Transformations between Composite and Visitor implementations in Java

Akram Ajouli, Julien Cohen, Jean-Claude Royer

Transformations between Composite and Visitor implementations in Java

Akram Ajouli
ASCOLA (EMN - INRIA - LINA)
Nantes, France
Akram.Ajouli@mines-nantes.fr

Julien Cohen
LINA (UMR 6241, CNRS, Univ. Nantes, EMN)
Nantes, France
Julien.Cohen@univ-nantes.fr

Jean-Claude Royer
ASCOLA (EMN - INRIA - LINA)
Nantes, France
Jean-Claude.Royer@mines-nantes.fr

***Abstract*—Basic automated refactoring operations can be chained to perform complex structure transformations. This is useful for recovering the initial architecture of a source code which has been degenerated with successive evolutions during its maintenance lifetime. This is also useful for changing the structure of a program so that a maintenance task at hand becomes modular when it would be initially crosscutting.**

We focus on programs structured according to Composite and Visitor design patterns, which have dual properties with respect to modularity. We consider a refactoring-based round-trip transformation between these two structures and we study how that transformation is impacted by four variations in the implementation of these patterns. We validate that study by computing the smallest preconditions for the resulting transformations. We also automate the transformation and apply it to JHotDraw, where the studied variations occur.

***Keywords*-refactoring; design patterns**

I. Introduction

The complexity of program maintenance depends on the quality of its structure. But, for a maintenance task at hand, some design choices that cannot be qualified as good or bad could also impact that complexity. This is illustrated by the case of Composite and Visitor patterns. Despite their usefulness in facilitating reuse and maintainability, each one is more suitable for a specific kind of maintenance. While the Composite (as well as Interpreter pattern and simple class hierarchies) offers modular maintenance with respect to data types, the Visitor pattern provides modular maintenance with respect to functions [5]. These two patterns can be good at design time because you usually do not know at that moment if you will face more maintenance on the data axis or on the function axis in the future. These (micro-)architectures are complementary. Automatic switching between the two structures at the source code level allows to benefit from the best pattern with respect to a maintenance task at hand [4].

Such a behavior-preserving transformation is given by Ajouli [1] and has been validated by a static analysis by Cohen and Ajouli [3]. However, that transformation is designed for a given implementation of the pattern (methods with no parameters, no returned values, abstract class at the root of the composite structure and a single level in the composite hierarchy). Since various design choices for implementing those patterns are found in real softwares, the described transformation does not apply directly.

Our contribution in this paper is the extension of the transformation presented by Ajouli [1] to take into account four variations in the implementations of the Composite and Visitor patterns. We validate the built transformations by computing their minimum preconditions and by applying them to JHotDraw in which the four variations occur.

For each variation considered on the Composite pattern, we discuss how it is reflected on the dual Visitor implementation, on the round-trip transformation and on its preconditions (Sec. III). Then, we validate these changes in the transformation algorithms by applying them to JHotDraw both practically and formally (Sec. IV). Before addressing the variations of the patterns and transformations, we review the basic transformation on a toy example in Sec. II.

II. Transformation between Composite and Visitor

To illustrate each pattern variation, we start with a toy implementation of the Composite pattern, given in Fig. 1. This program is composed by an abstract class, `Figure`, with two declared business methods, `print` and `show`, and two subclasses. One of these classes, `Group`, contains references to `Figure` objects, and the business methods in that classes make recursive invocations on those objects. This is a simple implementation of the Composite pattern. Fig. 2 gives a program with the same semantics but which implements the Visitor pattern.

We consider the two transformations given in Fig. 3 from [1] to switch between these two implementations of the Composite and Visitor structures. These transformations are built by composing elementary behavior-preserving refactoring operations from IntelliJ IDEA (a similar transformation is also possible with the refactoring operations of Eclipse). This composition is based on a meaningful orchestration of these operations in order to get the right structure (briefly explain below). The chain of operations is automated by using the API of IntelliJ IDEA (some refactoring operations are extended or modified in order to satisfy the fully automation of the transformation).

```
abstract class Figure {

  abstract  void print();

  abstract  void show();}
```

```
class Rectangle extends Figure{

 void print(){System.out.println("Rectangle");}

 void show(){System.out.println("Rectangle:" + this);}}
```

```
class Group extends Figure {
 ArrayList<Figure> children
        = new ArrayList<Figure> () ;
 void print() {
   System.out.println("Group:");
   for (Figure child : children) {child.print();}}

 void show(){
   System.out.println("Group " + this);
   for (Figure child : children) {child.show();}
   System.out.println("(end)");}}
```

Figure 1. A program structured according to Composite Design pattern.

```
abstract class Figure {
 void print(){accept(new PrintVisitor());}

 void show(){accept(new ShowVisitor());}

 abstract void accept(Visitor v);}
```

```
class Rectangle extends Figure{

 void accept(Visitor v){v.visit(this);}}
```

```
class Group extends Figure {
 ArrayList<Figure> children = new ArrayList<Figure>();

 void accept(Visitor v) {v.visit(this);}}
```

(a) Data classes.

```
abstract class Visitor {

 abstract void visit(Rectangle r);

 abstract void visit(Group g);}
```

```
class PrintVisitor extends Visitor {
 void visit(Rectangle r){System.out.println("Rectangle");}

 void visit(Group g) {
  System.out.println("Group:");
  for (Figure child : g.children){child.accept(this);}}}
```

```
class ShowVisitor extends Visitor{
 void visit(Rectangle r){
                System.out.println("Rectangle:"+r);}
 void visit(Group g){
  System.out.println("Group "+g);
  for (Figure child : g.children){child.accept(this);}
  System.out.println("(end)");}}
```

(b) Visitor classes.

Figure 2. Visitor structure of the program shown in the Fig. 1.

1) ForAll m in $\mathbb{M}$ do **CreateEmptyClass**(vis(m))
2) ForAll m in $\mathbb{M}$ do **CreateIndirectionInSuperClass**(S,m, aux(m))
3) ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do **InlineMethodInvocations**(c, m, aux(m))
4) ForAll m in $\mathbb{M}$ do **AddParameterWithReuse**(S, aux(m), vis(m))
5) ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do **MoveMethodWithDelegate**(c, aux(m), vis(m), "visit")
6) **ExtractSuperClass**($\mathbb{V}$, "Visitor")
7) ForAll m in $\mathbb{M}$ do **UseSuperType**(S, aux(m), vis(m), "Visitor")
8) **MergeDuplicateMethods**(S, $\{aux(\text{m})\}_{\text{m}\in\mathbb{M}}$, "accept")

(a) Base Composite→Visitor transformation.

I) ForAll v in $\mathbb{V}$ do **AddSpecializedMethodInHierarchy**(S, "accept","Visitor",v)
II) **DeleteMethodInHierarchy**(S,accept,"Visitor")
III) ForAll c in $\mathbb{C}$ do **PushDownAll**("Visitor","visit",c)
IV) ForAll v in $\mathbb{V}$, c in $\mathbb{C}$ do **InlineMethod**(v,"visit",c)
V) ForAll m in $\mathbb{M}$ do **RenameMethod**(S,accept,vis(m),aux(m))
VI) ForAll m in $\mathbb{M}$ do **RemoveParameter**(S,aux(m),vis(m))
VII) ForAll m in $\mathbb{M}$ do **ReplaceMethodDuplication**(S,m)
VIII) ForAll m in $\mathbb{M}$ do **PushDownImplementation**(S,m)
IX) ForAll m in $\mathbb{M}$ do **PushDownAll**(S,aux(m))
X) ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do **InlineMethod**(c,aux(m))
XI) ForAll v in $\mathbb{V}$ do **DeleteClass**(v)
XII) **DeleteClass**("Visitor")

(b) Base Visitor→Composite transformation.

Figure 3. Base Algorithms for reversible transformation from Composite to Visitor.

We use the following notations to abstract the algorithm from the given example:

- $\mathbb{M}$: set of business methods, here $\mathbb{M}$ ={print,show}.
- $\mathbb{C}$: set of Composite hierarchy classes except its root, here $\mathbb{C}$ ={Rectangle, Group}
- S: root of the Composite hierarchy, here S = Figure.
- vis: function that generates a visitor class name from a business method name, here vis(print) = PrintVisitor.
- $\mathbb{V}$: set of visitor classes, here $\mathbb{V} = \{vis(m)\}_{m\in\mathbb{M}}$ ={PrintVisitor, ShowVisitor}.
- aux: function used to generate names of temporary methods, here aux(print) = printAux.

A. Composite to Visitor

The Composite→Visitor algorithm of Fig. 3(a) is explained with three stages: preparing for moving business code (steps 1 to 4); moving the business code to the Visitor classes (step 5) and recovering the conventional structure of the Visitor pattern (steps 6 to 8).

Steps 1 to 4: Preparing for moving business code: First, we create an empty visitor class (step 1) for each business method of the program (PrintVisitor and ShowVisitor). Then, in order to preserve the interface, we introduce a delegator for the business methods (step 2). The initial business method is now split into the *delegator* which keeps the name of the business methods, its type and its defining class and the *business code* contained in the deleguee method. In the following, we call *auxiliary* methods those deleguee methods that contain the business code.

The introduction of delegators have transformed direct recursive invocations into indirect recursion. We replace invocations of delegators by invocations of auxiliary methods in business code to recover direct recursion and avoid using the delegator in business code (step 3).

Finally, to be able to move business code to the right visitor classes, we introduce into each auxiliary method its target class by adding to it the visitor class type and name as a dummy parameter (step 4).

Step 5: Moving business code to visitor classes: Move auxiliary methods containing the business code to visitor classes and rename them into "visit". The famous double dispatch involved in the Visitor pattern is created by keeping (again) a delegator in the originating class for each moved method.

Steps 6 to 8: Recovering Visitor structure: We extract a super-class for visitor classes (step 6). That class will contain the abstract declarations of the *visit* methods. In the delegators (which are in the Composite side), we change the type of the parameter to the new super-class `Visitor` (step 7). Finally, we unify all delegators in Composite side into a single method *accept* (step 8).[1]

The resulting program is the one of Fig. 2 which implements the Visitor pattern.

B. Visitor to Composite

The base Visitor→Composite transformation is given in Fig. 3(b). Again, the key point is to moving the business code back to the composite hierarchy. For this reason, we explain the whole algorithm again in three stages: preparing the move, performing the move and recovering the target structure.

Steps I and II: Preparing for moving business code: Duplicate the method `accept(Visitor)` in the whole composite hierarchy into overloaded methods (see footnote 1). Each overloaded method takes one of the visitor classes as parameter (step I). These methods perform the same code as the initial `accept(Visitor)` method, which is deleted since its role is delegated to the new methods (end of step I). At this point, all the invocations of the *visit* method are done on subclasses of `Visitor`, so that we can delete the abstract declaration of *visit* methods from the abstract `Visitor` (step III). This will allow to inline visit methods in composite classes (next step).

[1]**MergeDuplicateMethods** and **AddSpecializedMethodInHierarchy** are composite refactoring operations described in [2].

Step III: Move business code to Composite classes: Inline the *visit* methods of the visitor classes and delete them from these classes. This boils down to moving the business code back to composite classes (inside what was delegators before). At this point, the visitor classes are empty (no methods). They cannot be deleted now because they are still referenced, they are deleted in the two last steps.

Steps IV to XI: Recovering Composite structure: In all the hierarchy, rename overloaded methods *accept* (from step I) to methods with temporary different names to remove the overloading: the refactoring operation determines statically which instances of the overloaded methods was referred to by the delegator, and adjusts the invocation to the convenient deleguee accordingly (step V).

Then, remove the visitor parameters of these methods (step VI). Indeed, the parameter is not used, and the type of the parameter is not used anymore to resolve overloading.

Now we need to remove the delegation between the interface for the business method (delegator) and the business code (*deleguee*, renamed at step V):

- Replace any recursive invocation of the deleguee methods by invocations of the corresponding method (step VII).
- Push down the body of the delegators to the sub-classes (step VIII). This removes the dynamic dispatch which would prevent future inlining.
- To be able to delete the deleguee methods when they will be inlined, delete their declarations from the abstract super-class (step IX).
- Inline the deleguee methods in the concrete composites classes and delete their declarations (step X).
- Delete the visitor hierarchy since it is not used anymore (steps XI andXII).

After performing this transformation, we find back the initial program of Fig. 1 except a few changes in the layout and the comments.

C. Precondition

These transformations are validated in [3] by inferring a minimum precondition that ensures that all the preconditions of successive atomic refactoring operations will be satisfied at run time, and that after performing a round-trip transformation, the resulting program is in a state which satisfies the initial precondition, so that the transformation can be applied again.

The computation of the minimum precondition is based on the calculus of Kniesel and Koch [9] and on a formal description of the refactoring operations of IntelliJ IDEA. Those computed preconditions are valid under the hypothesis that the formal description of the operations are faithful with respect to the underlying tool (which has been tested but not formally proven).

The full precondition (available in [3]) is a conjunction of 49 predicates. Here is an extract of that precondition:

ExistsClass(Rectangle)
$\wedge$ ExistsMethodDefinition(Rectangle,show)
$\wedge\neg$ExistsMethodDefinition(Rectangle,accept)
$\wedge\neg$ExistsMethodDefinition(Rectangle,printAux)
$\wedge$ ...

For example, the proposition $\neg$ExistsMethodDefinition(Rectangle, printAux) is related to a temporary method name introduced in step 2 and indicates that such methods must not initially exist.

In the following, we always consider the round-trip transformation for the computation of the preconditions.

III. Pattern Variations

The previous transformation algorithms can be applied only to programs satisfying the computed preconditions. In particular, the business methods must have no parameter, must return *void*, the superclass in the data type must be an abstract class, and it must have only one level of subclasses. In the following, we relax these restrictions and show how the transformations take these changes into account.

A. Methods with parameters

1) Considered variation: We consider that some business methods in the Composite structure have parameters, as exemplified by the following method setColor :

```
// in Figure
abstract void setColor(int c)
```

```
// in Rectangle
int color;
void setColor(int c) { this.color = c; }
```

```
// in Group
void setColor(int c) {
   for (Figure child : children){child.setColor(c);} }
```

Note that the parameter c of the method setColor is passed to each recursive call (in the class Group).

2) Target structure: In the Visitor structure (Fig. 2), the visitor object, which is created by the interface methods of the class Figure, is passed recursively as parameter of *accept* and as receiver of *visit* invocations. So, to take the parameter c into account, we put it into the state of that visitor object, so that it is available during the traversal:

```
class SetColorVisitor extends Visitor{

 final int c;

 SetColorVisitor (int c){ this.c = c ; }

 void visit(Rectangle r){ r.color = c; }

 void visit(Group g){
  for(Figure child : g.children){child.accept(this);} }}
```

The method setColor of the Figure abstract class passes the parameter c to the constructor of the class SetColorVisitor, then passes the resulting visitor object (with c in its state) to the *accept* method:

```
// in Figure
void setColor(int c) { accept(new SetColorVisitor(c)); }
```

The implementation of *accept* in Rectangle and Group is left unchanged.

3) Composite→Visitor Transformation: The refactoring operation of step 4 of the basic transformation (Fig. 3(a)) add a visitor parameter to the methods that becomes *accept* later. Here, we do not want to add the visitor parameter to the initial method parameter (such as c), but we want to replace the initial parameter with the visitor. To do that we apply the operation **IntroduceParameterObject** (step 4.A below). Note that the refactoring operation **IntroduceParameterObject** could not be used with methods without parameters.

For that reason, we distinguish methods with parameters and methods without parameters and we introduce the following notation to introduce different treatments in the transformation algorithm:

- $\mathbb{M}_{\mathbb{P}}$: set of methods with parameters, here $\mathbb{M}_{\mathbb{P}} = \{$setColor(int c)$\}$.
- $\mathbb{M}_{\mathbb{W}}$: set of methods without parameters, with $\mathbb{M}_{\mathbb{P}} \cup \mathbb{M}_{\mathbb{W}} = \mathbb{M}$ and $\mathbb{M}_{\mathbb{P}} \cap \mathbb{M}_{\mathbb{W}} = \emptyset$.

Introducing a parameter object of type A to a method m(B b) for example creates a class A, moves the parameter b to A as an instance variable and finally changes m(B b) into m(A a). Any old access to b in the body of m will be replaced by a.b.

The initial step 1 is omitted for methods with parameters because the operation **IntroduceParameterObject** creates the new class (step 1.A below replaces step 1).

Here are the deviations from the basic algorithm for this variation:

1.A) ForAll m in $\mathbb{M}_{\mathbb{W}}$ do **CreateEmptyClass**(vis(m))
(replaces step 1)
4.A) ForAll m in $\mathbb{M}_{\mathbb{P}}$ do
IntroduceParameterObject(S, aux(m), vis(m))
ForAll m in $\mathbb{M}_{\mathbb{W}}$ do
AddParameterWithReuse(S, aux(m), vis(m))
(replaces step 4)

4) Visitor→Composite Transformation: Before deleting visitor classes (step XI) we have to check that there is no references to them in the Composite hierarchy. For the methods without parameters, we just remove the parameters corresponding to the visitor (step VI.A : restriction of step VI to methods without parameters) since at this moment those methods do not use that parameter. For example, at this moment (before step VI), the intermediate method for print in Rectangle is as follows:

```
// in Rectangle
void printaux(PrintVisitor v){
        System.out.println("Rectangle");}
```

For the methods with parameters, instead of deleting the visitor parameter, we have to inline the occurrences of visitor classes to recover the initial parameter c. After applying step X (before deleting visitor classes), the method setColor is as follows:

```
// in Rectangle
void setColor(int c){
      this.color = new SetColorVisitor(c).c;}
```

At this point we apply the operation **InlineParameterObject** which will replace new SetColorVisitor(c).c by c (step XI.A), and then we can delete visitor classes (step XII).

Here is the extension of the back transformation:

VI.A	ForAll m in $\mathbb{M}_{\mathbb{W}}$ do **RemoveParameter**(S,aux(m),vis(m))	(replaces step VI)
XI.A	ForAll m in $\mathbb{M}_{\mathbb{P}}$ do **InlineParameterObject**(S, aux(m), vis(m))	(before step XI)

5) Computed Precondition: The generated preconditions for this variations are related to the methods that have parameters, such as ExistsMethodDefinitionWithParams(Figure,setColor,[int c]). This constraint is imposed by the operation **IntroduceParameterObject** (step 4.A) since that operation works only with methods with parameters (the set $\mathbb{M}_{\mathbb{P}}$ contains setColor(int c)).

B. Methods with different return types

1) Considered variation: We consider now business methods with different return types. For example we consider a program with two methods: Integer eval() and String show().

2) Target Structure: Since we have methods with different return types, we cannot use void to the *accept* method. One solution is to have an *accept* method variant for each return type by the means of overloading. But this breaks the beauty of the Visitor pattern (one *accept* method for each business method instead of one *accept* method to implement an abstract traversal). To avoid that, we use generic types as done in Oliveira et al. [12]. In the abstract class Figure, the *accept* method becomes generic:

```
abstract <T> T accept(Visitor <T> v)
```

Note that the returned type is bound by the type of the visitor class which appears as parameter. Each visitor class represents a business method and its return type. The parameterized visitor structure is as follows:

```
abstract class Visitor <T> {...}
```

```
class EvalVisitor extends Visitor <Integer> {...}
```

```
class ShowVisitor extends Visitor <String> {...}
```

Remark: Because of the restriction in the use of generic types in Java, returned types which are raw types, such as int or bool, must be converted to object types such as Integer or Boolean. In the case of void, one can use Object and add a return null statement (we use a refactoring operation to do that).

3) Composite→Visitor Transformation: We use the following notations in the algorithm corresponding to this variation:

- $\mathbb{R}$: Set of methods and their corresponding return types, here $\mathbb{R}$ ={(show,String), (eval,Integer)}.

In step 6 of the basic algorithm, the operation **ExtractSuperClass** creates a new abstract class and pulls up abstract declarations of visit methods. In the considered variation, we have to use an extension of the pull up operation that introduces generic types in the super class to be able to insert abstract declarations for methods with different return types.

To deal with this variation we apply the operation **ExtractSuperClassWithoutPullUp** then the operation **PullUpWithGenerics**[2] instead of the operation **ExtractSuperClass** of the step 6 (step 6.B).

6.B	**ExtractSuperClassWithoutPullUp**($\mathbb{V}$, "Visitor") ; ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do **PullUpWithGenerics**(vis(m), "visit","Visitor")	(replaces 6)

4) Visitor→Composite Transformation: At the step I of the base algorithm, we must specify the return type of each *accept* method. The convenient return types could be identified directly from return types of visit methods existing in concrete visitors. This is done by the operation **AddSpecialisedMethodWithGenerics** (step I.B).

I.B	ForAll v in $\mathbb{V}$ do **AddSpecializedMethodWithGenerics**(S,"accept",$\mathbb{R}$, "Visitor",v)	(replaces I)

5) Computed Precondition: The only difference with the basic preconditions is the check of return types which should not be raw types. This precondition is required by the operation **PullUpWithGenerics**.

C. Hierarchy With multilevel

1) Considered variation: We consider that the Composite hierarchy has multiple levels, with a random repartition of business code: some business methods are inherited, and some other are overridden.

For example, we consider the class Rectangle has a subclass ColoredRectangle where the method print is overridden whereas the second method show is inherited:

```
class ColoredRectangle extends Rectangle{
 int color;
 ColoredRectangle (int c){ this.color = c; }

 void print{System.out.println(
      "Rectangle colored with " + color); } }
```

2) Target Structure: In order to have in visitor classes one *visit* method for each class of the Composite hierarchy, the code of the method show() defined in Rectangle in the Composite structure and inherited by ColoredRectangle, is placed in the methods visit(ColoredRectangle c) and visit(Rectangle r) in ShowVisitor:

[2]http://plugins.jetbrains.com/plugin/?idea_ce&id=6889

```
class ShowVisitor extends Visitor{
 void visit(Group g){...}

 void visit(Rectangle r){
           System.out.println("Rectangle :"+ r +".");}

 void visit(ColoredRectangle c){
          System.out.println("Rectangle :"+ c +".");}}
```

3) Composite→Visitor Transformation: In order to push down a duplicate of the inherited method to the right subclass, we apply the operation **PushDownCopy** (step 1.C) before running the basic algorithm.

We use the following notations in the algorithm corresponding to this variation:

- $i(c)$: a function that gives the list of inherited methods of a class ; here $i($ColoredRectangle$)$ = $\{$show()$\}$.
- $s(c)$: a function that gives the superclass of a class.

1.C	ForAll c in $\mathbb{C}$, ForAll m in $i(c)$ do **PushDownCopy**(c,m,$s(c)$)	(before 1)

4) Visitor→Composite Transformation: First we apply the basic algorithm. Then, in order to get back the initial structure we delete methods (step XII.C) that were initially added in these classes in the step 1.C of the forward transformation.

XII.C	ForAll (c,m) in $\mathbb{C}$, ForAll m in $i(c)$ do **DeleteMethod**(c,m)	(after XII)

Remark: The refactoring operation that performs that deletion should rely on the fact that the code of the deleted method is the same as the code in the super class.

5) Computed Precondition: The precondition that characterizes the transformation for this variation is: the method show must not be overridden in the class ColoredRectangle and must not call any overloaded method called with the argument this in the class Rectangle. This is due to the fact that, if the method Rectangle::show have any overloaded method with the argument this, we could not copy this method to the class ColoredRectangle since the argument this will refer to that class and which could change the behavior of this method which is supposed to keep the same behavior either in the class Rectangle or in the class ColoredRectangle.

D. Interface instead of Abstract Class

1) Considered Variation: We now consider that the root of the Composite hierarchy is not an abstract class but an interface and that there is an intermediary abstract class between it and its subclasses. This architecture is found in real softwares: libraries are often provided by the means of an interface and compiled byte-code (*Facade* pattern).

We suppose that there are no other subclasses implementing the interface.

```
interface Figure{
 void print();
}
```

```
abstract class AbstractFigure implements Figure {
 abstract void print();
}
```

```
class Group extends AbstractFigure{

ArrayList<Figure> children = ...

 void print(){
     ...
     for(Figure child : children){ child.print();}}
}
```

2) Target Structure: Here is a possible target structure corresponding to the considered variation:

```
interface Figure{
 void print();
 void accept(Visitor v);
}
```

```
abstract class AbstractFigure implements Figure{
 void print(){accept(new PrintVisitor());}
}
```

```
class Group extends AbstractFigure{
 ArrayList<Figure> children = ...
 void accept(Visitor v){v.visit(this);}
}
```

```
class PrintVisitor extends Visitor{
  void visit(Group g){
     for(Figure child : g.children){ child.accept(this);}}
}
```

Note that the loop in visit(Group) is done on objects of type Figure (not AbstractFigure).

3) Composite→Visitor Transformation: To reach the target structure, we have to create a delegator print(){printaux(..)} in the class AbstractFigure and inline the recursive call of print in Group (steps 2 and 3). But that recursive call refers to the method print declared in the Figure interface whereas the delegator is defined in the abstract class AbstractFigure. To solve that, we introduce a downcast to the class AbstractFigure in the recursive call to print as follows: ((AbstractFigure) child).print() (step 3.D). This makes the inlining by the refactoring tool possible. This downcast is legal because we suppose that the interface has no other implementation than the abstract class.

After creating the method accept (step 8), we pull up its declaration to the interface Figure, then we delete the downcast (step 8.D).

3.D	ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do **IntroduceDownCast**(c,m,S)	(before 3)
8.D	**pullupAbstractMethod**(S, "accept", I) ForAll v in $\mathbb{V}$ do **DeleteDownCast**(v,"accept")	(after 8)

Real practice of the transformation: The algorithms shown above represent the ideal solution to get a Visitor structure. In fact, there is no operation in the refactoring tools we use to manage downcasts. In order to automate the full transformation, we do not use downcasts and do not inline the delegator. As a result we get a Visitor with indirect recursion as follows:

```
// In Figure
   void print();
```

```
// In AbstractFigure
   abstract void accept(Visitor v);
   void print(){accept(new PrintVisitor();}
```

```
// In PrintVisitor
   void visit(Group g){
     for(Figure child : g.children){ child.print();}}
```

We can see that at each recursive invocation a new instance of a Visitor is created. The result is legal but shows a poor use of memory. This problem disappears when the initial Composite structure is recovered. Moreover, if needed, the downcast can be introduced manually (or the refactoring operation can be implemented).

So, in practice, the variation in the algorithm is: do not apply step 3 (nor 3.D); do not apply step 8.D (but step 8).

4) Visitor→Composite Transformation: After the *practical* Composite→Visitor transformation, the base Visitor→Composite transformation can be applied without performing the step VII.

After the full Composite→Visitor transformation described above (with downcasts), we also have to add and remove some downcasts to recover the Composite structure (before step VII and after VII, *the detail is not given for reason of space*).

5) Computed Precondition: We consider the practical algorithms without downcasts. Because we do not apply inlinings in step 3, the constraints IsRecursiveMethod(Group,print) and IsRecursiveMethod(Group,show) have disappeared from the computed minimum precondition.

E. Support for Precondition Generation

To generate the minimum preconditions to ensure the correctness of our transformations, we described 24 refactoring operations with 480 *backward description* rules (we use the concept of *backward descriptions* from the work of Kniesel and Koch [9]). The specification of each refactoring operation (preconditions and *backward descriptions*) are given in [2].

IV. USE CASE : JHOTDRAW

In this section we we apply our transformation to the JHotDraw framework.

1) Overview: In JHotDraw, there is a Composite structure with 18 classes and 6 business methods which shows the four variations presented above. We aliment the transformation algorithm with the following data:

- $S =$ AbstractFigure.
- $\mathbb{C} = \{$ EllipseFigure, DiamondFigure, RectangleFigure, RoundRectangleFigure, TriangleFigure, TextFigure, BezierFigure, TextAreaFigure, ... $\}$.
- $\mathbb{M}_{\mathbb{P}} = \{$ basicTransform (AffineTransform tx), contains(Point2D.Double p), setAttribute(AttributeKey key,Object value), findFigureInside(Point2D.Double p), addNotify(Const "Drawing d),removeNotify(Drawing d)$\}$.
- $\mathbb{M}_{\mathbb{W}} = \emptyset$.
- $\mathbb{R} = \{$ (basicTransform,Void), (contains,Boolean), (setAttribute, Void), (findFigureInside,Figure), (addNotify, Void), (removeNotify, Void)$\}$.
- $s($LineConnectionFigure$) = \{$BezierFigure$\}$
  $s($...$)=$ ...
- $i($LineConnectionFigure$) = \{$findFigureInside, setAttribute,contains$\}$,
  $i($...$) =$ ...

2) From Composite to Visitor: To switch from the Composite structure of JHotDraw to its Visitor structure we apply the following sequence of steps: 1.C ; 2 ; 4.A ; 5 ; 6.B ; 7 ; 8.

3) From Visitor to Composite: To recover the initial structure, we apply the following: steps I.B ; II; III; IV ; V ; VI.A; VIII ; IX ; X ; XI.A ; XI ; XII ; XII.C.

4) Generated Precondition: We have computed a minimum precondition that ensures the correctness of the round-trip transformation. That precondition, given in [2], is a conjunction of 1852 propositions.

V. RELATED WORK

Transforming Design Pattern Implementations: Roberts et al. [13] use sequences of basic refactoring operations to introduce design patterns in existing programs, including the Visitor pattern. Ó Cinnéide and Nixon [11] provide automatic preconditions generation for such sequences, without considering Visitors however. Hills et al. [6, 7] implement the reverse transformation: they remove the Visitor structure from a real interpreter. Kerievsky [8] provides a catalog of guidelines to introduce design patterns by refactoring sequences, including the Visitor.

Variations in Composite and Visitor implementations: Kerievsky [8] introduces Visitors in two variations of class hierarchies. The first one is close to the base architecture we consider (Sec. II), but misses some features of the Composite pattern: 1) the business methods initially do not have abstract declarations in the superclass of the hierarchy and 2) there is no recursion in the class structure. Moreover, only one business method is considered (3). As a result, there are three differences between Kerievsky's algorithm and our base algorithm:

1) Our steps 2 and 8 that make the *accept* method appear are done in a different way in Kerievsky [8].
2) Without recursion, our step 3 is pointless, and in step 4 the "*any var*" option of the **AddParameter** operation is also useless (see **AddParameterWithReuse** in [2]).
3) Only one Visitor class is introduced in Kerievsky, so that he does not need to add an interface for the Visitor hierarchy (steps 6 to 8 in our algorithm).

Also, the algorithms of Kerievsky are only generic guidelines and are not formally analysed.

The second variation takes place in a very different program, so that comparing the algorithms is not relevant.

VI. Conclusion

The contributions of this article are the following:

- We have selected four common variations in the implementation of the Composite pattern and we have shown how these variations reflect in the Visitor pattern.
- For each variation, we have extended the previously defined transformation. The resulting transformations are automated and invertible.
- For each variation, we have checked the validity of the adapted transformations by computing the minimum precondition that ensures their success.
- We have applied the four extensions of the algorithms on a real-size use case. The resulting algorithm is also validated by computing its precondition.
- This use case also shows that the algorithm extensions can be used together without conflicting interaction between them.

The potential impact of this work in software industry could be estimated with the following future work:

- Estimate the repartition of variations of Composite and Visitor patterns in industrial softwares and compare it to the variations we have considered. See which uncovered variations are important in industry.
- Estimate the value of changing the structure of a source code in order to ease its maintenance by experimenting it with real programmers, on real programs.

Of course, to be applicable in industry, we also have to provide some refactoring operations that are currently missing (see sections III-C and III-D).

Besides the potential industrial benefits, studying pattern variations and their impact in dual architectures has a pedagogical interest: it can be used to better understand the patterns themselves [8]. In particular, the Visitor pattern is not trivial to understand. This work shows for instance how visitors with or without a state are related to methods with or without parameters (variation A) and how generic types are needed in visitors when the business methods have different return types (variation B).

As a future work, it would be usefull to use refactoring inference tools such as [10] to find changes in the transformations automatically.

References

[1] A. Ajouli. An automatic reversible transformation from composite to visitor in Java. In *Conférence en IngénieriE du Logiciel (CIEL 2012)*, 2012.

[2] A. Ajouli and J. Cohen. Refactoring Composite to Visitor and Inverse Transformation in Java. Research Report hal-00652872 v3, July 2013. URL http://hal.archives-ouvertes.fr/hal-00652872.

[3] J. Cohen and A. Ajouli. Practical use of static composition of refactoring operations. In *ACM SAC*, pages 1700–1705, Portugal, Mar. 2013.

[4] J. Cohen, R. Douence, and A. Ajouli. Invertible program restructurings for continuing modular maintenance. In *CSMR*, pages 347–352, 2012.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] M. Hills, P. Klint, T. Van Der Storm, and J. Vinju. A case of visitor versus interpreter pattern. In *TOOLS'11*, pages 228–243. Springer-Verlag, 2011.

[7] M. Hills, P. Klint, and J. J. Vinju. Scripting a refactoring with rascal and eclipse. In *Workshop on Refactoring Tools (WRT)*, pages 40–49. ACM, 2012.

[8] J. Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[9] G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3): 9–51, Aug. 2004.

[10] I. Moghadam and M. Ó Cinnéide. Automated refactoring using design differencing. In *CSMR*, pages 43–52, 2012.

[11] M. Ó Cinnéide and P. Nixon. Composite refactorings for Java programs. In *Workshop on Formal Techniques for Java Programs, ECOOP*, 2000.

[12] B. C. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. *SIGPLAN Not.*, 43(10):439–456, 2008.

[13] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.