Understanding Design Patterns Density with Aspects: A Case Study in JHotDraw using AspectJ

Simon Denier, Pierre Cointe

Understanding Design Patterns Density with Aspects

a Case Study in JHotDraw using AspectJ

Simon Denier and Pierre Cointe

OBASCO
École des Mines de Nantes, INRIA, LINA,
4, rue Alfred Kastler, Nantes, France
{sdenier, cointe}@emn.fr

Abstract. Design patterns offer solutions to common engineering problems in programs [1]. In particular, they shape the evolution of program elements. However, their implementations tend to vanish in the code: thus it is hard to spot them and to understand their impact. The problem becomes even more difficult with a "high density of pattern": then the program becomes easy to evolve in the direction allowed by patterns but hard to change [2]. Aspect languages offer new means to modularize elements. Implementations of object-oriented design patterns with AspectJ have been proposed [3]. We aim at testing the scalability of such solutions in the JHotDraw framework. We first explore the impact of density on pattern implementation. We show how AspectJ helps to reduce this impact. This unveils the principles of aspects and AspectJ to control pattern density.

1 Introduction

Design patterns [1] are well-known couples of problem-solution for program engineering. They shape the structure and the interface of their targets, and redefine some behaviors. Most design patterns aim at decoupling concerns, in particular to allow separate evolution. However, the shape they impose disallows evolution in other directions. Also, the programmer must often make a tradeoff between the impact of the pattern and the properties he wants from it, which results in distortions from the standard pattern. Then implementations of design patterns suffer from lack of traceability: some elements tend to be lost and pattern identity itself is hard to trace back to the model — such that the pattern is said to "vanish" in the code [4].

The impact of design patterns on implementation, their tendency to shape evolution, and the difficulty to trace them in the code raise questions when software grows in complexity. But the implications of design patterns in complex software is not well understood. Most prominent work includes the study of relationships between design patterns, such as [1] (section "Design pattern relationships") and [5], who proposes a classification of different kind of relationships.

This includes patterns making use of other patterns in their implementation as well as interactions between two patterns. [2] shows in the context of JUnit that mature frameworks tend to have a high density of patterns: then they are "easy to use, but hard to change". Implementations become so entangled that it is nearly impossible to think of a pattern alone and that they can lose some of their flexibility.

Aspect-oriented languages à la AspectJ [6, 7] offer new means to modularize software elements. [3] shows some general aspectizations of the GoF design patterns [1]. Aspect-oriented languages allow:

- modularization of crosscutting pattern elements;
- better separation between a generic (reusable) part and a specific part;
- language-level detection and visualization tools for interactions;
- pluggability of modules to replace a pattern implementation by another.

Our aim is to test the scalability of such aspectizations of design patterns in a real application. We experiment with the JHotDraw framework[1] as it is a well documented "design exercise" involving many design patterns.

Our guideline is to look for an incremental and reversible development of JHotDraw. We start with a basic yet functional framework (called the *base* thereafter). We then "compose" new modules into the base, incrementally enhancing the framework, but still with the option to come back to earlier versions. By doing so we underline design choices which happen through the whole program when more functions are composed together, but get lost because there is no mean to trace such choices to the module they support. Such a guideline allows for a deeper separation of concerns highlighting the development process and product versions.

Section 2 presents JHotDraw *base* (internals and design patterns), as well as the specific example of the "invalidation" concern. Section 3 examines the invasive impact of two additional concerns on the base, related to the Observer, Composite and Decorator patterns. Section 4 shows how such impact can be modularized with AspectJ constructs. We follow with some discussions in Sect. 5, related work in Sect. 6 and conclude in Sect. 7.

2 An Overview of the JHotDraw framework

2.1 General Architecture

Figure 1 shows main interfaces and relationships involved in the JHotDraw framework *base*. It gives a feeling of how a JHotDraw application works and what can be extended. We now explain the responsibilities and collaborations of each interface, in particular with regard to framework extension:

- **DrawingEditor** is the base interface for the application. It maintains a link to the active `DrawingView` and to the current tool. Extensions usually define the GUI, instantiate tools and drawing views.

[1] We use JHotDraw 5.3 – available at `http://www.jhotdraw.org/`

– **DrawingView** displays one drawing. It holds interactions between the user and the drawing, as well as graphics with Swing. This is apparent as it is linked to `JPanel`, maintains a link to the list of currently selected figures, and has access to the current tool *via* the editor. The default implementation class fulfills all these roles.
– **Drawing** acts as a container and a uniform layer to manage a set of figures.
– **Figure** is a central entity to the user. Depending on the application, it can spawn a large tree of derived figures, such as rectangle, circle, line, or more structured figures.
– **Tool & Handle** allow to create or manipulate `Figure`s, either as a whole (select, move) or focusing on a specific property (size, radius). Tools and handles are notified of user interactions by the drawing view. They too can spawn a large tree of derived classes depending on the application.

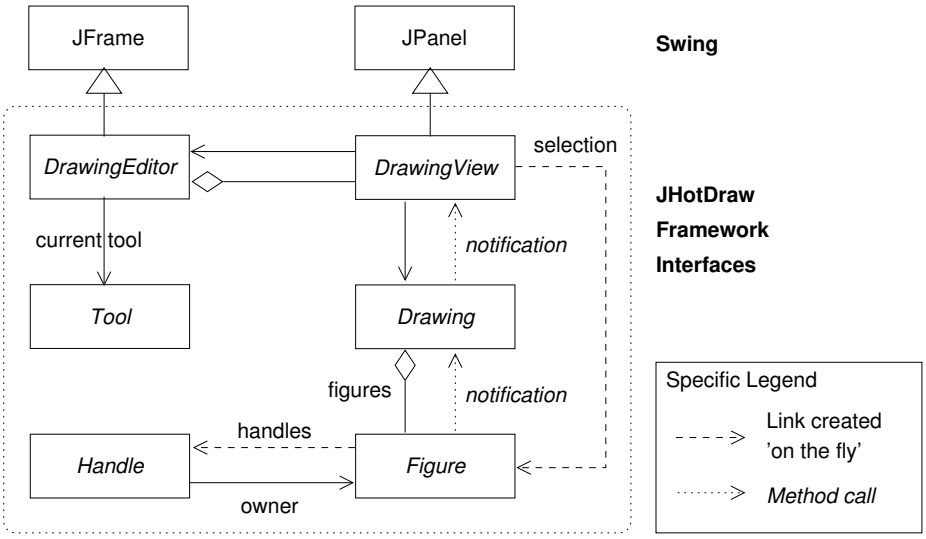

Fig. 1. A synthetic diagram of main interfaces and relationships in JHotDraw. The inheritance relationship between Swing class `JFrame` and JHotDraw interface `DrawingEditor` is a shortcut to the real relationship between `JFrame` and a `DrawApplication` class implementing `DrawingEditor`. The same is true for `JPanel` and `DrawingView`

2.2 Design Patterns Involvement

We now quickly sketch how design patterns are involved in this general architecture (Fig. 1) and in the underlying implementation. The design patterns exposed

here are documented in the code. The list below shows the importance of patterns to define relationships (MEDIATOR, OBSERVER) as well as to extend the framework (STATE, PROTOTYPE, STRATEGY, ADAPTER, FACTORY METHOD). These patterns build up the JHotDraw framework *base*.

- **Mediator:** `DrawingEditor` is a mediator between the current tool and the drawing view.
- **Strategy:** some `DrawingView` activities such as painting the drawing or grid constraining are configured with strategies.
- **Observer:** there are two occurrences of the OBSERVER pattern. One is lying between figures and the drawing, the other between a drawing and a drawing view. They basically serve to update the drawing and the view in response to figure modifications. One such concern is detailed in Sect. 2.3.
- **State & Prototype:** by switching between tools for the current tool state, the user changes the behavior he wants to apply in the drawing view context. Creation tools create figures by copying a prototype figure.
- **Adapter & Factory Method:** `Handler` adapts the `Figure` interface to respond to mouse events. The strong link between a figure and its specific handles is enforced by a factory method in `Figure`.

Finally two other patterns are worth mentioning for the purpose of this article: an occurrence of the COMPOSITE pattern with `CompositeFigure`, to manipulate a group of `Figure`s as a single entity; and an occurrence of the DECORATOR pattern with `DecoratorFigure`, which adds singularities on the target figure (a border for example).

We now focus on the relationship between `DrawingView`, `Drawing` and `Figure`. We will survey how design patterns are involved in a particular concern.

2.3 Updating a View: the Invalidation Concern

Whenever a figure changes, the display view has to be updated to reflect those changes. Following a classic optimization, the area to be redrawn is clipped in order to speed up refreshing and avoid screen flashing. The process of updating a view takes two steps: first compute the clipping area and announce it to Swing, then draw on the graphics context when required by Swing. We call the first step the invalidation concern: its purpose is to collect damaged area from figures before sending a repaint request to Swing.

The obvious way to do that is to let figures announce their own clipping area whenever they change. They should notify their `Drawing`. The clipping area of a figure can be simply defined by its bounding rectangle. The clipping area for the drawing can be defined by the union of all clipping areas notified between two updates.

Obviously, this concern can be implemented with an OBSERVER occurrence between `Figure` and `Drawing`:

- when a figure is added to a drawing (after its creation for example), the drawing is registered as an observer for the figure;

– when performing some actions (such as move, change color), the figure notifies its drawing with its bounding box;
– when notified, the drawing adds the bounding box to its clipping area;
– when a figure is deleted, it deregisters the drawing as observer.

Another occurrence of OBSERVER stands between `Drawing` and `DrawingView`:

– a drawing registers its drawing view as observer;
– on request the drawing sends its clipping area to the drawing view (which forwards to Swing); after this point the clipping area can be reset.

Notice how the invalidation concern is itself decomposed in two steps: collecting the clipping area in `Drawing` and notifying Swing in `DrawingView`. This relationship is summarized in Fig. 1 by the two arrow lines for notification.

3 Study of Pattern Density in JHotDraw

The functionalities described in Sect. 2 define the JHotDraw framework base. We examine two additions to this base, both related to design patterns and the invalidation concern. Our goal is to understand how the current framework (which already contains these additions) differs from the basic one and to examine the impact in term of implementation. We do this in the spirit of incremental evolution exposed in the introduction.

3.1 Impact of Composite and Decorator on Invalidation

In the base framework all figures are direct children of the drawing and refer to it for the invalidation process. When the COMPOSITE pattern is used to manipulate a group of figures as a single entity, figures trees can be constructed (`GroupFigure`, Fig. 2). The same is true for the DECORATOR pattern with `BorderDecorator`. Then, since we can have any number of levels between the drawing and figures, does it affect the invalidation concern?

We first consider `GroupFigure`. It just merges clipping areas of figures underneath. There is no difference in merging at the group level or at the drawing level. So invalidation concern is not affected: figures can directly notify the drawing, and `GroupFigure` is not involved in this OBSERVER pattern. This solution is labelled A in Fig. 2.

On the contrary, `BorderDecorator` has the property to redefine the clipping area of the figure underneath. It enhances the bounding box by the size of its border. How do we notify the drawing that the clipping area of a figure should be enhanced? Obviously, solution A does not work since `BorderDecorator` does not have a chance to notify its change. We can think of other solutions:

– B: all `Figure`s notify the `Drawing` each time a command is transmitted;
– C: base `Figure`s notify their direct parent `Figure` of the change; parent can change the notification; recursively, the notification traverses the hierarchy to the `Drawing`;

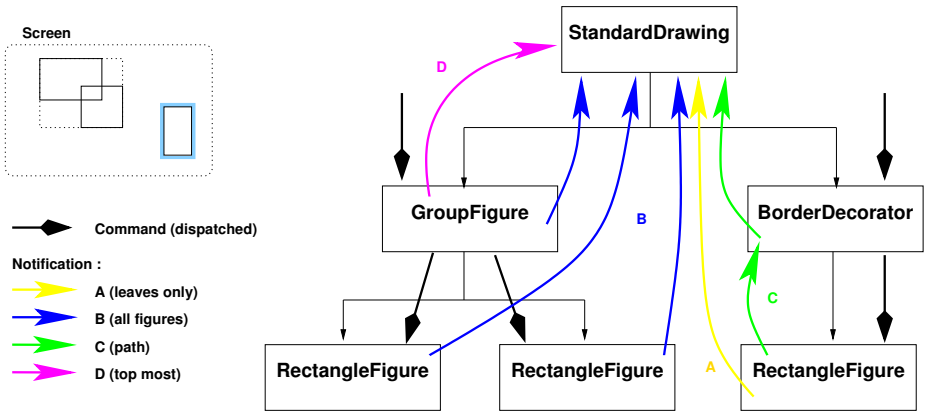

Fig. 2. Figures organized in a tree with Composite and Decorator patterns. In the upper left corner is a sample of figures display. The object diagram shows different strategies to deal with notification of invalidation (Sect. 3.1)

– D: in some cases, a parent `Figure` can directly notify the `Drawing`.

Solution B is very easy to implement, since all figures classes can inherit from an abstract class with the link to drawing. However, `GroupFigure` and `BorderDecorator` will trigger notifications every time: they can trigger false notifications since they do not know when their children really change. Notifications of children are redundant with those of `GroupFigure` and `BorderDecorator` yet it is impractical to inhibit them for temporary time.

Solution C is much more elegant with respect to false notification: `GroupFigure` and `BorderDecorator` will notify only if they receive notifications from children. This is the time for `BorderDecorator` to grow the clipping area. However, this has a cost in term of implementation: `GroupFigure` and `BorderDecorator` are subjects but also observers. This deeply changes the design as we now have many observers which are linked together in a chain, up to the drawing. In fact, the design for the invalidation concern is now that of a Chain of Responsibility pattern [1].

Solution D aims at reducing redundancy. A command such as "move figures" will automatically trigger changes in target figures. Then a `GroupFigure` can trigger the notification at the top level, computing the clipping area, and avoid notifications in levels underneath. The difficulty is to temporarily disable such notifications: however, the benefits seem too low for the cost of implementation in object-oriented languages.

The current JHotDraw framework chooses solution C. Figure 3 gives some details on the implementation of this solution. It allows to easily redefine for each `Figure` observer how it handles notifications (`CompositeFigure`, lines 17–18) and, in particular, if it changes them (`BorderDecorator`, lines 26–29). However, code complexity is increased. First, we notice that the change is only needed for the purpose of invalidation with `BorderDecorator`; but the change also affects

the composite class, due to the chaining solution. Second, as said above, the impact of the COMPOSITE and DECORATOR patterns is to transform the OBSERVER pattern into a CHAIN OF RESPONSIBILITY pattern, where each handler can be seen as an observer of its children.

```
1  abstract class AbstractFigure implements Figure { // Subject role
       private FigureChangeListener observer;
3      (...)
       public void moveBy(int dx, int dy){ invalidate(); (...) }
5      public void invalidate(){
         Rectangle r = displayBox(); // clipping area
7        observer.figureInvalidated(new FigureChangeEvent(this, r));
       }
9      public abstract Rectangle displayBox();
   }
11 interface FigureChangeListener { // Observer role
       public void figureInvalidated(FigureChangeEvent e); }
13
   class CompositeFigure extends AbstractFigure // Composite role
15   implements FigureChangeListener {
       (...)
17     public void figureInvalidated(FigureChangeEvent e){
           observer.figureInvalidated(e);} // simple forward
19 }
   class GroupFigure extends CompositeFigure {...}// Composite extension
21
   class DecoratorFigure extends AbstractFigure // Decorator role
23   implements FigureChangeListener {...}
   class BorderDecorator extends DecoratorFigure {// Decorator extension
25     (...)
       public void figureInvalidated(FigureChangeEvent e){
27         Rectangle r = e.getInvalidatedRectangle();
           r.grow(factorx, factory); // grow by size of border
29         observer.figureInvalidated(new FigureChangeEvent(this, r));}
   }
```

Fig. 3. Implementation of the OBSERVER pattern impacted by COMPOSITE and DECORATOR patterns in the current JHotDraw framework. `AbstractFigure` defines main parts of the *Subject* role, including the reference (line 2) to the *Observer* role, which is reified by the `FigureChangeListener` interface. `figureInvalidated` (line 12) is the notification method for the invalidation concern. Both `CompositeFigure` and `DecoratorFigure` inherit from `AbstractFigure` to be subjects and implement `FigureChangeListener` to be observers. While the default behavior for `figureInvalidated` is to forward the event (see `CompositeFigure`, lines 17–18), `BorderDecorator` must redefine this method to take account of its specifity (lines 26–29)

3.2 Evolving to Multiple Drawing Views

The base JHotDraw framework allows to build single-window applications (Fig. 6, left). There is only one drawing view, which can be managed by a SINGLETON pattern. This considerably simplifies the implementation of the OBSERVER pattern between a drawing and its drawing view (Fig. 4).

An extension of JHotDraw allows to build MDI (Multiple Document Interface) applications, allowing multiple drawing views on the same drawing (Fig. 6, right). The implementation is primarily supported by Swing and internal frames.

```
class StandardDrawing (...) implements Drawing {
    (...)
    public void figureInvalidated(FigureChangeEvent e){
        StandardDrawingView.instance().drawingInvalidated(
                new DrawingChangeEvent(this,
                                       e.getInvalidatedRectangle()));}
}
```

Fig. 4. Simple implementation of the *Subject* role for the `Drawing`–`DrawingView` OBSERVER pattern, with `DrawingView` as a SINGLETON pattern

The extension is almost modular since the base framework for single windows is not modified – except for the OBSERVER pattern in Fig. 4 which does not allow multiple observers per drawing. We need to change its implementation according to Fig. 5. This new implementation works in both singleton and multiple cases, but we have lost the simple choice of the single window framework.

```
class StandardDrawing (...) implements Drawing {
    (...)
    private Vector<DrawingView> observers
        = new Vector<DrawingView>();
    // when a view is linked to a drawing, it must call this method
    // to register itself as an observer
    public void addObserver(DrawingView view) { ... }
    public void removeObserver(DrawingView view) { ... }
    public void figureInvalidated(FigureChangeEvent e){
        for( DrawingView view: observers )
            view.drawingInvalidated(
              new DrawingChangeEvent(this,
                                     e.getInvalidatedRectangle()));}
}
```

Fig. 5. Implementation of the *Subject* role for the `Drawing`–`DrawingView` OBSERVER pattern, modified to handle multiple `DrawingViews`. For brevity, the original code has been rewritten using Java 5 generics and the new `for` loop

3.3 Impact of Pattern Density

Pattern density is a sign that the program design becomes complex, but it does not mean that patterns themselves are complex: the combination of the COMPOSITE, DECORATOR and OBSERVER patterns which form a CHAIN OF RESPONSIBILITY pattern is fairly easy to configure. The OBSERVER pattern is even simpler with a SINGLETON pattern. However, our short study shows that such a combination can have deep impact on implementation.

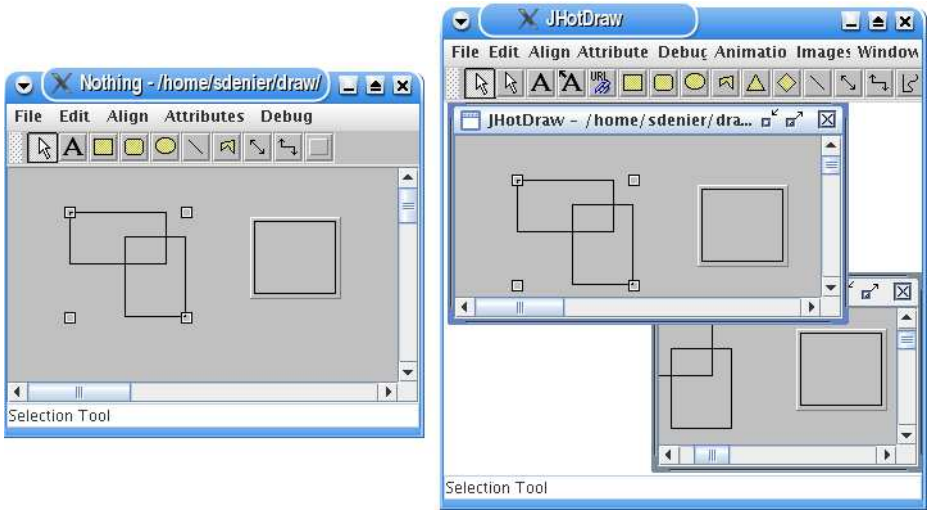

Fig. 6. Two JHotDraw applications: on the left, single view per drawing; on the right, multiple views on the same drawing.

4 Pattern Density with Aspects

We now investigate the invalidation concern and the above additions with AspectJ. We want those additions to be both incremental and reversible. The process is three-fold and can be summed up as:

1. a "classic" aspectization of OBSERVER patterns for the invalidation concern;
2. configuration of OBSERVER pointcuts to deal with COMPOSITE and DECORATOR patterns;
3. use of modularity and pluggability of aspects to deal with the presence or absence of the SINGLETON pattern.

4.1 Aspectization of the Invalidation Concern

We extract the whole invalidation concern from the base classes (resp. interfaces): `AbstractFigure` (resp. `Figure`), `StandardDrawing` (resp. `Drawing`), and `StandardDrawingView` (resp. `DrawingView`). This also includes many call points to the `invalidate` method (see `AbstractFigure.moveBy` in Fig. 3), scattered through the `Figure` hierarchy[2].

The `DrawingDamage` aspect (Fig. 7) structurally modifies `Drawing` classes to introduce a field called `damageArea` (line 2) and its control logic. The introduced `addDamage` method saves and merges clipping areas in this field (lines 3–5). The introduced `getAndResetDamage` method retrieves the clipping area of the drawing and resets it on purpose of the refresh process (lines 6–8).

[2] We count up to forty-one invalidating calls scattered across seventeen classes, with standard extension such as `GroupFigure` and `BorderDecorator` included.

```
aspect DrawingDamage {
    private Rectangle Drawing.damageArea;
    void Drawing.addDamage(Rectangle newDamage){
        if (damageArea == null) damageArea = newDamage;
        else damageArea.add(newDamage); }
    Rectangle Drawing.getAndResetDamage(){
        Rectangle r = damageArea; damageArea = null;
        return r; }
}
```

Fig. 7. The `DrawingDamage` aspect, which introduces new field and methods in `Drawing` subclasses for the invalidation concern

The `GetFigureDamage` aspect (Fig. 8) supports observation between a figure and its drawing. Similar to the reference to a `FigureChangeListener` (see Fig. 3, line 2), it introduces in each figure a reference to a drawing (`myListeningDrawing`, line 3). Registration is directly performed *via* pointcut and advice (lines 4–8). Notification pointcuts extract all previous method calls to `invalidate` which where scattered in `Figure`s methods (lines 12–20). The description by pointcuts is not especially shorter but is localized in the aspect. Finally the invalidate action triggered by advice makes use of the damage interface introduced in `Drawing` by `DrawingDamage` (lines 24–27).

The `RepairSingleView` aspect (Fig. 9) supports the second observer and the refresh logic (refresh logic was not shown in Sect. 3.2 but follows the same principle). We consider the singleton case for the drawing view: there is no need for an observer reference. Pointcuts extract requests for screen update (usually after an user operation — line 3). The advice notifies the singleton observer (lines 8–10) which then performs the Swing request (lines 15–16).

The code above shows no more than common benefits we expect from aspects: scattered code for notifications, structure and methods relevant to the invalidation concern are localized in aspects. We should note that the invalidation concern and the Observer pattern are typical examples of crosscutting concerns. We now examine issues from Sects. 3.1 & 3.2 with the help of AspectJ.

4.2 Revisiting Composite and Decorator Interactions

We consider the four strategies envisionned in Sect. 3.1 for invalidation of figures. Code from Fig. 8 implements **solution B** by default. Indeed `GroupFigure` and `BorderDecorator` are `Figure` *via* their respective superclass. The `GetFigureDamage` aspect is oblivious to the dynamic type of `Figure` instances.

It follows that **solution A** requires more effort. We must explicitly exclude `CompositeFigure` and `DecoratorFigure` from invalidate pointcuts. For example the `changed` pointcut must be rewritten as:

```
pointcut changed(Figure f):
    this(f) && execution(void Figure+.setAttribute(..))
    && !this(CompositeFigure) && !this(DecoratorFigure);
```

```
1  aspect GetFigureDamage {
       // Registration of drawing (observer) in figure
3      private Drawing Figure.myListeningDrawing;
       pointcut registerFigure(Drawing d, Figure f):
5          execution( Figure CompositeFigure.add(Figure) )
           && this(d) && args(f);
7      after(Drawing d, Figure f): registerFigure(d, f) {
           f.myListeningDrawing = d; }
9      (...)

11     // Notification of changes
       pointcut willChange(Figure f):
13         (execution(void Figure+.displayBox(Point, Point))
           || execution(void Figure+.moveBy(..))) && this(f);
15     before(Figure f): willChange(f){ invalidate(f); }
       after(Figure f):  willChange(f){ invalidate(f); }
17
       pointcut changed(Figure f):
19         this(f) && execution(void Figure+.setAttribute(..));
       after(Figure f):  changed(f){ invalidate(f); }
21     (...)

23     // Action on notification
       void invalidate(Figure f){
25         if( f.myListeningDrawing!=null ){
             f.myListeningDrawing.addDamage( f.displayBox() );
27         } (...) }
   }
```

Fig. 8. Sample from `GetFigureDamage` aspect, which supports the observer relationship from figures to their drawing. Pointcuts and advice are used both for registration and notification of the observer. Pointcut `willChange` (lines 12–15) stands for actions which invalidate both the old bounding box (where the figure used to be) and the new bounding box: such actions (move, resize) are advised before and after their execution (lines 15–16). Pointcut `changed` (lines 18–19) is used solely for actions which modify the inner appearance of the figure but not its bounding box: then notification occurs only after action (line 20)

```
   aspect RepairSingleView {
2      // Notifications (request for update)
       after(): execution(void StandardDrawingView.mousePressed(..)){
4          repairDamage(StandardDrawingView.instance().drawing()); }
       (...)
6
       // Action on notification
8      private void repairDamage(Drawing d){
           Rectangle r = d.getAndResetDamage();
10         StandardDrawingView.instance().repairDamage(r); }
   }
12
   class StandardDrawingView extends JPanel implements DrawingView {
14     (...)
       public void repairDamage(Rectangle r) { // Swing request
16         if (r != null) { repaint(r.x, r.y, r.width, r.height); }}
   }
```

Fig. 9. Sample from `RepairSingleView` aspect with singleton view configuration

This strategy is initially not interesting and loses even more appeal following such constraints. The `this(Type)` predicate can be translated as a dynamic `this instanceof Type` test in some cases.

Solution C is interesting: we do not need to transform the OBSERVER pattern into the CHAIN OF RESPONSIBILITY pattern to get the same effect. The case involves solely `BorderDecorator`: we only target figures which have a `BorderDecorator` in the chain of parents. If there is to be a change down in the chain, necessarily the clipping area will be that of the top most decorator. The process of detecting the top most decorator and passing it down to the triggering figure can be managed by pointcuts:

```
pointcut targetaction():
    execution(void BorderDecorator.setAttribute(..));
pointcut topmostdecorator(BorderDecorator bd):
    this(bd) && targetaction()
    && !cflowbelow(targetaction());
pointcut changed(Figure f):
    execution(void Figure+.setAttribute(..))
    && cflowbelow(topmostdecorator(f));
after(Figure f): changed(f) { invalidate(f); }
```

The `topmostdecorator` pointcut captures any execution of method `setAttribute` which are *not* in the control flow of another `BorderDecorator`: the decorator is then the top most. The `changed` pointcut will capture any execution of `setAttribute` which are under a `BorderDecorator`. But, instead of notifying invalidate with the current figure, it will use the parameter of `topmostdecorator`. The clipping area retrieved by `invalidate` (Fig. 8, line 26) will be that of the decorator.

The `changed` pointcut captures all executions below the top most decorator, including other decorators. This is not intended: only "leaves" (such as `RectangleFigure`) will trigger real modifications. Currently there is no mean in the AspectJ language to capture leaves in the control flow. An extension to the language is proposed in [8]. We could also use `!this(DecoratorFigure)` such as in solution A.

Solution D is more simple. We do not want all figures to trigger notifications, when we are sure that they will change. We simply trigger notifications for top most calls. For example move command can be notified at the top most level, by a figure, a composite or a decorator. The `willChange` pointcut can be rewritten:

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

Preliminary conclusion shows that the AspectJ pointcut language is expressive enough to implement the four notification strategies. Contrary to the object solution, there is no need to change `Figure` subclasses, `CompositeFigure` and `Decoratorfigure`. However, there is the hidden cost of using AspectJ dynamic

construct such as `cflow`. Currently we lack quantitative benchmarks on the performance of `cflow` with respect to the CHAIN OF RESPONSIBILITY solution, although this is not perceptible in the context of JHotDraw.

4.3 Pluggability of Aspects: Revisiting Multiple Views

Same as the observer in Fig. 4, `RepairSingleView` does not work with multiple views. Yet, we simply build a new `RepairMultipleViews` aspect (Fig. 10). Contrary to Fig. 5, `StandardDrawing` is not changed. The framework user can choose at weaving time which configuration (singleton or multiple views) he needs. A drawback is that there is no reuse between `RepairSingleView` and `RepairMultipleViews`, so that some change in base code could impact both aspects. However, it is possible to share some definitions (such as pointcuts for notification) using AspectJ abstract aspect and extension mechanism.

```
1  aspect RepairMultipleViews {
       // (De)registration of drawing views in drawing
3      private List<DrawingView> Drawing.listeningViews
           = new LinkedList<DrawingView>();
5      pointcut linkViewToDrawing(DrawingView view, Drawing drawing):
           execution(void DrawingView+.setDrawing(Drawing))
7          && this(view) && args(drawing);
       before(DrawingView v, Drawing d): linkViewToDrawing(v, d){
9          (...)
           v.drawing().listeningViews.remove(v);
11         d.listeningViews.add(v); }

13     // Notifications (request for update)
       after(DrawingView v): this(v)
15         && execution(void StandardDrawingView.mousePressed(..)){
           repairDamage(v.drawing()); }
17     (...)

19     // Action on notification
       private void repairDamage(Drawing d){
21         Rectangle r = d.getAndResetDamage();
           for (DrawingView view : d.listeningViews) {
23             view.repairDamage(r); }}
   }
```

Fig. 10. Sample from `RepairMultipleViews` aspect for multiple views (MDI) configuration. `Drawing` manages a list of drawing views which are its observers (lines 3–11). Since a view displays one drawing at a time, its registration on a new drawing involves its deregistration from the previous drawing (lines 10–11). Another change from Fig. 9 is the capture of the contextual view during notification (line 14)

5 Discussions

Before concluding, we present two subjects of discussion inspired by this work. They are complementary to this study but, to this day, rely much on subjective opinion.

5.1 Specificity of the AspectJ Solution

The specificity of the cflow-based AspectJ solution in Sect. 4.2 can be compared to a language where inspection of the execution stack is possible. Of course, a cflow construct can be easily emulated in such a language. However, the cflow construct combined with aspects allows to easily "compose" modules and patterns without modifying the base code. The fact that such a modification can be modularized simply with a stack inspector remains to be evaluated.

Intertype declaration (previously introduction) is another feature of AspectJ which is frequently used in pattern implementation (see Figs. 7, 8 or 10). This feature can partially emulate mixin or trait-like reuse [9].

5.2 Avoiding Implementation Overhead in a Field of Patterns

Without a reusable pattern library, programmers need to implement design patterns over and over: this leads to "implementation overhead" [10] when it comes to patterns with heavy, repetitive elements. When pattern density rises and the same pattern is being used over the same classes, there is a natural tendency in object-oriented languages to fuse concerns together in order to reuse pattern implementation and reduce the overhead. Thus reusability is enhanced at the depends of separation of concerns. We expose two such cases:

- the OBSERVER pattern in `Figure` is reused in a "figure connection" concern. Such connections are transversal to the invalidation concern. Typically observers in connection concern implement void methods for the invalidation notification and vice-versa;
- `StandardDrawing` implements the COMPOSITE pattern to manipulate `Figure`. In fact, it extends the `CompositeFigure` to reuse its structure and behavior, redefining some methods to accomodate for its nature of `Drawing`. This leads `Drawing` to copy the interface of `CompositeFigure` in a brittle relation. `StandardDrawing` also inherits from `Figure` a nonsensical subject role.

6 Related Work

The OBSERVER pattern serves as an exercise of choice for aspect languages features. The instantiation model of Caesar [11] follows more closely the object model of design patterns. Reflex [12] offers a metaphor of metaobjects as observers of hooksets. Many works, such as [13] and [14], deal with modularity and reusability of aspects in the context of design patterns: they contain valuable ideas on the way to configure generic aspects for use.

Few other patterns have been studied. One interesting case is the MEMENTO pattern, for which different attempts with AspectJ have been made [15]. To date the sole extensive study of single design patterns implementation with aspects is in [3]. It also contains some evaluation on "composition transparency" for those new implementations, that is the property to define multiple occurrences

of the same pattern while keeping them separate. However, it does not explore the issues of density and composition with other patterns.

[16] revisits the case of pattern density in JUnit [2]. It follows a different guideline than ours by not aspectizing the pattern but the supported concern. It remains to be shown whether such solutions can be generalized as design patterns.

7 Conclusion

Summary of problems we review about pattern implementation includes crosscutting of implementation, invasive modification of a pattern by application of another pattern, and tangling of concerns when reuse occurs to reduce overhead. One could argue that such problems are not specific to the implementation of design patterns. However, these are symptoms following the density of design patterns. These problems must be studied at the level of patterns and software design to promote their reusability. Software designers should be aware of such impacts:

- composition of patterns mean you have to reconsider forces so that you select another pattern, with the same concern (see Sect. 3.1);
- lack of reusable patterns itself could lead to tangling concerns in order to reduce implementation overhead.

Overall, there is a feeling that the difficulty in a dense field of patterns does not lie within pattern themselves (which remain what they are) but between them.

We notice AspectJ provides a sum of technologies, some of which (cflow, introduction) are not specific to aspects and exist in other languages. Nonetheless, this sum allows to cleanly modularize new concerns and compose them back and forth. It allows to avoid transformations of patterns described above, so that we were able to retain the basic JHotDraw framework and configure it by selecting aspects. We believe such an approach is valuable in software engineering to trace design choices during the development process.

The case of implementation overhead (Sect. 5.2) links to a reusable pattern library. We have implemented a composition of Composite, Iterator, and Visitor patterns which remains to be evaluated in the context of JHotDraw (`StandardDrawing` and `CompositeFigure`). The approach is to build reusable compositions based on the reusable single patterns.

Aspectization of patterns opens a new perspective: traceability is enhanced and, in particular, we could benefit from interaction detection [17] and visualization tool[3]. Detection of interactions can lead to automation:

- presence of the Singleton pattern links to a simple implementation of the Observer pattern;
- automatic configuration of pointcuts with cflow-like construct whenever Composite or Decorator patterns are detected;

[3] See AspectJ plugin for Eclipse – `http://www.eclipse.org/ajdt/`

– automatic registration of the OBSERVER pattern based on registration in the COMPOSITE pattern.

Acknowledgements. We would like to thank the anonymous reviewers for their comments, which help to improve the quality of this article.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts (1994)
2. Gamma, E., Beck, K.: JUnit: A Cook's Tour (2002) http://junit.sourceforge.net/doc/cookstour/cookstour.htm.
3. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proc. of OOPSLA 2002, ACM Press (2002) 161–173
4. Soukup, J.: Implementing patterns. In: Pattern languages of program design. ACM Press/Addison-Wesley Publishing Co., USA (1995) 395–412
5. Zimmer, W.: Relationships between design patterns. In Coplien, J.O., Shmidt, D.C., eds.: Pattern Languages of Program Design. Addison-Wesley (1994)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: Proc. of ECOOP 2001, LNCS 2072, Springer-Verlag (2001) 327–353
7. Colyer, A., Clement, A., Harley, G., Webster, M.: eclipse AspectJ. the eclipse series. Addison-Wesley (2005)
8. Douence, R., Teboul, L.: A crosscut language for control-flow. In: Proc. of GPCE 2004, LNCS, Springer-Verlag (2004)
9. Denier, S.: Traits programming with AspectJ. RSTI - L'objet **11**(3) (2005) 69–86
10. Bosch, J.: Design patterns as language constructs. Journal of Object-Oriented Programming **11**(2) (1998) 18–32
11. Ostermann, K., Mezini, M.: Conquering aspects with Caesar. In Akşit, M., ed.: Proc. of AOSD 2003, ACM Press (2003) 90–99
12. Tanter, É., Noyé, J., Caromel, D., Cointe, P.: Partial behavioral reflection: Spatial and temporal selection of reification. In Crocker, R., Steele, Jr., G.L., eds.: Proc. of OOPSLA 2003, ACM Press (2003) 27–46
13. Clarke, S., Walker, R.J.: Composition patterns: An approach to designing reusable aspects. In: Proc. of ICSE 2001. IEEE Computer Society (2001) 5–14
14. Lieberherr, K., Lorenz, D.H., Ovlinger, J.: Aspectual collaborations: Combining modules and aspects. Computer Journal of the British Computer Society **46**(5) (2003) 542–565
15. Marin, M.: Refactoring JHotDraw's undo concern to AspectJ. In: Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004). (2004)
16. Isberg, W.: Aop pointcut patterns in the JUnit Cook's Tour (2005) http://junit.sourceforge.net/doc/cookstour/cookstour.htm.
17. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In Batory, D., Consel, C., Taha, W., eds.: Proc. of GPCE 2002. LNCS 2487, Springer-Verlag (2002) 173–188