Expression and Composition of Design Patterns with AspectJ

Simon Denier, Pierre Cointe

Expression and Composition of Design Patterns with AspectJ

Simon Denier — Pierre Cointe

OBASCO Group
Ecole des Mines de Nantes – INRIA, LINA
4, rue Alfred Kastler
44307 Nantes Cedex 3, France
{Simon.Denier,Pierre.Cointe}@emn.fr

ABSTRACT. *Design patterns are well-known couples of problems-solutions for software engineering. By nature, they often lack support from languages and this further complicates the study of their composition in the code. Aspect-oriented languages provide new mechanisms for modularization, which can help to improve design patterns implementation. (Hannemann* et al., *2002) is the first extensive study of patterns aspectization with AspectJ. We notice some AspectJ idioms are needed in order to implement object relationships. We give a more reusable* VISITOR *pattern. We highlight a reusable composition of* COMPOSITE *and* VISITOR *patterns and expressive interactions of the* OBSERVER *pattern with a tree structure. We thus show that modularization by aspects helps composition of design patterns.*

RÉSUMÉ. *Les motifs de conception sont des couples bien connus de problèmes-solutions pour l'ingénierie des programmes. Par nature ils ne sont pas supportés par les langages, ce qui complique l'étude de leur composition dans le code. Les langages d'aspects fournissent de nouveaux mécanismes de modularisation, qui peuvent aider à l'implémentation des motifs de conception. (Hannemann* et al., *2002) est la première étude extensive de l'aspectisation des motifs avec AspectJ. Nous remarquons que AspectJ nécessite des idiomes afin de pouvoir implémenter les relations interobjets. Nous donnons une version plus réutilisable du motif* VISITOR. *Nous soulignons la composition réutilisable des motifs* COMPOSITE *et* VISITOR, *ainsi que des interactions expressives entre un motif* OBSERVER *et une structure en arbre. Nous montrons ainsi que la modularisation par les aspects facilite la composition des motifs de conception.*

KEYWORDS: *Design Patterns, Aspects, Composition, Interaction, Reusability.*

MOTS-CLÉS : *motifs de conception, aspects, composition, interaction, réutilisation.*

RSTI - L'objet – 12/2006. JFDLPA 2005, pages 41 to 61

1. Introduction

Design patterns (Gamma *et al.,* 1994) are well-known couples of problems-solutions for software engineering. They give some clues on benefits and tradeoffs of design implementation in object-oriented languages. They offer abstract descriptions that cover the gap between the model and the implementation, facilitating a mutual understanding. They are supportive of code micro-architectures, which enable variability and evolutivity.

However, object-oriented implementations of design patterns have failed in two ways. First, it is hard to maintain a link between the model and the implementation. Some concrete elements are hardly modularized and tend to be lost in the code. The pattern is said to vanish (Soukup, 2005) – in a modern stance, it lacks traceability. Second, lack of modularization means design patterns implementations are hardly reusable (Tatsubori *et al.,* 1998; Albin-Amiot, 2003).

Language features impact design patterns implementations, to the point where some patterns have built-in support and seem straightforward (Chambers *et al.,* 2000). Aspect-oriented languages provide new mechanisms to enable modularization of crosscutting concerns (Kiczales *et al.,* 1997). Such mechanisms can capture patterns or help to modularize them. (Hannemann *et al.,* 2002) provides the first extensive study of design patterns implementation with AspectJ, an extension of Java for aspect-oriented programming (Kiczales *et al.,* 2001; Colyer *et al.,* 2005). It claims that most design patterns from (Gamma *et al.,* 1994) get benefits from aspectization, in particular better modularity, and gives some insights on their crosscutting nature.

We review the aspectized patterns from (Hannemann *et al.,* 2002) and get the feeling that these implementations present programming idioms for AspectJ. These idioms capture objects relationships – such as a design pattern structure – and implement it in the aspect. In our opinion this shows that the aspect model in AspectJ is heterogeneous with respect to an object model: aspect instances can not be controlled as object instances and this hampers the use of AspectJ. Such critics have been made in (Mezini *et al.,* 2003). However, we also find that AspectJ mechanisms allow to bypass these object idioms and provide better implementations for the VISITOR pattern.

Another topic of interest is the composition of design patterns. As examplified in the Java AWT framework, such compositions are frequent and their understanding can be crucial. This includes patterns making use of others in their implementation as well as interaction between two patterns. To our knowledge few work has studied this field: most prominent include (Gamma *et al.,* 1994) (section "Design pattern relationships") and (Zimmer, 2005), who proposes a classification of the different kind of relationships between design patterns. Composition leads to the pattern density phenomenon: patterns provide leverage but their implementations become so entangled that it is nearly impossible to think of a pattern alone and that the program becomes hard to change (Gamma *et al.,* 2002).

The modularity gain from aspects promises a better understanding of composition and interactions between design patterns. We start to evaluate some composition cases, such as a well-known collaboration between COMPOSITE and VISITOR patterns, and "unforeseen" interactions with an OBSERVER pattern. We highlight the reusability and expressiveness of such compositions. We also experiment with object-aspect and aspect-aspect compositions of design patterns, to once more reveal the heterogeneity between aspects and objects in AspectJ.

Section 2 provides a quick overview of aspects concepts and some specificities of AspectJ. Section 3 reviews the usage and idioms of AspectJ to implement a design pattern with the OBSERVER pattern as example. We also review the DECORATOR pattern, which departs from idioms with benefits and drawbacks. We provide a new implementation of the VISITOR pattern, which improves on reusability and genericity, literally standing between a pattern and multiple-dispatch mechanism. Section 4 examines two cases of composition. First, we provide a reusable composition of COMPOSITE and VISITOR patterns. Second, we study new expressiveness of the OBSERVER pattern in face of a tree structure composed by COMPOSITE and DECORATOR, both as objects and as aspects. Section 5 gives related work and Section 6 concludes.

2. Overview of Aspects and AspectJ

We begin with a quick overview of aspects concepts and illustrations in AspectJ, an extension of Java. For readers interested by AspectJ, there are numerous ressources (Kiczales *et al.,* 2001), including book (Colyer *et al.,* 2005) and online sites[1].

The main intuition behind AOP is to introduce a *join point* model defining interesting events during execution of a program. These can be method call, method execution, constructor call, field access... A *pointcut language* selects specific join points, based for example on class and method name. In AspectJ, pointcuts can expose some of the execution context via variables. They can also be composed with logical operators (and, or, not). An *advice language* defines code to be ran at join points captured by pointcuts. In AspectJ, an advice is linked to a pointcut and parameterized by pointcut variables. Apart from this declaration, advice are defined like methods. An *aspect* is primarely a modular unit composed of pointcuts and advice. Like a class, it can contain ordinary member definitions such as fields and methods. An AspectJ aspect declaration has a form similar to that of a Java class declaration. Finally, the process by which an aspect is "plugged" into a program is called *weaving*. AspectJ allows weaving at compile-time or load-time, but not runtime.

For the purpose of this paper, there are two specificities of AspectJ we must review. First are *Inter-Type Declarations* (ITDs), which are a subset of structural reflection. They allow to introduce new members inside existing classes without modifying class source. A noteworthy feature of AspectJ is that such introductions can be made in

1. See `http://www.eclipse.org/aspectj/` or the AOP@Work series on `http://www-128.ibm.com/developerworks/`

combination with an interface: any class implementing this interface will "inherit" these new members. This makes ITDs akin to mixin inheritance or traits (Denier, 2005).

Second is the *instantiation model* of aspects in AspectJ. Like classes, aspects define instances to operate at runtime. Unlike classes, users do not explicitely control when an aspect instance is created and its scope of execution. Instead, this is an option to be chosen in the aspect declaration. Figure 1 shows two such options. By default an aspect defined a singleton instance, which runs on any target join points. But the aspect can also declare a `perthis(aPointcut)` modifier: in this case one aspect instance will be lazily created for each object crosscutted by `aPointcut`. Whenever a join point is picked up inside one of these objects, its associated aspect instance will run the advice. Such a per-object aspect instance will never run advice outside of its object context. We retain that the AspectJ instantiation model controls both the creation and the scope of an aspect instance.

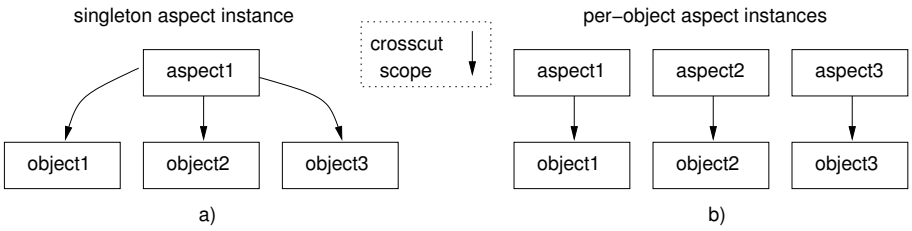

Figure 1. *Two options among others of the AspectJ instantiation (and scope) model. a) is the singleton model, where an aspect defines a unique instance, which can crosscut any objects. b) is the per-object model, where an aspect defines one instance per crosscutted object: the aspect instance can only crosscut its object*

3. Expression of Single Design Patterns

An examination of the catalog of aspectized design patterns from (Hannemann *et al.,* 2002) reveals many similarities between implementations. These similarities are more complex than straightforward language use and seem rather specific to AspectJ. We abstract them as idioms, that is "good practice" particular to AspectJ. Prominent idioms for design patterns implementation in AspectJ are:

– an aspect modularize the whole design pattern (including roles and collaborations), possibly deferring specialization to subaspects and classes;

– roles are declared by Java interfaces;

– relationships between roles can be implemented with Inter-Type Declarations (a subset of structural reflection) or centralized by the aspect in a single structure;

– collaborations are a mix of pointcuts, advice and regular methods.

Section 3.1 exposes such idioms for the OBSERVER pattern. The use of interfaces as roles and the centralized structure for relationships are especially idiomatic to (Hannemann *et al.,* 2002). Overall, these idioms allow to implement objects relationships inside an aspect.

3.1. *Anatomy of an Aspectized Pattern: the* OBSERVER *Pattern*

(Gamma *et al.,* 1994) cites the OBSERVER pattern as useful to:

> *"define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."*

We choose this particular pattern as it is: a well-known design pattern; an example of the benefits of aspects for pattern implementation; an example of AspectJ mechanisms and idioms. To describe the AspectJ-based implementation, we follow three sections in (Gamma *et al.,* 1994) which best relate to a general implementation of a design pattern: **Structure**, **Participants** and **Collaborations**. Figure 2 shows a reusable implementation of the OBSERVER pattern, which differs slightly from the one found in (Hannemann *et al.,* 2002). In particular, we present a solution using Inter-Type Declarations (ITDs). Figure 3 shows how to specialize this implementation for use.

First, a whole **design pattern** (including roles and collaborations) can be modularized in a single aspect. Some of them, such as the OBSERVER pattern (Figure 2), are reusable and can be specialized by aspect extension and class inheritance (Figure 3).

Participants or *Roles*, quickly describe entities involved in the pattern solution and their responsibilities. As such, they are interesting in order to trace and to understand how classes are involved with the pattern. In AspectJ, Java interfaces are used to define such roles (lines 3 and 8). Java interfaces are transversal to the inheritance hierarchy, so that it does not need modification. Also, AspectJ allows to apply roles (interfaces) from aspects (Figure 3, lines 2 and 3).

Structure is primarely a representation of *Relationships* between roles, such as inheritance and aggregation. Most design patterns work by decoupling concerns and assigning them to different roles, with an indirection. Then, relationships between roles must be defined to allow them to collaborate. A typical relationship, aggregation, is for a role instance to hold a reference in a field to another role instance. For example, subjects in the OBSERVER pattern hold a list of reference to their observers.

ITDs are a straightforward way to implement such relationships. For example, the list of references to Observers is implemented in `Subject` interface line 4 (Figure 2). Line 5 shows the introduction of an accessor. After compilation and weaving, all classes implementing this interface will get these implementations. A possible problem is that *public* ITDs (line 5) can conflict with elements from target class.

Another solution to implement relationships while avoiding ITDs conflict is to centralize them in a single structutre within the aspect. For example, the OBSERVER

aspect in (Hannemann *et al.,* 2002) holds references from all subjects to their observers: each subject is a key to its list of observers in a `HashMap`. The use of hash tables to centralize aggregation relationships is idiomatic in (Hannemann *et al.,* 2002).

Either ITDs or aspect centralization need some way to manage relationships: the registration process (Figure 2, lines 20 to 22, and Figure 3, lines 7 and 8) shows how these can be managed by pointcuts and advice.

Collaborations describe how roles work together to fulfill the pattern intent. Collaborations are usually defined in object-oriented languages by message passing and method definition. Aspects tend to partially replace methods with advice and method calls with pointcuts. Figure 2 shows such a case lines 13 to 17: the `stateChanges` pointcut capture notification points and the advice performs the notification process. Then, the task is fulfilled by calling `update` on observers. The dependency extraction which results from pointcuts is a key benefit from aspects.

```
1  public abstract aspect OberverProtocol {
   /* Role and structure for Subject */
3  protected interface Subject { }
   private Vector Subject.observers = new Vector();
5  public  Vector Subject.getObservers() {return observers;}

7  /* Role and update interface for Observer */
   protected interface Observer {
9   public void update(Subject s);
   }
11
   /* Notification process */
13 abstract pointcut stateChanges(Subject s);
   after(Subject s): stateChanges(s) {
15  for (int i = 0; i < s.getObservers().size(); i++)
     ((Observer)s.getObservers().elementAt(i)).update(s);
17 }

19 /* Registration process */
   abstract pointcut registerObserver(Observer o, Subject s);
21 after(Observer o, Subject s): registerObserver(o, s) {
    s.observers.addElement(o); }
23 }
```

Figure 2. *An abstract, reusable version of the* OBSERVER *pattern with AspectJ. We use inter-type declarations to implement the structure in the* `Subject` *interface. Notice how the registration is also defined as an advice*

```
1  public aspect FigureObserver extends ObserverProtocol {
    declare parents: Figure implements Subject;
3   declare parents: Drawing implements Observer;

5   pointcut stateChanges(Subject s):
     execution(void Figure.setAttribute()) && this(s);
7   pointcut registerObserver(Observer o, Subject s):
     execution(void Drawing.addFigure()) && this(o) &&args(s);
9  }
   public class Drawing{ // is Observer via FigureObserver
11  public void update(Subject s) { ... }
   }
```

Figure 3. *Specialization of AspectJ* OBSERVER *pattern. The aspect applies roles and defines pointcuts, while the observer class simply implements its update method*

3.2. *Benefit and Drawback for the* DECORATOR *Pattern*

The intent of the DECORATOR pattern, as defined in (Gamma *et al.,* 1994), is to:

> "attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

Figure 4 gives an example of the DECORATOR pattern from (Hannemann *et al.,* 2002). Many decorators can be defined on the same join point, ordered by precedence and applied or removed at weaving.

```
   public aspect BracketDecorator {
2   pointcut printCall(String s):
     execution(void ConcreteOutput.print(String)) && args(s);
4   void around(String s): printCall(s) {
     s = "[" + s + "]"; // Decorates the string
6    proceed(s); }
   }
```

Figure 4. *Basic* DECORATOR *pattern, adapted from Hannemann*

With respect to the object-based implementation, the benefits are obvious:

1) only decorated methods are advised, eliminating the "implementation overhead" (Bosch, 1998) of numerous forwarding methods;

2) there is no indirection for non-decorated methods, reducing the runtime cost;

3) there is no need to insert a Decorator instantiation.

This solution is great when subclassing is not allowed or to apply a DECORATOR pattern on the whole program. However, with respect to the intent above, this implementation does not allow to *dynamically* attach a decorator to *one* object.

We propose an extension in Figure 5. The main idea it that each instance has an `active` state (line 2), which controls the decorator behavior (line 8). A key construct of this implementation is the `perthis` modifier (line 1): it allows to associate one aspect instance to each `ConcreteOutput` object, effectively defining one `active` state per object. In any case, `proceed` allows the action to finally proceed to the decorated object (line 10). `decorate` and `strip` methods from `BrackDecorator` aspect allow to dynamically apply the decorator behavior (line 13).

```
1  public aspect BracketDecorator perthis(printCall(String)){
     private boolean active = false;
3    public void decorate(){ this.active=true; }
     public void strip(){ this.active=false; }
5    pointcut printCall(String s):
       execution(void ConcreteOutput.print(String)) && args(s);
7    void around(String s): printCall(s) {
       if( this.active )
9        s = "[" + s + "]";
       proceed(s); } // proceed in all cases
11 }
   // Registration usage:
13   BracketDecorator.aspectOf(aConcreteOutput).decorate();
```

Figure 5. *New* DECORATOR *pattern with dynamic application possibility*

Still, this solution lacks two properties of the object-based DECORATOR pattern:

– no dynamic reordering of decorators, since precedence is statically defined;

– no recursive composition (a `BracketDecorator` around a `BracketDecorator` around a ...), since we can not instantiate an aspect at will.

Contrary to the OBSERVER pattern and other patterns in AspectJ, the above DECORATOR pattern is less idiomatic: no role, no structure for relationships. In this case, the aspect *is* the decorator, whereas it tends to be an encapsulating module in other patterns. The reason is that `around` advice allows straightforward decoration, which considerably simplifies the implementation. The drawback, as seen above, is that it is not possible to get a full control on the instantiation and composition of decorators.

3.3. *Revisiting the* VISITOR *Pattern*

(Gamma *et al.,* 1994) announces:

> *"[Visitor can] represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."*

It is well-known as a replacement for multi-methods, or more precisely double-dispatch, in languages which do not support it.

The reusable VISITOR pattern in (Hannemann *et al.,* 2002) follows the guidelines from Section 3.1. This primarely includes a `VisitorPrococol` aspect, which defines roles-interfaces for *Visitor*, *Node* and *Leaf* (where nodes are non-terminal and leaves are terminal) and ITDs of `accept(Visitor)` in nodes in the usual way – `Node.accept(Visitor)` (resp. `Leaf.accept(Visitor)`) calls `Visitor.visitNode(Node)` (resp. `Visitor.visitLeaf(Leaf)`). A first remark is that it does not use pointcut and advice – only ITDs. A second remark is that this visitor only distinguish two kinds of node, which severely restricts its genericity. This solution suffers from the same shortcoming than the object-oriented visitor: it does not allow to easily change the visited structure.

We propose in Figure 6 a more generic and reusable VISITOR pattern, which slightly differs in principle from the classic pattern. There is no more indirection by an `accept` method in `VisitableNode`. Two interfaces are still defined for roles (lines 2 and 3), but we do not discriminate between nodes at this level. We still use an ITD to introduce the `visit` method in `Visitor` (line 4). The novelty is that we also define a pointcut, `visit`, on this method (lines 5 and 6). The purpose of the `visitor` pointcut and `perthis` modifier is explained below.

```
1  public abstract aspect VisitorProtocol perthis(visitor()) {
     public interface VisitableNode {}
3    public interface Visitor {}
     public void Visitor.visit(VisitableNode node) {}
5    public pointcut visit():
       execution( void Visitor.visit(VisitableNode) );
7    public abstract pointcut visitor();
   }
```

Figure 6. *The new abstract pattern for* VISITOR

Figure 7 shows how such a visitor can be defined. It is broken up between the `CountingVisitor` class, which in particular defines state and accessors, and the `CountingVisitorBehavior` aspect, which defines the behavior corresponding to the visited structure (see Figure 9 for structure definition). The `visitor` pointcut (lines 7 & 8) targets the `CountingVisitor` class, so that an aspect instance of

`CountingVisitorBehavior` is created for each object (thanks to the `perthis` modifier from Figure 6, line 1). The goal is to create an exclusive relationship between a visitor aspect and a visitor object: the aspect only crosscuts its associated object and can refer to it with `myVisitorInstance` (lines 9 to 11).

The purpose of the crosscutted `visit` method defined in Figure 6 appears lines 13, 16 and 21: this join point creates a dynamic association between a visitor and a node, which is advised by `CountingVisitorBehavior`. Dynamic dispatch occurs thanks to the `args` predicate, which works on dynamic types of parameters. The last advice shows a default behavior, crosscutting on any node which is not captured by the above advice.

```
public class CountingVisitor implements Visitor {
 private int count = 0;
 void incr(){ count++; }
 public int report(){ return count; }
}
aspect CountingVisitorBehavior extends VisitorProtocol {
 public pointcut visitor():
  execution(CountingVisitor.new(..));
 private CountingVisitor myVisitorInstance;
 after(CountingVisitor v):  visitor() && this(v){
  myVisitorInstance = v; }

 void around(TestLeaf l): visit() && args(l){
  System.out.println("Visiting a TestLeaf " + l);
  myVisitorInstance.incr(); }
 void around(TestGroup c): visit() && args(c){
  System.out.println("Visiting a TestComposite " + c);
  myVisitorInstance.incr(); }
 // Other nodes handled the same way
 // Default behavior with respect to advice precedence
 void around(TestElement c): visit() && args(c){
  System.out.println("Visiting a TestClass " + c);
  myVisitorInstance.incr(); }
}
```

Figure 7. *Customization of the* VISITOR *pattern for a particular structure (see also Section 4.1.2)*

The use is simple: instantiate a `CountingVisitor` and traverse the structure with it, calling `visit` on each node (see Figure 9). Notice that we make no hypothesis on how the structure is traversed and when the `visit` method is called. Following (Gamma *et al.,* 1994), this can be the responsibility of the structure, the visitor (in non-terminal nodes) or an external iterator.

3.4. *Synthesis on Aspects, AspectJ and Design Patterns*

Apart from dependency inversion (Nordberg III, 2001) and crosscutting modularization (Hannemann *et al.,* 2002), new mechanisms (useful although not specific) which come with aspects can improve a whole design pattern solution. This is the case with the DECORATOR pattern and particularly with our new VISITOR pattern. In particular, pointcut predicates offer new expressiveness, especially on dynamic properties. Design patterns whose behavior change based on the dynamic context could be improve in the same way: this could include, for example, STRATEGY, COMMAND, or CHAIN OF RESPONSIBILITY patterns.

We notice that idioms primarely target roles and relationships. Then the pattern aspect works as a singleton. The aspectized DECORATOR pattern, which violates these idioms to simplify its implementation, suffer from problems on instantiation and composition. Finally, our new VISITOR pattern needs a dual definition of class and aspect in order to work: the visitor class can be created on will and provides the scope for the visitor aspect. These examples highlight the heterogeneous nature of aspect instances in AspectJ: they are hardly usable as normal object instances, which can hamper their own design. We believe this is one of the main weakness of AspectJ. However, this problem is rather specific to AspectJ and not to aspect languages in general.

4. Composition of Design Patterns

In this section we experiment two cases for composition of design patterns involving aspects. The first case aims at producing a reusable composition based on simple reusable patterns. We take COMPOSITE and VISITOR as they are often used together. The second case looks at interactions which can interest the OBSERVER pattern when dealing with COMPOSITE and DECORATOR patterns. However, we survey in this case the aspectual interactions with COMPOSITE and DECORATOR both as objects and as aspects. Indeed, we show that the AspectJ expression of these interactions depends on the aspect or object nature of the pattern implementation.

4.1. *Composition of* COMPOSITE *and* VISITOR *Patterns*

The intent of the COMPOSITE pattern, citing (Gamma *et al.,* 1994), is to:

> *"compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."*

The class structure is straightforward: the `Composite` class subclasses and defines a one-to-many relation to a `Component` class. This allows recursive composition: a `Composite` instance can hold references to any type of `Component` instances, including other `Composites`. The `Composite` class inherits the interface of `Component`

but usually overrides each method in order to forward it to its children. Then, a method called on a `Composite` recursively goes down until reaching non-Composite instances. In a language which lacks multi-methods, closures or first-order methods, this leads to implementation overhead as each method must define the same loop to forward itself.

4.1.1. *The Aspectized* COMPOSITE *Pattern*

```
public abstract aspect CompositeProtocol {
 // Roles
 public interface Component {}
 public interface Composite extends Component {}
 public interface Leaf      extends Component {}

 // Centralized structure
 private Map perComponentChildren = new WeakHashMap();
 private Vector getChildren(Component s) {
  Vector children = (Vector)perComponentChildren.get(s);
  (...) // lazy instantiation of children vector if none
  return children; }

 // Registration interface
 public void addChild(Composite cs, Component cn) {
  getChildren(cs).add(cn); }
 public void removeChild(Composite cs, Component cn) {...}
 public Iterator traverseChildren(Component c)  {
  (...) return getChildren(c).iterator(); }

 // Collaboration with Visitor
 declare parents: Component extends VisitableNode;
 public void recurseAll(Component c, Visitor visitor){
  visitor.visit((VisitableNode) c);
  for (Iterator ite = traverseChildren(c); ite.hasNext();)
   recurseAll((Component) ite.next(), visitor); }
 // Other collaborations with Visitor...
}
```

Figure 8. *The abstract, reusable version of* COMPOSITE*; a noteworthy feature is the integrated* VISITOR *pattern*

The goal of the aspectization (Figure 8) is to promote a reusable definition of the COMPOSITE pattern. Then, it can be applied on existing classes, removing the implementation overhead of defining the structure and the registration logic. To further reduce implementation overhead, it must also define a "forward loop" which can be parameterized by function. A solution is to combine the COMPOSITE pattern with a VISITOR pattern, which we overview in this section and the next one.

We start with the solution of (Hannemann *et al.,* 2002), using interfaces for roles (lines 3 to 5), and the idiomatic centralized structure. Indeed the relationships between `Composite` and their children are all stored in the `perComponentChildren` map. For each `Composite`, the map stores a `Vector` of `Component` children (line 8).

In its solution, Hannemann then proceeds by defining a whole new `Visitor` *inside* its COMPOSITE pattern, copying but not reusing its genuine VISITOR pattern. We prefer to reuse our lighter, more generic solution. Thus, we simply include an interface to collaborate with the VISITOR pattern defined in Section 3.3. Collaborations need only one preliminary declaration: we declare a `Component` as a `VisitableNode` for `Visitor` (line 22). `recurseAll` (lines 23 to 26) is an example of such a collaboration: it performs the call to `visit` (which triggers the `Visitor` action) recursively on all subcomponents. Another collaboration, for example, is to make a simple forward, non-recursive, to children (the visitor will decide itself whether or not to perform a recursive visit).

4.1.2. *Customization and Usage with the* VISITOR *Pattern*

```
// Base classes
class TestClass {}
class TestLeaf extends TestClass  {}
class TestComposite extends TestClass {}

// Application of the Composite pattern
aspect TestManager extends CompositeProtocol {
 declare parents: TestClass extends Component;
 declare parents: TestLeaf extends Leaf;
 declare parents: TestComposite extends Composite; }

// Usage of Composite and Visitor patterns
 TestComposite root = new TestComposite();
 TestManager.aspectOf().addChild(root, new TestLeaf());
 CountingVisitor visitor = new CountingVisitor();
 TestManager.aspectOf().recurseAll(root, visitor);
 System.out.print("Number of nodes: " + visitor.report());
```

Figure 9. *Customization and use of* COMPOSITE-VISITOR

Figure 9 shows how to apply the COMPOSITE pattern and use it with a VISITOR pattern. We start with base classes `TestClass`, `TestLeaf`, and `TestComposite` (lines 2 to 4 – the inheritance relationship is not mandatory). The `TeamManager` aspect extends `CompositeProtocol` (line 7) to define a COMPOSITE pattern on these classes: it does so by matching each class with its role interface (lines 8 to 10). These are the only required steps to use `CompositeProtocol`.

From lines 13 to 17, we show typical tasks with a `Composite`: instantiation, child registration. Then we recurse over the whole structure, using the `CountingVisitor` defined in Figure 7, to count the number of nodes.

4.2. *The* OBSERVER *Pattern with* DECORATOR *and* COMPOSITE *Patterns*

(Gamma *et al.,* 1994) already cites the natural composition of DECORATOR and COMPOSITE patterns. They form together rich tree structures. Another case for composition is to plug an OBSERVER pattern on such a structure. With aspects, an `Observer` can crosscut on the whole tree, adding some expressiveness hardly allowed by traditional object-based `Observer`. The purpose of this section is to examine the expressiveness of AspectJ when composing the `Observer` aspect with `Composite` and `Decorator`, first as objects, second as aspects. This uncovers the fact that AspectJ needs to treat objects and aspects in different manner.

4.2.1. *Some Examples of Tree-Wide Observation*

In this section, we use a tree of `Figures` (in the context of a structured drawing program, for example) and the `Observer` defined in Figure 3. We make the hypothesis we have some `CompositeFigure` and `DecoratorFigure` classes (Figure 10). We refer to (Denier *et al.,* 2006) for a concrete case and the rationale behind such a composition.

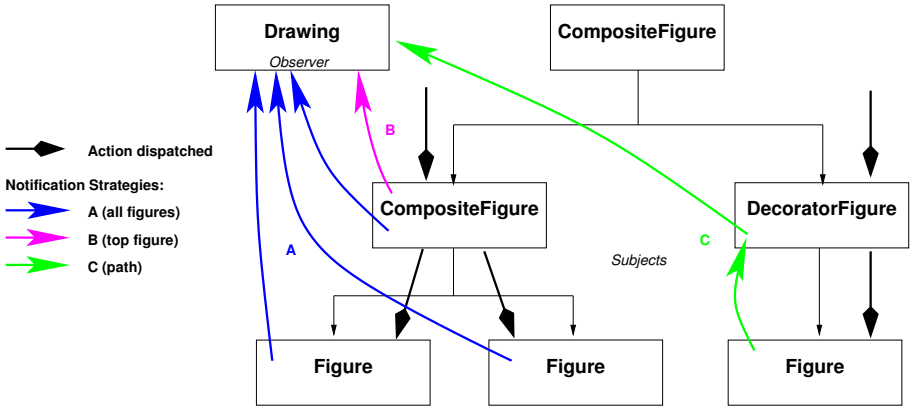

Figure 10. *This object diagram shows different strategies for notification of an `Observer`. `Drawing` is an observer of the tree structure. Figures are organized in a tree with* COMPOSITE *and* DECORATOR *patterns. An action launched on any figures is recursively dispatched to its children*

Obviously, we can make the `FigureObserver` aspect notifies for any action in any `Figures`, including `CompositeFigure` and `DecoratorFigure` (Figure 10, la-

bel A). This triggers redundant notifications. However, there are other possibilities to precisely customize the crosscut:

1) some notifications occur at the top level and do not need to be raised down in the tree – for example, `CompositeFigure` can make global notifications, regardless of its children (Figure 10, label B);

2) some notifications occur within tree leaves but need parameterization according to top level nodes – for example, `DecoratorFigure` in the path of action takes over notification *following* notification from its child (Figure 10, label C).

Following the above strategies, we can improve on the definition of a notification: first where and when do we notify; second which `Figure` is deemed responsible for the notification (it will be transmitted as a parameter of notification). We notice that strategies can be defined for each action on a case by case basis using aspects.

4.2.2. *An Aspectual `Observer` with `Decorator` and `Composite` as Objects*

The object-oriented solutions of DECORATOR and COMPOSITE patterns define a recursive control flow whenever a method is called on the tree. Indeed their basic principle for implementation of methods is to forward the message to their children. AspectJ can then crosscut the tree using its `cflow` constructs.

The **strategy labelled B** targets "top-level calls". The solution is straightforward and well-known in AspectJ. Oblivious to the type of figure crosscutted (whether a leaf figure, a `CompositeFigure` or a `DecoratorFigure`, it will only catch the first call on a recursive control flow (not necessarily the tree root). The `action` pointcut below defines the target of notification. The `topLevelNotify` pointcut implements strategy B and retrieves the figure responsible for the notification. In this case the figure is directly retrieved from the current join point.

```
pointcut action(): execution(void Figure+.move(..));
pointcut topLevelNotify(Figure f):
    this(f) && action() && !cflowbelow(action());
```

At first glance the **strategy labelled C** follows the contrary line. The `decoratedAction` pointcut takes care of join points which are *beneath* a `DecoratorFigure`. However, the figure retrieved in this case is the figure underneath, not the `DecoratorFigure`. This violates the definition of strategy C, which tells the `DecoratorFigure` is responsible when its children notify.

```
pointcut decoratedAction(Figure f):
    this(f) && execution(void Figure+.setAttribute(..))
&& cflowbelow(execution(void DecoratorFigure.setAttribute(..)));
```

We need another solution for strategy C. The process is two-step. First we retrieve the top most `DecoratorFigure` (by definition of strategy C, the top most decorator

is always responsible for other figures). `decoratorAction` and `topDecorator` are exactly the same pointcuts as for solution B, *except* that these are not used as notification pointcuts. The real notification pointcut follows: `decoratorNotify` targets the action in the control flow of `DecoratorFigure`, but retrieves the responsible figure from the `topDecorator` pointcut.

```
pointcut decoratorAction():
    execution(void DecoratorFigure.setAttribute(..));
pointcut topDecorator(DecoratorFigure df):
    this(df) && decoratorAction()
    && !cflowbelow(decoratorAction());

pointcut decoratorNotify(Figure f):
    execution(void Figure+.setAttribute(..))
    && cflowbelow(topDecorator(f));
```

A general benefit from the use of a specific control flow language is that 1) it is easier and safer than using, if possible, stack inspection; 2) it allows to easily discriminate for each method between A, B or C strategies. In a traditional object-oriented solution, the cost of implementing such mechanisms across many classes for each action is too high: usually one strategy is implemented and all actions stick to it. (Denier *et al.,* 2006) shows how C notifications are implemented using a CHAIN OF RESPONSIBILITY pattern.

4.2.3. *Observer, Decorator and Composite as Aspects*

We now examine whether the notification strategies for the OBSERVER pattern can be implemented in the presence of the aspectized DECORATOR (Section 3.2, especially dynamic as in Figure 5) and COMPOSITE (Section 4.1) patterns. The property to remember is that strategies B & C imply a recursive structure and recursive behaviors.

The aspectual COMPOSITE pattern provides these structure and behaviors. We simply need to adapt the pointcut syntax to the programming style for actions. Indeed, actions can be implemented using either for-loop or `recurseAll` with an aspectual `Visitor`. The later case asks for a redefinition of pointcut. The `action` pointcut now targets the `recurseAll` method from the `CompositeProtocol` aspect. The figure retrieved and the precise action targeted are now arguments of this method: in particular, the visitor type (`MoveVisitor`) is used to discriminate the action.

```
pointcut action():
    execution(void CompositeProtocol.recurseAll(..));
pointcut topLevelNotify(Figure f):
    args(f,MoveVisitor) && action() && !cflowbelow(action());
```

As underlined in Section 3.2, the aspectual DECORATOR pattern does not allow recursive composition. In fact, it works not by a recursive relation but by crosscutting each instance of target classes. Our pointcuts for strategies B & C in Section 4.2.2 are useless. One solution could be for the `Observer` to test the `Decorator` aspect for its current components (Figure 5, see lines 2 and 9) – this is not as refined as using `cflow`. However, such a solution is not necessarily needed: retrieving a responsible `Figure` for notification generally implies a later call to this `Figure`, for some purpose. This later call will then be advised by the oblivious `Decorator`, which will perform the decorated effect (or so we assume).

4.3. *Analysis of Composition by Aspects*

The different cases exposed above ask for an examination of their compositional form. The case of COMPOSITE and VISITOR aims at testing the composition of reusable patterns, in order to produce a reusable composition. The case is successful with respect to this objective. We attribute this to the new VISITOR pattern, which is not hampered by a static structure.

The case of OBSERVER pattern with COMPOSITE and DECORATOR patterns is more puzzling: the use of AspectJ allows more expressiveness at a low cost, although it clearly depends on the implementation of the structural patterns. Whenever some solutions use a structure of objects, we see an impact on the pointcuts of the OBSERVER pattern. This is the case for both the `Composite` object and aspect, as well as for the `Decorator` object. However, handling an instance with the `Decorator` aspect seems problematic: OBSERVER pattern pointcuts need no modification but we assume that the obliviousness nature of `Decorator` will solve the case. This shows there is some heterogeneity in the way we can handle object instances and aspect instances. Overall, we notice that only the customization (by pointcut) of the OBSERVER pattern needs changes. Both COMPOSITE and DECORATOR patterns are unchanged.

5. Related Work

Many approaches have been tried – and debated (Chambers *et al.,* 2000) – to ease design patterns use. These include tools and models to generate (Albin-Amiot, 2003), detect (Guéhéneuc, 2003), and refactor design patterns in object-oriented languages. Language approach is different, as languages do not directly target design patterns, but rather revisit pattern solutions with their mechanims. In this section we focus on work related to languages and aspects.

5.1. *Implementation of Design Patterns in Object Languages*

Several attempts to express design patterns at language level have been proposed. For example (Tatsubori *et al.,* 1998) use the OpenJava MOP to reify some design

patterns in Java source code. We expect a more declarative way to proceed by using an aspect language such as AspectJ. In (Bosch, 1996) author use a language which facilitates the expression of inter-class relationships. As design patterns can be seen as collaborating objects, this kind of languages helps efficiently in the implementation of some design patterns. Aspect languages promise a more general approach and also better mechanisms to modularize (and maybe reuse) design patterns implementation.

5.2. *Expression of Design Patterns in Aspects Languages*

The OBSERVER pattern is often used to present and evaluate new features of aspect languages. Caesar (Mezini *et al.,* 2003) comes with an instantiation model which seems to circumvent the role idioms of AspectJ. Reflex (Tanter *et al.,* 2003) can arbitrarily define and link metaobjects to hooksets, contrary to the constrained AspectJ model.

The Demeter language – DJ for its variant in Java (Lieberherr *et al.,* 2001) – has long been know to embody the VISITOR pattern. It focus on object structures whereas AspectJ focus more on control flow. Recently, (Lieberherr *et al.,* 2005) shows similarities between DJ and AspectJ with regards to the relational aspect of their pointcut languages. We believe that our experiments in AspectJ, such as COMPOSITE and VISITOR (Section 4.1), or COMPOSITE and OBSERVER using control flow (Section 4.2), provide an illustration of how dynamic structures are related to static structures and how to exploit them.

Few work has studied other patterns. One interesting case is MEMENTO, for which different attempts with AspectJ have been made (Marin, 2004). To date the sole extensive study of single design patterns implementation with aspects is in (Hannemann *et al.,* 2002). However, none of the work above has studied the composition of design patterns.

5.3. *Engineering Properties in Aspect Programming*

Work such as (Clarke *et al.,* 2001; Lieberherr *et al.,* 2003) deals with modularity and reusability of aspects: they often rely on design patterns (such as OBSERVER) in their examples. This suggest reusable design patterns libraries with aspects. Evolutivity is also a problem for aspects as well as for design patterns. Many pieces of work have focused in particular on robustness of pointcuts (Gybels *et al.,* 2003; Aldrich, 2004; Ostermann *et al.,* 2005).

(Kniesel *et al.,* 2005) argues that generic metaprogramming for aspects is more useful for some design patterns, such as DECORATOR and VISITOR, than reusability of aspects. Kniesel *et al* also advocates that the systematic use of interfaces as roles in AspectJ hamper genericity. We support this claim for the VISITOR pattern, but we present a rewriting of this pattern which is more generic while not relying on the powerful tools of metaprogramming.

6. Conclusion

This articles provides an overview of AspectJ idioms commonly used to implement design patterns. We show an extension of the AspectJ DECORATOR pattern, enabling dynamic attachment to objects. We highlight a more generic and reusable VISITOR pattern, which allows multiple dispatch and thus improves on the classic object-based visitor. Our visitor also works with the COMPOSITE pattern in a reusable composition. The OBSERVER pattern gains expressiveness in face of tree structure built by COMPOSITE and DECORATOR patterns.

Basically, aspects provide better modularization of design patterns, improving traceability and dependency control. Our primary results show that such modularization can be beneficial for the composition of design patterns. Interactions can be localized into aspects, such as for the OBSERVER pattern with COMPOSITE and DECORATOR patterns, and reusable compositions can be defined.

Language mechanisms which come with aspects, although not specific, also improve solutions. For example, we use dynamic type predicates to enable a better VISITOR pattern, and `cflow` constructs to give expressive semantics on tree structure crosscutting. Aspects still provide modularization for these powerful expressions, which help to cope with complexity of implementation. Finally, we notice that aspects have a large scale of grain for adaptation: an aspect can be expressive relative to a whole object structure (such as a tree) as well as to a single method in a class.

However, many of our examples were impacted by the instantiation model of AspectJ: the DECORATOR pattern do not allow recursive composition and is oblivious to the OBSERVER pattern; the VISITOR pattern requires a class to scope the aspect. Many design patterns implementations from (Hannemann *et al.,* 2002) rely on idioms to implement relationships between objects. We understand this as a sign of the heterogeneous nature of AspectJ, which does not allow to control aspect instances as class instances. These shortcomings obviously result from the AspectJ language design – aiming at simplification and robustness – and as such seem specific to AspectJ, not to aspects in general. Nevertheless, we consider these shortcomings to highlight an important characteristic of aspects languages: how do they define aspect instantiation and scope (during execution)? One should be careful when choosing an aspect language, as this likely constraints aspect control, impact and definition.

Another guideline is that aspect languages do not support all design patterns. We do not expect aspect languages to perform as large as metaprogramming, for example, when it comes to statically upgrade a class (although AspectJ ITDs allow some modifications).

In (Denier *et al.,* 2006), we have started to study composition of design patterns through pattern density and framework development. To generalize composition, we need some formalization or at least some classification. Then we can proceed with an evaluation (of the frequency) of such compositions on the current set of design patterns. Eventually, we expect a catalog of common composition.

The instantiation shortcomings and the use of idioms should not hide opportunities for new solutions triggered by language mechanisms, such as for the VISITOR pattern. We expect patterns concerned with dynamic adaptation to be especially sensitive: this includes STRATEGY, COMMAND, or CHAIN OF RESPONSIBILITY patterns.

It could be interesting to look for a partial automation of aspect composition. For example, the registration process of the OBSERVER pattern can be coupled with that of the COMPOSITE pattern once composed. Tools for detection of aspect interactions can be used in this process.

Acknowledgements

The authors would like to thank the anonymous reviewers for their comments as well as Mikal Ziane, whose remarks help to further improve this article.

7. References

Akşit M. (ed.), *Proc. 2^{nd} Conf. on Aspect-Oriented Software Development (AOSD-2003)*, ACM Press, 2003.

Albin-Amiot H., Idiomes et patterns Java : Application à la synthèse de code et à la détection, PhD Thesis, Ecole des Mines de Nantes and Université de Nantes, 2003.

Aldrich J., "Open Modules: Modular Reasoning in Aspect-Oriented Programming", *Proc. of Foundations of Aspect-Oriented Languages (FOAL AOSD'04)*, 2004.

Bosch J., "Language support for design patterns", *Proc. of TOOLS Europe'96*, Prentice-Hall, 1996, p. 197-210.

Bosch J., "Design Patterns as Language Constructs", *JOOP*, vol. 11, n° 2, 1998, p. 18-32.

Chambers C., Harrison B., Vlissides J., "A debate on language and tool support for design patterns", *Proc. of the 27^{th} Symposium on Principles of Programming Languages (POPL'00)*, ACM Press, 2000, p. 277-289.

Clarke S., Walker R. J., "Composition Patterns: An Approach to Designing Reusable Aspects", *Proc. of 23^{rd} Int'l Conf. Software Engineering (ICSE)*, 2001, p. 5-14.

Colyer A., Clement A., Harley G., Webster M., *eclipse AspectJ*, Addison-Wesley, 2005.

Coplien J., Shmidt D. (eds), *Pattern Languages of Program Design*, Addison-Wesley, 2005.

Denier S., "Traits Programming with AspectJ", *L'objet*, vol. 11, n° 3, 2005, p. 69-86.

Denier S., Cointe P., "Understanding Design Patterns Density with Aspects", *Proc. of the 5^{th} Workshop on Software Composition (SC'06)*, LNCS, Springer-Verlag, 2006. To appear.

Gamma E., Beck K., "JUnit: A Cook's Tour", 2002.

http://junit.sourceforge.net/doc/cookstour/cookstour.htm

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Massachusetts, 1994.

Guéhéneuc Y.-G., Un cadre pour la traçabilité des motifs de conception, PhD Thesis, Ecole des Mines de Nantes and Université de Nantes, 2003.

Gybels K., Brichau J., “Arranging Language Features for Pattern-based Crosscuts”, Akşit (2003), p. 60-69.

Hannemann J., Kiczales G., “Design pattern implementation in Java and AspectJ”, *Proc. of the 17^{th} ACM conference on Object-oriented programming, systems, languages, and applications*, ACM Press, 2002, p. 161-173.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., “An overview of AspectJ”, J. L. Knudsen (ed.), *Proc. of the 15^{th} European Conference on Object-Oriented Programming (ECOOP’01)*, vol. 2072 of *LNCS*, Springer-Verlag, 2001, p. 327-353.

Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M., Irwin J., “Aspect-Oriented Programming”, M. Aksit, S. Matsuoka (eds), *Proc. of the 11^{th} European Conference on Object-Oriented Programming (ECOOP’97)*, vol. 1241 of *LNCS*, Springer Verlag, 1997, p. 220-242.

Kniesel G., Rho T., “Generic Aspect Languages - Needs, Options and Challenges”, *Proc. of the 2^{nd} Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005)*, 2005. http://www.lifl.fr/jfdlpa05/kniesel.pdf

Lieberherr K. J., Palm J., Sundaram R., “Expressiveness and Complexity of Crosscut Languages”, *Proc. of the 4^{th} workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, 2005.

Lieberherr K., Lorenz D. H., Ovlinger J., “Aspectual Collaborations: Combining Modules and Aspects”, *Computer Journal of the British Computer Society*, vol. 46, n° 5, 2003, p. 542-565.

Lieberherr K., Orleans D., Ovlinger J., “Aspect-Oriented Programming with Adaptive Methods”, *Communications of the ACM*, vol. 44, n° 10, 2001, p. 39-41.

Marin M., “Refactoring JHotDraw’s Undo concern to AspectJ”, *Proc. of the 1^{st} Workshop on Aspect Reverse Engineering (WARE2004)*, 2004.

Mezini M., Ostermann K., “Conquering Aspects With Caesar”, Akşit (2003), p. 90-99.

Nordberg III M. E., “Aspect-Oriented Dependency Inversion”, *Proc. of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, 2001.

Ostermann K., Mezini M., Bockisch C., “Expressive Pointcuts for Increased Modularity”, *Proc. of the 19^{th} European Conference on Object-Oriented Programming (ECOOP’05)*, vol. 3586 of *LNCS*, Springer-Verlag, 2005, p. 214-240.

Soukup J., “Implementing patterns”, Coplien and Shmidt (2005), p. 395-412.

Tanter É., Noyé J., Caromel D., Cointe P., “Partial Behavioral Reflection: Spatial and Temporal Selection of Reification”, R. Crocker, G. L. Steele Jr. (eds), *Proc. of the 18^{th} ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003)*, ACM Press, 2003, p. 27-46.

Tatsubori M., Chiba S., “Programming support of design patterns with compile-time reflection”, *Proc. of the Workshop on Reflective Programming in C++ and Java at OOPSLA’98*, 1998, p. 56-60.

Zimmer W., “Relationships between design patterns”, Coplien and Shmidt (2005), p. 345-364.