

Fluid AOP

Task-specific Modularity

by

Terry Hon

B.Sc., The University of British Columbia, 2004

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

March, 2007

© Terry Hon 2007

Abstract

Most aspect-oriented programming technology used today uses a linguistic approach that enables programmers to write modular crosscutting code. Two limitations of these approaches are that there is only one decomposition present for a code-base and that they require developers to adopt a new (or extended) programming language. We propose fluid AOP to modularize crosscutting concerns without these limitations.

Fluid AOP provides mechanisms in the IDE for creating constructs that localize a software developer's interaction for a specific task. These constructs act as fluid aspects of the system. They are editable representations of the subset of the code-base that the developer needs to interact with to perform a task. We present three fluid AOP prototypes and provide comparisons between them; as well as comparisons between the fluid AOP, linguistic AOP, and non AOP approaches.

Table of Contents

Abstract	ii
Table of Contents	iii
List of Figures	v
Acknowledgements	vi
1 Introduction	1
1.1 Motivation	1
1.2 Fluid AOP	2
1.3 Thesis Statement	3
1.4 Overview	3
2 Related Work	4
2.1 Code Navigation	4
2.2 Linking Code Clones	5
2.3 Alternate Editor or Decomposition	5
3 Three Prototypes in Fluid AOP	8
3.1 Common Features	9
3.1.1 Aspect Editor	9
3.1.2 Pointcuts	10
3.2 Before/After/ITD	10
3.2.1 Advice	11
3.2.2 ITD	12
3.2.3 Multiple Advice or ITDs	14
3.3 Gather	15
3.4 Overlay	17
3.4.1 Variations in Overlay	18

Table of Contents

4	Join Point Models	20
4.1	Nature of the Join Points	20
4.2	Means of Identifying the Join Points	20
4.3	Means of Semantic Effect at Join Points	20
4.3.1	Before/After/ITD	21
4.3.2	Gather	21
4.3.3	Overlay	21
5	Implementation	22
5.1	Parser	22
5.2	Name binding and Type Checking	23
5.3	Join Point Matcher	23
5.4	Code Preparation and Linking	23
5.4.1	Before/After/ITD	24
5.4.2	Gather	24
5.4.3	Overlay	24
5.4.4	Overlay Groups	25
5.5	Limitations	26
6	Additional Example	27
6.1	Consistency Enforcement	27
7	Comparison	30
7.1	Fluid AOP	30
7.2	Fluid AOP vs. Linguistic AOP	32
7.3	Fluid AOP vs. OOP	32
8	Contribution	33
9	Future Work	35
9.1	Extending the Fluid AOP Space	35
9.2	Improving Current Implementations	35
9.2.1	Overlay Groups	35
9.2.2	More Overlay Features	36
9.2.3	Graphical Aspect Editor	36
9.3	Improving Scalability	37
	Bibliography	38

List of Figures

2.1	Screenshot of fluid document presentation	6
2.2	Screenshot of a fluid source code view	6
3.1	Outline view for aspect with overlay	9
3.2	Before/After/ITD prototype screenshot	11
3.3	Point class with display field introduced	13
3.4	Aspect with ITD and after returning advice	14
3.5	Modified Aspect with ITD with annotation inserted	15
3.6	Gather declaration	16
3.7	Overlay declaration	17
3.8	Overlay declaration with parameter renaming	19
5.1	Implementation stages	22
6.1	Undo methods in JHotDraw	28
6.2	The updated overlayed groups for undo methods	29

Acknowledgements

There are many people that have helped me directly and indirectly to get me to this point.

First and foremost I want to thank my supervisor, Gregor Kiczales, for all his guidance and time that he has put in my research. Your advice on and off the research has been invaluable to me. Secondly I would like to thank Gail Murphy for being my second reader and for always being there when I have questions.

Thanks also to Andrew Eisenberg who gave me advice throughout my research and for being the first to give comments about my thesis draft. I have to also thank Mik Kersten for helping me get started and for always answering all my Eclipse questions.

Thanks to everyone in SPL, especially Arjun, Thomas, Rainer, Ryan and Brett, for all your input in my research and for listening patiently to my countless complaints. I also have to thank Hermie and Holly for all the help and support for the past couple years.

Thanks also to Baharak and Sara for being the good friends that you are and for always urging me to try new things.

Finally I have to especially thank my family, in particular my parents and my brother for always taking care of me and pushing me to achieve the best that I can. I would not be here without you!

Chapter 1

Introduction

1.1 Motivation

Programmers perform numerous tasks in the code development phase of the software development cycle. These tasks can fall into the following categories: code evolution, code comprehension and documentation.

Recent work in AOP has shown that different concerns in a code-base can crosscut each other's natural structure. The same is true of software development tasks – while they sometimes align with the primary decomposition of the source code (such as when a single sorting method needs to be refined), they sometimes crosscut that structure (such as reconciling the naming convention of one package with another package). This is because the structure of the concern that the task is addressing crosscuts the structure of the overall system. This crosscutting nature causes many of the actions performed during a task to be performed in multiple locations, making the task harder to perform and more prone to error.

Throughout this dissertation, we use the term concern to mean a subset of a program that a software developer is interested in at a certain point in time. This concern might correspond to the implementation of a feature or it could be any other kind of slice of the program that a software developer is interested in. We use the term task to refer to a set of actions to modify or understand a part of a concern. These tasks can involve modification of existing programs, creation of alternative realizations, modifications of existing interfaces [6] as well as navigation to understand code. Finally two concerns are said to crosscut each other with respect to the dominant decomposition of the system if the implementation of the two concerns directly overlay in the primary decomposition, but neither entirely covers the other.

Note that we focus only on artefacts in the development phase. There is ongoing research [1, 7, 13, 20] that looks at modularization of crosscutting concerns at the design or architectural level, but we do not address those levels in this research.

Most of the approaches which address crosscutting concerns in the development phase focus on enabling modular crosscutting code. These ap-

proaches use linguistic techniques [18] to make it possible for the implementation of two concerns to be cleanly separated even when those concerns crosscut (with respect to the dominant decomposition). An example of this is the AspectJ [11] extension to Java.

Linguistic approaches like AspectJ provide mechanisms that enable software developers to write more modular implementations of crosscutting concerns; however these crosscutting concerns are not separated on a task-specific basis. There is only one decomposition that is present for a system at any given time. If another decomposition is desired, the system can be refactored into the new decomposition, but the previous decomposition is lost. Concerns that are the focus of specific tasks can thus still be scattered and tangled if they crosscut the primary decomposition.

The second limitation of linguistic AOP is that it requires software developers to adopt a new (or extended) programming language in order to create crosscutting modules. Like any other language adoption, a range of tools needs to be updated to accommodate the new language features. These tools include compilers, editors, IDEs, code formatters, documentation generators, and others. This large set of tool changes can make adopting AOP difficult, especially in an industrial context.

1.2 Fluid AOP

In this research we propose fluid AOP as a solution to the limitations of linguistic AOP described above. Specifically, we suggest that fluid AOP can provide a kind of modularization that can support task-specific work.

Fluid AOP provides mechanisms in the IDE for creating constructs that localize a software developer's interaction for a specific task. These constructs are *crosscutting views* in that they can gather content that would otherwise be scattered throughout the system; in particular the constructs are not constrained by the file decomposition of the system. These constructs are *effective* in that the views they provide are editable. This allows editing tasks to be done locally; they are also modular since there is a single construct for each task. Different constructs can present different slices of the same system; thus multiple decompositions of the same system can be present simultaneously by generating different constructs. These constructs conceptually localize the captured crosscutting concern. In the rest of this document, we refer to them as fluid aspects.

Since the mechanism for modularizing crosscutting concerns is a fluid aspect, which is an editable view of the system, the underlying code remains

the same; no new programming language has to be adopted, possibly making the transition in adopting AOP smoother.

1.3 Thesis Statement

This thesis aims to support the following hypothesis:

Crosscutting concerns can be modularized for specific tasks by IDE support that localizes scattered code fragments of interest into a single editable representation.

1.4 Overview

The remainder of the document is organized as follows. The related work is described in Chapter 2. The three fluid AOP prototypes that we implemented are described in Chapter 3; their corresponding join point models are described in Chapter 4. Chapter 5 describes the implementation of the three prototypes. Chapter 6 provides additional examples from a medium-sized code-base. Chapter 7 provides comparison between the three prototypes, as well as comparing fluid AOP with AspectJ and plain Java. Chapter 8 summarizes the contributions of this work. Finally we discuss several future research directions in Chapter 9.

Chapter 2

Related Work

Approaches to providing task localization of crosscutting concerns through the IDE fall into three categories. The first is to provide a localized way to navigate among the code in question, which we will discuss in section 2.1. The second is for the editor to provide new mechanisms to connect related or duplicated code, discussed in section 2.2. In section 2.3 we look at the final category which is to provide a kind of editor that includes views that do not necessarily correspond to the actual structure of the file that it is displaying.

2.1 Code Navigation

JQuery [8] provides browsable tree views that are user defined. It lets software developers select the content of the tree view by specifying a query in a query language called TyRuBa [23]. The ordering of the tree hierarchy can also be modified to match the tasks that they are currently working on. This tool allows a software developer to perform exploration and navigation tasks within a single tree view. It also allows them to see the path in which it gets to the current query, which minimizes chances of getting lost between searches. This tree view can be used to group together parts of the system that correspond to a crosscutting concern, letting software developers navigate between the parts within a single structure without getting lost. However, reasoning between multiple editors is still needed to perform a task.

Mylar [10] is a task-focused UI for Eclipse that provides software developers with only the information that is relevant to her current task. It leaves views and editors less cluttered than before which allows software developers to navigate more easily between parts of a crosscutting concern since they now appear to be closer to each other.

JQuery, Mylar and related approaches [4, 16] provide an easier way for software developers to navigate between code. Mylar thins out the elements of the primary decomposition, but leaves the structure of that decomposition intact. So a developer might only see one method in a file, and a related

method in a second file. But there are still two files, and the mechanical and mental overhead of switching between the files to reason about or edit the methods is still there. (Although reduced because of the thinning out.) With JQuery, the structure of the primary decomposition remains the same. The query tool allows user to form a navigator between subsets of the decomposition, but the mental overhead of switching between the files still exists.

2.2 Linking Code Clones

Linked editing [22] is an editor-based approach for dealing with duplicated code. The tool (CodeLink) allows software developers to select multiple duplicated code fragments and link them together persistently. From then on the tool will affect the same edit on all the fragments when one is edited. It also provides selective elision of clones where the programmer can hide the redundant common regions between the linked code with ellipses, leaving only the differences visible. The clones in CodeLink are selected manually by the software developer; in particular it does not have a semantic way for software developers to select these code fragments. Once the code clones are located, the link between them is persistent.

Simultaneous editing [14] is used in LAPIS (Lightweight Architecture for Processing Information Structure) which automates repetitive text editing. It allows software developers to specify a set of regions to edit by either describing a pattern of the text in those regions or by manual selection. The system then selects all the regions corresponding to the specification and subsequently all edits done on one of the selected regions will cause equivalent changes to the other selected regions. Simultaneous editing allows selection to be done by using a pattern but this pattern only describes text property of the code, but does not use the semantic information of the system. Also the selection is not persistent so one has to remember the pattern every time one needs to work with the same selection.

2.3 Alternate Editor or Decomposition

HyperJ [17] allows simultaneous decompositions of programs that differ from each other. It provides mechanisms for specifying which components to include for a concern. However the specification of join points to be included and how they compose with each other is separate from the code of the join points themselves.

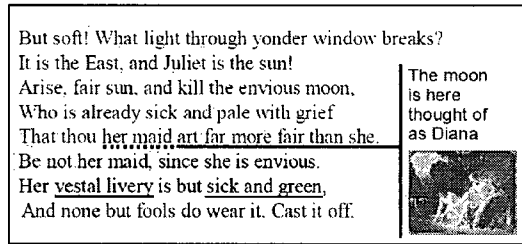


Figure 2.1: In Fluid Documents [2] the margin is used to show supporting materials. (Screenshot taken from [2])

The idea of fluid documents [2] was used in the context of literary documents. The idea is that the primary material is present in the editor, along with supporting material when the user requests that information. The supporting material is displayed at different indentation and font. (See Fig. 2.3) When space is limited, the supporting material can appear while overlapping with the existing content. An example of primary material is a literary passage and the supporting material could be an explanation about a specific terminology or alternate meaning to a word.

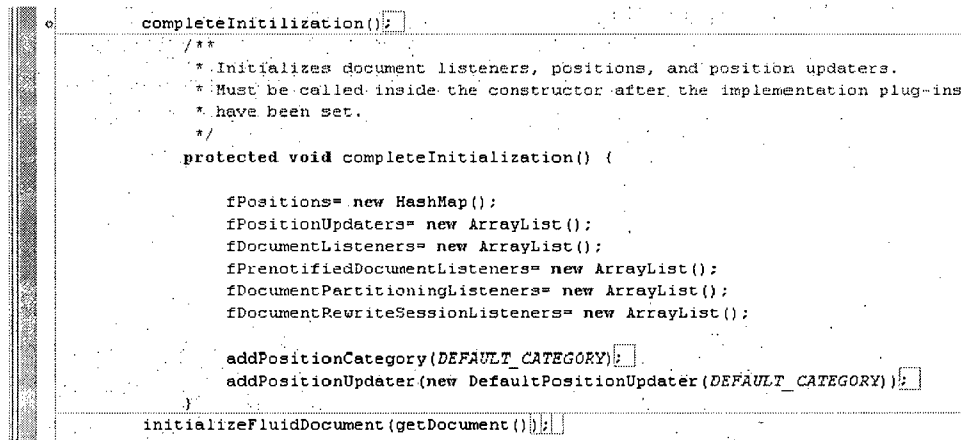


Figure 2.2: The screenshot shows an expansion of a method invocation in Fluid Source Code Views [5] which reveals the target method definition in context. (Screenshot taken from [5])

Fluid source code view [5] applies the concept of fluid documents to

source code. It reduces navigation by presenting a fluid view that inlines scattered code for “just in-time” comprehension. These fluid views are similar to our approach in that they are not limited by the actual file decomposition. The difference is that fluid source code view always has the same set of information available and software developers can choose to collapse some parts, whereas our approach lets software developers specify what information to display in the fluid aspect view. Also there is no way of eliminating duplicated changes in fluid source code view since there is no mechanism for representing a crosscutting slice as one entity.

In [9] the authors describe the Decal prototype which provides two mutually crosscutting and effective source code views. The first view is called the modules view which provides a decomposition of program structure in terms of modular units which crosscut classes. The other view is called the classes view which is similar to the traditional object-oriented view, showing decomposition of the system in terms of classes. These views are effective in that software developers can edit them directly and change the code-base. They are also first-class with clear semantics defined for all possible edit operations between the views, whereas other works such as Masterscope [21] and Stellation [3] have arbitrary grouping as views and the semantics of some editing operations are not well defined. Virtual source files are the source files that the views are displaying where these source files can overlap with each other; Decal internally keeps track of one internal structure of the code-base. This allows software developers to simultaneously have access to two decompositions of the same system. The crosscutting view is predefined by the tool designer and cannot be changed by the software developer and since the underlying structure of the code-base is a new representation, existing tools will have to be modified to be used with Decal.

In [19] the author describes MView, a source-code editor that provides dynamic crosscutting views of a Java system. These views are comprised of code fragments, annotations, and a structured presentation framework that shows the relationships between the fragments. View components in the editor refer directly to the Java model of the program, allowing software developers to edit code directly in the editor. It provides a program slice in a single editor, even if the slice is scattered in the code-base. Similar to the fluid source code view, there is no way to eliminate duplicated changes in the MView editor since the editor only groups together the code fragments.

Chapter 3

Three Prototypes in Fluid AOP

In order to investigate how fluid AOP support can aid in providing task-specific modularity, we implemented three prototypes of fluid AOP tools. These three prototypes address different kinds of development tasks; in this chapter we describe the prototypes using three scenarios where different editable views can localize the interactions required to perform the tasks. In Chapter 4 we will describe the join point models [12] underlying each of the prototypes. We focus on code-base written in Java for our prototypes.

The three scenarios that we have chosen are all illustrated using the same small code-base – a simple drawing application that has a display and a set of shapes that can be created and shown on the display. But each scenario should be familiar to developers as a common task in software development. They demonstrate that different tasks can require software developers to interact with different slices of the code-base. Each scenario describes a task that has to be performed on the drawing application. The first one describes a code evolution task where a new display updating concern is added; the second scenario involves a code comprehension task where one tries to understand an existing logging concern, the third scenario involves a code evolution task where the software developer wants to modify the existing logging concern. The difference between the first and the third scenarios is that the first is a feature addition task and the third is a feature modification task.

There are numerous common features that exist in all three prototypes and we describe these in Section 3.1. In the following subsections, we describe the features of each prototype.

3.1 Common Features

3.1.1 Aspect Editor

All three prototypes provide an additional editor with which software developers can interact. This editor works with files with extension `.fa`. These files store textual representations of the fluid aspect. These files include one or more pointcut definitions which are used to identify the code fragments that make up the concern of interest, as well as advice, intertype, gather or overlay declarations which specify the editable representation that the software developer is interested in. In all three prototypes, the representations are editable because they are linked back to the original files in which the code fragments appear.

We chose to use an editor as the means of interacting with the software developer because the various representations that we use to display code are also editable, which makes it natural for them to be presented in an editor. Secondly since we have the `.fa` files which consist of all the information to recreate the view, we can allow users to save these files and reuse them later to recreate the same editable representations.

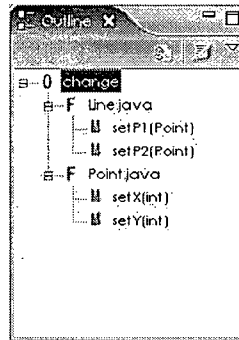


Figure 3.1: The outline shows that an overlay is declared for the pointcut `change`. `Change` matches code fragments in `Line.java` and `Point.java` and there are 2 `set` methods in each of the java files.

As a complementary part of the editor, we modified the outline view of the editor. The outline view provides a way for software developers to navigate between the aspect editor and the files where the matched code fragments belong to. Fig. 3.1.1 shows the outline view for an aspect that contains an overlay declaration. The icons differentiate between files and

methods in the outline view with **F** and **M**, as well as differentiating between the different declarations in the aspect, such as **O** for overlay, **G** for gather and **A** for advice declarations.

3.1.2 Pointcuts

The pointcut language used in all three prototypes is the same. The language we chose resembles the AspectJ pointcut language. The language has the following grammar:

```
pointcut ::= keyword '(' method_or_field_signature ')'
keyword  ::= declaration | invocation
method_or_field_signature ::= method_signature
                               | field_signature
method_signature ::= visibility_keyword type
                               identifier '(' param_list ')'
field_signature  ::= visibility_keyword type identifier
param_list      ::= (type identifier)*
```

The visibility keyword matches either **public**, **private** or **protected**. Type and identifier are string patterns that match a type or an identifier. We implemented the wildcard character ***** to match a string of any length consisting of any characters. We also implemented the sub-type pattern where **A+** matches all subclasses of **A** as well as the class **A** itself.

For example, the pointcut to identify the code fragments corresponding to the method declarations for **set** methods in any subclass of **Shape** would look like:

```
declaration(public void Shape+.set*(*))
```

3.2 Before/After/ITD

Consider the implementation of display updating for a drawing application. Whenever an attribute of a shape object changes, the display should be updated accordingly. In order to implement this, an observer pattern is used. In particular we focus on the change signaling behaviour, where the display object is an observer and the shape objects notify the display whenever there is a change. Therefore, display update invocation has to be inserted at

the end of all method declarations that change an attribute of a shape object. The display updating method should be invoked before these methods return, which is well suited for using an after returning advice.

The first fluid AOP prototype provides a fluid version of the pointcuts and advice model in AspectJ and other languages.

3.2.1 Advice

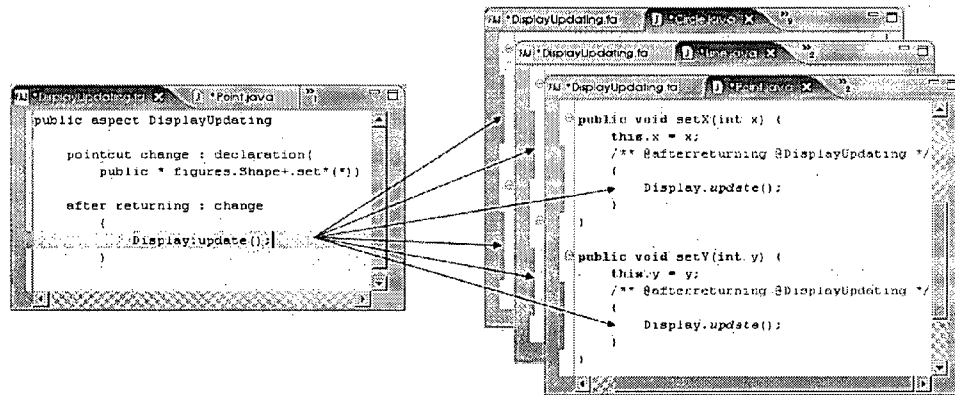


Figure 3.2: The aspect editor is shown on the left of the arrows and files of the corresponding code fragments are shown on the right. Editing in the body of the after returning advice also edits the advice blocks in the matched code fragments.

In this prototype, declaring an after advice first causes the editor to find all the code fragments matched by the pointcut. Then, within each of those blocks, it finds the after advice block. The after advice block is the block of code corresponding to the implementation of the after advice body. For each matched code fragment, there should be exactly one after advice block corresponding to each after advice. If such a block does not already exist (if this is the first time this advice is specified or the code fragment is a new addition to the code-base), then a new block is inserted at the end of the matched fragments; the body of the block is initialized to have the same content as the body of the advice. Once all the blocks are located, they are linked back to the original advice body. The resulting effect is that editing one of the blocks will edit all the blocks in unison. Fig. 3.2 shows a combination of the display updating aspect as well as some shape classes

with inserted advice blocks.

An additional comment is added preceding each block. This comment has two important functions. One is to act as an identifier for the tool to locate the inserted blocks during the linking phase. The second is to make it explicit to the software developers that the inserted blocks belong to a fluid aspect while reading the base code. This is helpful since if a software developer is interacting with the code without the tool, she would still know the inserted blocks are inserted as part of the concern for display updating causing her to be more cautious while changing code inside the blocks. On the other hand if the software developer is currently using the tool, the comments would remind her that editing the code inside the block would affect the other blocks belonging to the same advice.

Before advice has a similar effect except that the advice block is inserted at the beginning of the join point instead of at the end. We also implemented after returning advice, where the advice block is inserted just before the return statement of the method declaration.

To ensure that the semantics of an after or after returning advice is correctly implemented, sometimes the code fragment has to be refactored before the advice block can be inserted. In after advice, the original method body is surrounded by a try block and a finally block is added. The advice block is inserted into the finally block. This is to ensure that the advice body is executed even if an error or exception has been thrown.

For after returning advice, if the method declaration has no return statement at the end of the body, then the advice block is inserted at the end of the method body, as seen in Fig. 3.2. If the method declaration has a single return statement that returns nothing or returns a variable or field reference, then the advice body is inserted just before the return statement. If the return statement returns any other values, the prototype splits the return statement into a variable declaration where the variable is initialized to the value that was originally returned, and a return statement that returns the variable. The advice block is then inserted in between the variable declaration and the return statement. Finally if there are multiple return statements, then an advice block will be inserted to each return statement at the appropriate location with the corresponding refactoring as specified above.

3.2.2 ITD

Consider now what happens if the drawing tool needs to support multiple displays, i.e. each shape could be displayed in a different display object.

Instead of calling a static update method when the display should be updated, we want to invoke a non static method which might take the shape that triggers the update as an argument.

To perform this implementation within the same aspect editor, we use another feature of the prototype. An intertype declaration (ITD) is used to define a field or a method of a class.

An ITD consists of a field or method declaration along with a type pattern that specifies which class(es) the method or field should be declared in. The type pattern used is the same as the one used in a pointcut declaration.

Declaring an ITD first causes the editor to find all the classes matched by the type pattern. For each matched class, there should be exactly one declaration corresponding to each ITD. If such a declaration does not already exist (if this is the first time that this ITD is specified or if this class is a new addition to the code-base), a new field or method declaration is inserted at the beginning of the class declaration.

Once all the method or field declarations are located, they are linked back to the original ITD body. The resulting effect is that editing one of the declarations will edit all the blocks in unison. Fig. 3.3 shows the introduced display field in the Point class, a subclass of Shape. Fig. 3.4 shows the final display updating aspect with the ITD and the modified after returning advice.

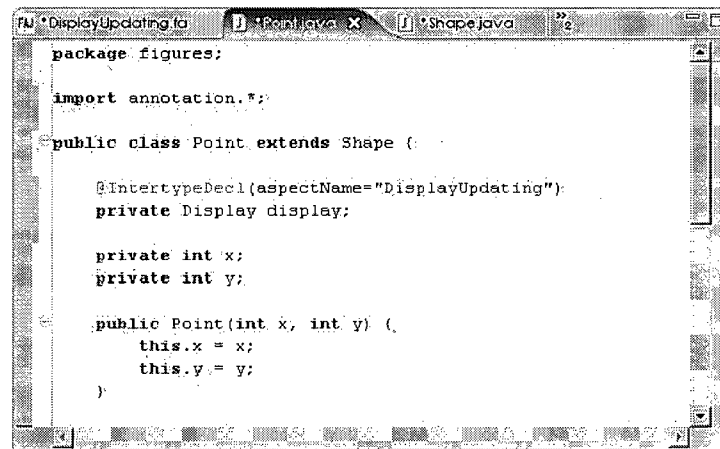


Figure 3.3: The display field is introduced by the aspect DisplayUpdating.

Notice that instead of using a comment before an introduced declaration, an annotation is used. We would have liked to use an annotation for advice blocks as well, since annotations carry more semantic meaning in the code

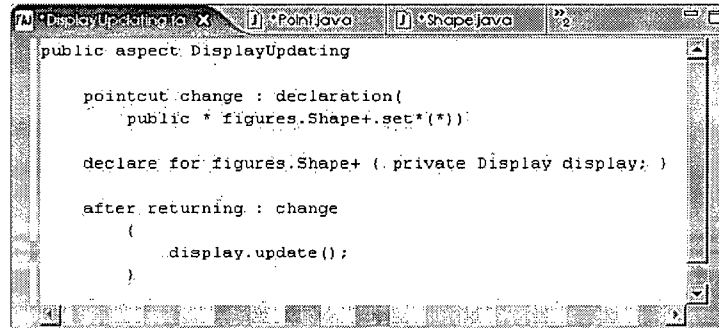


Figure 3.4: The pointcut definition, ITD and after returning advice is shown. The body of the advice has been changed to invoke the non-static update method.

then a comment and software developers might be less likely to overlook an annotation as compared to a comment. However Java 1.5¹ does not allow annotations in front of an arbitrary block so we had to use comments instead. The aspect name is included in these annotations, similar to the comments for before or after advice.

After the introduced field or method declaration is inserted in the appropriate classes, they are linked back to the ITD such that editing the ITD declaration will also change the introduced declarations as a uniform slice.

Before and after advice, together with ITD mean this prototype now presents more possible editable representations for localizing concerns by allowing software developers to perform edits for scattered code within the aspect editor. A developer can both see and edit all the advice and ITD by looking at the single fluid aspect.

3.2.3 Multiple Advice or ITDs

When there are multiple advice declarations or ITDs inside one fluid aspect, we need to record more than just the aspect name to identify exactly the advice or ITD. A solution for this is to insert an identifier that is unique within a single aspect, which we chose to be a counter of the number of declarations that have been defined in the aspect so far. We will focus on ITD for the remainder of this section but the same change can be applied to advice. Instead of adding the identifier information as an attribute to the annotation, it is added in the comment instead.

¹Java JSR 308 may allow annotations on blocks

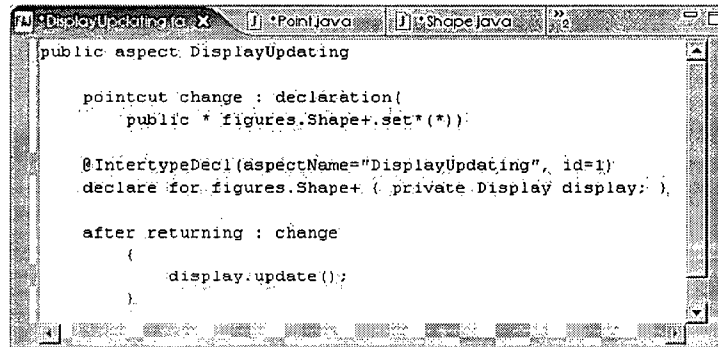


Figure 3.5: The pointcut definition, ITD and after returning advice is shown. An annotation is inserted preceding the ITD to reflect the id that is assigned to the ITD.

The annotation that appears before the inserted method or field declaration now contains an attribute for the aspect name as well as the identifier. In order to connect this identifier back to the ITD, we also need to record the identifier in the fluid aspect view. Fig 3.5 shows the fluid aspect with this new annotation.

3.3 Gather

The addition of concerns that do not already exist in the system, like display updating, can be localized by using advice or ITDs. In this section, we look at a task that cannot be localized with the previous prototype, and show that it can be localized by using the **gather** fluid AOP prototype.

Consider a scenario where a software developer is examining a system for which she believes there is a logging concern already implemented. She wants to understand the logging concern better by inspecting logging calls. There is no mechanism in the advice and ITD fluid AOP prototype for presenting all the code fragments within one editor for her to inspect. Thus we created another prototype that provides a gather mechanism.

The **gather** construct collects all the code fragments identified by the pointcut and displays them within a single editor. Each code fragment is displayed in its entirety in the order that they are identified. This view of the code fragments is editable – the user can directly edit any of the code fragments and this has the effect of changing the file that it originally appears in. Linked editing is used to achieve this, similar to the advice body

and ITD linking in the previous prototype.

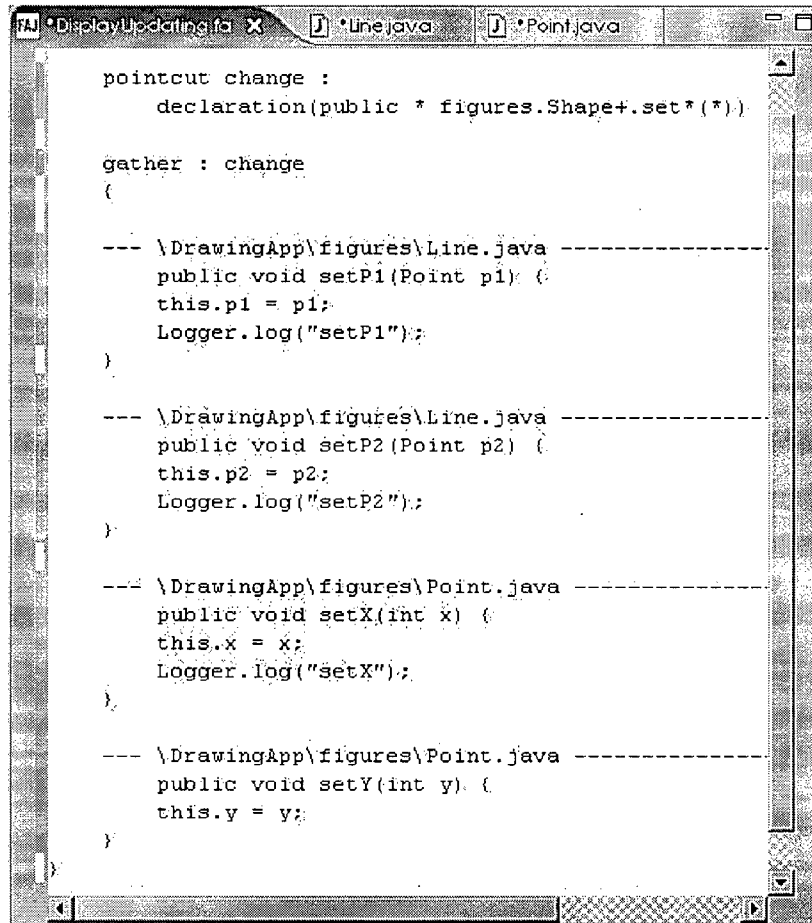


Figure 3.6: Body of the gather declaration is populated by editable representation of all the matched code fragments. Notice that the last `set` methods is missing a log invocation as compared to the other `set` methods.

In Fig. 3.6 a **gather** construct is shown. The pointcut used is identical to the one used in the after returning advice example. Each matching code fragment is completely shown inside the **gather** construct. A label containing the file name in which the code fragment originally appears is inserted before each code fragment. Editing the content in the **gather** construct body has the same effect as editing the actual code fragment.

Now the code comprehension task can be performed within a single ed-

itor. Looking at the code in Fig. 3.6 it is easy to see that all but one `set` method declaration has a `log` invocation at the end of its body. This makes the inconsistency in this logging concern easy to discover; and fixing this bug can be done locally in the same construct. This bug would have required the developer to compare code in multiple files otherwise.

3.4 Overlay

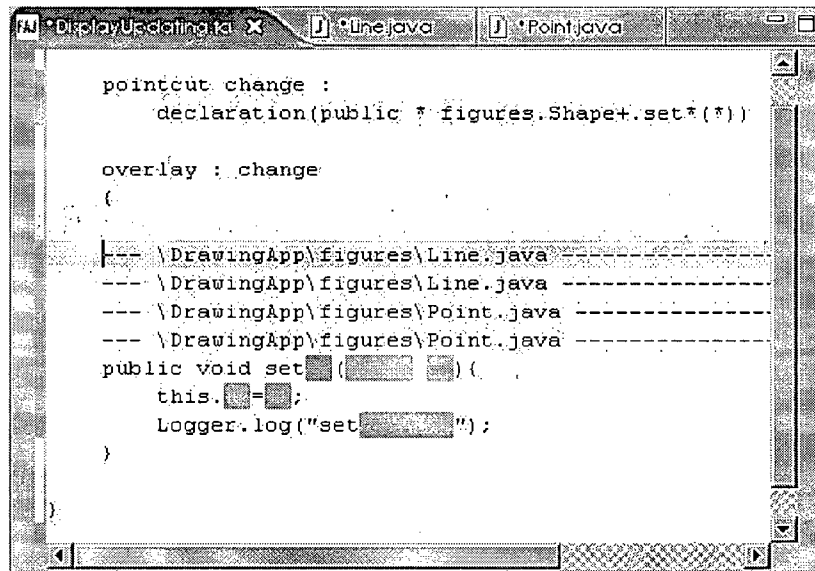


Figure 3.7: All the `set` methods of shape classes are overlaid and one representation is inserted into the overlay declaration. Parts that are different among the code fragments are grayed out.

The third fluid AOP prototype goes further than **gather** in terms of bringing scattered code fragments together for linked editing. It displays a single view for all identified code fragments. In order to generate this view, we have to calculate a single abstraction of the identified code fragments. It is impossible to show one abstraction that has all the original information that each code fragment possesses, since the code fragments are not guaranteed, and will most likely not be, identical. The idea is that this abstraction shows the similarities among the join points and hides the parts that are different.

In our prototype a top-down approach is used to compare the code fragments. We first produce the abstract syntax tree (AST) corresponding to

each code fragment. We then do a pair-wise recursive comparison. For two nodes of the same type, if they are leaves, then we use string comparisons to compare the text of the nodes; if they are non leaves, then the children are compared. Nodes of different types are not equal and their children are not visited.

After the comparisons are completed and the abstraction is generated, the similar parts are displayed and linked back to the corresponding parts in the original code fragments, and the rest are grayed out.

Consider the task of modifying the logging concern from the **gather** example. One wants to change the argument being passed to the logger by consistently adding a description before the method name. This task can be performed as multiple edits within one **gather** construct since all the code fragments are collected into one construct. With the **overlay** construct, this change can be done as a single edit.

The **overlay** construct is shown in Fig. 3.7. The view shows that all identified code fragments are **set** method declarations; they each assign a value to a field and end with a log method invocation. To perform the modification task, one can edit the log invocation directly in the **overlay** construct and the change will be propagated to all the original code fragments.

3.4.1 Variations in Overlay

Simple overlay can be useful when the set of join points is small and the join points are similar to each other. Otherwise, the overlaid structure will show mostly grayed out parts which do not provide enough information. We implement a variation in the way simple overlay is displayed. The comparison to generate the overlay remains the same as before, but instead of displaying just a grayed out box for method parameters, we name these boxes with the order in which they are declared, i.e. each grayed out parameter is displayed as *\$<parameter position>*. For example the first parameter of an overlaid method declaration is displayed as \$1 and the second is displayed as \$2 and so on. With this new way of display, if the parameters are used uniformly between all the overlaid method bodies, despite the parameter names being different, that information is still present in the overlay. Fig. 3.8 shows the new overlay display.

A second variation is a combination of using **gather** and **overlay**. Since overlay is most effective when the comparisons are done on a set of similar code fragments, this variation tries to guarantee that the groups are similar. Before the comparisons are performed, another step is implemented to group similar code fragments before comparing nested code fragments and

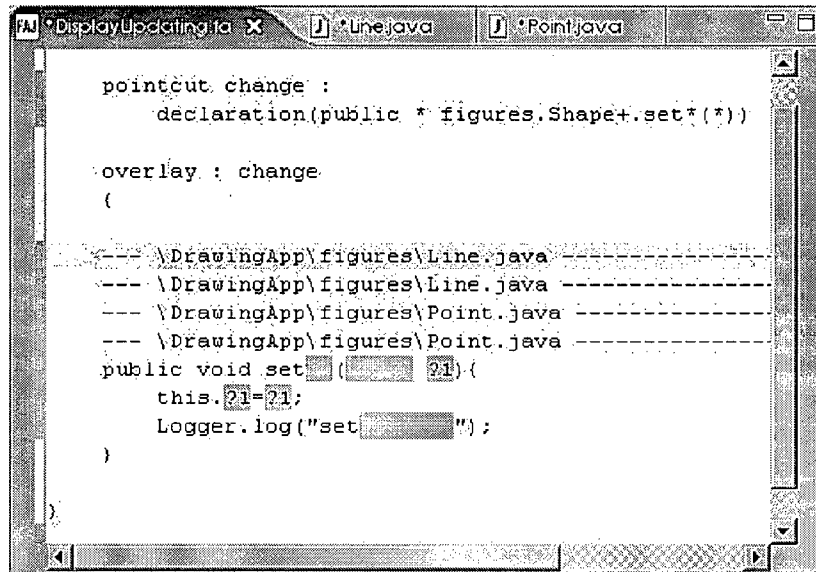


Figure 3.8: The set methods are overlaid as before except that the parameter of the method has been displayed as ?1.

constructing the overlay. See Chapter 5 for details on how the groups are formed and Chapter 6 for an example that uses this feature.

Chapter 4

Join Point Models

In AOP approaches, a join point model (JPM) [12] is used to describe different AOP mechanisms in general terms. In this chapter we describe the three prototypes in terms of their join point models, using this simple three-part ontology:

1. the nature of the join points
2. the means of identifying the join points
3. the means of semantic effect at the join points

4.1 Nature of the Join Points

For all three prototypes, the join points are static code fragments. These code fragments include method declarations and method invocations. For the before/after/ITD model, code fragments corresponding to field declarations are also join points. Since all three prototypes are only concerned with static code fragments (i.e. static join points), the three corresponding join point models are all static join point models.

4.2 Means of Identifying the Join Points

The means of identifying join points are identical in all three models. They all use the same pointcut language for method signature based identification of method declarations and method invocations. Type patterns are used to identify types in method declarations as well as for identifying the classes that the method or field declaration should be introduced in for ITD.

4.3 Means of Semantic Effect at Join Points

This is the part that differs the most among the three join point models. The following sections describe the semantic effect of each model.

4.3.1 Before/After/ITD

This join point model supports before, after and after returning advice, as well as intertype declarations; therefore, there are four means of effects.

The effect of a before, after or after returning advice is to insert advice body blocks at the appropriate location of the identified join points. These locations are discussed in detail in the prototype description. Linked editing is then enabled between those inserted blocks and the advice body itself. The effect is that editing the advice body edits the inserted advice blocks around all the join points.

Similarly the effect of intertype declarations is to insert definitions into the identified classes, and then enable linked editing between the introduced declarations and the intertype declaration itself. The final effect is that editing the intertype declaration edits the introduced declarations in all the identified classes.

All four constructs have the effect of producing a uniform slice of the code-base where a single edit effects simultaneous changes to the rest of the components in the slice.

4.3.2 Gather

In this join point model, the means of semantic effect is a **gather** construct that groups together views of the identified join points. These copies are linked to the join points where they appear originally in the code-base, so that editing the view edits the actual join points.

Instead of producing a uniform slice of the code-base as in advice/ITD, this construct collects the whole slice and produces a single view for it. However, editing a part of the construct only changes the join point that it corresponds to.

4.3.3 Overlay

The means of semantic effect for the **overlay** join point model is an overlay construct that groups together views of the identified join points, but instead of displaying all the views showing the original code fragments, only a single abstraction of the views is displayed.

In this abstract view, some parts of the original join points are shown, in particular the similar parts of the join points are shown. It also shows additional information about the join points in that one can find out how the join points compare to each other by inspecting the **overlay** construct and locating which parts are grayed out and which parts are not.

Chapter 5

Implementation

We have implemented the three prototypes as Eclipse² plug-ins. The fluid aspect editor is an extension of the standard Eclipse text editor.³ The components of the implementation of the prototypes are presented in Fig. 5.

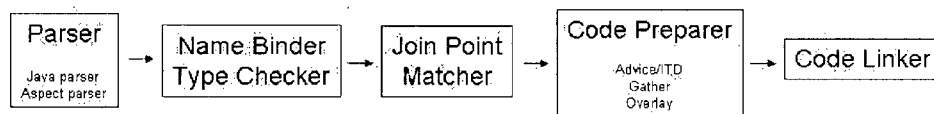


Figure 5.1: Stages of the implementation architecture

5.1 Parser

To parse the content of the aspects, we implemented a parser in addition to the existing Java parser. This parser is responsible for parsing the pointcut definitions as well as parsing the advice, intertype, gather or overlay declaration. We used the ANTLR framework⁴ to generate the parser.

We used the Eclipse Java parser⁵ to parse Java code into abstract syntax trees, which are defined in the DOM package.⁶ We created additional AST classes for pointcut and advice, as well as overlaid AST nodes which we will discuss in more detail in section 5.4.3.

²URL: www.eclipse.org

³org.eclipse.ui.editors.text.TextEditor

⁴ANTLR - ANother Tool for Language Recognition. URL: www.antlr.org

⁵org.eclipse.jdt.core.dom.ASTParser

⁶org.eclipse.jdt.core.dom

5.2 Name binding and Type Checking

Since we require name binding and type information in the join point matching phase, we also implemented a name binder and a type checker. This is implemented with three visitors. The first visitor is responsible for gathering all class declarations as well as recording all the method and field declarations of each class. The second visitor performs most of the name binding where all variable references are referred back to their declarations, as well as some of the field references and some of the method invocations. The last visitor ensures type restrictions are met by looking up the type of each expression and determining if there are any type conflicts. The remaining field references and method invocations that are not bound in the previous visitor are also bound once the type information becomes available.

After this phase, each reference is connected back to its declaration and there is enough information for the join point matcher to determine if any given AST in the code-base satisfies a pointcut.

5.3 Join Point Matcher

The AST corresponding to the whole code-base is traversed once to determine which code fragments are matched by pointcuts appearing in fluid aspects. Only the ASTs of the right kind are checked against the pointcut. For example if the `invocation` pointcut is used, then only ASTs corresponding to method invocations are checked against the pointcut. All the information is present to generate the method or field signature at an AST for comparison with the method or field pattern specified in a pointcut.

The pattern in the pointcut is then converted to a regular expression and the AST is matched if the regular expression matches the corresponding signature of the AST.

5.4 Code Preparation and Linking

This stage of the implementation is responsible for preparing the code both in the code-base and within the fluid aspects so that linked editing can be used to connect the aspect to the rest of the matched code fragments in the code-base.

5.4.1 Before/After/ITD

For advice declarations, first the advice block corresponding to each join point is located. If the block does not already exist, then the text of the advice declaration body is inserted into the appropriate location along with a comment with the information as discussed in the prototype discussion. The location in which the block is inserted into depends on the type of advice. Refactoring of the join point, if necessary, is done before the location can be determined.

Once all the blocks are located, their contents are compared. Currently our prototype requires that linked blocks be identical to each other. An error is thrown if this is not the case. Once all the blocks are found to be identical, they are then linked back together with the advice declaration body by using the JDT linked editing mechanism.⁷ The linked editing mechanism takes care of keeping the linked blocks consistent; any edit in one block will result in the same change occurring in all other blocks.

For intertype declarations, first the tool ensures that an annotation has been inserted for each ITD. If not a new annotation is inserted with the aspect name as its attribute. Then for each identified class that the ITD applies to, the field or method declaration being introduced is inserted at the beginning of the class if it is not introduced already. The declarations are then linked back to the original ITD with the same linking mechanism as advice. Note that a library containing the definition of the annotation has to be included on any project that uses this tool otherwise the annotations that are inserted for an introduced method or field declaration will be undefined.

5.4.2 Gather

For **gather** declaration, an editable view of the text of each identified AST is shown by inserting the code fragment of the AST inside the **gather** body. The name of the file that the AST appears in precedes the code fragment. To allow for editing of the AST in the **gather** body, the code fragment inside the **gather** declaration is linked back to the original code fragment.

5.4.3 Overlay

Overlay constructs are generated after a set of code fragments are compared with each other and the similarities and differences are located. The comparisons are done on the ASTs instead of at the text level since white spaces

⁷org.eclipse.jface.text.link

and other syntactic formatting components are ignored in the ASTs; we can also easily compare the types of the ASTs and obtain their children without having to reparse the text each time.

In the simple overlay, the overlay operation always takes a set of code fragments and generates a single overlaid structure. This structure is generated by `OverlayPreparer`, which implements something similar to an AST visitor. Instead of traversing a single AST structure, it traverses the whole list in parallel. At each AST level, the node of each AST at that level is compared to each other to make sure that they are the same kind of AST node. If the kind of one AST differs from the others then a grayed-out node is returned. The methods corresponding to the visit methods of the visitor now return AST nodes instead of boolean values. For non-leaf node, a new node of the same kind is returned with the result of overlaying the children as the new children. For leaf nodes, either a replicate of the nodes is returned or a grayed-out node is returned depending on the similarity of the nodes.

To minimize the changes to the existing AST classes, and since there are different restrictions for the kinds the children of some AST nodes must be, different kinds of grayed-out nodes are added to the AST package. For example the child representing the condition of an if statement has to be an expression, and the child representing the then clause has to be a statement. Different kinds of grayed-out nodes are added into the AST package to accommodate the different types. There are grayed-out nodes for expression, modifier, simple name, variable declaration, statement, string literal and type.

Once the overlaid structure is created, it is then linked back to the original code fragments. Since the prototype utilizes the linked editing provided by the Eclipse JDT plugin, only identical parts between the original code fragments and the overlaid structure are linked together. In other words, none of the grayed-out nodes are linked back to their corresponding code fragments. White spaces also become a problem when compound AST nodes are linked together. This is overcome by linking all the children of the nodes first. Then if the regions of two links are adjacent to each other, the two links are combined into a link between the combined regions of the original links of the text.

5.4.4 Overlay Groups

The splitting of a set of code fragments into groups based on similarity is implemented by comparing certain attributes of the code and then dividing

the code fragments based on the similarities of those attributes. Currently, the attributes that are used to compare the fragments are hard-coded in the implementation. Some of these attributes include number of children, the AST kind and method names (if the AST is a method declaration or a method invocation).

In simple overlay a single overlay construct is generated for a set of code fragments. When a comparison between two nodes shows that the values of one of the attributes being compared are different, either the comparison will proceed as before and return a grayed-out node if that attribute is not one that is used to separate the groups, or an **OverlayException** is thrown.

The **OverlayException** is caught where the generation of the overlaid structure is initiated. The AST that causes the exception to be thrown (i.e., the AST that differs from the ASTs that have been compared already) is removed from the group and the overlay generation process is restarted with the new group. This continues until an overlaid structure is successfully generated. Then we perform the same steps for the remaining ASTs until all ASTs have been compared and are represented in one of the overlaid structures.

5.5 Limitations

We have identified certain limitations of our approach in the prototypes. Since the linked editing provided by Eclipse requires that all the documents that are linked be opened, our prototypes will not work for a set of join points that are scattered over more than six hundred editors; Eclipse throws an editor handle exception indicating that too many editor handles have been opened.

Secondly, since all linked text is changed simultaneously whenever one location is changed, the memory usage becomes very large when running the prototypes with fluid aspects with many join points. Currently when we run the prototypes on a medium-sized system there is a slight but not significant delay when editing constructs that are linked to multiple different files. Most of the delay occurs in the preparation and link creation stage and this delay ranges from 1 to 5 seconds.

Chapter 6

Additional Example

In the preceding chapters we have looked at scenarios based on the small figure drawing example presented in Chapter 3. In this chapter we present an additional example which uses JHotdraw⁸ as the code-base. JHotdraw is an open source drawing tool written in Java that comprises of 57360 lines of source code.

6.1 Consistency Enforcement

The combination of **overlay** and **gather** can be used to enforce consistency among related code fragments. We will now describe a process through which we enforce an ordering policy for the undo methods.

We start by creating an overlay for the undo methods. When we first do this, with similar fragments grouped together, the identified code fragments are separated into 6 different groups. (See Fig. 6.1)

Upon closer inspection, we observed that some groups differ just because of the ordering of statements, in particular the position of the if statements. One group has the following ordering:

```
if (! <condition>)
    return false;
<statements>
```

And the other group has the following ordering:

```
if (<condition>)
    <statements>
return false;
```

Since the order of the if statement and the return false statement are inconsistent in the two sets, they are separated into two groups. In our

⁸JHotdraw - URL: www.jhotdraw.org



Figure 6.1: Part of the overlaid undo methods - the undo method in `InsertImageCommand` is inconsistent with the similar undo methods.

scenario, we judged that making the code-base more consistent would help with understanding of the code.

This was done by reordering the if statement and the statements following the if statement in one of the groups. This edit itself can be done

using the overlay. Fig. 6.2 shows the new overlay group, which keeps the undo method in InsertImageCommand.java in the same group as other undo methods that test for the condition `super.undo()`.

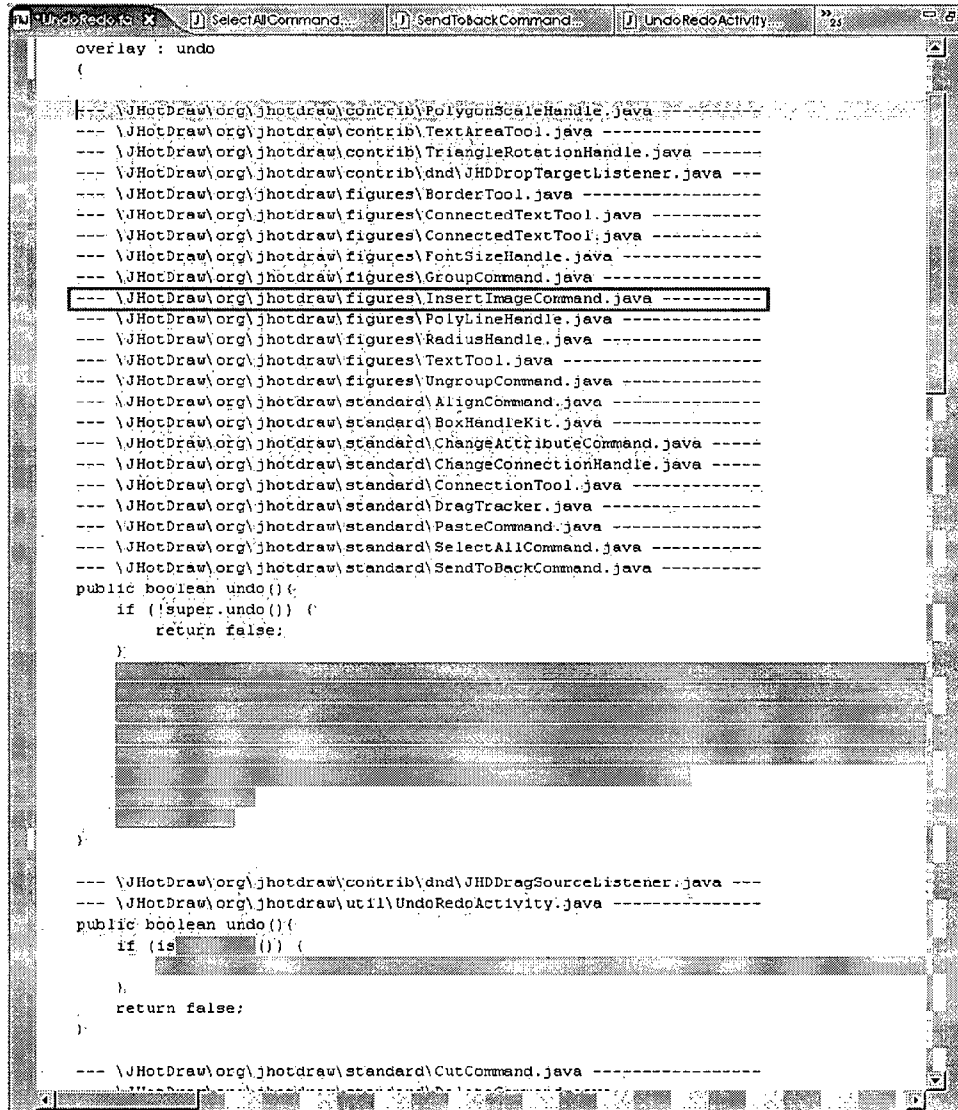


Figure 6.2: The updated screenshot of the overlaid groups - The first group now includes InsertImageCommand which has an if statement with similar conditions as the other method declarations.

Chapter 7

Comparison

In the introduction we defined the problem to be addressed as the two limitations that are currently present in linguistic AOP approaches. These two limitations are AOP requires new programming language adoption and only one decomposition of the software system can be present at any one time.

Our hypothesis is that IDE-based support of providing AOP, in particular IDE support for modularizing specific tasks by localizing the editable presentation of the program source, can overcome those two limitations.

The discussion in this chapter is intended to evaluate how the different prototypes we have developed solve those problems, compared to each other, compared to using linguistic AOP and compared to using OOP (plain Java). We also compare additional features that are desirable in IDE support. These features are modularization in editing and viewing of crosscutting concerns, as well as modularization in viewing of join points. Another desired property is for the tool to reduce the load of a developer having to perform duplicated editing tasks by replacing them with a single edit task that accomplishes the same.

7.1 Fluid AOP

In *before/after/ITD* JPM, the editing of a given concern is modularized because a software developer only needs to perform edits in the aspect view inside advice bodies or ITD bodies. There is also a modularized view of the concern in the aspect since one can deduce the join points from the pointcut, as well as deduce the effects of the aspect by inspecting the type of advice and the advice bodies; however, if one needs any information about the join points other than the signature of the method, such as the first statement of each method body, then the aspect view is insufficient. Navigation to scattered code is still needed to find such information about join points.

This JPM is best suited for performing modularization of a concern that is uniform across all join points, and where the concern is quite separate from the core decomposition, so that no additional information about the

join points other than the pointcut that is used to match them is required and no additional navigation of code is required for performing edits in this concern.

In the **gather** JPM, join points, or at least an editable view of the join points, appear in the aspect itself. All the information about a join point can be found from the aspect view; therefore the viewing of the join points is modularized. (Although information about the context in which the join points appear still requires navigation to the scattered primary home of those join points.) The editing and viewing of a crosscutting concern is also modularized since it can all be done within one single aspect view. One property this JPM lacks is the ability to eliminate duplicated changes since each change to a different join point has to be done manually. Even though the changes can be done in close proximity to others, one still needs to go and make every change.

This JPM appears well suited for an aspect in which only a small set of join points is of interest. Since all the information about each join point is shown inside the **gather** construct, it is likely useful for code understanding and as a starting point in narrowing subset of code for investigation without losing any information from too much abstractions.

Finally in the **overlay** JPM, duplicated changes are greatly reduced since changes that can be done uniformly for all join points in an equivalence class can be done on the **overlay** construct. Not all information in a join point is displayed in this JPM; only the similar parts are displayed. However more information in relation with the other join points can be discovered from looking at the **overlay** construct; one can find out which parts of the join points are similar and which parts are different.

The viewing and editing of crosscutting concern are also modularized, even though the concern is not as structured as the **advice/ITD** JPM. In **overlay**, the crosscutting concern can be any subset of the parts that are displayed fully in the overlayed structure, as opposed to in **advice**, the **advice** block is the action of the concern.

The effectiveness of **overlay** depends greatly on the similarity of the identified join points. If the join points are drastically different, then all that will be shown for the overlay would be a grayed out box, which provides no additional information to the software developer. As shown in the example section, **overlay** can be useful after more information about the identified join points is discovered from inspecting the **gather** construct first, then either generate a single overlay if all the join points are similar to provide a meaningful abstraction, or then generate several overlays that distinguish the significant differences between the join points.

7.2 Fluid AOP vs. Linguistic AOP

One of the advantages of linguistic AOP comes from the ability to delay binding till the late stages of the compilation process or even up to run time. This late binding allows for pluggable aspects. In fluid AOP, since binding occurs at the editing phase, it is more difficult to remove aspects once they have been inserted.

Another difference between the two AOP approaches is that in linguistic AOP, one code fragment can only be part of one aspect. In Fluid AOP, the same code fragment can appear in multiple aspects, since the aspects are just alternate views of the system.

On the other hand, a strength of fluid AOP is that after aspect editing, no special compiler is needed to compile the code. This may ease adoption of AOP since one person on a team can use fluid AOP without causing the rest of the team to use a new compiler.

Currently the prototypes we have developed have a less expressive means of identifying join points than AspectJ. The current prototypes only have pointcuts that match declarations of method and field and invocation of method. In Section 9 we discuss more pointcut declarations that we expect will be useful in Fluid AOP but which are not yet implemented. Since the join points in Fluid AOP differ from AspectJ, it is natural that a different set of pointcuts are needed.

7.3 Fluid AOP vs. OOP

Since fluid AOP is a support in an IDE and does not change the language of the code-base, all existing OOP tools work as before on the code; therefore all the advantages that OOP provides are also present with fluid AOP approach.

In addition, fluid AOP provides fluid aspects which can modularize code that appears scattered or tangled in the code-base. This makes practical and frequent tasks easier to perform. For example the ability to group together multiple join points and perform a single uniform edit to these join points make refactoring (for example adding method parameter, or renaming) easier and less error prone. In particular, one feature of renaming that fluid AOP can handle is systematically renaming part of a set of method names. For example one can rename all the `set` methods from starting with `set` to starting with `mutate`.

Fluid AOP can also serve as another tool for aiding in code exploration and understanding, as seen in Chapter 3.

Chapter 8

Contribution

In this research we propose an IDE-based approach to modularizing crosscutting concerns for specific tasks. We consider modularization of a concern in this context to mean localization of code that a software developer has to view or edit to perform a task. We illustrated the feasibility of fluid AOP by presenting three join point models with corresponding prototype implementations. These three join point models are separate but can be used together as shown in the example in Chapter 6. We have outlined some key components in the implementation of our prototype for each JPM.

Fluid AOP provides a novel way of modularizing crosscutting concerns, by providing fluid editable views on demand, hence giving software developers the option of slicing the code-base in multiple ways to help suit current task of interest.

We also presented multiple prototypes that show ways to display multiple code fragments inside one editor. The most novel construct in these prototypes is the **overlay** construct. An editable code level presents one abstraction of multiple code fragments. This construct shows a combination of similarities and differences between the code fragments that are not immediately evident when browsing between the code with typical OO IDE support.

These fluid AOP prototypes show that even though the underlying code-bases are semantically and syntactically identical to OOP modulo the aspects, additional information is inserted in the code-base that can be helpful. In before/after/ITD, comments or annotations are inserted in the code to signify the existence of an aspect. For developers that are not using fluid AOP tools, it becomes clear that the advice blocks or introduced methods or fields are placed there to implement a crosscutting concern.

The fluid aspect files (with extension .fa in our prototypes) can also serve as a way to make aspects explicit in the code-base. Even though these files do not have compile-time effects in the code-base, it provides information for identifying the elements that make up the implementation of the aspect.

Through the design for the various prototypes, we have come to learn that simple text comparison in **overlay** only works for a small set of very simi-

lar code fragments. We also learned that using the linked editing mechanism in Eclipse and extending the text editor in Eclipse are good starting points in building our prototypes but to improve the performance and features of the prototypes, we will have to implement our own linking mechanism and editor. These are discussed further in Chapter 9.

Chapter 9

Future Work

This is only the beginning of the exploration of fluid AOP. There are many directions which we can take to extend this work. Some are described in this chapter.

9.1 Extending the Fluid AOP Space

We currently have three join point models that form a subpart of the fluid AOP space. One future research direction could be to extend the subpart of fluid AOP by creating more JPMs. By the way we define a join point model, there are three components we can expand to create new JPMs.

The first is to expand the set of join points. Currently the join points are only code fragments corresponding to method declarations, method invocations and field declarations. Possible useful join points can include code fragment corresponding to any statement or expression to make the join point granularity smaller.

The second is to expand our pointcuts. An example of a possible useful pointcut is one that identifies the surrounding method declaration of a method invocation instead of identifying the method invocation itself.

Finally we can explore new ways to display matched join points. Currently we have `advice`, `gather` and `overlay` and we tried to combine `overlay` and `gather`. Future work can look into new kinds of constructs to display the identified code fragments.

9.2 Improving Current Implementations

9.2.1 Overlay Groups

The current implementation of grouping into equivalence classes in `overlay` is implemented by hard coding the various factors in the implementation of the tool. To change those factors requires going in and updating the source code. We will explore different ways to let a software developer control the

individuation criteria for groups. One solution is to give software developers a knob-like control which controls how many groups will be presented. Instead of using the comparison step to decide how many groups there will be, we can first separate the groups by a clustering algorithm, then use the current **overlay** (the version that takes a set of code fragments and return exactly one overlaid construct) to generate the overlaid construct.


[15] describes algorithms for clustering elements which require that each element be converted to a vector which can then be used to calculate distances between elements. To make use of these algorithms, we can have a vector with length equal to the number of different AST node types possible. Each code fragment can then be converted into a vector by filling in how many nodes of each type are present as children of the code fragment. Then we could use the clustering algorithm to decide the grouping and then present the overlaid construct of each group. The clustering algorithms enable software developers to specify exactly how many number of groups should be presented.

9.2.2 More Overlay Features


We can also improve the display of overlay, for example instead of graying out the different parts completely, we can show a list of the differences. We also plan to design more meaningful overlaid structures and better similarity comparer for creating them. Some possible options are to include more renaming mechanisms or overlaying different structures, such as overlaying method declarations with method invocations.

Another part of **overlay** that can be improved is that currently all the file names that the overlaid code fragments belong to are displayed preceding an overlaid construct to show which files are involved. This list can become too large to be useful if the set of files involved is large. One way to make this list more scalable is to let developers collapse the list and expand it only when they are interested in the files involved.

9.2.3 Graphical Aspect Editor

Currently the aspect editors in the three prototypes are all plain text editors. Using a graphical editor, for example we can use the Eclipse Graphical Editing Framework (GEF)⁹ to implement this editor, will provide more flexible ways to present information for the user. We can utilize different graphics to represent different results in overlay. A gray box  can be used to mean

⁹URL: www.eclipse.org/gef

that all join points have an element at that location but they have different content; a half gray box  can be used to mean that some join points have an element at that location with different content, but some join points do not have that element.

9.3 Improving Scalability

One limitation to our implementation of the prototypes is that it does not scale for large code-base. One step that has to be taken is to implement our own mechanism for linking code fragments. When the code-base gets larger, simultaneous updating of all join points no longer become feasible. Editing changes should be propagated either when the affected file is brought to view or when user specifies that the changes should be propagated.

Another option to minimize the delay in linked editing is to make the granularity of change propagation be larger. Currently it is at the level of a single key stroke. We can instead try to group together key strokes that occur in close proximity to each other and delay refreshing all the affected documents till a group of changes have been made or the user pauses.

Bibliography

- [1] Joao Araújo, Jon Whittle, and Dae-Kyoo Kim. Modeling and composing scenario-based requirements with aspects. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International (RE'04)*, pages 58–67, Washington, DC, USA, 2004. IEEE Computer Society.
- [2] Bay-Wei Chang, Jock D. Mackinlay, Polle T. Zellweger, and Takeo Igarashi. A negotiation architecture for fluid documents. In *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 123–132, San Francisco, California, United States, 1998. ACM Press.
- [3] Mark C. Chu-Carroll, James Wright, and David Shields. Supporting aggregation in fine grained software configuration management. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 99–108, Charleston, South Carolina, USA, 2002. ACM Press.
- [4] Robert DeLine, Amir Khella, Mary Czerwinski, and George Robertson. Towards understanding programs through wear-based filtering. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 183–192, St. Louis, Missouri, 2005. ACM Press.
- [5] Michael Desmond, Margaret-Anne Storey, and Chris Exton. Fluid source code views for just in-time comprehension. In *Software Engineering Properties of Languages and Aspect Technologies Workshop*, Bonn, Germany, 2006.
- [6] A. Nico Habermann and David Notkin. Gandalf: software development environments. *IEEE Trans. Softw. Eng.*, 12(12):1117–1127, 1986.
- [7] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

- [8] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, Boston, Massachusetts, 2003. ACM Press.
- [9] Doug Janzen and Kris De Volder. Programming with crosscutting effective views. In *ECOOP '04: Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 197–222, Oslo, Norway, 2004. Springer-Verlag.
- [10] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, Portland, Oregon, USA, 2006. ACM Press.
- [11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
- [12] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.
- [13] Jacques Klein, Loïc Hérouët, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 27–38, Bonn, Germany, 2006. ACM Press.
- [14] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.
- [15] Lefteris Moussiades and Athena Vakali. Pdetect: A clustering approach for detecting plagiarism in source code datasets. *The Computer Journal*, 48(6):651–661, 2005.
- [16] Mikkel Rønne Jakobsen and Kasper Hornbæk. Evaluating a fisheye view of source code. In *CHI '06: Proceedings of the SIGCHI conference*

- on *Human Factors in computing systems*, pages 377–386, Montréal, Québec, Canada, 2006. ACM Press.
- [17] Harold Ossher and Peri Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 734–737, Limerick, Ireland, 2000. ACM Press.
 - [18] Andres Diaz Pace and Marcelo Campo. Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, 2001.
 - [19] Philip J. Quitslund. Beyond files: programming with multiple source views. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 6–9, Anaheim, California, 2003. ACM Press.
 - [20] Awais Rashid, Ana Moreira, and Joao Araújo. Modularisation and composition of aspectual requirements. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 11–20, Boston, Massachusetts, 2003. ACM Press.
 - [21] Warren Teitelman and Larry Masinter. The interlisp programming environment. *Computer*, 14(4):25–33, April 1981.
 - [22] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.
 - [23] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.