Refactoring JHOTDRAW's Undo concern to ASPECTJ

Marius Marin
Software Evolution Research Lab
Delft University of Technology
The Netherlands
A.M.Marin@ewi.tudelft.nl

Abstract

In this paper we discuss an approach to the aspect-oriented refactoring of the Undo *concern in an open-source Java system. A number of challenges and considerations of the proposed solution are analyzed for providing useful feedback about how the employed aspect language could better support the refactoring to aspects. We also consider the* unpluggability *property of a concern as an estimate of its refactoring costs and propose a number of research questions to measure the improvements due to aspect refactoring.*

1. Introduction

Aspect oriented programming(AOP) is aimed at overcoming the modularization limitations of object orientation, and in particular at reducing code tangling and scattering. Refactoring is a technique for improving the internal structure of the code without affecting its external behavior [1]. Refactoring object oriented systems to aspects is a natural step towards AOP adoption. However, it is important to see how the existing approaches to AOP can support the refactoring process. Starting from this consideration, we propose an aspect solution to the *Undo* crosscutting concern in an open-source Java system, using ASPECTJ [1] as the implementation language. The analyzed system is JHOTDRAW [2], a model framework for two-dimensional graphics of around 18,000 non-comment lines of code.

The case for the aspect refactoring of the *Undo* concern in JHOTDRAW was introduced in our previous work [2], where *fan-in* analysis was employed to identify crosscutting concerns. The results have shown about 30 undo activities defined for various elements of the graphical framework. A classification of these elements would comprise *command*, *tool*, and *handle* classes as well as one class for dragging figures. We will discuss the refactoring of the *commands* group as the largest in terms of defined undo activities and also as a very common (undoable) task in a drawing application.

[1]www.eclipse.org/aspectj/
[2]jhotdraw.org, v.5.4b1

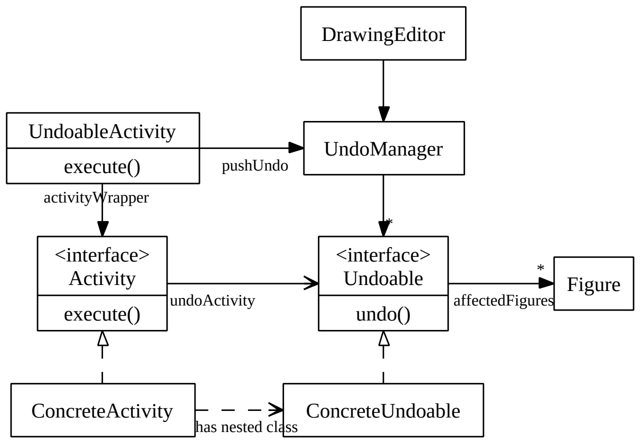


Figure 1. Participants for *undo* in JHOTDRAW.

2. Current Undo implementation

A number of activities in JHOTDRAW, such as handling font sizes and colors, image rotation, or inserting the clipboard's content into a drawing, support the undo functionality. A representation of the elements in the implementation of the undo concern is given in figure 1.

The *Activity* components participate in the implementation of the *Command* design pattern. The pattern provides a generic interface (*Activity*) for the operations to be executed when menu items are selected by the user, which allows to separate the user-interface from the model. Item-selection actions result in invocations to the *execute* method of the associated, specific activities. Many of these activities also have support for undo functionality, which in JHOTDRAW is implemented by means of nested (undo) classes. The nested class knows how to undo the given activity and maintains a list of *affected figures* whose state is also affected if the activity must be undone. Whenever the activity modifies its state, it also updates fields in its associated undoactivity needed to actually perform the undo. The application supports repeated undo operations (the *Undo Command*) by recording the last executed commands in reversed order. This is achieved by wrapping the commands that can be undone into an *Undoable Command* object, which serves three roles: first, it assumes the request to execute the command, second, delegates the command's execution to the wrapped command, and last, acquires a reference to the undo activity associated with the wrapped command and pushes it into a stack managed by an *UndoManager* object. When executing an *Undo Command*, the top undo activity in the stack is extracted and, after the execution of its *undo()* method, is pushed into a redo stack managed by the same *UndoManager* object.

The *Command* hierarchy in JHOTDRAW, shown in figure 5, implements the design pattern bearing the same name. The (12) undo-able commands store a reference to their associated undo activity. These references are obtained in the control flow of the command's execution through dedicated factory methods.

Given the described implementation, it is apparent that the primary decomposition of *Command* is crosscut by a number of elements, as follows:

(1) the field declared by *AbstractCommand* for storing the reference to the associated undo activity,

(2) the accessors for this field implemented by the same class,

(3) the *UndoActivity* nested classes implemented by most of the concrete commands that support undo functionality,

(4) the factory methods for the undo activities declared by each concrete command that can be undone,

(5) the references to the before enumerated elements from non-undo related members, e.g., the *execute()* method of the command class.

These crosscutting elements are outlined in the figures 2 and 6 for two command classes: (1) *ChangeAttribute Command* modifies the predefined attributes of a figure, such as the text color or the font size for a *Text Figure*, and (2) *Paste Command* is an activity that supports the insertion of the clipboard content into the active drawing of the graphics editor. The same elements are also used as criteria for grouping the command classes in figure 5, as it will be described in section 4.

```
public class ChangeAttributeCommand extends AbstractCommand {

    //constructor and private fields ...

    //the command's execute() method
    public void execute() {
        super.execute();

        setUndoActivity(createUndoActivity());
        getUndoActivity().setAffectedFigures(
             view().selection());
        FigureEnumeration fe = getUndoActivity().
            getAffectedFigures();
        while (fe.hasNextFigure()) {
            fe.nextFigure().setAttribute(fAttribute, fValue);
        }

        view().checkDamage();
    }
    // Factory method for undo activity
    protected Undoable createUndoActivity() {
        return new ChangeAttributeCommand.UndoActivity(
            view(), fAttribute, fValue);
    }
    public static class UndoActivity extends UndoableAdapter {
        //implementation of the undo nested functionality...
        public void undo() {...}; // ...
    }
}
```

Figure 2. The original ChangeAttributeCommand class to change a figure's attribute.

```
public void execute() {
    //super.execute(); - added by a separate aspect, not
    //undo-related, with a higher priority than the undo aspect
    FigureEnumeration fe = view().selection();
    while (fe.hasNextFigure()) {
        fe.nextFigure().setAttribute(fAttribute, fValue);
    }

    view().checkDamage();
}
```

Figure 3. The refactored ChangeAttributeCommand.

```
public privileged aspect ChangeAttributeCommandUndoActivity {

    pointcut inChangeAttributeCommand(ChangeAttributeCommand cmd) :
        this(cmd) &&
        execution(void ChangeAttributeCommand.execute());

    before(ChangeAttributeCommand cmd) :
        inChangeAttributeCommand(cmd) {
            cmd.setUndoActivity(cmd.createUndoActivity());
            cmd.getUndoActivity().setAffectedFigures(
                cmd.view().selection());
    }

    Undoable ChangeAttributeCommand.createUndoActivity() {
        return new ChangeAttributeCommandUndoActivity.
            UndoActivity(view(), fAttribute, fValue);
    }

    public static class UndoActivity extends UndoableAdapter {
        // the same implementation as for the original nested class
    }
}
```

Figure 4. The aspect solution for ChangeAttributeCommand.

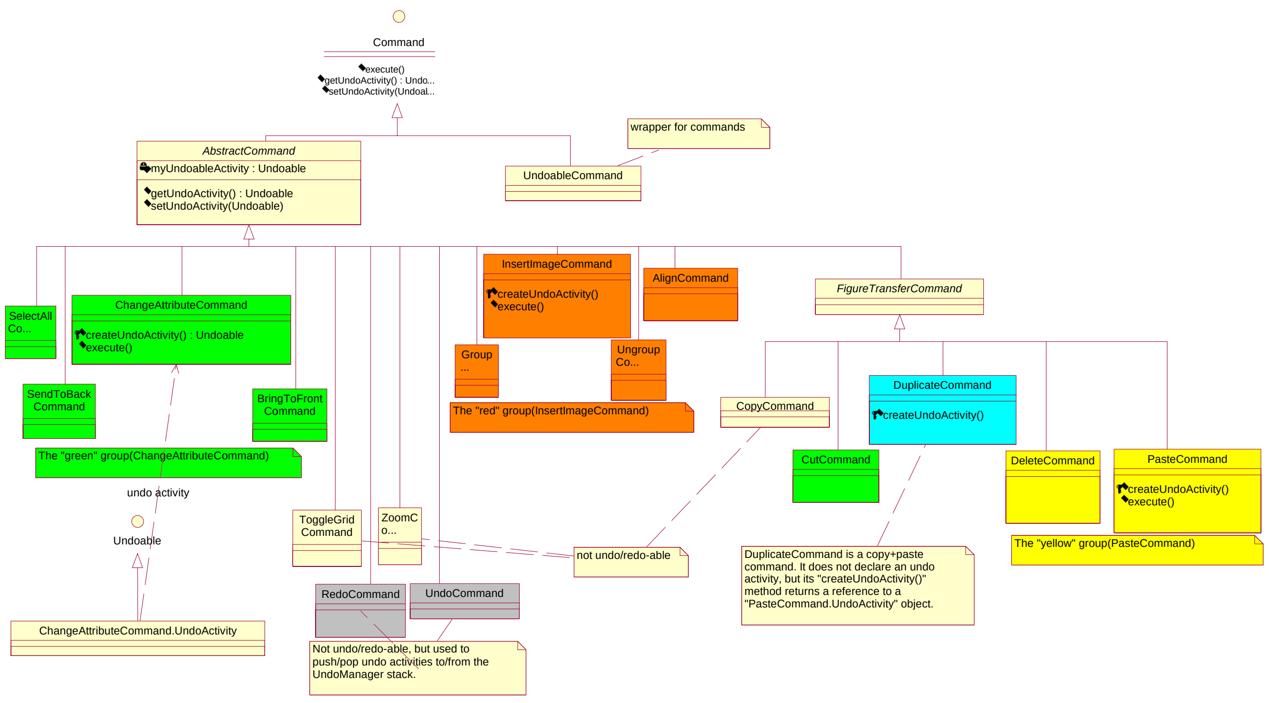

Command
execute()
getUndoActivity() : Undo...
setUndoActivity(Undoal...
wrapper for commands
AbstractCommand
myUndoableActivity : Undoable
getUndoActivity() : Undoable
setUndoActivity(Undoable)
UndoableCommand
InsertImageCommand
createUndoActivity()
execute()
AlignCommand
FigureTransferCommand
SelectAll Co...
ChangeAttributeCommand
createUndoActivity() : Undoable
execute()
Group ...
Ungroup Co...
The "red" group(InsertImageCommand)
CopyCommand
DuplicateCommand
createUndoActivity()
SendToBack Command
BringToFront Command
The "green" group(ChangeAttributeCommand)
CutCommand
DeleteCommand
PasteCommand
createUndoActivity()
execute()
undo activity
Undoable
ToggleGrid Command
ZoomC o...
not undo/redo-able
The "yellow" group(PasteCommand)
DuplicateCommand is a copy+paste command. It does not declare an undo activity, but its "createUndoActivity()" method returns a reference to a "PasteCommand.UndoActivity" object.
RedoCommand
UndoCommand
ChangeAttributeCommand.UndoActivity
Not undo/redo-able, but used to push/pop undo activities to/from the UndoManager stack.

Figure 5. Command hierarchy in JHOTDRAW.

3. Example refactorings

3.1. Tirsen's Generic Undo Aspect

A general solution for handling the undo functionality was proposed by Tirsen [3]. It mainly consists of keeping track of the fields that are set when a command executes. However, the solution suffers from several limitations when considering the *undo* implementation in JHOTDRAW: it does not capture all the state modifications caused by a command's execution, such as changes in data structures, and it requires filtering the set fields, as not all these fields are of interest to the undo process. Given the complexity of the undo process in JHOTDRAW and the way it is handled, the approach is problematic.

3.2. A Simple Case: Undoing the ChangeAttribute Command

The systematic refactoring we propose for the undo functionality consists of several steps. First, an undo-dedicated aspect is associated to each undo-able command. The aspect will implement the entire undo functionality for the given command, while the undo code is removed from the command class. By convention, each aspect will consistently be named by appending "UndoActivity" to the name of its associated command class to enforce the relation between the two, as in figure 4. In a successive step, the command's nested *UndoActivity* class moves to the aspect. The factory methods for the undo activities (*createUndoActivity()*) also move to the aspect, from where are introduced back, into the associated command classes, using inter-type declarations. Lastly, the undo setup is attached to those methods from which was previously removed, namely the *execute()* method, by means of an advice. Figure 2 shows the original implementation of the command, while figures 3 and 4 illustrate the refactored class and the aspect solution, respectively.

3.3. A Complex Case: Undoing the Paste Command

The general strategy outlined above for the case of the *ChangeAttribute Command* requires some supplementary steps for commands with a higher degree of tangling for the undo functionality. An interesting case for its complexity is that of the *PasteCommand* class, shown in figure 6. Both the command's main logic and the undo-related setup depend on common condition checks. The proposed solution looks for a clean separation of the two concerns, hence it captures the calls that set the variables checked in the command's execution, and re-uses the same values when executing the undo functionality as a separate, post-command operation. The common conditions, emphasized in figure 6, are also shown in figure 7 and have the associated pointcuts marked in figure 8. The aspect defines its own set of variables that are set to the same values as the ones checked in the control flow of the advised (*execute*) method.

4. Levels of Unpluggability

The refactoring we propose tries to stay close to the original design of the application and to ensure an easy migration to the aspect-based solution. After identifying the crosscutting concern, this is removed from the system and re-added in an aspect-specific manner, as previously discussed. However, the concern's removal has different levels of complexity for various commands. Given the identified elements of the crosscutting concern, it is possible to distinguish common characteristics for grouping the commands as a complexity assessment. The classification shown as colors in figure 5 is based on two main criteria:

1. the degree of tangling of the undo setup in the command's logic, particularly the activity's *execute()* method, and

2. the impact of removing the undo-related part from its original site, which can be estimated by the number of references to the factory method and to the methods of the nested undo activity.

These characteristics define the concern as *unpluggable*; that is, the core logic of the method executing the command is separated in the method's flow from the crosscutting undo elements, thus making possible to have the command executing correctly after removing the lines of code appertained to undo.

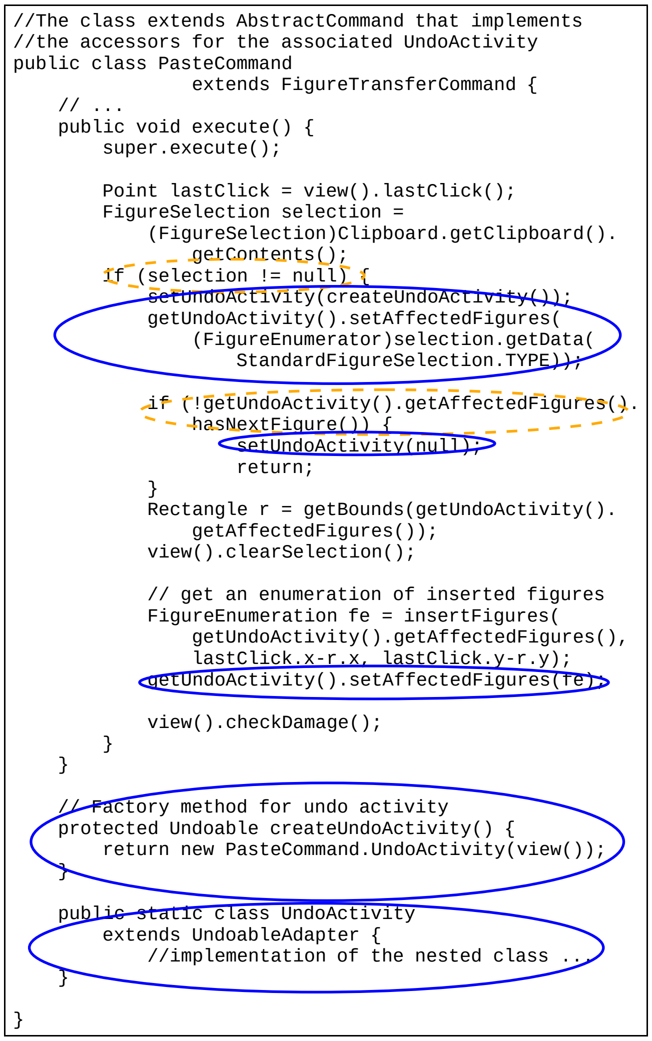
The "green"(ChangeAttributeCommand) group exhibits a number of properties that permit a clean feature extraction:

- the references to the nested undo activities and the factory methods for these activities are exclusively from inside the enclosing class, or from other (extending) undo activities,

- the undo-related code in the enclosing classes is unpluggable as previously described, and thus suitable for extraction and refactoring by means of advice constructs [3]

- the other methods related to the crosscutting functionality of undo (*set/getUndoActivity*) are inherited from top level classes (and not overridden locally) where they can be refactored by means of introduction.

The "red" group (*InsertImageCommand*) of commands does not exhibit the undo unpluggability. The commands



```
//The class extends AbstractCommand that implements
//the accessors for the associated UndoActivity
public class PasteCommand
                extends FigureTransferCommand {
    // ...
    public void execute() {
        super.execute();

        Point lastClick = view().lastClick();
        FigureSelection selection =
            (FigureSelection)Clipboard.getClipboard().
                getContents();
        if (selection != null) {
            setUndoActivity(createUndoActivity());
            getUndoActivity().setAffectedFigures(
                (FigureEnumerator)selection.getData(
                    StandardFigureSelection.TYPE));

            if (!getUndoActivity().getAffectedFigures().
                hasNextFigure()) {
                    setUndoActivity(null);
                    return;
            }
            Rectangle r = getBounds(getUndoActivity().
                getAffectedFigures());
            view().clearSelection();

            // get an enumeration of inserted figures
            FigureEnumeration fe = insertFigures(
                getUndoActivity().getAffectedFigures(),
                lastClick.x-r.x, lastClick.y-r.y);
            getUndoActivity().setAffectedFigures(fe);

            view().checkDamage();
        }
    }

    // Factory method for undo activity
    protected Undoable createUndoActivity() {
        return new PasteCommand.UndoActivity(view());
    }

    public static class UndoActivity
        extends UndoableAdapter {
            //implementation of the nested class ...
    }
}
```

Figure 6. The original PasteCommand class - command to insert clipboard's content into the drawing.

can not yield the expected results in the absence of the functionality defined by the nested undo-related classes. This dependency has been considered a candidate for a preliminary (object-oriented) refactoring with more implications for the original code, but able to produce the concerns' uncoupling.

The refactoring of the "yellow" group (PasteCommand) affects a larger number of classes. The multiple references from outside the class enclosing the *UndoActivity* to the corresponding factory method or to the undo constructor are specific to this group. Moreover, the undo-related calls from the various methods can be more tangled than for the "green" group.

[3]In practice, small local refactorings that eliminate one layer of indirection are needed before having the concern's statements separated from the rest of the code. In fi gures 2 and 3, for instance, the enumeration of the selected fi gures in the view, *fe*, is obtained differently.

```
public void execute() {
    super.execute();
    Point lastClick = view().lastClick();
    FigureSelection selection = (FigureSelection)
        Clipboard.getClipboard().getContents();
    if (selection != null) {
        //introduced variable for affected figures
        FigureEnumerator figEnum = (FigureEnumerator)
            selection.getData(StandardFigureSelection.TYPE);
        if (!figEnum.hasNextFigure())
            return;
        Rectangle r = getBounds(figEnum);
        view().clearSelection();
        figEnum.reset();
        //the 'fe' enumeration is not needed here anymore
        insertFigures(figEnum, lastClick.x-r.x,
                               lastClick.y-r.y);
        view().checkDamage();
    }
}
```

Figure 7. The refactored execute() method in PasteCommand.

5. Improved Language Support for Refactoring to Aspects

We generally appreciate the results of the refactoring process as leading to a cleaner separation of concerns and to a better modularization. By aspect-refactoring, the two concerns are separately modularized and the secondary concern of undo is no longer tangled into the implementation of the primary one. Our systematic approach is intended to ensure a gradual and possible automatic process of migration, with some of the steps turned into general refactorings, as for instance, migrating nested classes to aspects or extracting features into inter-type declarations. However, a number of drawbacks that, we think, can be overcome by a better aspect language support, can be discussed in relation to this experiment.

The original design uses static nested classes to enforce a syntactical relation between the undo activity and its enclosing command class. Since the ASPECTJ mechanisms do not allow introduction of nested classes, the post-refactoring association will only be an indirect one, based on naming conventions. This is a weaker connection than the one provided by the original solution.

Another drawback is the change of the visibility for the methods introduced from aspects, i.e. inter-type declarations. The visibility declared in the aspect refers to the aspect and not to the target class. For instance, it is not possible in ASPECTJ to introduce members into a class that are *protected* for that class. This is the case for the undo factory methods whose visibility cannot be preserved by the refactoring process. Having caller methods unable to access the callee after refactoring will require changes in the visibility that can weaken the boundaries imposed by the original design.

For the discussed case of the "yellow" group, code in

```
public aspect PasteCommandUndoActivity {
    //store the Clipboard's contents - common condition
    FigureSelection selection;

    pointcut execute_callClipboardgetContents() :
        call(Object Clipboard.getContents())
        && withincode(void PasteCommand.execute());

    after() returning(Object select) :
        execute_callClipboardgetContents() {
            selection = (FigureSelection)select;
    }

    //The variable stores the value returned by insertFigures()
    FigureEnumeration insertedFiguresEnumeration;

    pointcut execute_callinsertFigures() :
        call(FigureEnumeration FigureTransferCommand.
            insertFigures(FigureEnumeration, int, int))
        && withincode(void PasteCommand.execute());

    after() returning(FigureEnumeration figs) :
        execute_callinsertFigures() {
            insertedFiguresEnumeration = figs;
    }

    FigureEnumerator selectedData;

    pointcut execute_callselectiongetData() :
        call(Object FigureSelection.getData(String))
        && withincode(void PasteCommand.execute());

    after() returning(FigureEnumeration dataSel) :
        execute_callselectiongetData() {
            ArrayList al = new ArrayList();
            while(dataSel.hasNextFigure()) {
                al.add(dataSel.nextFigure());
            }
            dataSel.reset();
            selectedData = new FigureEnumerator((Collection)al);
    }

    pointcut executePasteCommand(PasteCommand cmd) :
        this(cmd) &&
        execution(void PasteCommand.execute());

    /**
     * Execute the undo setup.
     */
    void after(PasteCommand cmd) : executePasteCommand(cmd) {
        //the values for the variables that have to be checked here,
        //e.g., selection, have been captured by means of advices

        // the same condition as in the advised method
        if(selection != null) {
            cmd.setUndoActivity(cmd.createUndoActivity());
            cmd.getUndoActivity().setAffectedFigures(selectedData);
            // the same condition as in the advised method
            if (!cmd.getUndoActivity().getAffectedFigures().
                   hasNextFigure()) {
                cmd.setUndoActivity(null);
                return;
            }
            cmd.getUndoActivity().setAffectedFigures(
                insertedFiguresEnumeration);
        }
    }
    /**
     * Factory method for undo activity - cannot be protected anymore
     */
    Undoable PasteCommand.createUndoActivity() {
        return new PasteCommandUndoActivity.
                       UndoActivity(view());
    }

   //the nested class moves to the aspect
    public static class UndoActivity
        extends UndoableAdapter {
        // the undo actvity nested class
    }
}
```

Figure 8. The undo aspect for PasteCommand.

both the method's logic and the undo setup part is executed if a common condition holds. This means that the same condition will be checked in the advice executing the undo setup functionality and in the advised method, too. While we believe this is not a reason for concern from the design point of view, it can be from the perspective of the compiler work. In the same time, the conditions strengthen the relation between the two concerns, and affect the modular reasoning about the undo concern, which has to be aware of the execution particularities of its associated command.

5.1. Research questions

The downsides of the proposed aspect solution, despite an overall improvement, pose several questions.

> How to measure the code improvements due to refactoring to aspects? Is it possible to define a set of metrics for this?
>
> Would it be possible to use these metrics to compare different aspect solutions? How can these solutions be compared from the perspective of easy migration?
>
> What is a good aspect solution? Could we define a set of good practices in aspect oriented programming?

We think that some of these questions can be answered by improving the support of the aspect language for the refactoring process. Preserving the advantages of the original implementation will prove beneficial in eliminating potential tradeoffs. A set of good AOP practices is an open issue, and just as the language itself, is part of the evolution process of the aspect oriented technique. Reliable solutions to common problems, as the undo functionality one, are also critical to avoid intrusive code due to the language mechanisms. All these are important concerns for building confidence in an AOP adoption for existing systems.

6. Conclusions

The solution achieved by applying the aspect oriented techniques to refactor the *undo* concern in an existing, well-designed object-oriented system shows improvements in terms of modularity and separation of concerns. Yet, the downsides of the aspect-based solution raise questions about how the improvements can be quantified and what are the desired aspect solutions for specific crosscuttings. The *unpluggability* property gives a measure of how clear the concern is distinguished in the original code and is a good estimate of the refactoring costs.

References

[1] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[2] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*. IEEE Computer Society Press, 2004.

[3] J. Tirsen. Undo in AspectJ. Codehause Jutopia discussion forum, April 25 2004. `blogs.codehause.org`.