Refactoring Composite to Visitor and Inverse Transformation in Java

Akram Ajouli, Julien Cohen

To cite this version:

Akram Ajouli, Julien Cohen. Refactoring Composite to Visitor and Inverse Transformation in Java. 2011. hal-00652872v1

HAL Id: hal-00652872
https://hal.science/hal-00652872v1

Submitted on 16 Dec 2011 (v1), last revised 1 Jul 2013 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 4
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 10

4 Variants of Transformations for Various Pattern Instances **18**
4.1 Methods with Parameters 18
4.1.1 Composite→Visitor Transformation 18
4.1.2 Visitor Program 19
4.1.3 Visitor→Composite Transformation 19
4.2 Methods Returning Values 20
4.2.1 Composite→Visitor Transformation 20
4.2.2 Visitor Program 20
4.2.3 Visitor→Composite Transformation 20
4.3 Interface instead of Abstract class in the Composite structure 20
4.3.1 Composite→Visitor Transformation 20
4.3.2 Visitor Program 24
4.3.3 Visitor→Composite Transformation 24
4.4 Class Hierarchies with Several Levels 24
4.4.1 Composite→Visitor Transformation 25
4.4.2 Visitor Program 25
4.4.3 Visitor→Composite Transformation 25

5 Application to JHotDraw **28**

6 Related work **28**
6.1 Refactoring to Patterns 28
6.2 Building Complex Refactoring Operations 29
6.3 Design Patterns Discovery 29

7 Conclusion **29**

References **29**

A Refactoring Operations **31**
A.1 CreateEmptyClass . . . 31
A.2 CreateIndirectionInSuperClass . . . 31
A.3 AddParameter . . . 31
A.4 AddParameterWithReuse . . . 32
A.5 AddParameterWithDelegate . . . 32
A.6 MoveMethod . . . 32
A.7 MoveMethodWithDelegate . . . 33
A.8 RenameMethod . . . 33
A.9 ExtractSuperClass . . . 33
A.10 ExtractSuperClassWithGenerics . . . 34
A.11 GeneraliseParameter . . . 34
A.12 MergeDuplicateMethods . . . 35
A.13 PullUpAbstract . . . 35
A.14 PullUpConcrete . . . 35
A.15 InlineMethod . . . 36
A.16 InlineMethodInvocations . . . 36
A.17 AddSpecializedMethodInHierarchy . . . 36
A.18 DuplicateMethodInHierarchy . . . 37
A.19 DeleteMethodInHierarchy . . . 37
A.20 PushDownAll . . . 38
A.21 PushDownImplementation . . . 39
A.22 pushDownNotRedefinedMethod . . . 39
A.23 ReplaceMethodDuplication . . . 39
A.24 DeleteClass . . . 40
A.25 ExtractGeneralMethod . . . 40
A.26 InlineClass . . . 40

1 Introduction

Composite and Visitor patterns have dual properties with respect to modularity: while the Composite pattern (as well as Visitor 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 [CD11].

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 automatic yet, which makes the proposal of Cohen and Douence [CD11] not currently applicable in object oriented programs.

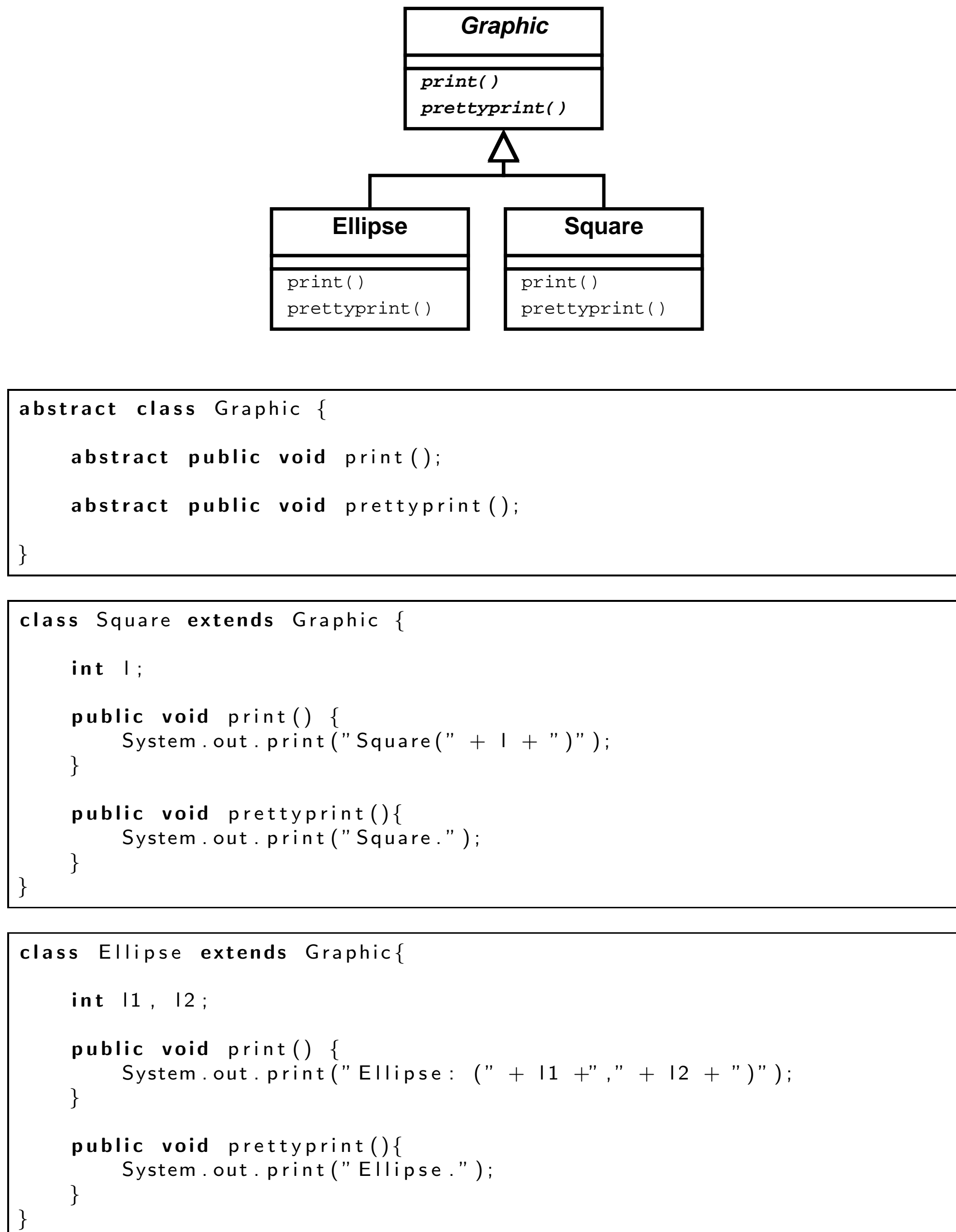
In this report we do preliminary work before automating refactoring based Composite↔Visitor transformations:

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.");
    }
}
```

Figure 1: Base Program (classic class hierarchy)

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

Here, $\mathsf{LM} = \{\mathsf{print},\mathsf{prettyprint}\}$, $\mathsf{LC} = \{\mathsf{Ellipse},\mathsf{Square}\}$ and $\mathsf{S}=\mathsf{Graphic}$.

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

```
1. ForAll m in LM, c in LC do
     Let visitorname = V(m) in
       MoveMethodWithDelegate(c, m, visitorname)
       RenameMethod(visitorname, m, "visit")
   done

2. AddAbstractSuperClass("Visitor", LV)

3. ForAll c in LC do
     PullUpAbstract(LV, "visit", c, "Visitor")

4. ForAll c in LC do
     ExtractMethod(c, LM, "accept")

5. ForAll m in LM do
     PullUpConcrete(LC, m, S)
```

Figure 2: Simple Class Hierarchy → Visitor transformation [MT04].

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[1] from the class hierarchy to visitor classes (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 we keep the original methods as delegators to visitor's methods in order not to change the interface of the class hierarchy (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 characteristic 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 operator. In the same way, these general guidelines (Fig. 2) must be *interpreted* by someone which will adapt them to his particular program.

We now consider that the operator uses a refactoring tool. We consider IntelliJ IDEA but the same remarks will apply to Eclipse unless otherwise stated.

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 (parameters or body).

The reason is that the receiver object cannot be inferred (this is an instance method).

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

[1] We call business code the code that defines the operations, here *print* and *prettyprint*, which is spread over several classes (with 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)

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).

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, that operation cannot be applied to several classes simultaneously. We have to extract a superclass from one class, then introduce inheritance manually. Since that operation is supported in Eclipse, there is good hope that this feature could be implemented in IntelliJ IDEA, otherwise, we can still use Eclipse for this step.[2]

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

```
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

[2]Eclipse supports *Extract Superclass* for several classes, but not *Extract Interface* for several classes, and there is a non-blocking bug on the introduction of *@Override* annotations.

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 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.25.

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.

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 (at the moment the algorithm is not automatic).

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 methods to handle take no parameter and do not return any result, and the traversal process is stateless.

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:

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

```java
abstract class Graphic {

        abstract public void print();

        abstract public void prettyprint();

}
```

```java
class Ellipse extends Graphic{

  public void print() {
    System.out.println("Ellipse :" + this);
  }

  public void prettyprint(){
    System.out.println("Ellipse corresponding to the object " + this + ".");
  }
}
```

```java
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)

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

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

    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 can be found problematic for the following reason: if we look at the definition of *Graphic.print()* (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 anticipate on the fact 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,[3] which impacts the rest of the algorithm (for instance, the pull-up of step 1 is already done).

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→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) =printAux and *aux*(prettyprint) =prettyprintAux.[4]

Note that two bugs are encountered with IntelliJ IDEA 10.5.2 in this algorithm (see *MoveMethodWithDelegate* and *GeneraliseParameter*, App. A). Until these two bugs are solved, a manual intervention is needed.

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

3.2 Visitor→Composite Transformation

Composite→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 steps (Fig 10):

1. We replace dynamic dispatch with static dispatch.
2. We in-line the business code from the *visitor* structure to the *composite* structure.
3. We 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.

[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.

[4] Of course, we should ensure that these names are not clashing with other names in the project.

1. ForAll m in LM do
   Let visitorname = V(m) in
   CreateEmptyClass(visitorname)
2. ForAll m in LM do
   Let auxname = aux(m) in
   CreateIndirectionInSuperClass(S,m, auxname)
3. ForAll m in LM, c in LC do
   Let auxname = aux(m) in
   InlineMethodInvocations(c, m, auxname)
4. ForAll m in LM do
   Let visitorname = V(m) and auxname = aux(m) in
   AddParameterWithReuse(S, auxname, visitorname, new visitorname())
5. ForAll m in LM, c in LC do
   Let visitorname = V(m) and auxname = aux(m) in
   MoveMethodWithDelegate(c, auxname, visitorname, "visit")
6. ExtractSuperClass(LV, "Visitor")
7. ForAll m in LM do
   Let visitorname = V(m) and auxname = aux(m) in
   GeneraliseParameter(S, auxname, visitorname, "Visitor")
8. Let LAUX = { aux(m) }$_{\mathsf{m} \in \mathsf{LM}}$ in
   MergeDuplicateMethods(S, LAUX, "accept")

Figure 7: Base Composite→Visitor transformation

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).

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. 14).

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).

```
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 :" + 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 9: Program with Visitor (visitor classes)

1. ForAll v in LV do
   addSpecializedMethodInHierarchy(S,accept,"Visitor", v)

   deleteMethodInHierarchy(S,accept,"Visitor")

2. ForAll c in LC do
   pushDownAll("Visitor","visit",c)

3. ForAll v in LV, c in LC do
   InlineMethod(v,visit,c)

4. ForAll m in LM do
   renameMethod(S,accept,V(m),aux(m))

5. ForAll m in LM do
   removeParameter(S,aux(m),V(m))

6. ForAll m in LM do
   replaceMethodDuplication(S,m)

7. ForAll m in LM do
   pushDownImplementation(S,m)

8. ForAll m in LM do
   pushDownAll(S,aux(m))

9. ForAll m in LM, c in LC do
   inlineMethod(c,aux(m))

10. ForAll v in LV 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)

```java
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);
}
```

```java
class Ellipse extends Graphic{

    public void accept(PrettyPrintVisitor v) {
        System.out.println("Ellipse corresponding to the object " + this + ".");    }

    public void accept(PrintVisitor v) {
        System.out.println("Ellipse :" + this);
    }
}
```

```java
class CompositeGraphic extends Graphic {

    ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();

    public void accept(PrettyPrintVisitor v) {
        System.out.println("Composite " + this + " composed of:");
        for (Graphic graphic : mChildGraphics) {
            graphic.accept(v);
        }
        System.out.println("(end of composite)");
    }

    public void accept(PrintVisitor v) {
        System.out.println("Composite:");
        for (Graphic graphic : mChildGraphics) {
            graphic.accept(v);
        }
    }

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

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

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 :" + this);
    }

    public void prettyprint() {
        System.out.println("Ellipse corresponding to the object " + 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);
    }

    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)");
    }
}
```

Figure 14: Result after Back Transformations

4 Variants of Transformations for Various Pattern Instances

In this section we present many structures or variants of either Composite pattern or Visitor pattern. At the same time we try to apply the basic algorithm of the switching among the two patterns in order to satisfy the transformation of each variant of the indicated design patterns.

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 our example (see Fig. 15).

4.1.1 Composite→Visitor Transformation

```
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.

At the step 4 of the Composite→Visitor algorithm of Fig. 7, replace the application of the operation *addParameterWithReuse* 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

```
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> getmChildGraphics() {
        return mChildGraphics;
 }
}
```

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

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)).

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:

```
    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,

```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 with color:"+ ellipse.color); }
}
```

```java
public class SetColorVisitor extends Visitor{
    private final int c;
    public SetColorVisitor(int c) {
        this.c = c;             }
    public int getC() {
        return c;      }
    public void visit(CompositeGraphic compositeGraphic) {
        for (Graphic graphic : compositeGraphic.mChildGraphics) {
            graphic.accept(this);
        }    }
    public void visit(Ellipse ellipse) {
        ellipse.color = getC();
    }
}
```

Figure 17: Visitor with methods having parameters (visitor classes)

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).

4.2.1 Composite→Visitor Transformation

At the step 6 of the base algorithm of Fig. 7, we use the operation *ExtractSuperWithGenerics* (see App. A.10). This operation is used to extract a super-class that supports 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 1 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.

```
abstract class Graphic{
    public abstract Integer perimeter ();
    public abstract String toString ();

}
```

```
class Ellipse extends Graphic{
  int perimeter;

  public Ellipse (int perimeter){
    this.perimeter=perimeter ;};

    public Integer perimeter () {
        return (perimeter);
    }

    public String  toString () {
        return ("Ellipse : " + Integer.toString(perimeter));
    }
}
```

```
class CompositeGraphic extends Graphic {

    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();

    public Integer perimeter() {
        int acc = 0 ;
        for (Graphic graphic : mChildGraphics) {
            acc += graphic.perimeter();
        }
        return acc;
    }

    public String toString(){
        String s ;
        s = new String ("Composite with: ");
        for (Graphic graphic : mChildGraphics) {
            s = s.concat(graphic.toString() + ", ");
        }
        System.out.println("(end)");
        return s;
    }

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

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

Figure 18: Composite with methods returning types

```java
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);
}
```

```java
class Ellipse extends Graphic{
  int perimeter;
  public Ellipse (int perimeter){
    this.perimeter=perimeter ;};

    public <T> T accept(Visitor<T> v) {
        return  v.visit(this);
    }
}
```

```java
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)

```java
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);
  }
}
```

```java
public class ToStringVisitor extends Visitor <String> {
  public String visit(CompositeGraphic compositeGraphic) {
   String s ;
   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)

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

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

```java
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> getmChildGraphics() {
        return mChildGraphics;
  }
}
```

Figure 22: Visitor structure for Interface instead of Abstract Composite (data 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).

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

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

```java
public class PrettyPrintVisitor extends Visitor {

    public void visit(CompositeGraphic compositeGraphic) {
        System.out.println("Composite " + compositeGraphic + " composed of:");
        for (AbstractComposite 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 23: Visitor structure for Interface instead of Abstract Composite (visitor classes)

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. 7, apply the operation *pushDownNotRedefinedMethod* (see App. A.22) 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.

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.

```java
abstract class Graphic {

        abstract public void print();
        abstract public void prettyprint();

}
```

```java
class Ellipse extends Graphic{

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

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

```java
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 :" + ellipse);
    }
    public void visit(ColoredEllipse coloredEllipse) {
        System.out.println("Ellipse :" + coloredEllipse.color);
    }

}
```

```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 " + ellipse + ".");
    }
    public void visit(ColoredEllipse coloredEllipse) {
        System.out.println("Ellipse :" + coloredEllipse.color);
    }
}
```

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

5 Application to JHotDraw

To validate our transformation algorithms, we apply them to JHotDraw [GI]. JHotDraw has been made to illustrate the use of design patterns (this is still a toy example, but which is larger than the previous one and which is not tailored to fit our transformation).

To know on which classes to apply the transformation, we can apply 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 6 classes of the class hierarchy.

The Composite→Visitor transformation applies successfully with the help of variations studies in Sec. 4, except for primitive types which have to be transformed into object types for using generics (variation *Methods Returning Values* , Sec. 4.2).

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

It took between 8 and 9 hours to apply the whole Composite→Visitor transformation. Most of time time is due to interaction (selection the entities to transform, selecting the refactoring operation and giving parameters) and can be automated. The computing time (check preconditions, generate transformed code, save files) was between 3 and 4 minutes.

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 an user. As 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 [HKVDSV11].

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 here, 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 [CD11].

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.

[CD11] Julien Cohen and Rémi Douence. Views, Program Transformations, and the Evolutivity Problem in a Functional Language. Research Report hal-00481941, `http://hal.archives-ouvertes.fr/hal-00481941/en/`, 19 pages, 2011.

[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/`.

[HKVDSV11] Mark Hills, Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. A case of visitor versus interpreter pattern. In *Proceedings of the 49th international conference on Objects, models, components, patterns*, TOOLS'11, pages 228–243, Berlin, Heidelberg, 2011. Springer-Verlag.

[JLB02] Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan Bae. An automated refactoring approach to design pattern-based program transformations in Java programs. In *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, APSEC '02, pages 337–, Washington, DC, USA, 2002. IEEE Computer Society.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[KK04] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(Issues 1-3):9–51, Aug. 2004. Special Issue on Program Transformation.

[Koc02] Helge Koch. Ein refactoring-framework fr Java (in german). Diploma thesis, CS Dept. III, University of Bonn, Germany, April 2002.

[KSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society.

[MGL06] Naouel Moha, Yann-Gael Gueheneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 297–300, Washington, DC, USA, 2006. IEEE Computer Society.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30:126–139, February 2004.

[OC00] Mel Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. PhD thesis, Trinity College, Dublin, Oct. 2000.

[OCN99] M. Ó Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 463–, Washington, DC, USA, 1999. IEEE Computer Society.

[OCN00] Mel Ó Cinnéide and Paddy Nixon. Composite refactorings for Java programs, 2000.

[PRSK10] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *IEEE International Conference on Software Maintenance (ICSM)*, Sept. 2010.

[RJ04] J. Rajesh and D. Janakiram. Jiad: a tool to infer design patterns in refactoring. In *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '04, pages 227–237, New York, NY, USA, 2004. ACM.

[SS03] Jason M. Smith and David Stotts. SPQR: Flexible automated design pattern extraction from source code. In *18th IEEE Intl. Conf. on Automated Soft. Eng.*, pages 215–224. IEEE Computer Society Press, 2003.

[TCSH06] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, 32:896–909, November 2006.

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 172–181, New York, NY, USA, 2006. ACM.

A Refactoring Operations

In this appendix, we define refactoring operations we use in our transformations. For each operation, we describe its behavior, and 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). All preconditions dealing with name clashes are left implied.

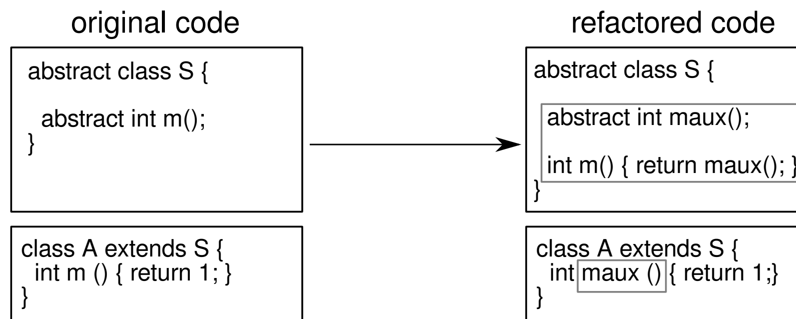
In addition, when operations take a method name as parameter, we consider that method name can be completed with parameter types to resolve overloading if needed.

A.1 CreateEmptyClass

CreateEmptyClass(classname c): Create an empty class c in the project.

Refactoring tools. *new Class* in Eclipse and IntelliJ IDEA.

A.2 CreateIndirectionInSuperClass



CreateIndirectionInSuperClass(class s, method m, newname n)

Refactoring tools. With IntelliJ IDEA: Use *Change Signature* on the method m in class s (specify to "delegate via overloading method", specify the new name n, specify the desired visibility).

With Eclipse:

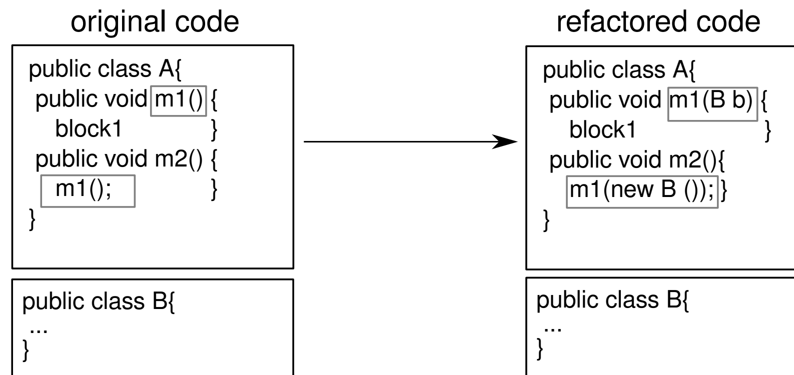
- Use *Change Method Signature* on the method m in class s (specify to "keep original method as delegate to changed method", and specify the new name n).
- Use *Pull Up* to remove the delegator method code that have been introduced in subclasses (the delegator code is the same in all the classes). Specify the method in the superclass must be removed replaced by the pulled up method, which is the same.
- Restore method invocations in client classes that have been changed (initial method invocations have been replaced by delegate method invocations that can be replaced by delegator invocations so that the initial client code is left unchanged).

  This step is manual, but it could be avoided by adapting Eclipse operation so that the client code is left unchanged when the change in the method signature is hidden with a delegator.

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.



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

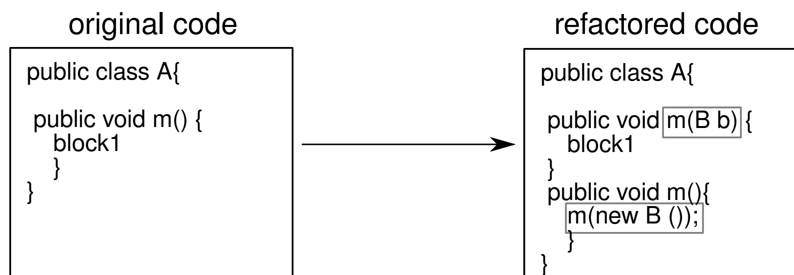
A.4 AddParameterWithReuse

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.

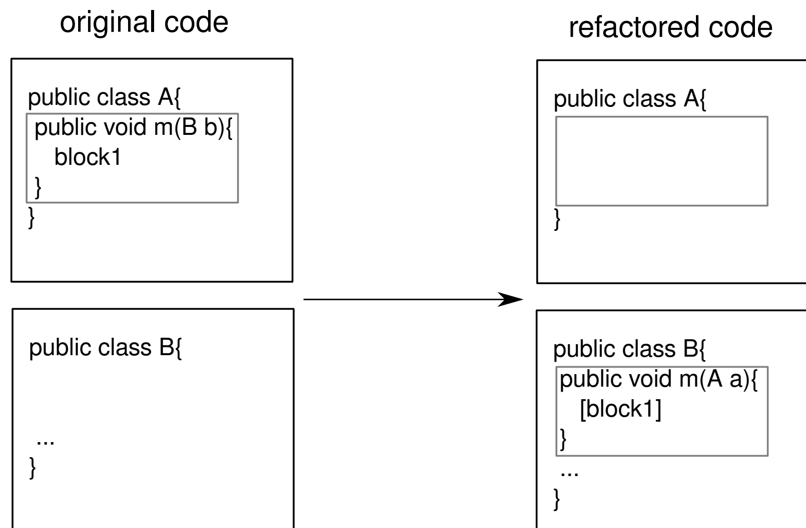
In IntelliJ IDEA, this is specified with the *Any Var* option in *Change Signature*. This is not supported by Eclipse.

Note that when several variables of the specified type are visible, the result in 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.

A.5 AddParameterWithDelegate



A.6 MoveMethod

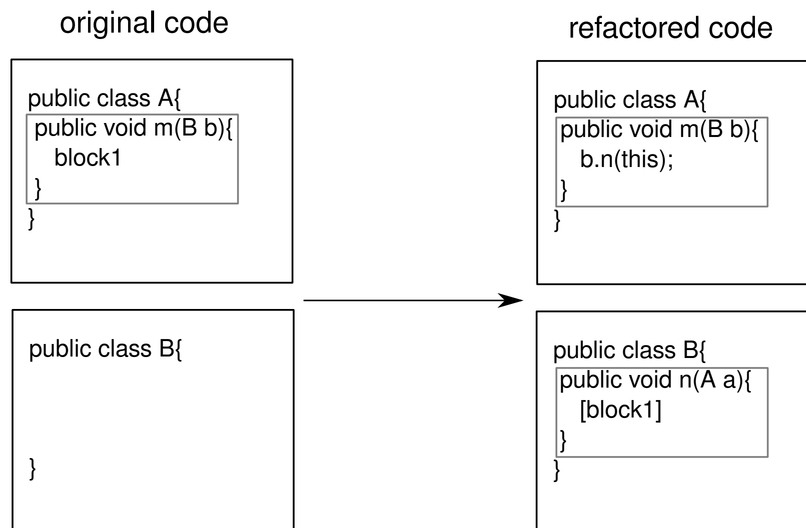


Refactoring tools. If the receiver object is not used in the body of the initial class, it will not be included as parameter in the destination class, so that you have to add it (see *AddParameter*).

A.7 MoveMethodWithDelegate

(*Move Method* in Fowler [Fow99])

MoveMethodWithDelegate(class c, method m, targetclass t, newname n): Transform a method m of a class c into a delegator to a method n in an other class t. The code of m has been moved to n (and adapted).



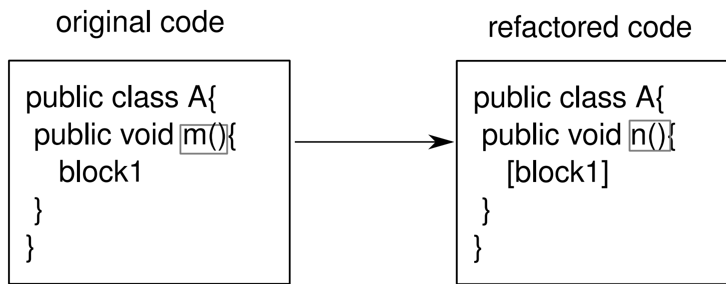
Refactoring tools. *Move* in Eclipse tool. In IntelliJ IDEA, first introduce a local delegate (with *Change Signature*), then *Move*.

Preconditions: An object of the destination class must appear as a parameter of the method m.

A.8 RenameMethod

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

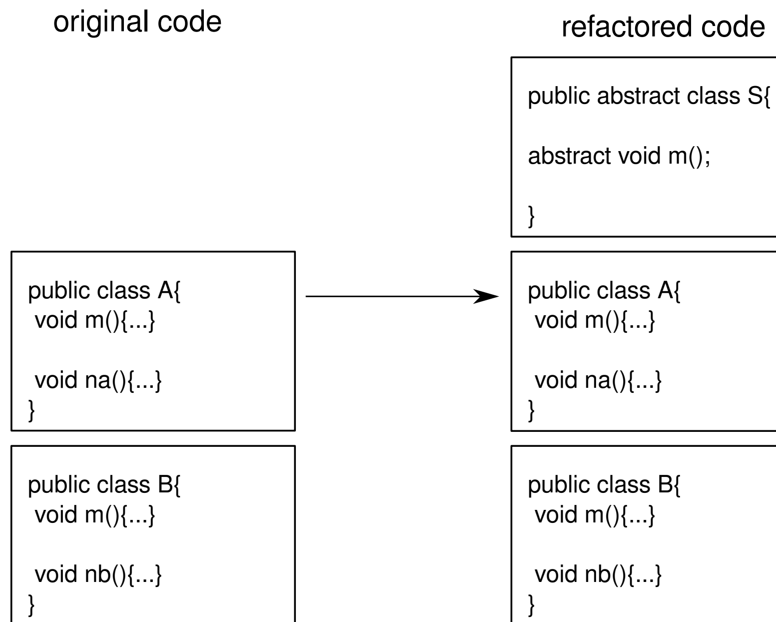
RenameMethod(class c, method m, newname n): Rename the method m of class m into n.



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

A.9 ExtractSuperClass

(*Extract Super Class* in Fowler [Fow99] and [Koc02])
ExtractSuperClass(set of classes C, newname s)



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 that the inheritance link must be set manually.

A.10 ExtractSuperClassWithGenerics

Same as *ExtractSuperClass* but unify different types in method parameters and return types by using parametric polymorphism (Java *Generic* types).

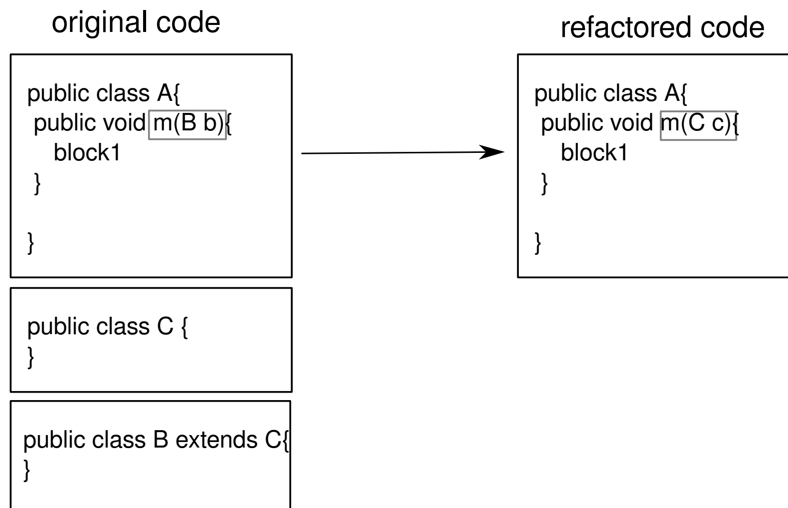
Refactoring tools. This is not supported by Eclipse nor IntelliJ IDEA.

A.11 GeneraliseParameter

GeneraliseParameter(class c, method m, type t, newtype s)
Modify the parameter type `t` of method `m` in class `c` by the type `s`.

Preconditions:

- The type `t` is a subtype of `s`.
- All method invocations on the parameter of type `t` in the body of the method must be possible on `s`.

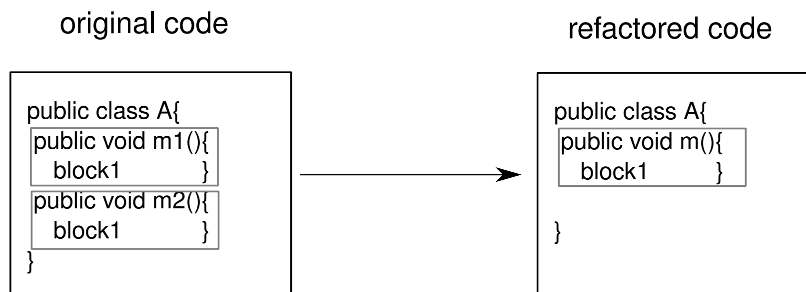


Refactoring tools. *Change Method Signature* in Eclipse tool and *Type Migration* in IntelliJ IDEA (or *Change Signature*).

A.12 MergeDuplicateMethods

MergeDuplicateMethods(class c, methods M, newname n)

Create one method which will replace a set of methods that have the same body.



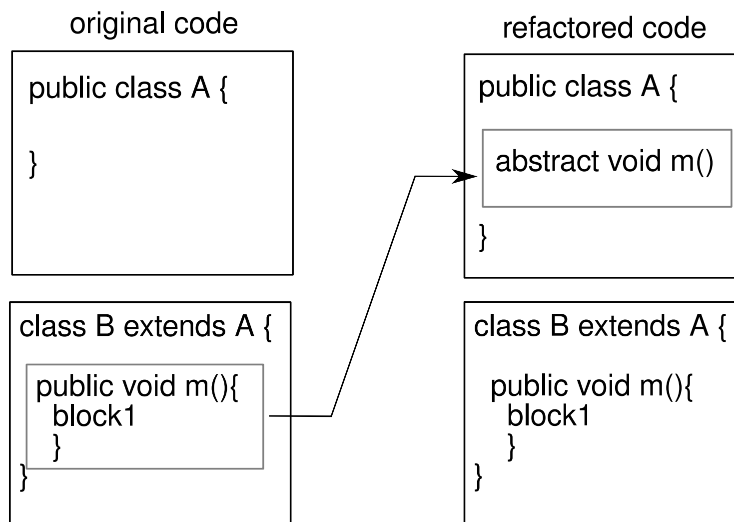
Refactoring tools. *Rename, Replace Method duplication, Extract Method, In-line* in Eclipse, *Rename, Replace Method Code Duplicates, Pull Up, Inline* in IntelliJ IDEA.

Preconditions: The two concerned methods must be semantically equivalent.

A.13 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.14 PullUpConcrete

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

Pull up a method which has the same implementation in a set of classes C to their superclass s: move the definition to s and remove it from the classes of C

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

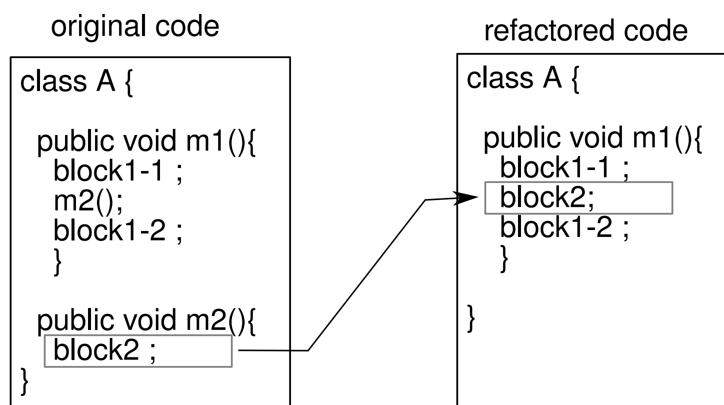
Preconditions:

- s is a superclass of each class in C.
- If m is defined is several classes of C, the code is the same.

A.15 InlineMethod

(*Inline Method* in [Fow99])

InlineMethod(class c, method m, types): 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.

Preconditions: The method is not polymorphic (abstract or overridden) [Fow99].

A.16 InlineMethodInvocations

Inline only some invocations, do not delete the method definition.

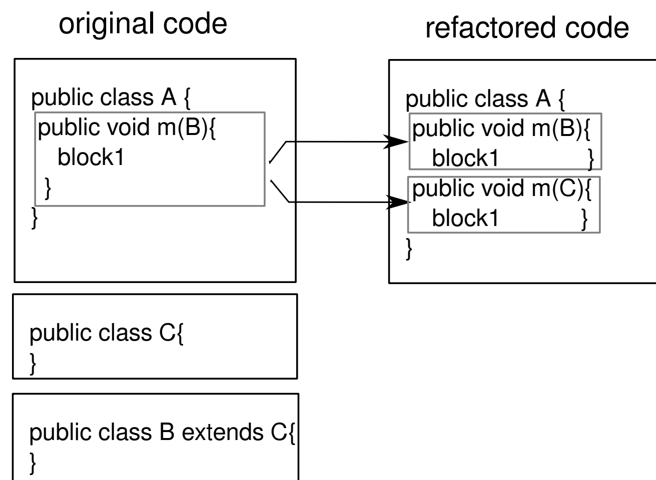
Refactoring tools. Inline in Eclipse and IntelliJ IDEA: select an invocation to inline and specify you want to inline only that one.

Preconditions: The method is not polymorphic (abstract or overridden) [Fow99].

A.17 AddSpecializedMethodInHierarchy

AddSpecializedMethodInHierarchy(class s, method m, type t, subtype t'): Get a new method from an existing method m by specializing one of its parameters of type t into a type t' which is a subtype of t.

This new duplication takes place in s and in all its subclasses that override m.



Refactoring tools. With IntelliJ IDEA:

1. Apply DuplicateMethodInHierarchy(c, m, temp-name) (see 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*. Here, the renaming introduces an overloading that could change the semantics of the program, but in this case, since the two methods have the same body, the behavior is preserved (some invocation may be dispatched on the new method, but the external behavior is the same).

Preconditions: t' is a subtype of t.

A.18 DuplicateMethodInHierarchy

Used only in AddSpecializedMethodInHierarchy.

DuplicateMethodInHierarchy(class c, method m, newname n)

This creates a duplicate of the method m in the class c with a the name n. All overriding methods in subclasses are also duplicated in these classes.

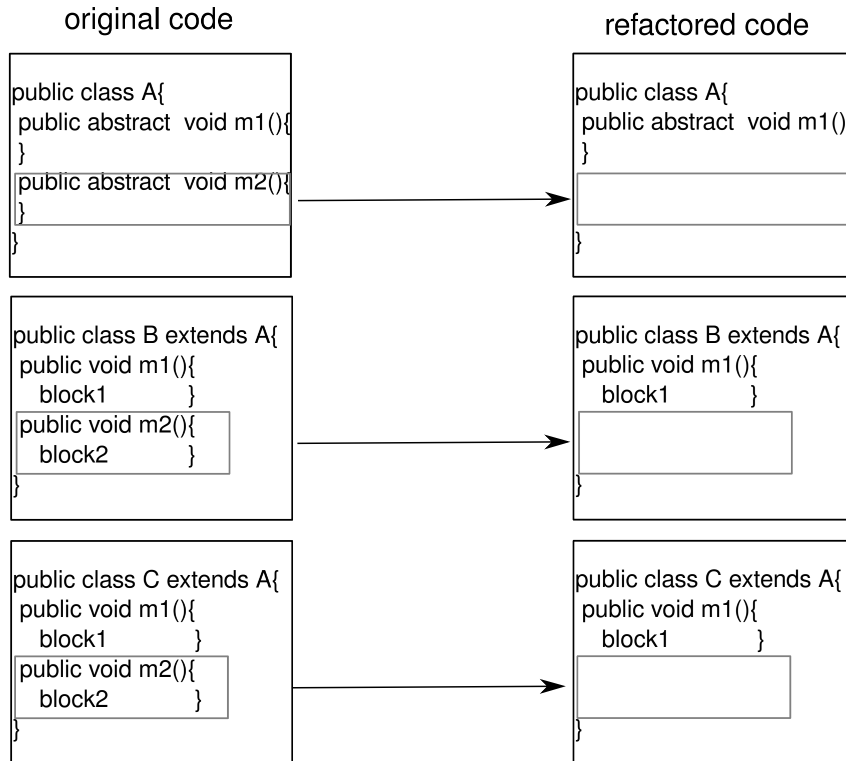
Refactoring tools. With IntelliJ IDEA:

1. For each implementation of the method m in c and its subclasses, 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*).

A.19 DeleteMethodInHierarchy

(*Delete Method* in Fowler [Fow99] and [Koc02])
DeleteMethodInHierarchy(class c, method m): Delete a method m from a class c and its subclasses.

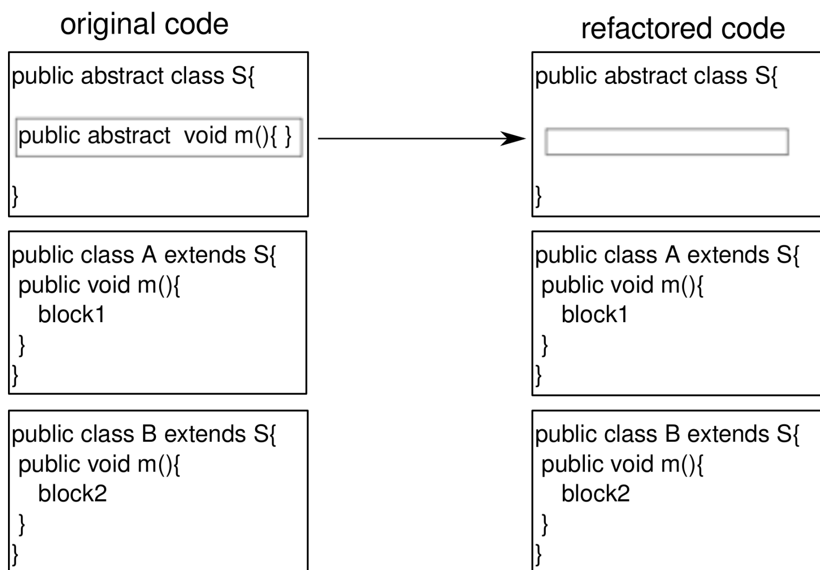


Refactoring tools. *Safe Delete* in IntelliJ IDEA and *Delete* in Eclipse.

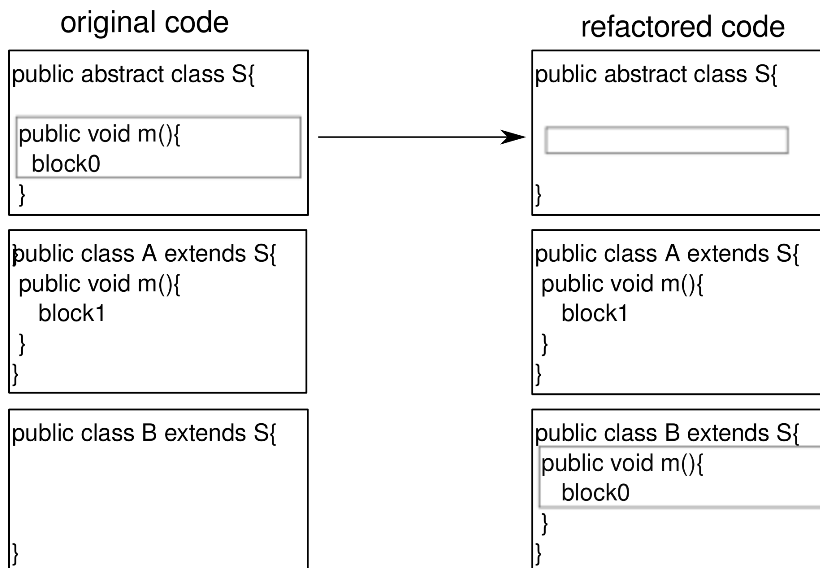
Preconditions: The method to be deleted must not be used.

A.20 PushDownAll

PushDownAll(class s, method m): Push down a method m from a class s to all 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:

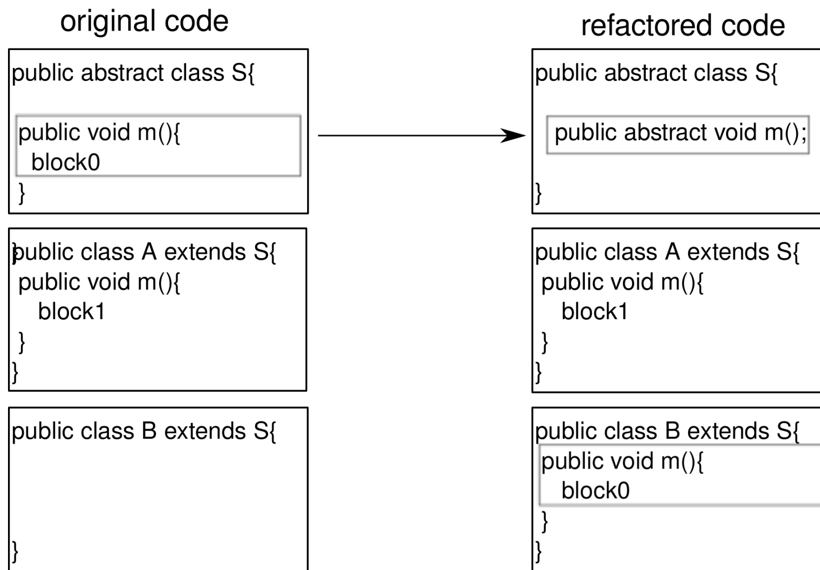


Refactoring tools. *Push Down* or *Push member Down* in Eclipse tool and IntelliJ IDEA.

Preconditions: The concerned method should not be used for the type `s`.

A.21 PushDownImplementation

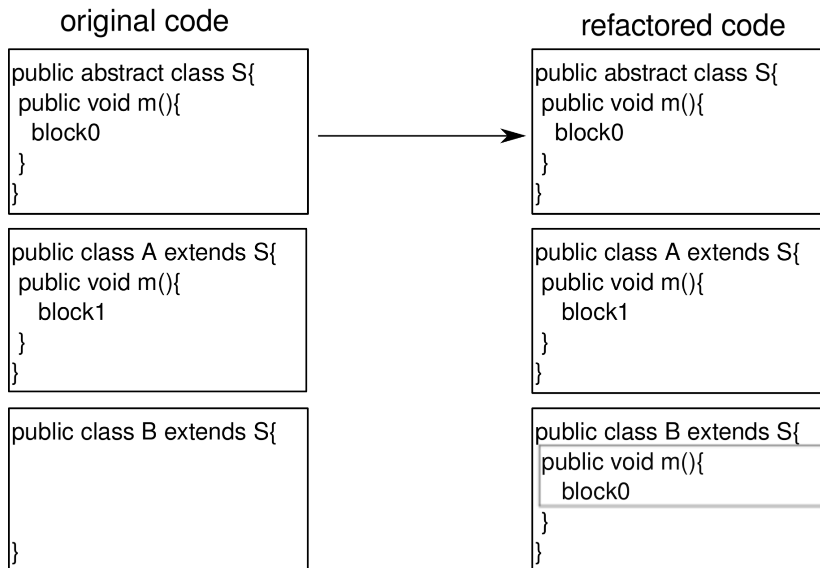
Same as *PushDownAll* but keep the method abstract in the superclass.



Preconditions: The method is not abstract.

A.22 pushDownNotRedefinedMethod

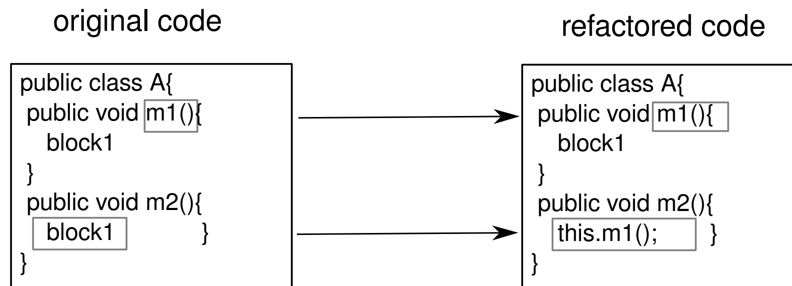
pushDownNotRedefinedMethod(class c, method m)
Duplicate the method m of class c into its subclasses.



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

A.23 ReplaceMethodDuplication

ReplaceMethodDuplication(class c, method m): Replace any occurrence of method m's body in c by an invocation of that method.



Refactoring tools. *Replace Method Duplication* in IntelliJ IDEA.

Preconditions: The method m must not be abstract.

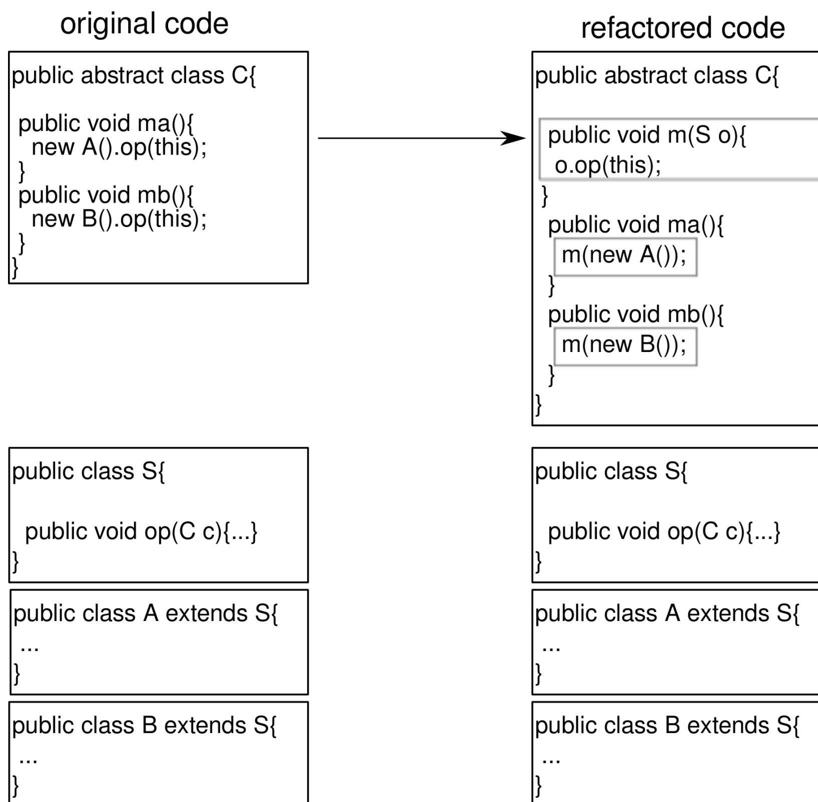
A.24 DeleteClass

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

Preconditions: The class is not referenced in the project.

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

A.25 ExtractGeneralMethod



A.26 InlineClass

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

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