrettyPrintVisitor v) {
        v.visit(this);
    }

    public void accept(PrintVisitor v) {
        v.visit(this);
    }

    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }

    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}

Figure 11: Reverse-State 1 (data classes)
public abstract class Visitor {
    
}

public class PrintVisitor extends Visitor {
    public void visit(CompositeGraphic compositeGraphic) {
        System.out.println("Composite:");
        for (Graphic graphic : compositeGraphic.mChildGraphics) {
            graphic.accept(this);
        }
    }
    public void visit(Ellipse ellipse) {
        System.out.println("Ellipse :" + ellipse);
    }
}

public class PrettyPrintVisitor extends Visitor {
    public void visit(CompositeGraphic compositeGraphic) {
        System.out.println("Composite " + compositeGraphic + " composed of:");
        for (Graphic graphic : compositeGraphic.mChildGraphics) {
            graphic.accept(this);
            System.out.println("(end of composite) ");
        }
    }
    public void visit(Ellipse ellipse) {
        System.out.println("Ellipse corresponding to the object " + ellipse + ".");
    }
}

Figure 12: Reverse-State 1 (visitor classes)

Remove Visitors and Recover Initial Structure (Fig. 10 steps 4 to 11). Once the business code has been moved into the convenient classes, the rest of the refactoring operations are common refactoring operations allowing to recover the composite structure (the important part is done before).

The result of this step is given in Fig. 14.

3.3 Result after Round Trip Transformation

After this transformation, the program conforms to the Composite pattern (Fig. 13).

A few more refactorings are necessary to recover exactly the original program: make private the fields that were made public during the Composite→Transformation, reorder method definitions.

Note also that some comments are altered or lost during the transformation (which is not shown by our example).
Figure 13: Reverse-State 2 (data classes)
abstract class Graphic {
    public abstract void print();
    public abstract void prettyprint();
}

class Ellipse extends Graphic {
    public void print() {
        System.out.println("Ellipse :");
    }
    public void prettyprint() {
        System.out.println("Ellipse corresponding to the object ");
    }
}

class CompositeGraphic extends Graphic {
    ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
    public void print() {
        System.out.println("Composite:");
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }
    public void prettyprint() {
        System.out.println("Composite ");
        for (Graphic graphic : mChildGraphics) {
            graphic.prettyprint();
        }
        System.out.println("(end of composite)");
    }
}

Figure 14: Result after Back Transformations
4 Variants of Transformations for Various Pattern Instances

In this section we consider some variants of either Composite pattern or Visitor pattern and we adapt the algorithm of transformation.

4.1 Methods with Parameters

In this section we consider that methods of interest have parameters. We consider a method `setColor` with an integer as parameter in the Composite hierarchy (see Fig. 15).

```java
abstract class Graphic {
    public abstract void print();
    public abstract void setColor(int c);
}

class Ellipse extends Graphic{
    protected int color;
    public void print() {
        System.out.println("Ellipse with color:"+ color);
    }
    public void setColor(int c){
        this.color = c;
    }
}

class CompositeGraphic extends Graphic{
    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void print() {
        System.out.println("Composite:");
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }
    public void setColor(int c){
        for (Graphic graphic : mChildGraphics) {
            graphic.setColor(c);
        }
    }
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}
```

Figure 15: Composite with methods having parameters.

4.1.1 Composite→Visitor Transformation

At the step 4 of the Composite→Visitor algorithm of Fig. 14 replace the application of the operation `addParameter-WithReuse` with parameters by the operation `Introduce Parameter Object`. This operation is offered by refactoring tools (Eclipse and IntelliJ IDEA). If we consider a method `m(A a, B b, C c)`, it replaces the parameters by an object of a new class `P` it creates and which contains instances variables `a`, `b` and `c`. Invocations `m(a,b,c)` are replaced by `m(new P(a,b,c)).`
abstract class Graphic {
    public void print() {
        accept(new PrintVisitor());
    }
    public void setColor(int c) {
        accept(new SetColorVisitor(c));
    }
    public abstract void accept(Visitor v);
}

class Ellipse extends Graphic {
    protected int color;
    void accept(Visitor v) {
        v.visit(this);
    }
}

class CompositeGraphic extends Graphic {
    ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void accept(Visitor v) {
        v.visit(this);
    }
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
    ArrayList<Graphic> getChildGraphics() {
        return mChildGraphics;
    }
}

Figure 16: Visitor with methods having parameters (data classes)

4.1.2 Visitor Program

The result program is shown by the figures 16 and 17.

4.1.3 Visitor → Composite Transformation

After performing the step 9 of the basic algorithm (Fig. 10) perform the following tasks:

1. Delete the superclass Visitor structure.

2. InlineClass(v) (for each visitor class v that corresponds to the business method with parameter).

After having performed the rest of the transformation of Fig. 10, we get the following code in the Ellipse class:

```java
public void setColor(int c) {
    final int c1 = c;
    color = new Object() {
        private final int c = c1;
        public int getC() {
            return c;
        }
    }.getC();
}
```

Here, we have to replace new Object(){...c1...}.getC() by c1. The reason is that we have replaced a parameter by an object containing the parameter with Add Object Parameter during the Composite→Transformation, and now we have to do the inverse, extract a component from an object. The same has to be done in CompositeGraphic. At the moment, we do this manually.

4.2 Methods Returning Values

In this section we consider that methods of interest return results, for instance we consider a method perimeter that returns an Integer and toString that returns a String (see Fig. 18).
This would require to have one accept method for each return type. To avoid this, we use generic types (see the visitor structure in Figs. 19 and 20). Note that the use of Java generic types is only possible with object types (such as `Integer`), not with raw types (such as `int`).

4.2.1 Composite → Visitor Transformation

In the step 6 of the base algorithm of Fig. 7, we use the operation `ExtractSuperClassWithoutPullUp` (see App. A.9.1) and then the operation `PullUpWithGenerics` (see App. A.16). The last operation is used to pull up methods having different return types and create the right generic types.

4.2.2 Visitor Program

The result program is shown by the figures 19 and 20.

4.2.3 Visitor → Composite Transformation

At the step 11 of the base algorithm of the Fig. 10, we must specify the return type of each `accept` method and replace the parameter type by the corresponding type (the operation `addSpecializedMethodInHierarchy` must change the return type in addition to the parameter type).

4.3 Interface instead of Abstract class in the Composite structure

In this section we consider that the top of the Composite hierarchy in an interface instead of abstract class (see Fig. 21). As we have to put some code in the superclass, we just introduce an abstract class in the hierarchy.

4.3.1 Composite → Visitor Transformation

Before performing the base algorithm of Fig. 7 create an abstract class that implements the interface of the Composite hierarchy. This is done as the following:

1. Extract a super-Class from composite classes.
2. Pull up the business methods as abstract methods to the super-Class.
3. Make the super-class implementing the interface of the Composite structure.
4. Change any use of type Interface to type super-Class (use `Type Migration` in IntelliJ IDEA to perform this operation).
5. Push down business method from the interface of composite structure (this will help to avoid any confusion or complexity of manipulating business methods in the abstract class).
Figure 18: Composite with methods returning types
abstract class Graphic{
    public Integer perimeter() {
        return accept(new PerimeterVisitor());
    }
    public String toString() {
        return accept(new ToStringVisitor());
    }
    public abstract <T> T accept(Visitor<T> v);
}

class Ellipse extends Graphic{
    int perimeter;
    public Ellipse (int perimeter){
        this.perimeter=perimeter ;
    }
    public <T> T accept(Visitor<T> v) {
        return v.visit(this);
    }
}

class CompositeGraphic extends Graphic {
    ArrayList<Graphic> mChildGraphics =
    new ArrayList<Graphic>();
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
    ArrayList<Graphic> getmChildGraphics () {
        return mChildGraphics;
    }
    public <T> T accept(Visitor<T> v) {
        return v.visit(this);
    }
}

Figure 19: Visitor with generics (data classes)

public class TotalPerimeterVisitor extends Visitor <Integer>{
    public Integer visit(CompositeGraphic compositeGraphic) {
        int acc = 0 ;
        for (Graphic graphic : compositeGraphic.mChildGraphics) {
            acc += graphic.accept(this);
        }
        return acc;
    }
    public Integer visit(Ellipse ellipse) {
        return (ellipse.perimeter);
    }
}

public class ToStringVisitor extends Visitor <String> {
    public String visit(CompositeGraphic compositeGraphic) {
        String s =
        new String ("Composite with : ");
        for (Graphic graphic : compositeGraphic.mChildGraphics) {
            s = s.concat(graphic.accept(this) + ", ");
        }
        System.out.println("(end)"");
        return s;
    }
    public String visit(Ellipse ellipse) {
        return ("Ellipse : " + Integer.toString(ellipse.perimeter));
    }
}

Figure 20: Visitor with generics (visitor classes)
interface Graphic {
    abstract public void print();
    abstract public void prettyprint();
}

class Ellipse implements Graphic{
    public void print() {
        System.out.println("Ellipse :" + this);
    }
    public void prettyprint(){
        System.out.println("Ellipse corresponding to the object " + this + ".");
    }
}

class CompositeGraphic implements Graphic {
    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void print() {
        System.out.println("Composite : ");
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }
    public void prettyprint(){
        System.out.println("Composite " + this + " composed of:");
        for (Graphic graphic : mChildGraphics) {
            graphic.prettyprint();
        }
        System.out.println(" (end of composite) ");
    }
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}

Figure 21: Composite with Interface
interface Graphic {
    void print();
    void prettyprint();
}

public abstract class AbstractComposite implements Graphic {
    public void print() {
        accept(new PrintVisitor());
    }
    public void prettyprint() {
        accept(new PrettyPrintVisitor());
    }
    public abstract void accept(Visitor v);
}

class Ellipse extends AbstractComposite implements Graphic{
    public void accept(Visitor v){
        v.visit(this);
    }
}

class CompositeGraphic extends AbstractComposite implements Graphic {
    ArrayList<AbstractComposite> mChildGraphics = new ArrayList<AbstractComposite>();
    public void accept(Visitor v){
        v.visit(this);
    }
    public void add(AbstractComposite graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(AbstractComposite graphic) {
        mChildGraphics.remove(graphic);
    }
    ArrayList<AbstractComposite> getChildGraphics () {
        return mChildGraphics;
    }
}

After doing the previous tasks, perform the base algorithm to reach the Visitor structure.

4.3.2 Visitor Program
The result program is shown by the figures 22 and 23.

4.3.3 Visitor→Composite Transformation
After performing the basic algorithm of back transformation (Fig. 10), change any use of the super-class type to the interface type (Type Migration in IntelliJ IDEA). Finally, delete the intermediate super-class.

4.4 Class Hierarchies with Several Levels
In this section we consider that the Composite class hierarchy has several levels (we add a subclass ColoredEllipse to Ellipse, which provides an overriding method for only one of the two business methods, see Fig. 24). The interest of this variant is that a subclass extends a composite and does not redefine all business methods. This subclass exists in different depth of hierarchy as the main composite class.

4.4.1 Composite→Visitor Transformation
Before performing the base algorithm of Fig. 21, apply the operation pushDownNotRedefinedMethod (see App. A.24) in order to push down the methods that exists in the composites but not redefined in the sub-classes. After that, the basic algorithm applies.
public class PrintVisitor extends Visitor{
    public void visit(CompositeGraphic compositeGraphic) {
        System.out.println("Composite:");
        for (AbstractCompositeGraphic : compositeGraphic.mChildGraphics) {
            graphic.accept(this);
        }
    }
    public void visit(Ellipse ellipse) {
        System.out.println("Ellipse:");
    }
}

public class PrettyPrintVisitor extends Visitor {
    public void visit(CompositeGraphic compositeGraphic) {
        System.out.println("Composite " + compositeGraphic + " composed of:");
        for (AbstractCompositeGraphic : compositeGraphic.mChildGraphics) {
            System.out.println("(end of composite) ");
            System.out.println("(end of composite) ");
        }
    }
    public void visit(Ellipse ellipse) {
        System.out.println("Ellipse corresponding to the object " + ellipse + ".");
    }
}

Figure 23: Visitor structure for Interface instead of Abstract Composite (visitor classes)

4.4.2 Visitor Program
The result program is shown by the figures 25 and 26.

4.4.3 Visitor→Composite Transformation
At the step 8 of the basic algorithm (see Fig. 10) use the operation pushDownNotRedefinedMethod in order to push down the auxiliary methods that exist in composites and are not redefined in their sub-classes.
abstract class Graphic {
    abstract public void print();
    abstract public void prettyprint();
}

class Ellipse extends Graphic {
    public void print() {
        System.out.println("Ellipse :" + this);
    }
    public void prettyprint() {
        System.out.println("Ellipse corresponding to the object " + this + ".");
    }
}

class ColoredEllipse extends Ellipse {
    int color;
    public void print() {
        System.out.println("Ellipse :" + color);
    }
}

class CompositeGraphic extends Graphic {
    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void print() {
        System.out.println("Composite: ");
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }
    public void prettyprint() {
        System.out.println("Composite " + this + " composed of:");
        for (Graphic graphic : mChildGraphics) {
            graphic.prettyprint();
        }
        System.out.println("(end of composite)");
    }
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}

Figure 24: Composite with multiple hierarchical levels.
abstract class Graphic {
    public void print() {
        accept(new PrintVisitor());
    }
    public void prettyprint() {
        accept(new PrettyPrintVisitor());
    }
    public abstract void accept(Visitor v);
}

class Ellipse extends Graphic{
    public void accept(Visitor v) {
        v.visit(this);
    }
}

class ColoredEllipse extends Ellipse{
    int color;
    public void accept(Visitor v) {
        v.visit(this);
    }
}

class CompositeGraphic extends Graphic {
    ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
    ArrayList<Graphic> getmChildGraphics() {
        return mChildGraphics;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

Figure 25: Visitor with multiple hierarchical levels (data classes)


```java
public class PrintVisitor extends Visitor {
    public void visit(CompositeGraphic compositeGraphic) {
        System.out.println("Composite: ");
        for (Graphic graphic : compositeGraphic.mChildGraphics) {
            graphic.accept(this);
        }
    }
    public void visit(Ellipse ellipse) {
        System.out.println("Ellipse : ");
    }
    public void visit(ColorEllipse coloredEllipse) {
        System.out.println("Ellipse : ");
    }
}
```

```java
public class PrettyPrintVisitor extends Visitor {
    public void visit(CompositeGraphic compositeGraphic) {
        System.out.println("Composite "+ compositeGraphic + " composed of: ");
        for (Graphic graphic : compositeGraphic.mChildGraphics) {
            graphic.accept(this);
        }
        System.out.println("(end of composite) ");
    }
    public void visit(Ellipse ellipse) {
        System.out.println("Ellipse corresponding to the object ");
    }
    public void visit(ColorEllipse coloredEllipse) {
        System.out.println("Ellipse : ");
    }
}
```

Figure 26: Visitor with multiple hierarchical levels (visitor classes)

5 Application to JHotDraw

To validate our transformation algorithms, we apply them to JHotDraw [GI].

To know on which classes to apply the transformation, we use a pattern detection tool. We have applied pattern4 [TCSH06]: it reports a Composite structure with 6 operations and it reports the superclass and the subclass that implements the “container”. The operations are defined by overriding methods in about 12 classes of the class hierarchy.

The semi-automatic Composite → Visitor transformation applies successfully with the help of variations studied in Sec. H (and with the limitations explained in these variations).

It takes between 8 and 9 hours to apply the whole Composite → Visitor transformation interactively. Most of time is due to interaction (selection the entities to transform, selecting the refactoring operation and giving parameters). The automated version takes a few minutes. We are focusing on automating the whole reversible transformation on JhotDraw.

A second instance of the pattern is found but we have not transformed it since it has only one operation defined.

6 Related work

6.1 Refactoring to Patterns

Using chains of elementary refactoring operations to introduce design patterns into programs is not new. The idea is first proposed by Batory and Takuda [BT95].

Ó Cinnéide [OC00] give transformation to introduce several patterns but not the Visitor (he considers in [OC00] that automating the introduction of a visitor pattern is impractical).

Kerievsky [Ker04] proposes two sets of guidelines to introduce Visitor patterns. The first one is similar to the one from Mens and Tourwé [MT04] described in Sec. 2. The second one applies to an “external accumulation”: instead of transforming an operation defined by overriding methods in the class hierarchy, it applies to an operation defined outside of the class hierarchy by a switch on the type of an object with instanceof and type casts. Neither Mens and Tourwé [MT04] nor Kerievsky [Ker04] give the inverse transformation.

Hills et al. [HKVDSV11] have transformed a program based on a Visitor pattern to introduce a Visitor pattern instead (the Visitor pattern is similar to the Composite pattern). Their transformation is automated, with a few interactions
with the user. Since their transformation is dedicated to a specific program and is not abstractly described, it requires some work to be applied to other programs.

Jeaon et al. [JLB02] provide automatic inference of sequence of refactoring operations allowing to reach design pattern based versions of programs. Sudan et al. [PRSK10] provide an inference of a sequence of refactoring operations allowing to pass from a given version of a program to a second given version. Such tools could be used to infer variations of our transformation algorithms for variations in initial programs, or to infer transformations between other patterns.

6.2 Building Complex Refactoring Operations

The transformations we aim at can be seen as complex/composed refactoring operations. As each refactoring operation has specific preconditions, and as we use a large number of elementary transformations, assistance for building such transformations would be valuable. Several works provide languages to build or compose refactoring operations. Ó Cinnéide and Nixon [OCN00] show how to compose elementary refactoring operations with pre/post-conditions, as well as Kniesel and Koch [KK04].

Verbaere et al. propose a language dedicated to building refactoring operations [VEdM06], and Klint et al. propose a language dedicated to program manipulation [KSV09], which they have used to build the Visitor→Interpreter transformation.

6.3 Design Patterns Discovery

To provide a fully automated transformation, detection of the occurrences of the initial design pattern must be automated. Several work exist in that domain. Smith and Scott provide a tool that discovers variants of a design pattern in a given program [SS03]. Such tools can be used to automatically provide inputs to our transformations.

On the opposite, some tools detect pattern precursors, anti-patterns or code smells [RJ04, MGL06], but, in our approach, we consider that the initial program has already a good design.

7 Conclusion

In this report:

- We have shown how to use refactoring operations to transform a Java program conforming to the Composite pattern (or Interpreter pattern) into a program (still in Java) conforming to the Visitor pattern and vice versa.
- We have explained how to use some refactoring tools (IntelliJ IDEA and Eclipse) to perform these transformations. We have seen that some basic refactorings are not supported by these tools.
- We have discussed some variations in transformations to adapt to variations in the initial programs.

This work is a first step towards automation of these transformations so that the user does not have to perform each basic refactoring with a refactoring tool. On the example of the JHotDraw program, automation can reduce transformation time from 8 hours to a few minutes. This kind of automated transformation can be used to provide different versions of a same programs with different properties with respect to modularity [CDA12].

References

[BT95] Don Batory and Lance Tokuda. Automated software evolution via design pattern transformations. Technical report, University of Texas at Austin, Austin, TX, USA, 1995.

[CA13] Julien Cohen and Akram Ajouli. Practical use of static composition of refactoring operations. In ACM Symposium On Applied Computing, Portugal, March 2013.

[CDA12] Julien Cohen, Rémi Douence, and Akram Ajouli. Invertible program restructurings for continuing modular maintenance. In Soft. Maintenance and Reengineering (CSMR), 16th European Conf. on, pages 347–352, 2012.

[Fow99] Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GI] Erich Gamma and IFA Informatik. JHotDraw as Open-Source Project. http://www.jhotdraw.org/
In this appendix, we define refactoring operations we use in our transformations. For each operation, we give an example to explain its behavior, and we tell how it is performed with IntelliJ IDEA or Eclipse. We give some preconditions when an operation applies only in a specific case. These preconditions are neither minimal (they can be refined into weaker conditions) nor complete (they are sufficient in our basic examples, but not in some situations we have not considered). Also, some preconditions dealing with name clashes are left implied.

The given backward descriptions are used for static composition of refactorings [KK04] (see App. B).
A.1 CreateEmptyClass

Overview: CreateEmptyClass (classname c): this operation is used to add a new class c.

Refactoring tools. new Class in Eclipse and IntelliJ IDEA.

Precondition. ¬existsType(c)

Backward Description.
ExistsClass(c) ⊃ ⊤
ExistsType(c) ⊃ ⊤
IsAbstractClass(c) ⊃ ⊥
ExistsMethodDefinition(c, Y) ⊃ ⊥
ExistsMethodDefinitionWithParams(c, Y, []) ⊃ ⊥
ExistsMethodDefinitionWithParams(c, Y, [T1]) ⊃ ⊥
ExistsMethodDefinitionWithParams(c, Y, [T1; T2]) ⊃ ⊥
ExistsMethodDefinitionWithParams(c, Y, [T1; T2; T3]) ⊃ ⊥
ExistsMethodDefinitionWithParams(c, Y, [T1; T2; T3; T4]) ⊃ ⊥
ExistsMethodDefinitionWithParams(c, Y, [T1; T2; T3; T4; T5]) ⊃ ⊥
IsInheritedMethod(c, Y) ⊃ IsVisibleMethod(java.lang.Object, Y, c)
IsInheritedMethodWithParams(c, Y, []) ⊃ IsVisibleMethod(java.lang.Object, Y, [], c)
IsInheritedMethodWithParams(c, Y, [T1]) ⊃ IsVisibleMethod(java.lang.Object, Y, [T1], c)
IsInheritedMethodWithParams(c, Y, [T1; T2]) ⊃ IsVisibleMethod(java.lang.Object, Y, [T1; T2], c)
IsInheritedMethodWithParams(c, Y, [T1; T2; T3]) ⊃ IsVisibleMethod(java.lang.Object, Y, [T1; T2; T3], c)
IsInheritedMethodWithParams(c, Y, [T1; T2; T3; T4]) ⊃ IsVisibleMethod(java.lang.Object, Y, [T1; T2; T3; T4], c)
IsInheritedMethodWithParams(c, Y, [T1; T2; T3; T4; T5]) ⊃ IsVisibleMethod(java.lang.Object, Y, [T1; T2; T3; T4; T5], c)
IsSubType(c, X) ⊃ ⊥ (X ≠ java.lang.Object)
ExtendsDirectly(c, X) ⊃ ⊥ (X ≠ java.lang.Object)
ExistsMethodDefinitionWithParams(B, Y, [c; T1]) ⊃ ⊥
ExistsMethodDefinitionWithParams(B, Y, [T1; c]) ⊃ ⊥
ExistsMethodDefinitionWithParams(B, Y, [T1; T2; c]) ⊃ ⊥
ExistsMethodDefinitionWithParams(B, Y, [T1; c; T2]) ⊃ ⊥
ExistsMethodDefinitionWithParamsWithParams(B, Y, [c; T1]) ⊃ ⊥
ExistsMethodDefinitionWithParamsWithParams(B, Y, [c; T1; T2]) ⊃ ⊥
ExistsMethodDefinitionWithParamsWithParams(B, Y, [c; T1; T2; c]) ⊃ ⊥
ExistsParameterWithName(B, Y, [c], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [c; T1], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [T1; c], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [T1; c; T2], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [T1; T2; c], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [c; T1; T2], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [c; T1], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [T1; c], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [T1; c; T2], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [T1; T2; c], P) ⊃ ⊥
ExistsParameterWithName(B, Y, [c; T1; T2], P) ⊃ ⊥
IsUsedMethodIn(c, B, Y) \mapsto \bot
IsUsedMethod(c, B, [T1]) \mapsto \bot
IsUsedMethod(c, B, [T1; T2]) \mapsto \bot
IsUsedMethod(c, B, [T1; T2; T3]) \mapsto \bot
IsUsedMethod(c, B, [T1; T2; T3; T4]) \mapsto \bot
IsUsedConstructorAsMethodParameter(c, B, Y) \mapsto \bot
IsUsedConstructorAsInitializer(c, B, Y) \mapsto \bot
IsUsedConstructorAsObjectReceiver(c, B, Y) \mapsto \bot
IsUsedConstructorAsMethodParameter(B, c, Y) \mapsto \bot
IsUsedConstructorAsInitializer(B, c, Y) \mapsto \bot
IsUsedConstructorAsObjectReceiver(B, c, Y) \mapsto \bot
IsSubType(B, c) \mapsto \bot
ExtendsDirectly(B, c) \mapsto \bot
MethodIsUsedWithMethod(B, Y, [P], [c]) \mapsto \bot
MethodIsUsedWithMethod(B, Y, [P], [c; T1]) \mapsto \bot
MethodIsUsedWithMethod(B, Y, [P], [T1; c]) \mapsto \bot
MethodIsUsedWithMethod(B, Y, [P], [T1; c; T2]) \mapsto \bot
MethodIsUsedWithMethod(B, Y, [P], [T1; T2; c]) \mapsto \bot
AllInvokedMethodsOnObjectOfMareDeclaredInC(c, T1, T2, T3) \mapsto \bot
AllInvokedMethodsWithParameterOnObjectOfMareNotOverloaded(c, T1, T2) \mapsto \bot
BoundVariableInMethodBody(c, T1, T2) \mapsto \bot
ExistsField(c, F) \mapsto \bot
ExistsMethodInvocation(c, Y, T1, X) \mapsto \bot
ExistsAbstractMethod(c, Y) \mapsto \bot
IsIndirectlyRecursive(c, Y) \mapsto \bot
IsVisibleMethod(c, Y, [T1], B) \mapsto \bot
IsVisibleMethod(c, Y, [T1; T2], B) \mapsto \bot
IsVisibleMethod(c, Y, [T1; T2; T3], B) \mapsto \bot
IsVisibleMethod(B, Y, [B], c) \mapsto \bot
IsVisibleMethod(B, Y, [B; T1], c) \mapsto \bot
IsVisibleMethod(B, Y, [T1; B], c) \mapsto \bot
IsVisibleMethod(B, Y, [T1; T2; B], c) \mapsto \bot
IsVisibleMethod(B, Y, [T1; B; T2], c) \mapsto \bot
IsVisibleMethod(B, Y, [B; T1; T2], c) \mapsto \bot
IsInverter(c, Y, T1, T2) \mapsto \bot
IsDelegator(c, Y, X) \mapsto \bot
IsAbstractClass(c) \mapsto \bot
IsUsedMethodIn(c, Y, B) \mapsto \bot
IsUsedMethodIn(B, Y, c) \mapsto \bot
IsPrimitiveType(c) \mapsto \bot
IsPublic(c, Y) \mapsto \bot
IsProtected(c, Y) \mapsto \bot
IsPrivate(c, Y) \mapsto \bot
IsUsedAttributeInMethodBody(c, X, Y) \mapsto \bot
IsGenericsSubtype(c, [T1], B, [T2]) \mapsto \bot
IsGenericsSubtype(c, [T1; T2], B, [T4; T3]) \mapsto \bot
IsGenericsSubtype(c, [T1; T2; T3], B, [T4; T5; T6]) \mapsto \bot
IsInheritedField(c, F) \mapsto \bot
IsOverridden(c, Y) \mapsto \bot
IsOverloaded(c, Y) \mapsto \bot
IsOverriding(c, Y) \mapsto \bot
IsRecursiveMethod(c, Y) \mapsto \bot
IsRecursiveMethod(c, Y) \mapsto \bot
HasReturnType(c, Y, T_1) \mapsto \bot
HasParameterType(c, T_1) \mapsto \bot
HasParameterType(B, c) \mapsto \bot
MethodHasParameterType(c, Y, T_1) \mapsto \bot
AllSubclasses(c, \{C_1; C_2; C_3\}) \mapsto \bot
ExtendsDirectly(c, java.lang.Object) \mapsto \top

A.2 CreateIndirectionInSuperClass

Overview: CreateIndirectionInSuperclass (classname s, subclasses \{a,b\}, methodname m, types \{t,t'\}, returntype q, newname n) : this operation is used to create an indirection of the method s::m to the method n in all the hierarchy.

Refactoring tools. With IntelliJ IDEA:

- Use Change Signature on the method m in class s (select “delegate via overloading method”, specify the new name n, specify the desired visibility).

With Eclipse:

- Use Change Method Signature on the method m in class s (specify to “keep original method as delegate to changed method”, and specify the new name n).

Precondition.

\[
(\text{ExistsClass}(s) \\
\land \text{IsAbstractClass}(s) \\
\land \text{ExistsMethodDefinitionWithParams}(s, m, \{t,t'\}) \\
\land \text{ExistsAbstractMethod}(s, m) \\
\land \neg \text{IsInheritedMethod}(s, n) \\
\land \neg \text{IsInheritedMethodWithParams}(s, n, \{t,t'\}) \\
\land \neg \text{ExistsMethodDefinitionWithParams}(s, n, \{t,t'\}) \\
\land \text{AllSubclasses}(s, \{a; b\}) \\
\land \text{HasReturnType}(s, m, q) \\
\land \neg \text{IsPrivate}(s, m) \\
\land \neg \text{IsPrivate}(a, m) \\
\land \neg \text{IsPrivate}(b, m) \\
\land \text{ExistsMethodDefinition}(s, m) \\
\land \text{ExistsMethodDefinition}(a, m) \\
\land \text{ExistsMethodDefinition}(b, m) \\
\land \neg \text{ExistsMethodDefinition}(s, n) \\
\land \neg \text{ExistsMethodDefinition}(a, n) \\
\land \neg \text{ExistsMethodDefinition}(b, n))
\]
Backward Description.

ExistsAbstractMethod(s, n) \rightarrow \top

IsIndirectlyRecursive

IsIndirectlyRecursive

ExistsMethodDefinitionWithParams(s, n, [t; t']) \rightarrow \bot

IsDelegator(s, m, n) \rightarrow \top

IsInheritedMethodWithParams(s, n, [t; t']) \rightarrow \bot

HasReturnType(s, n, q) \rightarrow HasReturnType(s, m, q)

HasReturnType(a, n, q) \rightarrow HasReturnType(s, m, q)

HasReturnType(b, n, q) \rightarrow HasReturnType(s, m, q)

ExistsMethodDefinition(s, n) \rightarrow ExistsMethodDefinition(s, m)

ExistsMethodDefinition(a, n) \rightarrow ExistsMethodDefinition(a, m)

ExistsMethodDefinition(b, n) \rightarrow ExistsMethodDefinition(b, m)

ExistsMethodDefinitionWithParams(s, n, [t; t']) \rightarrow \top

ExistsMethodDefinitionWithParams(a, n, [t; t']) \rightarrow \top

ExistsMethodDefinitionWithParams(b, n, [t; t']) \rightarrow \top

ExistsMethodDefinitionWithParams(s, n, [P1]) \rightarrow \bot

ExistsMethodDefinitionWithParams(a, n, [P1]) \rightarrow \bot

ExistsMethodDefinitionWithParams(b, n, [P1]) \rightarrow \bot

ExistsMethodDefinitionWithParams(s, n, [P1; P2]) \rightarrow \bot

ExistsMethodDefinitionWithParams(a, n, [P1; P2]) \rightarrow \bot

ExistsMethodDefinitionWithParams(b, n, [P1; P2]) \rightarrow \bot

ExistsMethodDefinitionWithParams(s, n, [P1; P2; P3]) \rightarrow \bot

ExistsMethodDefinitionWithParams(a, n, [P1; P2; P3]) \rightarrow \bot

ExistsMethodDefinitionWithParams(b, n, [P1; P2; P3]) \rightarrow \bot

ExistsMethodDefinitionWithParams(b, n, [P1; P2; P3; P4]) \rightarrow \bot

ExistsMethodDefinitionWithParams(b, n, [P1; P2; P3; P4]) \rightarrow \bot

ExistsParameterWithName(s, n, [t; t'], N) \rightarrow \bot

ExistsParameterWithName(a, n, [t; t'], N) \rightarrow \bot

ExistsParameterWithName(b, n, [t; t'], N) \rightarrow \bot

ExistsParameterWithName(s, n, [P1], N) \rightarrow ExistsParameterWithName(s, m, [P1], N)

ExistsParameterWithName(a, n, [P1], N) \rightarrow ExistsParameterWithName(a, m, [P1], N)

ExistsParameterWithName(b, n, [P1], N) \rightarrow ExistsParameterWithName(b, m, [P1], N)

ExistsMethodDefinition(a, m) \rightarrow \bot

ExistsMethodDefinition(b, m) \rightarrow \bot

ExistsMethodDefinitionWithParams(a, m, [t; t']) \rightarrow \bot

ExistsMethodDefinitionWithParams(b, m, [t; t']) \rightarrow \bot

ExistsMethodDefinitionWithParams(a, m, [P1]) \rightarrow \bot

ExistsMethodDefinitionWithParams(b, m, [P1]) \rightarrow \bot

IsIndirectlyRecursive(a, n) \rightarrow IsRecursiveMethod(a, m)

IsIndirectlyRecursive(b, n) \rightarrow IsRecursiveMethod(b, m)

IsOverridden(s, n) \rightarrow \bot

IsOverloaded(a, n) \rightarrow \bot

IsOverloaded(b, n) \rightarrow \bot

IsOverridden(s, n) \rightarrow \bot

IsOverridden(a, n) \rightarrow IsOverridden(a, m)

IsOverridden(b, n) \rightarrow IsOverridden(b, m)

IsOverridden(s, n) \rightarrow \bot

IsOverloaded(a, n) \rightarrow \bot

IsOverloaded(b, n) \rightarrow \bot
IsOverriding\( (a, n) \rightarrow \) IsOverriding\( (a, m) \)
IsOverriding\( (b, n) \rightarrow \) IsOverriding\( (b, m) \)
IsRecursiveMethod\( (s, n) \rightarrow \bot \)
IsRecursiveMethod\( (a, n) \rightarrow \bot \)
IsRecursiveMethod\( (b, n) \rightarrow \bot \)
AllInvokedMethodsWithParameter\( O \)InBodyOf\( M \)AreNotOverloaded\( (s, n, V) \rightarrow \)
AllInvokedMethodsWithParameter\( O \)InBodyOf\( M \)AreNotOverloaded\( (a, n, V) \rightarrow \)
AllInvokedMethodsWithParameter\( O \)InBodyOf\( M \)AreNotOverloaded\( (b, n, V) \rightarrow \)
AllInvokedMethodsWithParameter\( O \)InBodyOf\( M \)AreNotOverloaded\( (s, n, N) \rightarrow \)
AllInvokedMethodsWithParameter\( O \)InBodyOf\( M \)AreNotOverloaded\( (a, n, N) \rightarrow \)
AllInvokedMethodsWithParameter\( O \)InBodyOf\( M \)AreNotOverloaded\( (b, n, N) \rightarrow \)

IsPublic\( (s, n) \rightarrow \) IsPublic\( (s, m) \)
IsPublic\( (a, n) \rightarrow \) IsPublic\( (a, m) \)
IsPublic\( (b, n) \rightarrow \) IsPublic\( (b, m) \)
IsProtected\( (s, n) \rightarrow \) IsProtected\( (s, m) \)
IsProtected\( (a, n) \rightarrow \) IsProtected\( (a, m) \)
IsProtected\( (b, n) \rightarrow \) IsProtected\( (b, m) \)
IsPrivate\( (s, attribut) \rightarrow \bot \)
IsPrivate\( (a, attribut) \rightarrow \bot \)
IsPrivate\( (b, attribut) \rightarrow \bot \)
IsPrivate\( (s, n) \rightarrow \bot \)
IsPrivate\( (a, n) \rightarrow \bot \)
IsPrivate\( (b, n) \rightarrow \bot \)
IsOverriding\( (a, n) \rightarrow \) IsOverriding\( (a, m) \)
IsOverriding\( (b, n) \rightarrow \) IsOverriding\( (b, m) \)
IsDelegator\( (a, n, V) \rightarrow \) IsDelegator\( (a, m, V) \)
IsDelegator\( (b, n, V) \rightarrow \) IsDelegator\( (b, m, V) \)
IsDelegator\( (a, V, n) \rightarrow \) IsDelegator\( (a, V, m) \)
IsDelegator\( (b, V, n) \rightarrow \) IsDelegator\( (b, V, m) \)
IsInheritedMethodWithParams\( (a, n, [t; t']) \rightarrow \) IsVisibleMethod\( (s, m, [t; t'], a) \)
IsInheritedMethodWithParams\( (b, n, [t; t']) \rightarrow \) IsVisibleMethod\( (s, m, [t; t'], b) \)
MethodIsUsedWithType\( (a, n, \)\([t; t'], [t; t']) \) \rightarrow MethodIsUsedWithType\( (a, m, \)\([t; t'], [t; t']) \)
MethodIsUsedWithType\( (b, n, \)\([t; t'], [t; t']) \) \rightarrow MethodIsUsedWithType\( (b, m, \)\([t; t'], [t; t']) \)
IsUsedMethod\( (a, n, [t; t']) \rightarrow \) IsUsedMethod\( (a, m, [t; t']) \)
IsUsedMethod\( (b, n, [t; t']) \rightarrow \) IsUsedMethod\( (b, m, [t; t']) \)
IsUsedMethodIn\( (a, n, V) \rightarrow \) IsUsedMethodIn\( (a, m, V) \)
IsUsedMethodIn\( (b, n, V) \rightarrow \) IsUsedMethodIn\( (b, m, V) \)
IsInverter\( (a, n, V, V1) \rightarrow \) IsInverter\( (a, m, V, V1) \)
IsInverter\( (b, n, V, V1) \rightarrow \) IsInverter\( (b, m, V, V1) \)
ExistsMethodInvocation\( (a, V, V1, n) \rightarrow \) ExistsMethodInvocation\( (a, V, V1, m) \)
ExistsMethodInvocation\( (b, V, V1, n) \rightarrow \) ExistsMethodInvocation\( (b, V, V1, m) \)
IsIndirectlyRecursive\( (a, n) \rightarrow \) IsIndirectlyRecursive\( (a, m) \)
IsIndirectlyRecursive\( (b, n) \rightarrow \) IsIndirectlyRecursive\( (b, m) \)
BoundVariableInMethodBody\( (a, n, V) \rightarrow \) BoundVariableInMethodBody\( (a, n, V) \)
BoundVariableInMethodBody\( (b, n, V) \rightarrow \) BoundVariableInMethodBody\( (b, n, V) \)
IsOverridden\( (a, n) \rightarrow \) IsOverridden\( (a, m) \)
IsOverridden\( (b, n) \rightarrow \) IsOverridden\( (b, m) \)
A.3 AddParameter

(Add Parameter in Fowler [Fow99] et [Koc02])

AddParameter(class c, method m, parameterType t, parameterName n, defaultvalue e):
Add a parameter of type t to a method m in class c. In method invocations, use the expression e as new parameter.

| original code | refactored code |
|---------------|----------------|
| public class A{
  public void m1{
    block1
  }
  public void m2{
    m1();
  }
} | public class A{
  public void m1(B b){
    block1
  }
  public void m2{
    m1(new B());
  }
} |

Refactoring tools. Change Method signature in Eclipse tool and Change Signature in IntelliJ IDEA.

A.4 AddParameterWithReuse

Overview: AddParameterWithReuse (classname s, subclasses [a,b], methodname m, methodparameters [], paramType t, paramName p, usedvalueofparamType defaultvalue): this operation is used to add the parameter p of type t to the parameters of the method s::m, a::m and b::m. Same as AddParameter, but instead of adding a default value for the additional parameter in invocations, use any value with the specified type that is visible from the invocation site.

In IntelliJ IDEA, this is specified with the Any Var option in Change Signature. This is not supported by Eclipse.

Note that when several variables of the specified type are visible, the result is unspecified. In the example of use in this report, the type of the added parameter is a fresh type, and in recursive methods, the only variable of this type is the parameter being introduced so that there is not ambiguity.

Precondition.
¬BoundVariableInMethodBody(s, m, p)
¬ExistsClass(s)
¬ExistsMethodDefinition(s, m)
¬ExistsMethodDefinitionWithParams(s, m, [])
¬¬ExistsMethodDefinitionWithParams(s, m, [t])
¬¬IsInheritedMethodWithParams(s, m, [t])
¬¬ExistsParameterWithName(s, m, [], p)
¬¬ExistsType(t)
¬¬AllSubclasses(s, [a; b])

Backward Description.
ExistsMethodDefinitionWithParams(s, m, []) ⊨ ⊥
ExistsMethodDefinitionWithParams(s, m, [t]) ⊨ T
¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬---
A.5 AddParameterWithDelegate

**original code**
```
public class A{
  public void m() {
    block1
  }
}
```

**refactored code**
```
public class A{
  public void m(B b) {
    block1
  }
  public void m()
    m(new B());
}
```

A.6 MoveMethod

**original code**
```
public class A{
  public void m(B b){
    block1
  }
}
```

**refactored code**
```
public class B{
  public void m(A a){
    [block1]
  }
  ...
}
```

**Refactoring tools.** If the receiver object is not used in the body of the initial class, it will not be included as parameter in the destination class, so that you have to add it (see AddParameter).

A.7 MoveMethodWithDelegate

*(Move Method in Fowler [Fow99]*)

**Overview:** MoveMethodWithDelegate (classname s,attributes [att1,att2], targetclass a,methodtobemoved m, parameter-stypes [t,a], returntype q, movedmethod n, receivingobjectname o, newreceivingobjectname o'): this operation is used to move the method s::m to the class a and rename it as n.

Transform a method m of a class s into a delegator to a method n in an other class a. The code of m has been moved to n (and adapted).
Refactoring tools. *Move* in Eclipse tool. In IntelliJ IDEA, first introduce a local delegate (with *Change Signature*), then *Move*.

Precondition.

\[
(\text{ExistsClass}(s) \land \text{ExistsClass}(a) \\
\land \text{ExistsMethodDefinitionWithParams}(s, m, [t; a]) \\
\land \text{ExistsParameterWithName}(s, m, [t; a], o) \\
\land \neg \text{ExistsMethodDefinitionWithParams}(a, n, [t; s]) \\
\land \text{HasReturnType}(s, m, q) \\
\land \neg \text{IsPrivate}(s, m) \\
\land \neg \text{IsPrivate}(s, att1) \\
\land \neg \text{IsPrivate}(s, att2))
\]

Backward Description.

\[
\text{ExistsMethodDefinition}(a, n) \mapsto \top \\
\text{ExistsMethodDefinitionWithParams}(a, n, [t; s]) \mapsto \top \\
\text{HasReturnType}(a, n, q) \mapsto \text{HasReturnType}(s, m, q) \\
\text{ExistsParameterWithName}(a, n, [t; s], N) \mapsto \text{ExistsParameterWithName}(s, m, [t; a], N) \quad (N \neq o) \\
\text{ExistsParameterWithName}(a, n, [t; s], o') \mapsto \top \\
\text{ExistsParameterWithName}(a, n, [t; s], o) \mapsto \bot \\
\text{ExistsParameterWithName}(a, n, [t; s], s) \mapsto \top \\
\text{ExistsParameterWithName}(a, n, [t; s], a) \mapsto \bot \\
\text{ExistsParameterWithName}(a, n, [t; s], T) \mapsto \text{ExistsParameterWithName}(s, m, [t; a], T) \quad (T \neq a) \land (T \neq s) \\
\text{ExistsMethodInvocation}(s, m, a, n) \mapsto \top \\
\text{IsInverter}(s, m, a, q) \mapsto \top \\
\text{IsPrivate}(s, att1) \mapsto \bot \\
\text{IsPrivate}(s, att2) \mapsto \bot \\
\text{IsUsedAttributeInMethodBody}(s, att1, m) \mapsto \bot \\
\text{IsUsedAttributeInMethodBody}(s, att2, m) \mapsto \bot
\]

A.8 RenameMethod

(*Rename* in Fowler [Fow99] et [Koc02])

\[
\text{RenameMethod}(\text{class } c, \text{method } m, \text{newname } n) : \text{Rename the method } m \text{ of class } c \text{ into } n.
\]
Refactoring tools. Rename in Eclipse and IntelliJ IDEA.

We have identified two types of this operation. The first one does not accept the overloading, the second one accepts overloading.

### A.8.1 RenameInHierarchyNoOverloading

**Overview** RenameInHierarchyNoOverloading (class c, subclasses [a,b], method m, types [t,t'], newname n) : this operation is used to rename the method (c,a,b)::m into n if n does not already exist with another signature in the hierarchy.

**Precondition.**
(ExistsClass(c) ∧ ExistsClass(a) ∧ ExistsClass(b) ∧ ExistsMethodDefinition(c, m) ∧ ExistsMethodDefinitionWithParams(c, m, [t; t']) ∧ AllSubclasses(c, [a; b]) ∧ ~ExistsMethodDefinition(c, n) ∧ ~ExistsMethodDefinition(a, n) ∧ ~ExistsMethodDefinition(b, n) ∧ ~ExistsMethodDefinitionWithParams(c, n, [t; t']) ∧ ~ExistsMethodDefinitionWithParams(a, n, [t; t']) ∧ ~ExistsMethodDefinitionWithParams(b, n, [t; t']) ∧ ~IsOverloaded(c, m) ∧ ~IsOverloaded(a, m) ∧ ~IsOverloaded(b, m) ∧ ~IsInheritedMethod(c, n))

**Backward Description.**
ExistsMethodDefinition(c, n) \(\ast\) \(\top\)
ExistsMethodDefinitionWithParams(c, n, [t; t']) \(\ast\) \(\top\)
ExistsMethodDefinition(a, n) \(\ast\) ExistsMethodDefinition(a, m)
ExistsMethodDefinition(b, n) \(\ast\) ExistsMethodDefinition(b, m)
ExistsMethodDefinition(c, m) \(\ast\) \(\bot\)
ExistsMethodDefinition(a, m) \(\ast\) \(\bot\)
ExistsMethodDefinition(b, m) \(\ast\) \(\bot\)
ExistsMethodDefinitionWithParams(a, n, [t; t']) \(\ast\) ExistsMethodDefinitionWithParams(a, m, [t; t'])
ExistsMethodDefinitionWithParams(b, n, [t; t']) \(\ast\) ExistsMethodDefinitionWithParams(b, m, [t; t'])
ExistsMethodDefinitionWithParams(c, m, [t; t']) \(\ast\) \(\bot\)
ExistsMethodDefinitionWithParams(a, m, [t; t']) \(\ast\) \(\bot\)
ExistsMethodDefinitionWithParams(b, m, [t; t']) \(\ast\) \(\bot\)
IsInheritedMethod(a, n) \(\ast\) IsInheritedMethod(a, m)
IsInheritedMethod(b, n) \(\ast\) IsInheritedMethod(b, m)
IsDelegator(c, n, M) \(\ast\) IsDelegator(c, m, M)
IsDelegator(a, n, M) \(\ast\) IsDelegator(a, m, M)
IsDelegator(b, n, M) \(\ast\) IsDelegator(b, m, M)
A.8.2 RenameOverloadedMethodInHierarchy

Overview RenameOverloadedMethodInHierarchy (class c, subclasses [a,b], method m, newname n, types [t]) : this operation is used to rename the method (c,a,b)::m into n nevertheless n will be overloaded or not.

Precondition.
(ExistsClass(c)
∧ ExistsMethodDefinitionWithParams(c, m, [t])
∧ ¬IsInheritedMethodWithParams(c, n, [t])
∧ ¬ExistsMethodDefinitionWithParams(c, n, [t])
∧ ¬ExistsMethodDefinitionWithParams(a, n, [t])
∧ ¬ExistsMethodDefinitionWithParams(b, n, [t])
∧ ¬IsInheritedMethodWithParams(c, m, [t])
∧ AllSubclasses(c, [a; b]))

Backward Description.
ExistsMethodDefinition(c, m) ⇐ ⊥
ExistsMethodDefinition(c, n) ⇐ ⊤
IsOverriding(c, n) ⇐ ⊥
IsOverridden(c, n) ⇐ ⊥
IsPublic(c, n) ⇐ IsPublic(c, m)
ExistsMethodDefinitionWithParams(c, n, [t]) ⇐ ExistsMethodDefinitionWithParams(c, m, [t])
ExistsMethodDefinitionWithParams(a, n, [t]) ⇐ ExistsMethodDefinitionWithParams(a, m, [t])
ExistsMethodDefinitionWithParams(b, n, [t]) ⇐ ExistsMethodDefinitionWithParams(b, m, [t])
ExistsMethodDefinition(a, n) ⇐ ⊤
ExistsMethodDefinition(b, n) ⇐ ⊤
ExistsMethodDefinition(c, m, [t]) ⇐ ⊥
ExistsMethodDefinition(c, m, [t]) ⇐ ⊥
ExistsMethodDefinition(c, m, [t]) ⇐ ⊥
IsDelegator(c, n, V) ⇐ IsDelegator(c, m, V)
IsDelegator(a, n, V) ⇐ IsDelegator(a, m, V)
IsDelegator(b, n, V) ⇐ IsDelegator(b, m, V)
IsDelegator(c, V, n) ⇐ IsDelegator(c, V, m)
IsDelegator(a, V, n) ⇐ IsDelegator(a, V, m)
IsDelegator(b, V, n) ⇐ IsDelegator(b, V, m)
IsRecursiveMethod(c, n) ⇐ IsRecursiveMethod(c, m)
IsRecursiveMethod(a, n) ⇐ IsRecursiveMethod(a, m)
IsRecursiveMethod(b, n) ⇐ IsRecursiveMethod(b, m)
ExistsAbstractMethod(c, n) ⇐ ExistsAbstractMethod(c, m)
ExistsAbstractMethod(a, n) ⇐ ExistsAbstractMethod(a, m)
ExistsAbstractMethod(b, n) ⇐ ExistsAbstractMethod(b, m)
IsInheritedMethodWithParams(a, n, [t]) ⇐ IsVisibleMethod(c, m, [t], a)
IsInheritedMethodWithParams(b, n, [t]) ⇐ IsVisibleMethod(c, m, [t], b)
MethodIsUsedWithType(c, n, [t], [t]) → MethodIsUsedWithType(c, m, [t], [t])
MethodIsUsedWithType(a, n, [t], [t]) → MethodIsUsedWithType(a, m, [t], [t])
MethodIsUsedWithType(b, n, [t], [t]) → MethodIsUsedWithType(b, m, [t], [t])
IsUsedMethod(c, n, [t]) → IsUsedMethod(a, m, [t])
IsUsedMethod(a, n, [t]) → IsUsedMethod(a, m, [t])
IsUsedMethodIn(c, n, V) → IsUsedMethodIn(a, m, V)
IsUsedMethodIn(a, n, V) → IsUsedMethodIn(a, m, V)
IsUsedMethodIn(b, n, V) → IsUsedMethodIn(b, m, V)
HasReturnType(c, n, V) → HasReturnType(c, m, V)
HasReturnType(a, n, V) → HasReturnType(a, m, V)
HasReturnType(b, n, V) → HasReturnType(b, m, V)
IsInverter(c, n, V, V1) → IsInverter(c, m, V, V1)
IsInverter(a, n, V, V1) → IsInverter(a, m, V, V1)
IsInverter(b, n, V, V1) → IsInverter(b, m, V, V1)
ExistsMethodInvocation(c, V, V1, n) → ExistsMethodInvocation(a, V, V1, m)
ExistsMethodInvocation(a, V, V1, n) → ExistsMethodInvocation(a, V, V1, m)
ExistsMethodInvocation(b, V, V1, n) → ExistsMethodInvocation(b, V, V1, m)
IsIndirectlyRecursive(c, n) → IsIndirectlyRecursive(c, m)
IsIndirectlyRecursive(a, n) → IsIndirectlyRecursive(a, m)
IsIndirectlyRecursive(b, n) → IsIndirectlyRecursive(b, m)
BoundVariableInMethodBody(c, n, V) → BoundVariableInMethodBody(c, n, V)
BoundVariableInMethodBody(a, n, V) → BoundVariableInMethodBody(a, n, V)
BoundVariableInMethodBody(b, n, V) → BoundVariableInMethodBody(b, n, V)
IsOverridden(a, n) → IsOverridden(a, m)
IsOverridden(b, n) → IsOverridden(b, m)
IsUsedConstructorAsMethodParameter(V, c, m) → ⊥
IsUsedConstructorAsMethodParameter(V, a, m) → ⊥
IsUsedConstructorAsMethodParameter(V, b, m) → ⊥

A.8.3 RenameDelegatorWithOverloading

Overview RenameDelegatorWithOverloading (classname s, subclasses [a,b], method m, paratype t, paramName pn, super type- Ofparatype t’, newname n) : this operation is used to rename the method (c,a,b)::m into n and accepts overloaded methods. This operation is an ad-hoc use of the operation RenameOverloadedMethodInHierarchy (we need in this use more details about the signature of the method to be renamed).

Precondition.
(ExistsClass(s)
∧ ExistsClass(a)
∧ ExistsClass(b)
∧ ExistsMethodDefinition(s, m)
∧ ExistsMethodDefinitionWithParams(s, m, [t])
∧ AllSubclasses(s, [a, b])
∧ ¬ExistsMethodDefinitionWithParams(s, n, [t])
∧ ¬ExistsMethodDefinitionWithParams(a, n, [t])
∧ ¬ExistsMethodDefinitionWithParams(b, n, [t])
∧ ¬IsInheritedMethod(s, n)
∧ ¬ExistsMethodDefinitionWithParams(s, n, [t]))

Backward Description. ExistsMethodDefinition(s, n) → ⊤
ExistsMethodDefinitionWithParams(s, n, [t]) → ⊤
IsPublic(s, n) → IsPublic(s, m)
IsOverridden(b, n) ↦ IsOverridden(b, m)
IsUsedConstructorAsMethodParameter(V, s, m) ↦ ⊤
IsUsedConstructorAsMethodParameter(V, a, m) ↦ ⊤
IsUsedConstructorAsMethodParameter(V, b, m) ↦ ⊤

A.9 ExtractSuperClass
(Extract Super Class in Fowler [Fow99] and [Koc02])

Overview: ExtractSuperClass (subclasses[a,b], superclass s, methodsOfsubclasses [m,n], returnType t): this operation is used to extract a super-class s from the classes a and b and make an abstract declaration of methods a::m, a::n, b::m and b::n in this new super-class.

Refactoring tools. Extract Superclass in Eclipse tool and IntelliJ IDEA. In IntelliJ IDEA, the Extract Superclass operation cannot be applied to several classes simultaneously, so we have maintain the code of this operation in order to run it on several classes.

Precondition.
¬existsType(s)
∧ ExistsClass(a)
∧ ExistsClass(b)
∧ ExtendsDirectly(a, java.lang.Object)
∧ ExtendsDirectly(b, java.lang.Object)
∧ HasReturnType(a, m, t)
∧ HasReturnType(a, n, t)
∧ HasReturnType(b, m, t)
∧ HasReturnType(b, n, t)

Backward Description.
IsAbstractClass(s) ↦ ⊤
ExistsClass(s) ↦ ⊤
ExistsType(s) ↦ ⊤
ExistsMethodDefinition(s, X) ↦ (ExistsMethodDefinition(a, X)
∧ ExistsMethodDefinition(b, X))
ExistsMethodDefinitionWithParams(s, X, []) ↦ (ExistsMethodDefinitionWithParams(a, X, [])
∧ ExistsMethodDefinitionWithParams(b, X, []))
ExistsMethodDefinitionWithParams(s, X, [Y]) ↦ (ExistsMethodDefinitionWithParams(a, X, [Y])
∧ ExistsMethodDefinitionWithParams(b, X, [Y]))
IsUsedMethodIn(s, X, Y) ↦ ⊤
ExistsMethodDefinitionWithParams(X, Y, [s]) \mapsto \bot

IsUsedMethod(s, X, [Y]) \mapsto \bot

AllSubclasses(s, [a; b]) \mapsto \top

MethodIsUsedWithType(X, Y, [Z], [s]) \mapsto \bot

IsInheritedMethodWithParams(X, Y, [s]) \mapsto \bot

IsUsedConstructorAsMethodParameter(s, X, Y) \mapsto \bot

IsUsedConstructorAsInitializer(s, X, Y) \mapsto \bot

IsUsedConstructorAsObjectReceiver(s, X, Y) \mapsto \bot

IsPrimitiveType(s) \mapsto \bot

IsSubType(a, s) \mapsto \top

IsSubType(b, s) \mapsto \top

IsSubType(X, s) \mapsto \top

IsPublic(s, m) \mapsto \top

IsPublic(s, n) \mapsto \top

ExistsAbstractMethod(s, m) \mapsto \top

ExistsAbstractMethod(s, n) \mapsto \top

IsOverriding(s, m) \mapsto \bot

IsOverriding(s, n) \mapsto \bot

IsOverridden(s, m) \mapsto \top

IsOverridden(s, n) \mapsto \top

IsPrivate(s, m) \mapsto \bot

IsPrivate(s, n) \mapsto \bot

A.9.1 ExtractSuperClassWithoutPullUp

Overview: ExtractSuperClassWithoutPullUp (subclasses[a,b], superclass s): this operation is a specific variant of the operation extract super-class. It is simply used to extract a super-class without pull up the methods of sub-classes.

Precondition.
(\neg ExistsType(s)
\land ExistsClass(a)
\land ExistsClass(b)
\land ExtendsDirectly(a, java.lang.Object)
\land ExtendsDirectly(b, java.lang.Object))

Backward Description.
IsAbstractClass(s) \mapsto \top

IsSubType(a, s) \mapsto \top

IsSubType(b, s) \mapsto \top

IsSubType(X, s) \mapsto \top

IsSubType(X, a)

IsSubType(X, s) \mapsto \top

IsSubType(X, b)

A.10 GeneraliseParameter

Overview: GeneraliseParameter (classname s, subclasses [a,b], methodname m, paramName p, type t, supertype st): this operation is used to change the type t of the parameter p of the methods s::m, a::m and b::m into a super-type st. All the uses of the parameter which type is changed must be legal with the new type st (method invocations, object passed as parameter of other methods). The uses of the parameter which type is changed as parameter of methods must not result in a charge of the invoked code (static resolving of overloading).
Refactoring tools. *Change Method Signature* in Eclipse tool and *Type Migration* in IntelliJ IDEA (or *Change Signature*).

**Precondition.**
\[(\text{ExistsClass}(s) ∧ \text{ExistsClass}(a) ∧ \text{ExistsClass}(b) ∧ \text{ExistsMethodDefinition}(s, m) ∧ \text{ExistsMethodDefinition}(a, m) ∧ \text{ExistsMethodDefinition}(b, m) ∧ \text{IsSubType}(s, t) ∧ \text{AllSubclasses}(s, [a; b]) ∧ \text{IsLocalInvokedMethod}(s, m, p, t) ∧ \text{IsLocalInvokedMethod}(a, m, p, t) ∧ \text{IsLocalInvokedMethod}(b, m, p, t) ∧ \text{NotInvolvedInOverlaodingResolution}(s, m, p) ∧ \text{NotInvolvedInOverlaodingResolution}(a, m, p) ∧ \text{NotInvolvedInOverlaodingResolution}(b, m, p))\]

**Backward Description.**
\[\text{IsInverter}(s, m, t, V) \rightarrow \text{IsInverter}(s, m, st, V)\]
\[\text{IsInverter}(a, m, t, V) \rightarrow \text{IsInverter}(a, m, st, V)\]
\[\text{IsInverter}(b, m, t, V) \rightarrow \text{IsInverter}(b, m, st, V)\]
\[\text{ExistsMethodDefinitionWithParams}(s, m, [t]) \rightarrow \text{ExistsMethodDefinitionWithParams}(s, m, [st])\]
\[\text{ExistsMethodDefinitionWithParams}(a, m, [t]) \rightarrow \text{ExistsMethodDefinitionWithParams}(a, m, [st])\]
\[\text{ExistsMethodDefinitionWithParams}(b, m, [t]) \rightarrow \text{ExistsMethodDefinitionWithParams}(b, m, [st])\]
\[\text{ExistsMethodDefinitionWithParams}(s, m, [st]) \rightarrow \bot\]
\[\text{ExistsMethodDefinitionWithParams}(a, m, [st]) \rightarrow \bot\]
\[\text{ExistsMethodDefinitionWithParams}(b, m, [st]) \rightarrow \bot\]
\[\text{IsInheritedMethodWithParams}(a, m, [t]) \rightarrow \top\]
\[\text{IsInheritedMethodWithParams}(b, m, [t]) \rightarrow \top\]
\[\text{IsUsedConstructorAsMethodParameter}(t, s, m) \rightarrow \top\]
\[\text{IsUsedConstructorAsMethodParameter}(t, a, m) \rightarrow \top\]
\[\text{IsUsedConstructorAsMethodParameter}(t, b, m) \rightarrow \top\]
\[\text{IsUsedConstructorAsMethodParameter}(st, s, m) \rightarrow \bot\]
\[\text{IsUsedConstructorAsMethodParameter}(st, a, m) \rightarrow \bot\]
\[\text{IsUsedConstructorAsMethodParameter}(st, b, m) \rightarrow \bot\]
\[\text{IsOverridden}(a, m) \rightarrow \text{ExistsMethodDefinition}(a, m)\]
\[\text{IsOverridden}(b, m) \rightarrow \text{ExistsMethodDefinition}(b, m)\]
\[\text{IsOverriding}(a, m) \rightarrow \text{ExistsMethodDefinition}(a, m)\]
\[\text{IsOverriding}(b, m) \rightarrow \text{ExistsMethodDefinition}(b, m)\]
A.11 MergeDuplicateMethods

Overview MergeDuplicateMethods (classname c, subclasses \(a, b\), mergedMethods \(m, n\), newMethod m2, invertedType t, returnType q): this operation is used to merge methods m and n existing in the hierarchy to a single method m2. The formal description of this operation is built on the formal description of five refactoring operations since it is composed of these operations.

Algorithm of the operation The operation MergeDuplicateMethods is based on four steps:

\[
\text{MergeDuplicateMethods (}c, \{a, b\}, \{m, n\}, m2, t, q) = \\
1. \text{ReplaceMethodCodeDuplicatesInverter (}c, m, \{n\}, t, q) \\
2. \text{PullupConcreteDelegator(}c, \{a, b\}, n, m) \\
3. \text{InlineAndDelete(}c, n) \\
4. \text{RenameInHierarchyNoOverloading (}c, \{a, b\}, m, t, m2)
\]

Refactoring tools. Rename, Replace Method duplication, Extract Method, In-line in Eclipse, Rename, Replace Method Code Duplicates, Pull Up, Inline in IntelliJ IDEA:

Notes.

- ReplaceCodeDuplicates introduces a delegation. After that, the code for m2 is the same in all the subclasses of c (a delegation). Then the pull-up can be done without changing the semantics, which allows to inline (and remove) m2 afterwards.

- After the first pull-up, the IntelliJ IDEA pull-up warns that some code already exists. The first time, when some code replaces the abstract declaration, the refactorer manages to remove the abstract declaration. The next times,
when a second code comes in addition of the first one, the refactorer prefers to leave the two versions (which are identical in this case), so that we have to use safe delete to remove one of them.

We could provide an extension of the pull-up operation with the customized behavior.

Preconditions:
- The two concerned methods bodies must be syntactically equals.
- The new name must not introduce an overloading.
- The two methods must not be overloaded.

A.12 ReplaceMethodcodeDuplicatesInverter

Overview ReplaceMethodcodeDuplicatesInverter (classname c, method m, copies [n,m1],invertedtype t,returntype q) : this operation is used to replace c::[n,m1] by c::m.

Precondition.
(ExistsClass(c)
∧ (ExistsMethodDefinition(c, m)
∧ ExistsMethodDefinition(c, n)
∧ ExistsMethodDefinition(c, m1))
∧ (IsInverter(c, m, t, q)
∧ IsInverter(c, n, t, q)
∧ IsInverter(c, m1, t, q)))

Backward Description.
IsDelegator(c, n, m) ↦ ⊤
IsDelegator(c, m1, m) ↦ ⊤
ExistsMethodInvocation(c, n, c, m) ↦ ⊥
ExistsMethodInvocation(c, m1, c, m) ↦ ⊥
IsRecursiveMethod(c, n) ↦ ⊥
IsRecursiveMethod(c, m1) ↦ ⊥

A.13 SafeDeleteDelegatorOverriding

Overview SafeDeleteDelegatorOverriding (classname c, method m, superclass s, delegatee n) : this operations is used to remove useless overridings.

Precondition. (ExistsClass(c)
∧ ExistsClass(s)
∧ ExistsMethodDefinition(c, m)
∧ ExistsMethodDefinition(s, m)
∧ IsDelegator(c, m, n)
∧ IsDelegator(s, m, n)
∧ NotInvolvedInOverlaodingResolution(c, m, this))

Backward Description.
ExistsMethodDefinition(c, m) ↦ ⊥
IsInheritedMethod(c, m) ↦ ⊤
AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC(c, m, X, Y) ↦ ⊥
AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded(c, m, X) ↦ ⊥
BoundVariableInMethodBody(c, m, X) ↦ ⊥
ExistsParameterWithName(c, m, [X], Y) ↦ ⊥
ExistsParameterWithType(c, m, [X], Y) ↦ ⊥
ExistsMethodInvocation(c, m, X, Y) ↦ ⊥
A.14 PullUpAbstract

PullUpAbstract(set of classes C, method m, interface s)

Pull up a method implemented in a set of classes C to their superclass s: do not move the definitions, just declare the method abstract in s.

Refactoring tools. Pull up in Eclipse tool and IntelliJ IDEA.

Preconditions:

- s is a superclass of each class in C.
- m is defined in all the classes of C
### A.15 PullUpImplementation

**Overview:** `PullUpImplementation(a, [att1, att2], m, s)`: this operation is used to pull up the definition of the method `a::m` to `s` and delete it from `a`.

`class B extends A {
    public void m(){
        block1
    }
}

public class A {
    
}

---

**Refactoring tools.** *Pull Up* in Eclipse tool and IntelliJ IDEA.

**Precondition.**

\[\exists \text{Class}(a) \land \exists \text{Class}(s) \land \text{IsAbstractClass}(s) \land \exists \text{MethodDefinition}(a, m) \land \exists \text{AbstractMethod}(s, m) \land \text{AllInvokedMethodsOnObjectOfInBodyOfMAreDeclaredInC}(a, m, this, s) \land \text{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, m, this) \land \neg \text{IsPrivate}(a, m) \land \neg \text{IsUsedAttributeInMethodBody}(a, att1, m) \land \neg \text{IsUsedAttributeInMethodBody}(a, att2, m)\]

**Backward Description.**

\[\exists \text{MethodDefinition}(a, m) \mapsto \bot\]
\[\exists \text{MethodDefinition}(s, m) \mapsto \top\]
\[\exists \text{AbstractMethod}(s, m) \mapsto \bot\]
\[\text{IsDelegator}(s, m, X) \mapsto \text{IsDelegator}(a, m, X) \quad (m \neq X)\]
\[\exists \text{MethodDefinitionWithParams}(a, m, [X]) \mapsto \bot\]
\[\exists \text{MethodDefinitionWithParams}(s, m, [X]) \mapsto \top\]
\[\text{IsInheritedMethodWithParams}(a, m, [X]) \mapsto \top\]
\[\text{IsInheritedMethod}(a, m) \mapsto \top\]
\[\text{IsVisibleMethod}(s, m, [X], a) \mapsto \top\]
\[\text{IsPrivate}(a, m) \mapsto \bot\]
\[\text{IsOverridden}(a, m) \mapsto \bot\]
\[\text{IsOverriding}(a, m) \mapsto \bot\]
\[\text{IsVisible}(s, m, a) \mapsto \top\]
\[\text{IsOverloaded}(s, m) \mapsto \exists \text{MethodDefinition}(s, m)\]
\[\text{IsUsedAttributeInMethodBody}(a, X, m) \mapsto \bot\]
\[\text{IsOverridden}(a, m) \mapsto \bot\]
\[\text{IsOverloaded}(a, m) \mapsto \bot\]
\[\text{IsRecursiveMethod}(a, m) \mapsto \bot\]
\[\text{HasReturnType}(a, m, X) \mapsto \bot\]
MethodHasParameterType(a, m, X) \to \bot
MethodIsUsedWithTypem(a, m, [X], [X]) \to \bot
IsPrivate(a, att1) \to \bot
IsPrivate(a, att2) \to \bot

A.16 PullUpWithGenerics

Overview: PullupWithGenerics (classname s, subclassname a, [att1,att2],methodname m,returntype RT ,parameterType T): this operation is used to pull up the method a::m to s and then creates the parameter type T to the class s (as shown in the following figure). After performing this operation a polymorphism is created in the hierarchy (Java Generic types).

Refactoring tools. We provide this operation as a plugin for IntelliJ IDEA (Pull up method refactoring extension: http://plugins.intellij.net/plugin/?idea_ce&id=6889).

Precondition.
 ExistsClass(s)
 ∨ IsAbstractClass(s)
 ∨ ExistsClass(a)
 ∨ IsSubType(a, s)
 ∨ ¬ExistsAbstractMethod(s, m)
 ∨ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC(a, m, this, s)
 ∨ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded(a, m, this)
 ∨ ¬IsPrimitiveType(RT)
 ∨ ¬HasParameterType(a, RT)
 ∨ ¬IsUsedAttributeInMethodBody(a, att1, m) ∨ ¬IsPrivate(a, att1))
 ∨ (¬IsUsedAttributeInMethodBody(a, att2, m) ∨ ¬IsPrivate(a, att2)))

Backward Description.
 ExistsAbstractMethod(s, m) \to \top
 HasReturnType(s, m, T) \to \top
 MethodHasParameterType(s, m, T) \to \top
 HasParameterType(s, T) \to \top
extendsFromPrametricClass(a, s, RT) → ⊤
IsGenericsSubtype(a, [RT], s, [T]) → ⊤
IsPrivate(a, m) → ⊥
IsOverridden(a, m) → ⊤
IsOverriding(a, m) → ⊤

A.17 InlineAndDelete

(Inline Method in [(Row99)])

Overview: InlineAndDelete (classname s, methodname m): this operation is used to replace one or all invocations of a given method by its body and delete it.

Refactoring tools. In-line in Eclipse tool and IntelliJ IDEA.

Precondition.
(ExistsClass(c)
∧ ExistsMethodDefinition(c, m)
∧ ¬IsOverriding(c, m)
∧ ¬IsOverridden(c, m)
∧ ¬IsRecursiveMethod(c, m))

Backward Description.
ExistsMethodDefinition(s, m) → ⊥
ExistsMethodDefinitionWithParams(s, m, []) → ⊥
AllInvokedMethodsOnObject0InBodyOfMAreDeclaredInC(s, m, X, Y) → ⊥
AllInvokedMethodsWithParameter0InBodyOfMAreNotOverloaded(s, m, X) → ⊥
BoundVariableInMethodBody(s, m, X) → ⊥
ExistsParameterWithName(s, m, [X], Y) → ⊥
ExistsParameterWithType(s, m, [X], Y) → ⊥
ExistsMethodInvocation(s, m, X, Y) → ⊥
IsInheritedMethodWithParams(s, m, [X]) → ⊥
IsIndirectlyRecursive(s, m) → ⊥
IsVisibleMethod(s, m, [X], Y) → ⊥
IsInverter(s, m, X, Y) → ⊥
IsDelegator(s, m, X) → ⊥
IsUsedMethod(s, m, [X]) → ⊥
IsUsedMethodIn(s, m, X) → ⊥
IsUsedConstructorAsMethodParameter(X, s, m) → ⊥
IsUsedConstructorAsInitializer(X, s, m) → ⊥
IsUsedConstructorAsObjectReceiver\((X, s, m)\) \(\Rightarrow \bot\)
IsPublic\((s, m)\) \(\Rightarrow \bot\)
IsProtected\((s, m)\) \(\Rightarrow \bot\)
IsPrivate\((s, m)\) \(\Rightarrow \bot\)
IsUsedAttributeInMethodBody\((s, X, m)\) \(\Rightarrow \bot\)
IsOverridden\((s, m)\) \(\Rightarrow \bot\)
IsOverloaded\((s, m)\) \(\Rightarrow \bot\)
IsOverriding\((s, m)\) \(\Rightarrow \bot\)
IsRecursiveMethod\((s, m)\) \(\Rightarrow \bot\)
HasReturnType\((s, m, X)\) \(\Rightarrow \bot\)
HasParameterType\((s, m)\) \(\Rightarrow \bot\)
MethodHasParameterType\((s, m, X)\) \(\Rightarrow \bot\)
MethodIsUsedWithType\((s, m, [X], [X])\) \(\Rightarrow \bot\)

A.18 InlineMethodInvocations

Overview: InlineMethodInvocations(classname c, inlinedmethod m, classofinlinedmethod a, modifiedmethod n): this operation is used to in-line a method invocation of the method c::m inside the method a::n.

Refactoring tools. Inline in Eclipse and IntelliJ IDEA: select an invocation to inline and specify you want to inline only that one.

Precondition.
\((\text{ExistsClass}(c)) \land (\text{IsIndirectlyRecursive}(c, n)) \land (\text{ExistsMethodInvocation}(c, n, a, m)) \land (\text{ExistsMethodDefinition}(c, n))\)

Backward Description.
\(\text{ExistsMethodInvocation}(c, m, a, n) \Rightarrow \bot\)
\(\text{Is Recursive}(c, m) \Rightarrow \text{ExistsMethodInvocation}(a, n, c, m)\)
\(\exists \text{Indirectly Recursive}(c, m) \Rightarrow (\text{ExistsMethodInvocation}(a, n, C, X) \land (\text{ExistsMethodDefinition}(C, X, c, m)) \land (X \neq m)\)
\(\text{Is Used Method In}(a, n, m) \Rightarrow \bot\)

A.19 AddSpecializedMethodInHierarchy (Composed)

Overview: AddSpecializedMethodInHierarchy(class s, subclasses [a,b], methodname m, callermethods [n,o], inkovemethods [p,q], paramtype t, paramname pn, subtypesOfparamtype [t1,t2], newtype t'): this operation is used to get the method s::m(t' pn) instead of s::m(t pn). This new duplication takes place in s and in all its subclasses that override m.
Algorithm of the operation  The operation AddSpecializedMethodInHierarchy is based on three steps:

AddSpecializedMethodInHierarchy(class s, subclasses [a,b], methodname m, callermethods [n,o], inkovemethods [p,q], param-type t, paramname pn , subtypesOfparamtype [t1,t2],newtype t') =

1. DuplicateMethodInHierarchy s [a,b] m [p,q] [n,o] temporaryName [t]
2. SpecialiseParameter s [a,b] temporaryName t pn [t1,t2] t';
3. RenameDelegatorWithOverloading (s, [a,b], temporaryName,t', pn,t,m)

Refactoring tools.  With IntelliJ IDEA:

1. Apply DuplicateMethodInHierarchy(c, m, temp-name) (see A.20 below).
2. Apply Change Signature on the method temp-name in the class s, to change the parameter type t into t' (this change is propagated into subclasses).
   Note that the behavior preservation is not guaranteed by this operation in general, but here we introduce a new method so the behavior is not changed. Note also, that here we cannot use the operation Type Migration of IntelliJ IDEA: replacing a parameter type by one of its subtypes is not safe in general.
3. Rename temp-name into m in s with Rename. That renaming introduces an overloading. In general, this could change the semantics of the program, but in the case of this particular chain, and provided the preconditions given below are satisfied, the behavior is preserved (the two methods have the same body; some invocations may be dispatched on the new method, but the external behavior is the same).

A.20 DuplicateMethodInHierarchy

Overview:  DuplicateMethodInHierarchy(class s, subclasses [a,b], methodname m, callermethods [m1,m2], inkovemethods [p,q],newname n ,paramType [t]) : this operation is used to create a duplicate of the method s::m with the name n. All overriding methods in subclasses are also duplicated in these classes.

Refactoring tools.  With IntelliJ IDEA:

1. For each implementation of the method m in the subclasses of the class s, duplicate m by applying Extract Method on its body (give the new name, specify the desired visibility), then inline the invocation of method n that has replaced the method’s body.
2. Use Pull Members Up to make the new method appear in classes where the initial method is declared abstract (specify that it must appear as abstract) (see PullUpAbstract).

Precondition.

\[
\neg \exists \text{Class}(s) \\
\neg \exists \text{MethodDefinitionWithParams}(s, m, [t]) \\
\neg \exists \text{MethodDefinition}(s, m) \\
\neg \exists \text{MethodDefinitionWithParams}(s, n, [t])
\]
\( \neg \exists \text{MethodDefinitionWithParams}(a, n, [t]) \)
\( \neg \exists \text{MethodDefinitionWithParams}(b, n, [t]) \)
\( \neg \exists \text{InheritedMethodWithParams}(s, n, [t]) \)
\( \forall \text{AllSubclasses}(s, [a; b]) \)

Backward Description.
\( \exists \text{MethodDefinition}(s, n) \rightarrow \top \)
\( \exists \text{MethodDefinitionWithParams}(s, n, [t]) \rightarrow \top \)
\( \forall \text{AllInvokedMethodsWithParameterOfMethodNameAreNotOverloaded}(s, n, V) \rightarrow \top \)
\( \forall \text{AllInvokedMethodsOnObjectOfMethodNameAreDeclaredInC}(s, n, V, V1) \rightarrow \top \)
\( (\text{size}(\text{invokedmethodsInMethodName}) = 1) \)
\( \forall \text{BoundVariableInMethodBody}(s, n, V) \rightarrow \exists \text{BoundVariableInMethodBody}(s, m, V) \)
\( \exists \text{Public}(s, n) \rightarrow \exists \text{Public}(s, m) \)
\( \exists \text{ExistsParameterWithName}(s, n, [t], V) \rightarrow \exists \text{ExistsParameterWithName}(s, m, [t], V) \)
\( \exists \text{ExistsParameterWithParamaterType}(s, n, [t], T) \rightarrow \exists \text{ExistsParameterWithParamaterType}(s, m, [t], T) \)
\( \exists \text{IsIndirectlyRecursive}(s, n) \rightarrow \exists \text{IsIndirectlyRecursive}(s, m) \)
\( \exists \text{RecursiveMethod}(s, n) \rightarrow \exists \text{RecursiveMethod}(s, m) \)
\( \exists \text{IsInverter}(s, T, V) \rightarrow \exists \text{IsInverter}(s, m, T) \)
\( \exists \text{IsUsedAttributeInMethodBody}(s, V, n) \rightarrow \exists \text{IsUsedAttributeInMethodBody}(s, m, V) \)
\( \exists \text{MethodHasParameterType}(s, n, V) \rightarrow \exists \text{MethodHasParameterType}(s, m, V) \)
\( \exists \text{ExistsMethodDefinitionWithParams}(a, n, [t]) \rightarrow \exists \text{ExistsMethodDefinitionWithParams}(a, m, [t]) \)
\( \exists \text{ExistsMethodDefinitionWithParams}(b, n, [t]) \rightarrow \exists \text{ExistsMethodDefinitionWithParams}(b, m, [t]) \)
\( \exists \text{IsDelegator}(s, n, p) \rightarrow \top \)
\( \exists \text{IsDelegator}(s, n, q) \rightarrow \top \)
\( \exists \text{IsDelegator}(a, n, p) \rightarrow \top \)
\( \exists \text{IsDelegator}(a, n, q) \rightarrow \top \)
\( \exists \text{IsDelegator}(b, n, p) \rightarrow \top \)
\( \exists \text{IsDelegator}(b, n, q) \rightarrow \top \)
\( \exists \text{ExistsMethodDefinition}(s, n) \rightarrow \top \)
\( \exists \text{ExistsMethodDefinition}(a, n) \rightarrow \top \)
\( \exists \text{ExistsMethodDefinition}(b, n) \rightarrow \top \)
\( \exists \text{MethodIsUsedWithType}(s, n, [t], [t]) \rightarrow \bot \)
\( \exists \text{MethodIsUsedWithType}(a, n, [t], [t]) \rightarrow \bot \)
\( \exists \text{MethodIsUsedWithType}(b, n, [t], [t]) \rightarrow \bot \)
\( \exists \text{MethodIsUsedWithType}(s, n, [t], [T]) \rightarrow \bot \)
\( \exists \text{MethodIsUsedWithType}(b, n, [t], [T]) \rightarrow \bot \)
\( \exists \text{ExistsMethodInvocation}(s, m1, V, n) \rightarrow \top \)
\( \exists \text{ExistsMethodInvocation}(s, m2, V, n) \rightarrow \top \)
\( \exists \text{ExistsMethodInvocation}(a, m1, V, n) \rightarrow \top \)
\( \exists \text{ExistsMethodInvocation}(a, m2, V, n) \rightarrow \top \)
\( \exists \text{ExistsMethodInvocation}(b, m1, V, n) \rightarrow \top \)
\( \exists \text{ExistsMethodInvocation}(b, m2, V, n) \rightarrow \top \)
\( \neg \exists \text{InheritedMethodWithParams}(a, n, [t]) \rightarrow \exists \text{MethodDefinitionWithParams}(a, m, [t]) \)
\( \neg \exists \text{InheritedMethodWithParams}(b, n, [t]) \rightarrow \exists \text{MethodDefinitionWithParams}(b, m, [t]) \)
\( \neg \exists \text{InheritedMethodWithParams}(a, n) \rightarrow \neg \exists \text{MethodDefinitionWithParams}(a, m) \)
\( \neg \exists \text{InheritedMethodWithParams}(b, n) \rightarrow \neg \exists \text{MethodDefinitionWithParams}(b, m) \)
\( \neg \exists \text{IsOverriding}(a, n) \rightarrow \neg \exists \text{MethodDefinitionWithParams}(a, m) \)
\( \neg \exists \text{IsOverridden}(b, n) \rightarrow \neg \exists \text{MethodDefinitionWithParams}(b, m) \)
\( \neg \exists \text{IsOverridden}(a, n) \rightarrow \neg \exists \text{MethodDefinitionWithParams}(a, m) \)
\( \neg \exists \text{IsOverridden}(b, n) \rightarrow \neg \exists \text{MethodDefinitionWithParams}(b, m) \)

52
### A.21 DeleteMethodInHierarchy

(Delete Method in Fowler [Fow99] and [Koc02])

**Overview:** DeleteMethodInHierarchy (classname s, subclasses [a,b], method m, invokedmethodsInm [m1,m2], paramType t) : this operation is used to delete the method m from the hierarchy of classes s, a and b.

Refactoring tools. Safe Delete in IntelliJ IDEA and Delete in Eclipse.

**Precondition.**

(ExistsClass(s) ∧ ExistsMethodDefinitionWithParams(s, m, [t]) ∧ ¬MethodIsUsedWithType(s, m, [t], [t]) ∧ ¬MethodIsUsedWithType(a, m, [t], [t]) ∧ ¬MethodIsUsedWithType(b, m, [t], [t]) ∧ AllSubclasses(s, [a;b]))

**Backward Description.**

ExistsParameterWithType(s, m, [t], t) $\mapsto \bot$
ExistsParameterWithType(a, m, [t], t) $\mapsto \bot$
ExistsParameterWithType(b, m, [t], t) $\mapsto \bot$
ExistsMethodDefinitionWithParams(s, m, [t]) $\mapsto \bot$
ExistsMethodDefinitionWithParams(a, m, [t]) $\mapsto \bot$
ExistsMethodDefinitionWithParams(b, m, [t]) $\mapsto \bot$
ExistsMethodDefinition(s, m) $\mapsto \bot$
ExistsMethodDefinition(a, m) $\mapsto \bot$
ExistsMethodDefinition(b, m) $\mapsto \bot$
IsUsedMethod(s, m1, [V1]) $\mapsto \bot$
IsUsedMethod(s, m2, [V1]) $\mapsto \bot$
IsUsedMethod(a, m1, [V1]) $\mapsto \bot$
IsUsedMethod\((a, m_2, [V_1])\) \(\mapsto \perp\) 
IsUsedMethod\((b, m_1, [V_1])\) \(\mapsto \perp\) 
IsUsedMethod\((b, m_2, [V_1])\) \(\mapsto \perp\) 
IsUsedConstructorAsMethodParameter\((V_1, s, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsMethodParameter\((V_1, a, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsMethodParameter\((V_1, b, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsMethodParameter\((t, s, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsMethodParameter\((t, a, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsMethodParameter\((t, b, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsObjectReceiver\((t, s, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsObjectReceiver\((t, a, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsObjectReceiver\((t, b, m)\) \(\mapsto \perp\) 
IsInheritedMethod\((a, m)\) \(\mapsto \perp\) 
IsInheritedMethod\((b, m)\) \(\mapsto \perp\) 
AllInvokedMethodsOnObject\(O\)InBodyOf\(MA\)AreDeclaredIn\(C\)\((s, m, V_1, V_2)\) \(\mapsto \perp\) 
AllInvokedMethodsOnObject\(O\)InBodyOf\(MA\)AreDeclaredIn\(C\)\((a, m, V_1, V_2)\) \(\mapsto \perp\) 
AllInvokedMethodsOnObject\(O\)InBodyOf\(MA\)AreDeclaredIn\(C\)\((b, m, V_1, V_2)\) \(\mapsto \perp\) 
ExistsAbstractMethod\((a, m)\) \(\mapsto \perp\) 
ExistsAbstractMethod\((b, m)\) \(\mapsto \perp\) 
AllInvokedMethodsWithParameter\(O\)InBodyOf\(MA\)AreNotOverloaded\((s, m, V_1)\) \(\mapsto \perp\) 
AllInvokedMethodsWithParameter\(O\)InBodyOf\(MA\)AreNotOverloaded\((a, m, V_1)\) \(\mapsto \perp\) 
AllInvokedMethodsWithParameter\(O\)InBodyOf\(MA\)AreNotOverloaded\((b, m, V_1)\) \(\mapsto \perp\) 
BoundVariableInMethodBody\((s, m, V_1)\) \(\mapsto \perp\) 
BoundVariableInMethodBody\((a, m, V_1)\) \(\mapsto \perp\) 
BoundVariableInMethodBody\((b, m, V_1)\) \(\mapsto \perp\) 
ExistsParameterWithName\((s, m, \{t\}, V_1)\) \(\mapsto \perp\) 
ExistsParameterWithName\((a, m, \{t\}, V_1)\) \(\mapsto \perp\) 
ExistsParameterWithName\((b, m, \{t\}, V_1)\) \(\mapsto \perp\) 
ExistsMethodInvocation\((s, m, V_1, V_2)\) \(\mapsto \perp\) 
ExistsMethodInvocation\((a, m, V_1, V_2)\) \(\mapsto \perp\) 
ExistsMethodInvocation\((b, m, V_1, V_2)\) \(\mapsto \perp\) 
IsInverter\((s, m, V_1, V_2)\) \(\mapsto \perp\) 
IsInverter\((a, m, V_1, V_2)\) \(\mapsto \perp\) 
IsInverter\((b, m, V_1, V_2)\) \(\mapsto \perp\) 
IsDelegator\((s, V_1, m)\) \(\mapsto \perp\) 
IsDelegator\((a, V_1, m)\) \(\mapsto \perp\) 
IsDelegator\((b, V_1, m)\) \(\mapsto \perp\) 
IsUsedMethodIn\((s, m, V_1)\) \(\mapsto \perp\) 
IsUsedMethodIn\((a, m, V_1)\) \(\mapsto \perp\) 
IsUsedMethodIn\((b, m, V_1)\) \(\mapsto \perp\) 
IsUsedConstructorAsInitializer\((V_1, s, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsInitializer\((V_1, a, m)\) \(\mapsto \perp\) 
IsUsedConstructorAsInitializer\((V_1, b, m)\) \(\mapsto \perp\)
IsUsedConstructorAsObjectReceiver(V1, s, m) ↦ ⊥
IsUsedConstructorAsObjectReceiver(V1, a, m) ↦ ⊥
IsUsedConstructorAsObjectReceiver(V1, b, m) ↦ ⊥
IsUsedAttributeInMethodBody(s, V1, m) ↦ ⊥
IsUsedAttributeInMethodBody(a, V1, m) ↦ ⊥
IsUsedAttributeInMethodBody(b, V1, m) ↦ ⊥
IsOverridden(a, m) ↦ ⊥
IsOverridden(b, m) ↦ ⊥
IsOverloaded(a, m) ↦ ⊥
IsOverloaded(b, m) ↦ ⊥
IsOverriding(a, m) ↦ ⊥
IsOverriding(b, m) ↦ ⊥
IsRecursiveMethod(s, m) ↦ ⊥
IsRecursiveMethod(a, m) ↦ ⊥
IsRecursiveMethod(b, m) ↦ ⊥
HasReturnType(s, m, V1) ↦ ⊥
HasReturnType(a, m, V1) ↦ ⊥
HasReturnType(b, m, V1) ↦ ⊥
MethodHasParameterType(s, m, t) ↦ ⊥
MethodHasParameterType(a, m, t) ↦ ⊥
MethodHasParameterType(b, m, t) ↦ ⊥
MethodIsUsedWithType(s, m, [t], [t]) ↦ ⊥
MethodIsUsedWithType(a, m, [t], [t]) ↦ ⊥
MethodIsUsedWithType(b, m, [t], [t]) ↦ ⊥

A.22 PushDownAll

Overview: PushDownAll (classname s, attributes [att1,att2], subclasses [a,b], method m,paramType [t]): this operation is used to push down the method s::m to its subclasses and delete that method from s (in Push Down Method by Fowler [Fow99], methods are not necessarily pushed down to all the subclasses).

Variation for non-abstract methods:
Refactoring tools. *Push Down* or *Push member Down* in Eclipse tool and IntelliJ IDEA.

**Precondition.**

\[
\begin{align*}
&\exists \text{Class}(s) \land \exists \text{AbstractClass}(s) \\
&\land \exists \text{MethodDefinitionWithParams}(s, m, [t]) \\
&\land \neg \exists \text{UsedMethod}(s, m, [t]) \\
&\land \exists \text{AllSubclasses}(s, [a; b]) \\
&\land \neg \exists \text{Private}(s, m) \\
&\land \exists \text{MethodDefinitionWithParams}(s, m, [t]) \\
&\land \exists \text{MethodDefinitionWithParams}(s, m, [t]) \\
&\land \neg \exists \text{Private}(s, \text{att1}) \\
&\land \neg \exists \text{Private}(s, \text{att2})
\end{align*}
\]

**Backward Description.**

\[
\begin{align*}
&\exists \text{MethodDefinitionWithParams}(s, m, [t]) \rightarrow \bot \\
&\exists \text{UsedMethodIn}(s, m, C) \rightarrow \bot \\
&\exists \text{MethodDefinition}(s, m) \rightarrow \bot \\
&\exists \text{AbstractMethod}(s, m) \rightarrow \bot \\
&\exists \text{Delegator}(s, m, V1) \rightarrow \bot \\
&\exists \text{ReturnType}(s, m, V1) \rightarrow \bot \\
&\exists \text{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, V1, V2) \rightarrow \bot \\
&\exists \text{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, V1) \rightarrow \bot \\
&\exists \text{BoundVariableInMethodBody}(s, m, V1) \rightarrow \bot \\
&\exists \text{ParameterWithName}(s, m, [t], V1) \rightarrow \bot \\
&\exists \text{ParameterWithtype}(s, m, [t], V1) \rightarrow \bot \\
&\exists \text{MethodInvocation}(s, m, V1, V2) \rightarrow \bot \\
&\exists \text{Public}(s, m) \rightarrow \bot \\
&\exists \text{Protected}(s, m) \rightarrow \bot \\
&\exists \text{Private}(s, m) \rightarrow \bot \\
&\exists \text{Overloaded}(s, m) \rightarrow \bot \\
&\exists \text{UsedAttributeInMethodBody}(s, V1, m) \rightarrow \bot \\
&\exists \text{RecursiveMethod}(s, m) \rightarrow \bot \\
&\exists \text{IndirectlyRecursive}(s, m) \rightarrow \bot \\
&\exists \text{ReturnType}(s, m, V1) \rightarrow \bot
\end{align*}
\]
MethodHasParameterType$(s, m, V1) \mapsto \perp$
MethodIsUsedWithType$(s, m, [t], [t]) \mapsto \perp$
IsUsedConstructorAsMethodParameter$(V1, s, m) \mapsto \perp$
IsUsedConstructorAsMethodParameter$(V1, s, m) \mapsto \perp$
IsUsedConstructorAsObjectReceiver$(V1, s, m) \mapsto \perp$
ExistsMethodDefinitionWithParams$(a, m, [t]) \mapsto \top$
ExistsMethodDefinitionWithParams$(b, m, [t]) \mapsto \top$
IsOverriding$(a, m) \mapsto \perp$
IsOverriding$(b, m) \mapsto \perp$
IsOverridden$(a, m) \mapsto \perp$
IsOverridden$(b, m) \mapsto \perp$
IsInheritedMethodWithParams$(a, m, [t]) \mapsto \perp$
IsInheritedMethodWithParams$(b, m, [t]) \mapsto \perp$
IsVisibleMethod$(s, m, [t], a) \mapsto \perp$
IsVisibleMethod$(s, m, [t], b) \mapsto \perp$
IsVisible$(s, m, a) \mapsto \perp$
IsVisible$(s, m, b) \mapsto \perp$
IsInheritedMethod$(a, m) \mapsto \perp$
IsInheritedMethod$(b, m) \mapsto \perp$
IsPrivate$(s, att1) \mapsto \perp$
IsPrivate$(s, att2) \mapsto \perp$

A.23 PushDownImplementation

Overview: PushDownImplementation (classname $s$, attributes $[att1, att2]$, subclasses $[a, b]$, method $m$, paramType $[t]$): same as PushDownAll but keep the method abstract in the superclass.

| original code                                      | refactored code                          |
|----------------------------------------------------|------------------------------------------|
| public abstract class S{                           | public abstract class S{                  |
|     public void m(){                               |     public abstract void m();             |
|         block0                                    | }                                        |
| }                                                  |                                           |
| public class A extends S{                          | public class A extends S{                 |
|     public void m(){                               |     public void m(){                     |
|         block1                                    |         block1                           |
| }                                                  | }                                        |
| public class B extends S{                          | public class B extends S{                 |
| }                                                  | }                                        |

Precondition.

$(\exists \text{Class}(s))$
∧ $(\exists \text{MethodDefinition}(s, m))$
∧ $(\exists \text{MethodDefinitionWithParams}(s, m, [t]))$
∧ $\neg (\exists \text{AbstractMethod}(s, m))$
∧ $(\forall \text{Subclasses}(s, [a; b]))$
∧ $\neg (\exists \text{MethodDefinitionWithParams}(a, m, [t]))$
∧ $\neg (\exists \text{MethodDefinitionWithParams}(b, m, [t]))$
∧ $\neg \text{IsPrivate}(s, att1)$

57
∧ ¬IsPrivate(s, att2))

Backward Description.

- ExistsAbstractMethod(s, m) \iff \top
- AllInvokedMethodsOnObjectOfMAreDeclaredInC(s, m, V1, V2) \iff \bot
- AllInvokedMethodsWithParameterOfMAreNotOverloaded(s, m, V1) \iff \bot
- BoundVariableInMethodBody(s, m, V1) \iff \bot
- ExistsMethodInvocation(s, m, V1, V2) \iff \bot
- IsInheritedMethodWithParams(s, m, [t]) \iff \bot
- IsIndirectlyRecursive(s, m) \iff \bot
- IsUsedConstructorAsInitializer(V1, s, m) \iff \bot
- IsUsedConstructorAsObjectReceiver(V1, s, m) \iff \bot
- IsPrivate(s, m) \iff \bot
- IsUsedAttributeInMethodBody(s, V1, m) \iff \bot
- IsOverridden(s, m) \iff \bot
- IsOverriding(s, m) \iff \bot
- IsRecursiveMethod(s, m) \iff \bot
- MethodHasParameterType(s, m, V1) \iff \bot
- MethodIsUsedWithType(s, m, [t], [t]) \iff \bot
- ExistsMethodDefinitionWithParams(a, m, [t]) \iff \top
- ExistsMethodDefinitionWithParams(b, m, [t]) \iff \top
- ExistsMethodDefinition(a, m) \iff \top
- ExistsMethodDefinition(b, m) \iff \top
- IsPrivate(s, att1) \iff \bot
- IsPrivate(s, att2) \iff \bot

A.24 PushDownNotRedefinedMethod

pushDownNotRedefinedMethod(class c, method m)

Duplicate the method m of class c into its subclasses.

Refactoring tools. *Extract Method, Inline, Push Down, Rename* in Eclipse and IntelliJ IDEA.
A.25 ReplaceMethodDuplication

Overview: ReplaceMethodDuplication (classname s, subclasses [a,b], method m, copy n, paramType [t]) : this operation is used to replace any occurrence of method s::m.

Refactoring tools. Replace Method Duplication in IntelliJ IDEA.

Precondition.
(ExistsClass(s)
∧ ExistsMethodDefinition(s, m)
∧ ExistsMethodDefinition(s, n)
∧ IsDelegator(s, n, m)
∧ AllSubclasses(s, [a; b]))

Backward Description. IsUsedMethod(s, n, [t]) \(\rightarrow \perp\)
IsDelegator(s, n, m) \(\rightarrow \top\)
ExistsMethodInvocation(s, n, s, m) \(\rightarrow \perp\)
IsRecursiveMethod(s, n) \(\rightarrow \perp\)
IsRecursiveMethod(a, n) \(\rightarrow \perp\)
IsRecursiveMethod(b, n) \(\rightarrow \perp\)

A.26 DeleteClass

DeleteClass(class c): Delete a class c which is not used.

Overview: DeleteClass (classname a, classnamesuperclass s ,allclasses [s,a,b], classnamemethods [m,m1],othermethods [m2,n]) : this operation is used to delete the class a which is supposed to be not used.

Refactoring tools. Safe Delete in IntelliJ IDEA, Delete in Eclipse.

Precondition.
(ExistsClass(a)
∧ ¬ExistsMethodDefinitionWithParams(s, m2, [a])
∧ ¬ExistsMethodDefinitionWithParams(s, n, [a])
∧ ¬ExistsMethodDefinitionWithParams(b, m2, [a])
∧ ¬ExistsMethodDefinitionWithParams(b, n, [a])
∧ ¬IsUsedMethodIn(a, m, s)
∧ ¬IsUsedMethodIn(a, m1, s)
∧ ¬IsUsedMethodIn(a, m, b)
∧ ¬IsUsedMethodIn(a, m1, b)
∧ ¬IsUsedConstructorAsMethodParameter(a, a, s, m2)
∧ ¬IsUsedConstructorAsMethodParameter(a, a, s, n)
∧ ¬IsUsedConstructorAsMethodParameter(a, a, b, m2)
∧ ¬IsUsedConstructorAsMethodParameter(a, a, b, n)
∧ ¬IsUsedConstructorAsInitializer(a, a, s, m2)
∧ ¬IsUsedConstructorAsInitializer(a, a, s, n)
∧ ¬IsUsedConstructorAsInitializer(a, a, s, n))
∧ ¬\textit{IsUsedConstructorAsInitializer}(a, a, b, m_2)
∧ ¬\textit{IsUsedConstructorAsInitializer}(a, a, b, n)
∧ ¬\textit{IsUsedConstructorAsObjectReceiver}(a, a, s, m_2)
∧ ¬\textit{IsUsedConstructorAsObjectReceiver}(a, a, s, n)
∧ ¬\textit{IsUsedConstructorAsObjectReceiver}(a, a, b, m_2)
∧ ¬\textit{IsUsedConstructorAsObjectReceiver}(a, a, b, n)
∧ ¬\textit{IsSubType}(s, a)
∧ ¬\textit{IsSubType}(b, a)

\textbf{Backward Description.}
\textit{ExistsType}(a) \mapsto \bot
\textit{IsSubType}(a, s) \mapsto \bot
\textit{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, V_1, V_2, V_3) \mapsto \bot
\textit{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, V_1, V_2) \mapsto \bot
\textit{BoundVariableInMethodBody}(a, V_1, V_2) \mapsto \bot
\textit{ExistsParameterWithName}(a, V_1, [V_2], V_3) \mapsto \bot
\textit{ExistsMethodInvocation}(a, V_1, V_2, V_3) \mapsto \bot
\textit{ExistsMethodDefinitionWithParams}(a, V_1, [V_2]) \mapsto \bot
\textit{ExtendsDirectly}(a, s) \mapsto \bot
\textit{ExtendsDirectly}(V_1, a) \mapsto \bot
\textit{IsInheritedMethodWithParams}(a, V_1, [V_2]) \mapsto \bot
\textit{IsIndirectlyRecursive}(a, V_1) \mapsto \bot
\textit{IsVisibleMethod}(a, V_1, [V_2], V_3) \mapsto \bot
\textit{IsInverter}(a, V_1, V_2, V_3) \mapsto \bot
\textit{IsDelegator}(a, V_1, V_2) \mapsto \bot
\textit{IsAbstractClass}(a) \mapsto \bot
\textit{IsUsedMethod}(a, V_1, [V_2]) \mapsto \bot
\textit{IsUsedMethodIn}(a, V_1, V_2) \mapsto \bot
\textit{IsUsedConstructorAsMethodParameter}(V_1, a, V_2) \mapsto \bot
\textit{IsUsedConstructorAsInitializer}(a, V_1, V_2) \mapsto \bot
\textit{IsUsedConstructorAsObjectReceiver}(a, V_1, V_2) \mapsto \bot
\textit{IsUsedConstructorAsInitializer}(V_1, a, V_2) \mapsto \bot
\textit{IsUsedConstructorAsObjectReceiver}(V_1, a, V_2) \mapsto \bot
\textit{IsPrimitiveType}(a) \mapsto \bot
\textit{IsPublic}(a, V_1) \mapsto \bot
\textit{IsProtected}(a, V_1) \mapsto \bot
\textit{IsPrivate}(a, V_1) \mapsto \bot
\textit{IsGenericsSubtype}(a, [V_1], s, [V_2]) \mapsto \bot
\textit{IsGenericsSubtype}(V_1, [V_2], a, [V_3]) \mapsto \bot
\textit{IsGenericsSubtype}(V_1, [a], V_2, [V_3]) \mapsto \bot
\textit{IsInheritedField}(a, V_1) \mapsto \bot
\textit{IsOverridden}(a, V_1) \mapsto \bot
\textit{IsOverloaded}(a, V_1) \mapsto \bot
\textit{IsOverriding}(a, V_1) \mapsto \bot
\textit{IsRecursiveMethod}(a, V_1) \mapsto \bot
\textit{HasReturnType}(a, V_1, V_2) \mapsto \bot
\textit{HasParameterType}(a, V_1) \mapsto \bot
A.27 ExtractGeneralMethod

original code

```java
public abstract class C {
    public void ma() {
        new A().op(this);
    }
    public void mb() {
        new B().op(this);
    }
}
```

refactored code

```java
public abstract class C {
    public void m(S o) {
        o.op(this);
    }
    public void ma() {
        m(new A());
    }
    public void mb() {
        m(new B());
    }
}
```

A.28 InlineClass

**InlineClass(class c):** Inline one or more references to a given class c.

**Refactoring tools.** *Inline* in Eclipse and IntelliJ IDEA.

A.29 SpecialiseParameter

**Overview:** SpecialiseParameter(classname s, subclasses [a,b], methodname m, paramType t , paramName p, subtypes [st,q], new paramType st): this operation is used to change the type t of the parameter p of the methods s::m, a::m and b::m into one of its subtypes (st).

**Precondition.**

\[\text{IsSubType}(st, t)\]
\[\neg \text{MethodIsUsedWithType}(s, m, [t], [t])\]
\[\neg \text{MethodIsUsedWithType}(a, m, [t], [t])\]
\[\neg \text{MethodIsUsedWithType}(b, m, [t], [t])\]
\[\text{ExistsClass}(s)\]
\[\text{ExistsMethodDefinitionWithParams}(s, m, [t])\]
\[\text{ExistsType}(t)\]
\[\text{ExistsType}(st)\]
\(\neg \text{ExistsMethodDefinitionWithParams}(s, m, [st])\)
\(\neg \text{IsInheritedMethodWithParams}(s, m, [st])\)
\(\text{AllSubclasses}(s, \{a, b\})\)
\(\text{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, p)\)
\(\langle \neg \text{MethodIsUsedWithType}(s, m, [l], [q]) \lor \text{ExistsMethodDefinitionWithParams}(s, m, [q]) \rangle \)

### Backward Description.

\(\text{ExistsMethodDefinitionWithParams}(s, m, [st]) \mapsto T\)
\(\text{ExistsMethodDefinitionWithParams}(s, m, [l]) \mapsto \bot\)
\(\text{ExistsMethodDefinitionWithParams}(a, m, [l]) \mapsto \bot\)
\(\text{ExistsMethodDefinitionWithParams}(b, m, [l]) \mapsto \bot\)
\(\text{ExistsMethodDefinitionWithParams}(a, m, [st]) \mapsto \text{ExistsMethodDefinitionWithParams}(a, m, [l])\)
\(\text{ExistsMethodDefinitionWithParams}(b, m, [st]) \mapsto \text{ExistsMethodDefinitionWithParams}(b, m, [l])\)

\(\text{MethodIsUsedWithType}(s, m, [l], [l]) \mapsto \bot\)
\(\text{MethodIsUsedWithType}(a, m, [l], [l]) \mapsto \bot\)
\(\text{MethodIsUsedWithType}(b, m, [l], [l]) \mapsto \bot\)
\(\text{ExistsParameterWithName}(s, m, [st], p) \mapsto T\)
\(\text{ExistsParameterWithName}(a, m, [st], p) \mapsto T\)
\(\text{ExistsParameterWithName}(b, m, [st], p) \mapsto T\)
\(\text{ExistsParameterWithName}(a, m, [st], st) \mapsto T\)
\(\text{ExistsParameterWithName}(b, m, [st], st) \mapsto T\)

\(\text{IsInverter}(s, m, st, V) \mapsto \text{IsInverter}(s, m, t, V)\)
\(\text{IsInverter}(a, m, st, V) \mapsto \text{IsInverter}(a, m, t, V)\)
\(\text{IsInverter}(b, m, st, V) \mapsto \text{IsInverter}(b, m, t, V)\)
\(\text{IsInheritedMethodWithParams}(s, m, [st]) \mapsto T\)
\(\text{IsInheritedMethodWithParams}(a, m, [st]) \mapsto T\)
\(\text{IsInheritedMethodWithParams}(b, m, [st]) \mapsto T\)
\(\text{IsUsedConstructorAsMethodParameter}(st, s, m) \mapsto T\)
\(\text{IsUsedConstructorAsMethodParameter}(st, a, m) \mapsto T\)
\(\text{IsUsedConstructorAsMethodParameter}(st, b, m) \mapsto T\)
\(\text{IsUsedConstructorAsMethodParameter}(t, s, m) \mapsto \bot\)
\(\text{IsUsedConstructorAsMethodParameter}(t, a, m) \mapsto \bot\)
\(\text{IsUsedConstructorAsMethodParameter}(t, b, m) \mapsto \bot\)
\(\text{IsOverridden}(a, m) \mapsto \text{ExistsMethodDefinition}(a, m)\)
\(\text{IsOverridden}(b, m) \mapsto \text{ExistsMethodDefinition}(b, m)\)
\(\text{IsOverriding}(a, m) \mapsto \text{ExistsMethodDefinition}(a, m)\)
\(\text{IsOverriding}(b, m) \mapsto \text{ExistsMethodDefinition}(b, m)\)

### B Preconditions composition

In this section, we apply a calculus of minimal precondition to the sequence of basic refactoring operations that compose our transformation. We apply the calculus of Kniesel and Koch [KK04], based on the backward descriptions given in the previous appendix. We use that calculus to compute a minimum precondition that ensures that the round-trip transformation succeeds, which means we determine a set of programs on which we can ensure that the preconditions of all the component refactoring operations will be satisfied when applying the Composite → Visitor → Composite transformation any number of time (as explained in [CA13]).

That computed precondition is given in Fig. 29. The chains taken into account for the computation are given in Figs. 27 and 28. The difference between these chains and the algorithms of previous sections are:

- We add some parameters to the operations that were not made explicit before (in previous sections, the project was an implicit parameter).
• Composite operations are replaced by their component operations (**MergeDuplicateMethods** in step 8 of the **Composite → Visitor** chain).

• We split some operations of the tool into several abstract operations when the behavior of an operation depends on the state of the program. In that case, each possible behavior is represented by a different abstract operation. For instance, we have two operations for method renaming, one for overloaded methods and one for not-overloaded methods.

1. CreateEmptyClass(PrintVisitor)
   CreateEmptyClass(PrettyprintVisitor)

2. CreateIndirectionInSuperClass(Graphic, [Ellipse;CompositeGraphic;], print, [], void, printTmpVC)
   CreateIndirectionInSuperClass(Graphic, [Ellipse;CompositeGraphic;], prettyprint, [], void, prettyprintTmpVC)

3. InlineMethodInvocations(CompositeGraphic, printTmpVC, [], Graphic, print, [])
   InlineMethodInvocations(CompositeGraphic, prettyprintTmpVC, [], Graphic, prettyprint, [])

4. AddParameterWithReuse(Graphic, [Ellipse;CompositeGraphic;], printTmpVC, [], PrintVisitor, v)
   AddParameterWithReuse(Graphic, [Ellipse;CompositeGraphic;], prettyprintTmpVC, [], PrettyprintVisitor, v)

5. MoveMethodWithDelegate (Ellipse, [mChildGraphics;], PrintVisitor, printTmpVC, [PrintVisitor;], void, visit)
   MoveMethodWithDelegate (CompositeGraphic, [mChildGraphics;], PrintVisitor, printTmpVC, [PrintVisitor;], void, visit)
   MoveMethodWithDelegate (CompositeGraphic, [mChildGraphics;], PrettyprintVisitor, prettyprintTmpVC, [PrettyprintVisitor;], void, visit)
   MoveMethodWithDelegate (CompositeGraphic, [mChildGraphics;], PrettyprintVisitor, prettyprintTmpVC, [PrettyprintVisitor;], void, visit)

6. ExtractSuperClass ([PrintVisitor;PrettyprintVisitor;], Visitor, [visit;], void)

7. GeneraliseParameter (Graphic, [Ellipse;CompositeGraphic;], printTmpVC, v, PrintVisitor, Visitor)
   GeneraliseParameter (Graphic, [Ellipse;CompositeGraphic;], prettyprintTmpVC, v, PrettyprintVisitor, Visitor)

8. ReplaceMethodcodeDuplicatesInverter (Ellipse, printTmpVC, [prettyprintTmpVC;], Visitor, void)
   ReplaceMethodcodeDuplicatesInverter (CompositeGraphic, printTmpVC, [prettyprintTmpVC;], Visitor, void)
   PullupConcreteDelegator (Ellipse, [mChildGraphics;], prettyprintTmpVC, Graphic)
   SafeDeleteWithOverridden (CompositeGraphic, prettyprintTmpVC, Graphic)
   InlineAndDelete (Graphic, prettyprintTmpVC)
   RenameInHierarchyNoOverloading (Graphic, [Ellipse;CompositeGraphic;], printTmpVC, [Visitor;], accept)

---

**Figure 27:** Chain of refactoring operations for **Composite → Visitor** transformation analysis
1. DuplicateMethodInHierarchy (Graphic, [Ellipse;CompositeGraphic:], accept, [visit:], [print;prettyprint:], acceptPrintVisitoraddspecializedMethodtmp, [Visitort])
   SpecialiseParameter (Graphic, [Ellipse;CompositeGraphic:], acceptPrintVisitoraddspecializedMethodtmp, Visitor, [PrintVisitor;PrettyprintVisitor:], PrintVisitor)
   RenameDelegatorWithOverloading (Graphic, [Ellipse;CompositeGraphic:], acceptPrintVisitoraddspecializedMethodtmp, Visitor, [PrintVisitor;PrettyprintVisitor:], PrettyprintVisitor)
   DuplicateMethodInHierarchy (Graphic, [Ellipse;CompositeGraphic:], accept, [visit:], [print;prettyprint:], acceptPrettyprintVisitoraddspecializedMethodtmp, [Visitor:])
   SpecialiseParameter (Graphic, [Ellipse;CompositeGraphic:], acceptPrettyprintVisitoraddspecializedMethodtmp, Visitor, [PrintVisitor;PrettyprintVisitor:], PrettyprintVisitor)
   RenameDelegatorWithOverloading (Graphic, [Ellipse;CompositeGraphic:], acceptPrettyprintVisitoraddspecializedMethodtmp, PrettyprintVisitor, v, Visitor, accept)
   DeleteMethodInHierarchy (Graphic, [Ellipse;CompositeGraphic:], accept, [visit:], Visitor)

2. PushDownAll (Visitor, [PrintVisitor;PrettyprintVisitor:], visit, [Ellipse;])
   PushDownAll (Visitor, [PrintVisitor;PrettyprintVisitor:], visit, [CompositeGraphic;])

3. InlineMethod (Ellipse, visit, PrintVisitor, accept)
   InlineMethod (CompositeGraphic, visit, PrintVisitor, accept)
   InlineMethod (Ellipse, visit, PrettyprintVisitor, accept)
   InlineMethod (CompositeGraphic, visit, PrettyprintVisitor, accept)

4. RenameOverloadedMethodInHierarchy (Graphic, [Ellipse;CompositeGraphic:], accept, printTmpVC, [PrintVisitor:]
   RenameOverloadedMethodInHierarchy (Graphic, [Ellipse;CompositeGraphic:], accept, prettyprintTmpVC, [PrettyprintVisitor:]

5. RemoveParameter (Graphic, [Ellipse;CompositeGraphic:], printTmpVC, [PrintVisitor:], PrintVisitor, v)
   RemoveParameter (Graphic, [Ellipse;CompositeGraphic:], prettyprintTmpVC, [PrettyprintVisitor:], PrettyprintVisitor, v)

6. ReplaceMethodDuplication (Graphic, [Ellipse;CompositeGraphic:], print, printTmpVC, [])
   ReplaceMethodDuplication (Graphic, [Ellipse;CompositeGraphic:], prettyprint, prettyprintTmpVC, [])

7. PushDownImplementation (Graphic, [], [Ellipse;CompositeGraphic:], print, [])
   PushDownImplementation (Graphic, [], [Ellipse;CompositeGraphic:], prettyprint, [])

8. PushDownAll (Graphic, [Ellipse;CompositeGraphic:], printTmpVC, [])
   PushDownAll (Graphic, [Ellipse;CompositeGraphic:], prettyprintTmpVC, [])

9. InlineAndDelete (Ellipse, printTmpVC)
   InlineAndDelete (CompositeGraphic, printTmpVC)
   InlineAndDelete (Ellipse, prettyprintTmpVC)
   InlineAndDelete (CompositeGraphic, prettyprintTmpVC)

10. DeleteClass (PrintVisitor, Visitor, [Ellipse;CompositeGraphic;PrintVisitor;PrettyprintVisitor;Visitor;Graphic:], [visit:], [accept;eval;show;])
    DeleteClass (PrettyprintVisitor, Visitor, [Ellipse;CompositeGraphic;PrintVisitor;PrettyprintVisitor;Visitor;Graphic:], [visit:], [accept;eval;show;])

11. DeleteClass (Visitor, java.lang.Object, [Ellipse;CompositeGraphic;PrintVisitor;PrettyprintVisitor;Visitor;Graphic:], [visit:], [accept;eval;show;])

Figure 28: Chain of refactoring operations of the back transformation analysis
(~ExistsMethodDefinition(Graphic, accept) \\
∧ ¬ExistsMethodDefinition(Ellipse, accept) \\
∧ ¬ExistsMethodDefinition(CompositeGraphic, accept) \\
∧ ~IsInheritedMethod(Graphic, accept) \\
∧ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded(CompositeGraphic, prettyprint, this) \\
∧ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC(Ellipse, prettyprint, this, Graphic) \\
∧ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded(Ellipse, prettyprint, this) \\
∧ ¬ExistsType(Visitor) \\
∧ ExistsClass(Ellipse) \\
∧ ~BoundVariableInMethodBody(Graphic, prettyprint, v) \\
∧ ~BoundVariableInMethodBody(Graphic, print, v) \\
∧ IsRecursiveMethod(CompositeGraphic, prettyprint) \\
∧ ExistsClass(CompositeGraphic) \\
∧ IsRecursiveMethod(CompositeGraphic, print) \\
∧ ExistsMethodDefinitionWithParams(Graphic, prettyprint, []) \\
∧ ExistsAbstractMethod(Graphic, prettyprint) \\
∧ ~IsInheritedMethod(Graphic, prettyprintTmpVC) \\
∧ ~IsInheritedMethodWithParams(Graphic, prettyprintTmpVC, []) \\
∧ ¬ExistsMethodDefinitionWithParams(Graphic, prettyprintTmpVC, []) \\
∧ HasReturnType(Graphic, prettyprint, void) \\
∧ ExistsMethodDefinition(Graphic, prettyprint) \\
∧ ExistsMethodDefinition(Ellipse, prettyprint) \\
∧ ExistsMethodDefinition(CompositeGraphic, prettyprint) \\
∧ ~ExistsMethodDefinition(Graphic, prettyprintTmpVC) \\
∧ ~ExistsMethodDefinition(Ellipse, prettyprintTmpVC) \\
∧ ~ExistsMethodDefinition(CompositeGraphic, prettyprintTmpVC) \\
∧ ExistsClass(Graphic) \\
∧ IsAbstractClass(Graphic) \\
∧ ExistsMethodDefinitionWithParams(Graphic, print, []) \\
∧ ExistsAbstractMethod(Graphic, print) \\
∧ ~IsInheritedMethod(Graphic, printTmpVC) \\
∧ ~IsInheritedMethodWithParams(Graphic, printTmpVC, []) \\
∧ ~ExistsMethodDefinitionWithParams(Graphic, printTmpVC, []) \\
∧ AllSubclasses(Graphic, [Ellipse; CompositeGraphic]) \\
∧ HasReturnType(Graphic, print, void) \\
∧ ~IsPrivate(Graphic, print) \\
∧ ~IsPrivate(Ellipse, print) \\
∧ ~IsPrivate(CompositeGraphic, print) \\
∧ ExistsMethodDefinition(Graphic, print) \\
∧ ExistsMethodDefinition(Ellipse, print) \\
∧ ExistsMethodDefinition(CompositeGraphic, print) \\
∧ ~ExistsMethodDefinition(Graphic, printTmpVC) \\
∧ ~ExistsMethodDefinition(Ellipse, printTmpVC) \\
∧ ~ExistsMethodDefinition(CompositeGraphic, printTmpVC) \\
∧ ~ExistsType(PrettyprintVisitor) \\
∧ ~ExistsType(PrintVisitor))

Figure 29: Computed precondition for the round-trip transformation