An Automatic Reversible Transformation from Composite to Visitor in Java

Akram Ajouli

An Automatic Reversible Transformation from Composite to Visitor in Java

Akram AJOULI
INRIA - ASCOLA team (EMN - INRIA - LINA)
(Nantes, France)
akram.ajouli@mines-nantes.fr

Abstract

We build reversible transformations between Composite and Visitor design patterns in Java programs. Such transformations represent an automatic reversible switching between different program architectures with a guarantee of semantic preservation. In this paper, we detail the algorithms of the transformations implemented by composing elementary refactoring operations. The transformations were automated with the refactoring tool of a popular IDE: Intellij Idea.

Keywords: Program transformation, design patterns, refactoring.

1 Introduction

Design patterns are program structures used to provide reuse and maintainability [4]. Each design pattern is specifically useful for some kinds of evolution. For example, the Composite (as well as the Interpretor pattern or the simple class hierarchies) and the Visitor design patterns offer two dual axes of decomposition for programs: while the former allows modular maintenance with respect to data, the latter permits modular maintenance with respect to operations/functions. So, each one of these patterns favors one kind of modular maintenance (modular adaptive or corrective maintenance as well as modular extension).

We propose in this paper an automatic reversible transformation based on elementary refactoring operations between Composite and Visitor design patterns in Java programs. That transformation is designed to guarantee semantic preservation. The goal of our approach is to permit doing modular maintenance with respect to different axes of decomposition.

To illustrate our approach we consider the program shown in Fig. 1. This program is structured according to the Composite pattern. It contains an abstract class *Graphic* and its two subclasses *Ellipse* and *CompositeGraphic* (this class contains the recursion of the Composite pattern). The functions of the program are implemented by the two methods *print* and *prettyprint*. Fig. 2 shows a program that is functionally equivalent to the first one but which is structured according to The Visitor pattern (Fig. 3 shows a class diagram which gives a clear overview about this structure). The program functions in this structure are implemented by classes *PrintVisitor* and *PrettyprintVisitor*. Our transformations automate switching from a Composite structure to a Visitor structure and vice-versa by composing refactoring operations offered by the refactoring tool of a popular IDE: Intellij Idea[1]. In particular, they transform the program of Fig. 1 into the one of Fig. 2 and vice-versa. The target of such transformations is to get two different structures of the same program in order to decrease the maintenance cost by avoiding non modular maintenance tasks. This is defended in Cohen et al. [3] and we describe in this paper the mechanics behind the proposed transformations.

We first describe some related work (Sec. 2). Then, we describe the processes of the transformations (Sec. 3). Finally, we conclude and we present some further work (Sec. 4).

[1] http://www.jetbrains.com/idea/

```
abstract class Graphic {

 abstract  void print();

 abstract  void prettyprint(); }
```

```
class Ellipse extends Graphic{

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

 void prettyprint(){
  System.out.println("Ellipse :"+this+".");} }
```

```
class CompositeGraphic extends Graphic {

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

 void print() {
   System.out.println("Composite:");
   for (Graphic g : graphics) {g.print();} }

 void prettyprint(){
   System.out.println("Composite " +this);
   for (Graphic g : graphics) {g.prettyprint();}
   System.out.println("(end)");}

 void add(Graphic g) {graphics.add(g);    } }
```

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

```
abstract class Graphic {
 void print() {
      accept(new PrintVisitor());}
 abstract void accept(Visitor v);
 void prettyprint() {
      accept(new PrettyprintVisitor());}}
```

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

```
class CompositeGraphic extends Graphic {
 ArrayList<Graphic> graphics =
    new ArrayList<Graphic>();
 void accept(Visitor v) {v.visit(this);}
 void add(Graphic g) {graphics.add(g);}}
```

(a) Data classes.

```
abstract class Visitor {
 abstract void visit(Ellipse e);
 abstract void visit(CompositeGraphic c);}
```

```
class PrintVisitor extends Visitor {
 void visit(Ellipse e){System.out.println("Ellipse ");}
 void visit(CompositeGraphic c) {
  System.out.println("Composite:");
  for (Graphic g : c.graphics){g.accept(this);}} }
```

```
class PrettyprintVisitor extends Visitor{
 void visit(Ellipse e) {
  System.out.println("Ellipse :"+e+".");}
 void visit(CompositeGraphic c){
  System.out.println("Composite "+c);
  for (Graphic g : c.graphics){g.accept(this);}
  System.out.println("(end)");          } }
```

(b) Visitor classes.

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

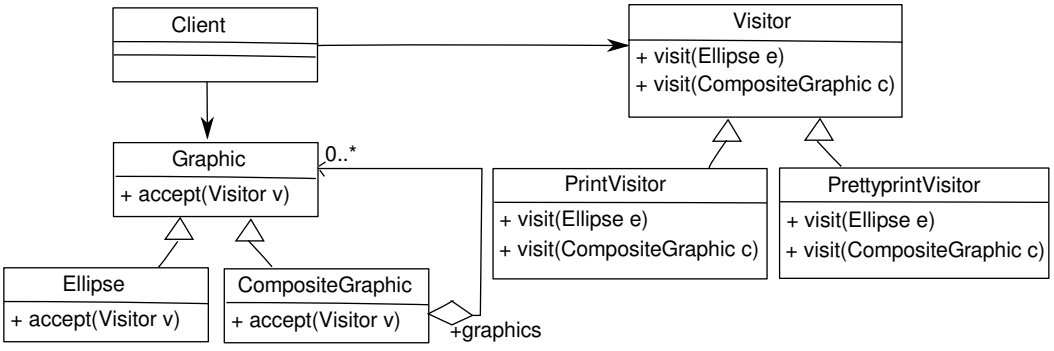

Figure 3: Class diagram of the Visitor structure.

2 Refactoring on design patterns

The idea of transforming design patterns is first discussed in the work of Takouda and Batory [2]. They propose an approach that aims to introduce patterns in object oriented programs. O'Cinnéide [9] proposes also transformations to introduce patterns in programs. Even these works support many patterns, they do not study the case of Visitor pattern. The former does not study it yet as a pattern that could be supported by their tool, the latter mentions that the automation of introducing this pattern is difficult (*"Overall Assessment: Impractical"* [9], page 151-152).

Introducing Visitor pattern in a program was studied in few works. Mens and Tourwé [8] propose an algorithm that allows to transform a simple class hierarchy (without recursion) to a structure that respects the Visitor design pattern. Kerievsky [6] proposes also three schemes that allow to introduce an instance of Visitor pattern to a program. But both of these two works are not automated and do not take in consideration the recursion existing in the Composite pattern.

Hills et al. [5] developed a tool that transforms a Visitor instance to a Composite instance. Their tool is used to transform a real program (their own transformation tool: Rascal [7]). These works are the nearest to ours. But they do not provide a reverse transformation.

In the following we detail the algorithms of the transformations. The Composite to Visitor transformation can be considered as an extension of the one proposed in The work of Mens and Tourwé [8] but it supports the recursion existing in the Composite pattern.

3 Transformation algorithms

We define two algorithms providing a reversible transformation from Composite to Visitor (see Figs. 4(a) and 4(b)). These algorithms are composed by elementary refactoring operations offered by the refactoring tool of Intellij Idea and a few other operations that have been implemented to make the transformations fully automated [2]. These algorithms could be performed from the GUI (semi-automatic) or directly from the API (fully automatic). The same thing could be done with Eclipse. In the algorithms, we use the following notations:

- $\mathbb{M}$: the set of business[3] methods. These methods could be detected automatically (in the example: $\mathbb{M}$ =$\{$*print*, *prettyprint*$\}$).
- $\mathbb{C}$: the set of the classes in the composite hierarchy except its root (in the example: $\mathbb{C}$ =$\{$*Ellipse*, *CompositeGraphic*$\}$).
- S: the root of the composite structure (in the example: S = *Graphic*).
- V: a function that gives a visitor class name from a business method name (in the example: V(*print*) = *PrintVisitor*).
- $\mathbb{V}$: the set of visitor classes, $\mathbb{V} = \{V(m)\}_{m \in \mathbb{M}}$ (in the example: $\mathbb{V}$ = $\{$*PrintVisitor*, *PrettyprintVisitor*$\}$).
- *aux*: a function used to generate names of temporary methods from business methods (in the example: *aux*(*print*) =*printAux*). These names are used only in intermediate states of the transformed program (the snapshot of some intermediate states of the program and the specification of the refactoring operations mentioned in the algorithm are given in Ajouli and Cohen [1]).

[2]The additional refactoring operations are available separately at `http://plugins.intellij.net/plugin/?idea_ce&id=6889`

[3]We use this term to precise the interesting methods in the program.

1. ForAll m in $\mathbb{M}$ do CreateEmptyClass(V(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), V(m))
5. ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do MoveMethodWithDelegate(c, aux(m), V(m), "visit")
6. ExtractSuperClass($\mathbb{V}$, "Visitor")
7. ForAll m in $\mathbb{M}$ do UseSuperType(S, aux(m), V(m), "Visitor")
8. MergeDuplicateMethods(S,$\{\ aux(\mathsf{m})\ \}_{\mathsf{m}\in\mathbb{M}}$, "accept")

(a) Composite to Visitor transformation.

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

(b) Visitor to Composite transformation.

Figure 4: Algorithms for reversible transformation from Composite to Visitor.

3.1 From Composite to Visitor

The algorithm of this transformation is shown in Fig. 4(a). This transformation consists of three main stages: preparing for moving business code; moving the business code to the Visitor structure and recovering the conventional structure of the Visitor pattern.

Steps from 1 to 4 (preparing for moving business code). For each business method create an empty visitor class (step 1). These classes will receive later the corresponding business code (in our example create two classes named *PrintVisitor* and *PrettyprintVisitor*). To keep the same interface, make the business methods delegating their codes to auxiliary methods (*printAux* and *prettyprintAux*) (step 2). These auxiliary methods will become *accept* in the final program (see Fig. 2). Then, to make the recursive calls to auxiliary methods explicit, replace each business method invocation existing in the sub-classes of the composite hierarchy by a call to its corresponding auxiliary method (step 3). Finally, in order to make the refactoring tool able to move the auxiliary methods to their right destinations, add the right references to these destinations in the methods (step 4) since the move operation must know the destination. To do that, make the type of each parameter refer to the corresponding target class that will receive the method (in the example, add to the methods *printAux* and *prettyprintAux* respectively parameters of type *PrintVisitor* and *PrettyprintVisitor*).

Step 5 (move business code from Composite to Visitor classes). Move auxiliary methods containing the business code to visitor classes. In order to create the double dispatch between the visitor structure and the composite structure, keep a delegation of each moved method in its original class (these delegations will constitute later the method *accept*). Also, rename all moved methods into "visit".

Steps from 6 to 8 (recovering Visitor structure). Extract a super-class from visitor classes (step 6). The extracted super-class will contain the abstract *visitor* methods whose implementations are in visitor classes. Then, replace references to visitor classes to a reference to the super-class of the visitor hierarchy (step 7). Finally, since auxiliary methods have equivalent bodies, replace them by a unique method called *accept* (step 8).

Result. After performing this transformation to the program having the Composite structure (Fig. 1), we get a program structured according to the Visitor pattern (Fig. 2). The resulted program is semantically equivalent to the initial one since the transformation is a composition of elementary refactoring operations. The automatic transformation took about 3 seconds.

3.2 From Visitor to Composite

We define now the algorithm that performs the reverse transformation. This algorithm transforms a Composite to a Visitor (see Fig. 4(b)). This transformation consists mainly in moving the business code to the composite structure and deleting any dependencies between the two structures. It is also based on the following three stages:

Steps 1 and 2 (preparing for moving business code). In order to make the method *accept* dealing directly with concrete visitors instead of the abstract Visitor, produce specialized methods from the method *accept*. Each new method has the same name as the initial method *accept* but its parameter type is changed to one of its sub-types (in the example, we get two additional methods *accept(PrintVisitor v)* and *accept(PrettyprintVisitor)* in the abstract class *Graphic*). Repeat this action according to the number of concrete visitor classes (step 1). The initial method *accept* can be deleted since its role is delegated to the new methods. In order to be able to in-line *visit* methods (to move business code into composite classes), delete all abstract *visit* methods from the abstract Visitor (step 2).

Step 3 (move business code from Visitor to Composite classes). In order to move the business code into composite classes, in-line the methods existing in the visitor classes and delete them from these classes.

Steps from 4 to 11 (recovering Composite structure). In order to avoid getting the same methods in the same scope when deleting the parameters of the two methods *accept*, rename the *accept* methods with names of dummy methods used in Sec. 3.1 (step 4). The name of each method will be fixed according to the method parameter type (for example: the method *accept(PrintVisitor v)* becomes *printAux(PrintVisitor v)*). In order to delete any use of visitors in the Composite structure, remove the dummy methods parameters (step 5). Then, in order to delete the delegation of the business code to the dummy methods: first, replace any invocation of dummy methods appearing in the composite classes by the respective invocation of the right delegator method (step 6) (example: replace *printAux()* by *print()*); then, make delegator methods abstract and push down their implementations to the sub-classes (step 7); after that, in order to be able to in-line the dummy methods, delete their declarations from the abstract composite (step 8) and then, in-line them in the concrete composites and delete their declarations (step 9). Finally, to get the conventional structure of the Composite pattern, delete visitor hierarchy since it is not used anymore.

Result. After performing this transformation to the program with the Visitor structure (Fig. 2), we get a program that respects the Composite structure and which is equivalent to the one shown in Fig. 1 except a few changes in the layout and the comments. The automatic transformation took also about 3 seconds.

4 Discussion and Further Work

We have automated reversible transformations between Composite and Visitor patterns which are supposed to guarantee semantic preservation. These transformations have been applied successfully to the programs of Figs. 1 and 2. They take a relevant time comparing to the semi-automatic transformations done by triggering each basic refactoring operation from the GUI. The fully automatic one takes 3 seconds in each way, while the semi-automatic one takes 12 minutes in each way. The refactoring operations missing from IntelliJ Idea have been implemented in order to make the transformations fully automated. The algorithms of the transformations have been extended in order to support some variations in the above patterns: methods having parameters, methods returning values (Visitor with generic types), class hierarchies with several levels, an interface instead of abstract class and access to private fields (see [1]).

As a future work, we aim to validate our algorithms by applying them to real programs and to real cases of software evolutions. We look also for formalizing these algorithms, proofing their correctness, calculating their minimum preconditions and studying their complexities.

Acknowledgement: We thank Dmitry Jeremov and Anna Kozlova (JetBrains) for their help with the refactoring tool of IntelliJ IDEA.

References

[1] Akram Ajouli and Julien Cohen. Refactoring Composite to Visitor and Inverse Transformation in Java. Technical report. Technical Report hal-00652872, `http://hal.archives-ouvertes.fr/hal-00652872`, 19 pages.

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

[3] Julien Cohen, Rémi Douence, and Akram Ajouli. Invertible Program Restructurings for Continuing Modular Maintenance. In *CSMR 2012*, Hungary, March 2012. 6 pages, Early Research Achievements Track.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman, 1995.

[5] 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. Springer-Verlag, 2011.

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

[7] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Proceedings of Source Code Analysis and Manipulation*, SCAM '09, pages 168–177. IEEE, 2009.

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

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