
Expression and Composition of Design Patterns with Aspects

Simon Denier* — Hervé Albin-Amiot^{*,**} — Pierre Cointe*

* *OBASCO Group*

École des Mines de Nantes – INRIA, LINA

4, rue Alfred Kastler

44307 Nantes CEDEX 3, France

{Simon.Denier,Herve.Albin-Amiot,Pierre.Cointe}@emn.fr

** *Sodifrance*

4, rue du Château de l'Éraudière

44324 Nantes – France

ABSTRACT. Design patterns are a powerful means to understand, model and implement OO micro-architectures. Their composition leads to architectures with interesting properties in terms of variability and evolutivity. However they are difficult to track, modularize and reuse as their elements tend to vanish in the code. We experiment aspect oriented programming as a modular technology to give new insights on the expression of design patterns. We first take a look at singular features of some design patterns to see how aspects deal with their representation. Then we study some cases of composition in the JHotDraw framework, where we analyze various interactions and the way aspects help to express them.

RÉSUMÉ. Les motifs de conception offrent un moyen puissant pour comprendre, modéliser et implémenter des micro-architectures objets. Leur composition donne des architectures aux propriétés de variabilité et d'évolutivité. Cependant ils sont difficiles à tracer, modulariser et réutiliser car leurs éléments ont tendance à disparaître dans le code. Nous utilisons la programmation par aspects pour porter un nouvel éclairage sur l'expression des motifs de conception. Nous observons d'abord comment les aspects représentent les éléments particuliers de certains motifs. Puis nous étudions quelques cas de composition à travers le cadriciel JHotDraw, pour lequel nous analysons diverses interactions et la façon dont les aspects les expriment.

KEYWORDS: design patterns, aspects, composition, reusability, traceability.

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

1. Introduction

Design patterns (Gamma *et al.*, 1994) have long been known as a descriptive catalog of design problems and their solutions expressed in the object-oriented way. They offer abstract descriptions that cover the gap between the model and the implementation, facilitating a mutual understanding. They are supportive of many micro-architectures in the code, for which they enable variability and evolutivity.

However object oriented implementations have failed in two ways. First the bridge between the model and the implementation is not “bulletproof”. Some concrete elements are hardly modularized and tend to be lost in the code. This results in the pattern vanishing in the code (Soukup, 1995) – or in modern stance, a lack of traceability. Another problem which follows is that patterns implementations are hardly reusable, though their generic descriptions are (Tatsubori *et al.*, 1998, Albin-Amiot, 2003). Generally there is no way to get a tangible representation of the pattern in the code.

Another topic of interest is the composition of design patterns. As exemplified 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 an interaction between two patterns. To our knowledge few works have studied this field: most prominent include (Gamma *et al.*, 1994) (section “Design pattern relationships”) and (Zimmer, 1994) who proposes a classification of the different kind of relationships between design patterns. Composition can lead to the pattern density problem: implementations become so entangled that it is nearly impossible to think of a pattern alone and that they lose some of their flexibility.

We propose to address those issues with the help of aspect-oriented programming. As a modular technology, aspects languages offer new mechanisms to express elements of design patterns implementation. Those mechanisms replace some constructs of patterns, or help to modularize the constructs; thus aspects are supportive of tracing and evolutivity by decoupling code. Moreover some works involve detection of interactions between aspects, which is of interest for composition. Aspects can benefit from insights of pattern-driven solutions, on the way to build robust implementations. Finally this enables the study of variability and configuration, as design patterns in the code are often generic infrastructures mixed with specialization for the target concern.

We start from the work of Hanneman & Kiczales (Hannemann *et al.*, 2002) on the expression of single design patterns. In Section 2 we try to get a step further by exploring variations offered by aspect languages. We experiment in particular with AspectJ (Kiczales *et al.*, 2001, Colyer *et al.*, 2005) and one of its feature, inter-type declarations (Cointe *et al.*, 2005). We draw some informal evaluation on design properties of aspects.

Then we follow in Section 3 with some composition samples which we extract from JHotDraw. JHotDraw is a Java GUI framework for structured graphics¹. It has been engineered as a “design exercise” involving many design patterns. This allows for the study of a rich yet accessible set of composition cases. In this preliminary study we experiment with OBSERVER, COMPOSITE, VISITOR and DECORATOR and investigate how they compose: we get aspectized composition as well as composition of aspects. We also regard expressiveness in interactions and observe in particular how pointcuts are affected.

2. Expression of Single Design Patterns

In this part we review most of the implementation features – including common variations – of two patterns, OBSERVER and COMPOSITE, both in an object-oriented and aspect-oriented ways. In order to facilitate the comparison, we divide each description in three panes: structure (participants), behavior (collaborations) and configuration. Although the last one is not part of the standard, we believe it to be relevant with regards to aspects, as configuration involves specific parts which tends to be scattered or tangled.

2.1. *The Many Ways of OBSERVER Pattern*

The OBSERVER pattern is described in (Gamma *et al.*, 1994) as a way to:

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

Most of the time the dependency management is not a primary concern for the observee. However this logic remains deeply embedded in the observee implementation so that it can be qualified as a crosscutting concern. Moreover programming features of aspect oriented languages fit nicely with concepts needed by the OBSERVER pattern. Together this explains why this pattern is an example of choice to demonstrate features of aspect oriented programming.

2.1.1. *OO View*

We review different features of OBSERVER as they occur in OO code.

Structure OBSERVER defines two types of participants, Observer and Observee.

It needs at least one structure per observee to maintain the dependency link.

This is often implemented directly in the observee (or one of its superclasses).

Rarely a general manager will maintain all links between participants.

1. See <http://www.jhotdraw.org/>

Behavior The notification process itself is a primary part of the implementation: it is linked to the structure maintaining the dependency. `Observers` often triggered themselves the notification. This results in advertizing all registered `Observers` *via* an update API.

`Observers` needs only to implement this generic update API. `Observers` need an API to access data in `Observees`: the original interface is generally sufficient. An interface to manage the dependencies (registration, deregistration of observers) is needed: it is implemented right where the structure lies (that is `Observees`).

Configuration This part is informally discussed in pattern, as they are specific to each case, but becomes important when dealing with aspects. First who has the responsibility to register the observers? Second who must perform the notification signals? Where and when are also questions of interest. These two requirements often involve crosscutting.

2.1.2. *AO View*

In this part we will see how the previous elements can be translated in terms of aspect oriented languages. It happens that most “crosscutting” features can be extracted to aspects. Figure 1 gives an example of `OBSERVER` implementation in AspectJ, illustrating some of the choices below.

Structure Depending on the features of the language, the structure can be automatically inserted at weaving time on behalf of the aspect (Figure 1). Or this can be managed by a central aspect instance: this case is also easier than in OO as the registration can be localized in the aspect.

The language can offer a way to directly manage the dependency by way of its instantiation model. That is an aspect instance can be associated to the observed instance at instantiation time, with different grain (class, object).

Behavior The notification process disappears completely, as it is replaced by the join point notification. Notification arises as an advice call. This, depending on the structure, can lead to a direct update (Figure 1) or to a dispatch to all observers. The update process is a very specific part left to observers (there is no crosscutting at this point). Registration/Deregistration may involve an interface on aspect, but can also be managed by pointcuts and advice (Figure 1).

Configuration With aspects one part gains a significant visibility: the points where notifications arise can now be described in terms of join points by mean of pointcuts. This part is also localized in the aspect. As already mentioned, the same can be true for registration. We can also explicitly apply roles by way of interfaces (Figure 1, not shown).

```

public abstract aspect ObserverProtocol {

    /* Role and structure for Subject */
    protected interface Subject { }
    private Vector Subject.observers = new Vector();
    public Vector Subject.getObservers() { return observers; }

    /* Role and update interface for Observer */
    protected interface Observer {
        public void update(Subject s);
    }

    /* Notification process */
    abstract pointcut stateChanges(Subject s);

    after(Subject s): stateChanges(s) {
        for (int i = 0; i < s.getObservers().size(); i++) {
            ((Observer)s.getObservers().elementAt(i)).update(s);
        }
    }

    /* Registration process */
    abstract pointcut registerObserver(Observer o, Subject s);

    after(Observer o, Subject s): registerObserver(o, s) {
        s.observers.addElement(o);
    }
}

```

Figure 1. An abstract, reusable version of OBSERVER with AspectJ. We use inter-type declarations from AspectJ to implement the structure in the *Subject* interface. Notice how the registration is also defined as an advice

2.2. The COMPOSITE Pattern

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

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

It is classified as a structural pattern. This primarily involves an inheritance relationship between a parent class, *Component*, and many subclasses; there are two kinds of subclasses, *Leaf* and *Composite*; *Composite* classes maintain a multiple aggregation relationship to other components. As such, COMPOSITE does not seem to be an interesting case for aspect.

2.2.1. OO View

Structure COMPOSITE defines three types of participants, `Component`, `Composite` and `Leaf`, the two last ones being subclasses of the first. This inheritance relationship is important as it allows for the uniformity of treatment.

Only `Composites` need a structure to hold references to their children. This structure can hold any `Component` subclasses, so as to be able to build a tree.

Behavior The behavioral aspect of COMPOSITE is much more informal: it implies that a generic interface is defined in `Component`. `Leaves` will implement this with their own definition, whereas `Composites` generally forward the request to their children.

COMPOSITES needs to define a child management interface² – although this is sometimes part of the generic `Component` interface. An iteration API to run through children is interesting.

Configuration Configuration is straightforward for COMPOSITE in the OO way, as it primarily involves an inheritance relationship to define roles. Then `Leaf` subclasses simply override methods from the generic API to define their specificity.

2.2.2. AO View

Since COMPOSITE involves few crosscutting, the intention is to extract reusable features as much as possible. Figure 2 gives the Hanneman solution, which contrasts with our OBSERVER implementation (Figure 1) by the language features used.

Structure Same as for OBSERVER, the structure for `Composite` can be either introduced at weaving time into classes, or implemented by an aspect instance which maintains a relationship with the `Composite` (Figure 2).

Behavior As the aspect is the only one knowing of the structure in `Composite`, it is mandatory to define an iteration API inside the aspect. Another solution is to define a dedicated VISITOR instance, which can perform the recurring behavior of forwarding to children in `Composite` (Figure 2). There is also the possibility to register children by way of pointcuts and advice.

Configuration Configuration is similar to OO with role imposition and refinement of actions (by instances of VISITOR – Figure 2). If significant, pointcuts will target points where children registration should occur.

2. The process to register a child should assert there is no loop in the tree structure – by registering an ancestor as a child. This is rarely (never) done in practice as it is assumed the pattern is used correctly.

```

public abstract aspect CompositeProtocol {

    /* Roles */
    public interface Component {}
    protected interface Composite extends Component {}
    protected interface Leaf extends Component {}

    /* Structure */
    private WeakHashMap perComponentChildren = new WeakHashMap();

    /* Registration API */
    private Vector getChildren(Component s) {
        ...
    }

    public void addChild(Composite composite,
                        Component component) {
        getChildren(composite).add(component);
    }

    public Enumeration getAllChildren(Component c) {
        return getChildren(c).elements();
    }

    /* Visitor pattern merged */
    protected interface Visitor {
        public void doOperation(Component c);
    }

    public void recurseOperation(Component c, Visitor v) {
        for (Enumeration enu = getAllChildren(c);
            enu.hasMoreElements();)
        {
            Component child = (Component) enu.nextElement();
            v.doOperation(child);
        }
    }
}

```

Figure 2. *The abstract, reusable version of COMPOSITE in Hanneman. A noteworthy feature is the integrated VISITOR pattern*

2.3. Evaluation of Properties

From the two cases above we draw some preliminary thoughts about the benefits of aspectized patterns but also on their potential drawbacks.

2.3.1. On Legibility, Understanding and Traceability

The legibility of a pattern is improved as far as its definition (including pointcuts) is localized in the aspect – thus there is no need of specific tool support to review it. Understanding a pattern alone is improved as long as it decouples its implementation from the base code. However this is not sufficient to get a full understanding of the pattern with its context: you still need tool support to locate joinpoints and introspect interactions at these joinpoints with other concerns.

If improvement in OBSERVER is efficient with the extraction of many notification points, the case is much more prone to discussion for COMPOSITE: indeed there is less tangling and scattering in the elements of COMPOSITE we extract to aspect. We can wonder if we gain some legibility by decoupling structure and interfaces from Composite.

Traceability relies on the ability to locate elements of patterns – which is immediate – but also on the ability to separate multiple instantiations of the same pattern based on their specificity. In AspectJ two mechanisms help with regards to this: first we can declare some aspects as abstract then specialize them with concrete subaspects, configuring pointcuts and methods. Second there is a range of instantiation model for aspects, from a particular cflow to a singleton instance: this helps in keeping the specific information at the level it belongs. However we must rely on the implementation: there is no enforcement in separating aspect instances and thus pattern instances.

2.3.2. On Variability, Evolutivity and Robustness

Variability takes into account the ease of defining and configuring different instantiations of the same pattern: in this sense it is closely related to tracing those multiple instantiations. One task which remains difficult when configuring a pattern is capturing the right context (for example, collecting parameters at runtime). This can even lead to the definition of more abstractions, such as pointcut and advice.

Evolutivity follows from the properties above: once a concern is localized and identified in an aspect, changes are localized to it. Variability and evolutivity benefit much of decoupling aspect implementation but also instantiation.

Robustness is a parallel concern of evolutivity: it questions how an aspect stands for changes to base code. However current aspect languages deal unevenly with this problem. The principal reason is that pointcuts are fragile as they rely much on syntactic features of the base code: thus, without tool support (versioning changes in shadow joinpoints between each compilation), they are unaware of changes. This question is also related to configuration: for example, how do we define pointcuts in OBSERVER to capture soundful notification points.

2.3.3. *On Reusability*

Reusability stands on the properties above: the truly generic, reusable part of the pattern implementation comes with traceability. But the pattern must also take into account variations in its specific parts. OBSERVER, with pointcut and method redefinition, stands well in most cases. COMPOSITE shows a bit harder case: it needs a VISITOR to be truly generic and reusable. We will come back on this in 3.4.

3. Composition of Design Patterns

Composition of patterns is a complex case in a traditional OO application, as pattern code tend to be diffused across multiple classes and intermingled with code from multiple concerns. But aspect oriented programming, as it offers means to abstract the behavioral relationship between concerns, should make it easier to model the way design patterns interact. This leads to the following problem: either we can express a composition of design patterns as a composition of their aspectized forms, or either we can only extract an aspectization of the composition. The first case is obviously more interesting as it allows for a reusable expression of composition. This section provides different cases.

3.1. *Description of the JHotDraw invalidation concern*

The case studies presented below are related with the invalidation concern of the JHotDraw framework. Invalidation is a subset of the screen refreshing process. As soon as figures in the drawing are about to change or have changed, they announce their clipping area in order to optimize the refreshment of screen via Swing. We quickly sketch some structures and mechanisms behind this JHotDraw micro-architecture.

3.1.1. *Drawing and Figures*

A typical JHotDraw application is built around the pair of Drawing and DrawingView. Drawing describes a two-dimensional space and contains Figures created by the user, so that it orders figures to paint themselves. RectangleFigure and arrow lines are examples of basic shape figures. A DrawingView instance is dedicated to one Drawing and can show it on screen. It is one of the main bridge between the base GUI framework (Swing for the time being) and the JHotDraw framework. There can be multiple DrawingView for the same Drawing.

Other figure types help to build structured drawings. First GroupFigure acts as a container of figures, such that it can manipulate a set of figures as a whole. Obviously GroupFigure as well as Drawing is an instance of the COMPOSITE pattern — both are subclasses of CompositeFigure. Second BorderDecorator, an instance of the DECORATOR pattern, can wrap other figures to enhance their appearance by a simple

border. The object graph representing the drawing is a tree, with basic figures as leaves, composites and decorators as nodes, and a `Drawing` instance as a root.

3.1.2. *Invalidation and* OBSERVER

This has an important impact on the way the invalidation concern is implemented. Not surprisingly there are instances of `OBSERVER` lying between figures and drawing views, as views are updated with damaged area from figures. The point is that parents in the tree register only as observers of their immediate children, thus creating a chain of observers from a leaf to the root. In particular this allows parents to take over notifications from children.³

Finally we notice that the invalidation concern is a good candidate for aspect. Invalidation is a minor concern of `DrawingView`. However its implementation, supporting `OBSERVER`, pervades through `Drawing` and the class hierarchy of figures. We can also take into account call points: as of version 5.3 of `JHotDraw`, forty-one invalidating calls are scattered across seventeen classes in the `JavaDrawApp` sample application.

3.2. *OBSERVER and* DECORATOR

First we consider the single `OBSERVER` pattern as an aspect, with a `DECORATOR` pattern being implemented in the object way. An instance of the `DECORATOR` is often located as an intermediary between its component and its hierarchy, so as to intercept all calls and either do something or forward to the component. However it can happen that the pattern is also concerned with crosscutting calls, as `OBSERVER` can perform.

Such is the case with a simple `BorderDecorator`, which wraps the figure with a border. Then the damaged area to be redrawn is the figure area augmented by the border size. But as `Observers` trap direct figure modifications, they will bypass `BorderDecorator` and then only retrieve the initial figure area. This is a matter of configuration of the interaction. However the case is simple: since the control flow of actions mimics the hierarchical structure of `DECORATOR`, we can use a pointcut based on the control-flow (as provided by `AspectJ`) to react to the interaction:

```
call(void Figure+.*(..)) &&
cflowbelow(call(void DecoratorFigure+.*(..)))
```

This pointcut will capture method calls in `Figures` which are called by `DecoratorFigure` methods. Then the aspect can handle this case with a specific advice.

The special case above can be simply eliminated if we consider `DECORATOR` as an aspect. The aspect instance wraps the object and intercepts some requests to the

3. This variation of `OBSERVER` building up a chain is quite common with `COMPOSITE`, and is very close to `CHAIN OF RESPONSIBILITY` in its behavior.

wrappee to insert before/after actions or replace the action and proceed: the DECORATOR will simply override all requests to its wrappee, including crosscutting actions. It also removes the burden of implementing direct forwarding methods. Notice this is a case of conflicting aspects: with AspectJ this can be solved by declaring the precedence of DECORATOR over OBSERVER. However this model does not work when we consider piled up instances of the same DECORATOR aspect.

3.3. OBSERVER *and* COMPOSITE

As described in 2.2, the presence of COMPOSITE leads to object tree. But what we specifically want to observe is modifications within nodes or, to put it another way, some actions that modify nodes state. COMPOSITE affects the behavior of an action so that its control flow mimics the object graph. Thus the behavior above can be summarized as observing a tree (be it an object graph or a control flow graph).

3.3.1. *Semantics for tree crosscutting*

Having such a tree structure allows the programmer to get some nice semantic effects, depending on the action. Takes for example a GroupFigure containing rectangles and arrow lines; then apply a changeAttribute action which targets the arrow mode for lines. Obviously only lines will change in response to the action — rectangles will ignore it. Invalidation will only takes place in lines, and there is no need to advertize it at the GroupFigure level. On the contrary, take a move action. In that case the whole set will be invalidated; such a change can be directly advertized by GroupFigure, without involving the subfigures. These two semantics are just two examples of what we can say about notifications in a tree path.

So from one object tree we focus on many action trees, each coming with its semantics for notification. How can we express that? Programming this in a OO way would impact many actions across many classes. With AOP we have the choice to use a general crosscut language for control flow, which will trigger notifications based on the site in the action tree (Douence *et al.*, 2004). The first semantics, which targets “non-recursive calls”, is expressed as the following pointcut (not existing in AspectJ):

```
call(void Figure+.changeAttribute(..)) &&
!pathabove(call(void Figure+.changeAttribute(..)))
```

It tells that we want all calls to changeAttribute except those which are on the path to the call, *i.e.* recursive calls. The AspectJ current pointcut language (1.2.1) could also express such a semantics, although in a more static way (not involving control-flow pointcut). In this case we must declare involved types, a less generic solution:

```
call(void Figure+.changeAttribute(..)) &&
withincode(void LineFigure+.changeAttribute(..))
```

The second semantics targets “top-level calls” and can be expressed as:

```
call(void Figure+.move(..)) &&
!cflowbelow(call(void Figure+.move(..)))
```

3.3.2. Configuration of composition

The need for such semantics comes from the composition of OBSERVER with COMPOSITE. However the pointcuts presented above are difficult to write, and as such error-prone. Since those semantics seem convenient for most cases of composition, can we find a better way to express those? An idea is to use Java 5 annotations to configure semantics on site, then declare an annotation-based pointcut as provided in the upcoming version of AspectJ⁴:

```
call(@Toplevel void Figure+.*(..)) &&
!cflowbelow(call(@Toplevel void Figure+.*(..)))
```

This pointcut captures any method which declares the @Toplevel annotation in the Figure type. The programmer just has to tag a “top-level action” (such as move) with @Toplevel. The aspect will automatically apply the right advice thanks to the above pointcut.

However notification configuration is not the only matter to think. Registration configuration (of observers) is also a concern. Notifications raise when changes occur within nodes; registration occurs when the tree itself changes. What is interesting here is that we can make a link between the registration process of COMPOSITE and the registration process of OBSERVER. As a figure is added to a drawing (via the COMPOSITE registration), we know we must register the drawing as a listener for the figure. So registration for OBSERVER can be managed by way of pointcuts and advice crosscutting COMPOSITE registration.

3.4. Preliminary Conclusions on Composition

Through the preceding sections we have reviewed different types of compositions: we begin with COMPOSITE (section 2.2) whose generic implementation makes use of VISITOR. So this is not to be strictly speaking a pure pattern implementation. It is the aspectized form of a composition of COMPOSITE and VISITOR. On the other side, we have seen with OBSERVER and DECORATOR a composition of aspectual forms: it is noticeable that such a composition eases the expression, as both patterns seem to work on the same level. Finally the case of OBSERVER and COMPOSITE is a third case: no matter how COMPOSITE is implemented, OBSERVER will crosscut it. This composition is asymmetrical.

4. See <http://eclipse.org/aspectj/doc/next/adk15notebook/index.html> for a description of annotations in the incoming AspectJ 5.

Another point of interest is configuration of patterns: we have seen in Section 2 that it gets highlighted by new features of aspect languages. What we observe in this section is that composition of patterns can involve the configuration of such a composition. Although we can not draw general conclusions by now, we have interesting pieces on the expressiveness of pointcut languages. It appears that patterns which involve a structural relationship between their participants, with deep impact on their behaviors, have lot of impact on OBSERVER. In this case a pointcut language based on control flow offers a fine grain of expression.

4. Related Works

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. Even if this experimentation is really interesting, we expect a more declarative way to proceed by using an aspect language such as AspectJ. In (Bosch, 1996) and (Ducasse, 1997), authors use a language (or an extension of an existing 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. However, aspect languages promise a more general approach and also better mechanisms to modularize (and maybe reuse) design patterns implementation.

The OBSERVER pattern serves as an exercise of choice for aspect languages features. Caesar (Ostermann *et al.*, 2003) pushes ahead its instantiation model. Reflex (Éric Tanter *et al.*, 2003) offers a metaphor of metaobjects as observers of hooksets. However few seems to have 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 (Hanemann *et al.*, 2002). However none of the works above have studied the composition of design patterns.

(Riehle, 1997) presents the notion of *Composite Design Patterns*, that is new patterns built on others showing a synergy. It focus on the description of such patterns with roles and not on their implementation in a language, nor does it explore the problem of unforeseen, conflicting interactions as we experiment.

Many works, such as (Clarke *et al.*, 2001) and (Lieberherr *et al.*, 2003), deal with modularity and reusability of aspects: they contain valuable ideas on the way to configure generic aspects for use.

The Demeter language — DJ for its variant in Java (Lieberherr *et al.*, 2001) — has long been know as a particular case: it embodies the VISITOR pattern rather than OBSERVER. However (Lieberherr *et al.*, 2005) shows similarities between DJ and AspectJ with regards to the relational aspect of their pointcut languages. We can take our composition of OBSERVER and COMPOSITE in 3.3, which translates structural relationships into behavioral ones, as some kind of illustration of that case.

The problem of aspect interactions is one the community is well aware. (Douence *et al.*, 2002) first formally define the problem and suggest operators to resolve conflicts. (Éric Tanter *et al.*, 2004) define such operators in their versatile AOP kernel. AspectJ comes with a simple precedence model to order advice execution at conflicting joinpoints. Also the AspectJ plugin for Eclipse (AJDT⁵) comes with a visualization tool which allows to see interactions of aspect with the base code.

5. Conclusion

By aspectizing design patterns we wanted to take a tour of implementation features offered by aspect languages. For now this mostly covers AspectJ with a glance at other languages. This reveals in particular how much of configuration is involved in the pattern implementation. We take a look at design properties to get a basic evaluation: locality and decoupling promote traceability and evolutivity, and partially understanding. Reusability depends much on the easiness of configuring reusable aspects. The problem of fragile pointcuts persists. Also some pattern implementations seem overkill, as COMPOSITE which needs VISITOR.

First experiments on composition of design patterns lead to interesting cases. In particular there is a need for configuration of composition, which involves aspect ordering as well as pointcut transformation. Indeed the presence of COMPOSITE or DECORATOR in the base code can have an impact on the OBSERVER pointcuts. We show in this case that a general pointcut language based on control flow gives easy and expressive semantics between participants. The experiment unveils different types of composition, from aspectized composition to composition of aspects. However we lack materials to draw a general tendency.

In future work we will experiment how configuration can be partially automated. For example, the registration process in OBSERVER is coupled with that of COMPOSITE once merged. Then, the fact that both patterns are aspectized and the use of interaction detection can trigger such a configuration. We plan to study other compositions, such as MEMENTO and COMMAND, or the composition of COMPOSITE, ITERATOR and VISITOR as aspects. CHAIN OF RESPONSIBILITY seems a good candidate for the kind of interaction involved by the composition of OBSERVER and COMPOSITE. We hope to make a catalog of common aspectization and composition of design patterns. Interesting properties for such compositions remain to be defined.

Acknowledgements

This work is part of the new AOSD network of excellence and its language laboratory (see <http://www.aosd-europe.net>).

5. See <http://www.eclipse.org/ajdt/>

6. References

- Albin-Amiot H., Idiomes et patterns Java: Application à la synthèse de code et à la détection, PhD thesis, École des Mines de Nantes and Université de Nantes, February, 2003.
- Bosch J., « Language support for design patterns », *Proceedings of TOOLS Europe '96*, Prentice-Hall, p. 197-210, 1996.
- Clarke S., Walker R. J., « Composition Patterns: An Approach to Designing Reusable Aspects », *Proc. 23rd Int'l Conf. Software Engineering (ICSE)*, p. 5-14, May, 2001.
- Cointe P., Albin-Amiot H., Denier S., « From (meta) objects to aspects », in , F. de Boer et al. (ed.), *In Proceedings of the 3rd International Symposium on Formal Methods for Components and Objects*, vol. 3657 of *Lecture Notes in Computer Science*, Springer-Verlag, p. 70-94, 2005.
- Colyer A., Clement A., Harley G., Webster M., *eclipse AspectJ*, the eclipse series, Addison-Wesley, 2005.
- Douce R., Fradet P., Südholt M., « A framework for the detection and resolution of aspect interactions », in , D. Batory, , C. Consel, , W. Taha (eds), *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 - Proceedings*, vol. 2487 of *Lecture Notes in Computer Science*, Springer-Verlag, Pittsburgh, PA, USA, p. 173-188, October, 2002.
- Douce R., Teboul L., « A crosscut language for control-flow », *GPCE 2004 - 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, Lecture Notes in Computer Science, Springer-Verlag, Vancouver, Canada, October, 2004.
- Ducasse S., « Réification des schémas de conception : une expérience », *Actes de LMO'97*, p. 95-110, 1997.
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Massachusetts, 1994.
- Hannemann J., Kiczales G., « Design pattern implementation in Java and AspectJ », *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, ACM Press, p. 161-173, 2002.
- Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., « An overview of AspectJ », in , J. L. Knudsen (ed.), *Proc. ECOOP 2001, LNCS 2072*, Springer-Verlag, Berlin, p. 327-353, June, 2001.
- Lieberherr K. J., Palm J., Sundaram R., « "Expressiveness and Complexity of Crosscut Languages" », *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, March, 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, p. 542-565, September, 2003.
- Lieberherr K., Orleans D., Ovlinger J., « Aspect-Oriented Programming with Adaptive Methods », *Communications of the ACM*, vol. 44, n° 10, p. 39-41, 2001.
- Marin M., « Refactoring JHotDraw's Undo concern to AspectJ », *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE2004)*, 2004.

- Ostermann K., Mezini M., « Conquering Aspects With Caesar », in , M. Akşit (ed.), *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, ACM Press, p. 90-99, March, 2003.
- Riehle D., « Composite Design Patterns », *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, ACM Press, p. 218-228, 1997.
- Soukup J., « Implementing patterns », *Pattern languages of program design*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, p. 395-412, 1995.
- Tatsubori M., Chiba S., « Programming support of design patterns with compile-time reflection », *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA'98*, Vancouver, Canada, p. 56-60, oct, 1998. ISSN 1344-3135.
- Zimmer W., « Relationships between design patterns », in , J. O. Coplien, , D. C. Schmidt (eds), *Pattern Languages of Program Design*, Addison-Wesley, 1994.
- Éric Tanter, Noyé J., Versatile Kernels for Aspect-Oriented Programming, Research Report n° RR-5275, INRIA, July, 2004.
- Éric Tanter, Noyé J., Caromel D., Cointe P., « Partial Behavioral Reflection: Spatial and Temporal Selection of Reification », in , R. Crocker, , G. L. Steele, Jr. (eds), *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003)*, ACM Press, Anaheim, California, USA, p. 27-46, October, 2003.