Separation of Concerns
with
Procedures, Annotations, Advice and Pointcuts

Gregor Kiczales

University of British Columbia
201-2366 Main Mall
Vancouver, BC V6R 1X4, Canada

gregork@acm.org

Mira Mezini

Technische Universität Darmstadt
Hochschulstrasse 10
D-64289 Darmstadt, Germany

mezini@informatik.tu-darmstadt.de

Abstract. There are numerous mechanisms for separation of concerns at the source code level. Three mechanisms that are the focus of recent attention – metadata annotations, pointcuts and advice – can be modeled together with good old-fashioned procedures as providing different kinds of bindings: procedure calls bind program points to operations, annotations bind attributes to program points; pointcuts bind sets of points to various descriptions of those sets; named pointcuts bind attributes to sets of points; and advice bind the implementation of an operation to sets of points. This model clarifies how the mechanisms work together to separate concerns, and yields guidelines to help developers use the mechanisms in practice.

1 Introduction

Programming language designers have developed numerous mechanisms for separation of concerns (SOC) at the source code level, including procedures, object-oriented programming and many others. In this paper we focus on three mechanisms – metadata annotations [4], pointcuts [16] and advice [33] – that are currently attracting significant research [9, 10, 19, 34] and developer interest [1, 11, 12, 14, 20].[1]

Our goal is understand what kinds of concerns each mechanism best separates, and how the mechanisms work together to separate multiple concerns in a system. We also seek to provide developers with answers to questions about what mechanism to use in any given situation. To enable this, we study how the three newer mechanisms, along with good old-fashioned procedures, separate concerns in a simple example.

[1] The paper assumes a reading familiarity with pointcuts and advice as manifested by AspectJ [16] as well as the Java 1.5 metadata facility [4]. Metadata annotations, pointcuts and advice can appear in a wide range of other languages [3, 13, 21, 28, 31] [8, 30], but we do not explicitly discuss that generalization here.

The study is focused on four key design concerns within the example. We present seven implementations of the example that use the mechanisms in different ways. We also present ten change tasks and how they are carried out in each implementation. Based on this, the paper provides:

1. An analysis of the degree to which the different mechanisms are able to separate and clarify the four design concerns in the seven implementations.

2. An analysis of the degree of locality of each change task for each implementation, and a comparison of that locality to the static separation.

3. A unified model of the four mechanisms showing how they work together to separate concerns.

4. An initial set of guidelines for using the mechanisms in development practice.

The paper is structured as follows: Section 2 presents the example, its four key design concerns and the seven implementations. Section 3 analyzes the static locality of the concerns in each implementation, and the locality of the change tasks for each implementation. Section 4 presents the unified model of the mechanisms. Section 5 presents the usage guidelines. We finish with related and future work and a summary.

2 The Example

Our comparison of the mechanisms is based on seven implementations of a simple graphical shapes example [16, 18]. In this example, a number of graphical shapes are shown on a display. Each shape has its own display state, and when that state changes, the display must be signaled so it can refresh itself. This design is shown in Figure 1.

The key objects in the design are the shapes and the display. There is an abstract Shape class, with concrete Point and Line subclasses. (Assume there are other concrete shapes such as Triangle. To save space they are not discussed here.) There is a single Display class, and, for simplicity, there is just a single system-wide display.

2.1 Four Design Concerns

In addition to concerns involving the functionality of the shapes, the design comprises four key design concerns, which are shown as dotted line boxes in Figure 1.

Refresh-Implementation – What is the behavior and implementation of the actual refresh operation?

Context-to-Refresh – What context from the actual display state change points should be available to the refresh implementation?

When-to-Refresh – When should the display be refreshed?

What-Constitutes-Change – What operations change the state that affects how shapes look on the display, i.e. their position?

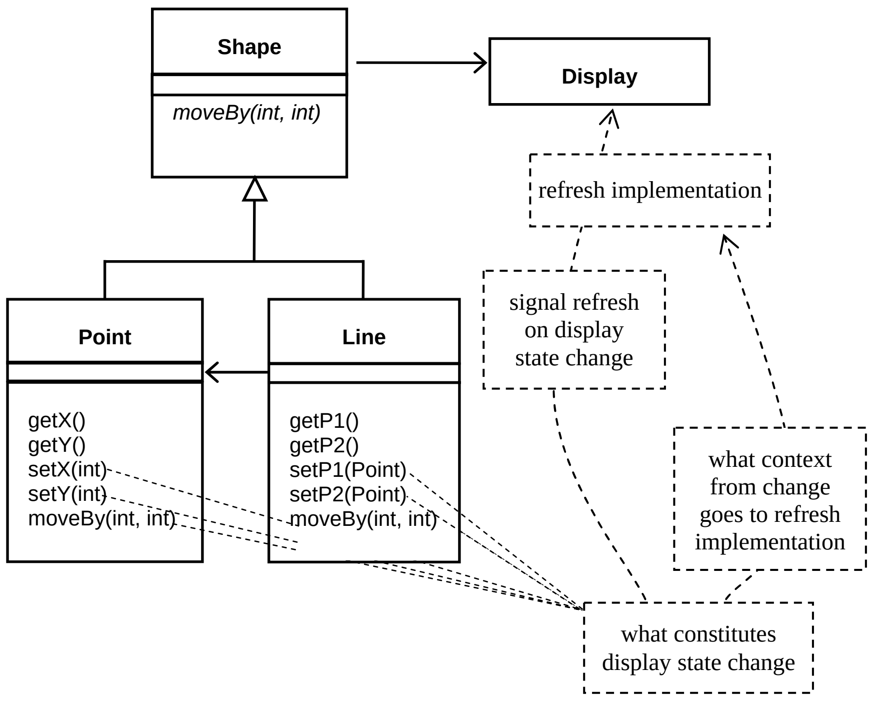

Figure 1. The design of the graphical shapes program, showing the main classes and two additional design concerns not separated as classes.

As is common, these concerns are interconnected. Our design resolves When-to-Refresh by deciding that refresh should happen immediately after each display state changes. This brings What-Constitutes-Change into focus as a concern that must be resolved. One could also argue the causality in the other direction, in that having thought about display state changes one then decides they should cause refreshes.

2.2 Seven Implementations

This rest of this section presents the code for seven implementations of the example. Discussion of the implementations is deferred until Section 3.

Straw-Man

The first implementation is a straw man, no good programmer would write this code today. Its purpose is to explicitly introduce procedures into the discussion.

In this implementation all of the methods that change display state directly include several lines of code that implement the actual display refresh. For example, the setX method looks like:

```
void setX(int nx) {
  x = nx;
  Graphics g = Display.getGraphics();
  g.clear();
  for( Shape s : Display.getShapes() ) {
    s.draw(g);
  }
}
```

GOFP

This implementation uses a good old-fashioned procedure (GOFP) to capture the refresh implementation. Each of the methods includes, at the end of the method, a call to a procedure (static method in Java) that refreshes the display.

```
void setX(int nx) {
  x = nx;
  Display.refresh();
}
```

The body of that procedure is the several lines of code that was duplicated in I1.

```
static void refresh() {
  Graphics g = getGraphics();
  g.clear();
  for( Shape s : getShapes() ) {
    s.draw(g);
  }
}
```

Annotation-Call

This implementation uses Java 1.5 metadata annotations [4]. Each method that changes display state has an annotation that says that executing the method should also refresh the display.

```
@RefreshDisplay
void setX(int nx) {
  x = nx;
}
```

A single after advice declaration serves to ensure that execution of methods with this tag calls Display.refresh(). The advice is written as:

```
after() returning: execution(@RefreshDisplay * *(..)) {
  Display.refresh();
}
```

There are other ways to associate run-time behavior with annotations, typically involving ad-hoc post-processors. We use advice in this paper because it is simple and compatible with the rest of the paper.

Annotation-Property

This implementation differs from the previous one only in the name of the annotation. Here the annotation name describes a property of the method – that it changes state that affects the display of the shape – rather than directly saying that executing the method should refresh the display. So the methods look like

```
@DisplayStateChange
void setX(int nx) {
  x = nx;
}
```

Again, a separate advice declaration says that execution of methods with the DisplayStateChange annotation should call Display.refresh().

```
after() returning: execution(@DisplayStateChange * *(..)) {
  Display.refresh();
}
```

Anonymous-Enumeration-Pointcut

This implementation uses an anonymous enumeration-based pointcut to identify method executions that change display state. So the methods have no explicit marking, and simply look like:[2]

```
void setX(int nx) {
  x = nx;
}
```

The entire implementation of signaling a display refresh consists of a single advice on an anonymous pointcut that explicitly enumerates the six methods; the body of the advice calls Display.refresh().

```
after() returning: execution(void Shape.moveBy(int, int)
                   || execution(void Point.setX(int))
                   || execution(void Point.setY(int))
                   || execution(void Line.setP1(Point))
                   || execution(void Point.setP2(Point)) {
  Display.refresh();
}
```

[2] Even though the method is not explicitly marked by the programmer, IDE support such as the ADJT Eclipse plug-in will show that the advice exists, for example with a gutter marker next to the method declaration [27].

Named-Enumeration-Pointcut

In this implementation the pointcut from the previous implementation is pulled out and given an explicit name. Again, the method bodies require no marking to enable display refresh signaling.

```
void setX(int nx) {
  x = nx;
}
```

The pointcut and advice are:

```
pointcut displayStateChange():
  execution(void Shape.moveBy(int, int)
  || execution(void Point.setX(int))
  || execution(void Point.setY(int))
  || execution(void Line.setP1(Point))
  || execution(void Point.setP2(Point));

after() returning: displayStateChange() {
  Display.refresh();
}
```

Named-Pattern-Pointcut

In this implementation only the pointcut differs from the previous implementation. Rather than enumerating the signatures of the methods that change display state, this implementation relies on the naming convention the methods follow to write a more concise pointcut. Again, the method bodies require no marking to enable display refresh signaling.

```
void setX(int nx) {
  x = nx;
}
```

The pointcut and advice are:

```
pointcut displayStateChange():
  execution(void Shape.moveBy(int, int)
  || execution(void Shape+.set*(..));

after() returning: displayStateChange() {
    Display.refresh();
}
```

The `execution(void Shape+.set*(..))` pointcut means execution of any method defined in Shape or a subclass of Shape, that returns void, has a name beginning with ‘set’, and takes any number of arguments.

3 Analysis of the Implementations

Our analysis of the different mechanisms is based on assessing the degree to which the seven implementations separate the four design concerns identified in Section 2.1. The assessment uses three criteria: locality of implementation, degree to which the implementation is explicit rather than implicit, and locality of change in a simple evolution experiment. The assessment of locality and explicit implementation is discussed in Section 3.1. The locality of change assessment is covered in Section 3.2. All three assessments are summarized in Table 1.

3.1 Locality and Explicit Representation

One way to compare how the implementations separate these design decisions is whether the code that implements the decision is localized. Another criterion is the degree to which the implementation of the decision is captured explicitly as opposed to implicitly. This analysis is summarized in the top part of Table 1.

The capture of Refresh-Implementation is implicit and non-localized in Straw-Man. There is no single place in the code that explicitly says that display refresh is implemented by the several lines of code. Instead, each method that the developer decided constitutes a display state change includes code that implements refresh. In the GOFP and subsequent implementations, the refresh procedure declaration captures this concern in an explicit and localized way. The declaration is read as saying “this is the refresh implementation – bind Display.refresh() to this code”.

The capture of Context-to-Refresh is implicit and non-localized in Straw-Man. No single place in the code explicitly says that no values from the change context are available to the display refresh implementation. In GOFP, the procedure declaration and every call to the procedure explicitly say that no arguments are passed, so this concern is explicit. But because this is expressed in the procedure and all the calls to it, it is non-localized. In the Annotation implementations there is a single call to the procedure, so this concern is captured explicitly and in two places. The same is true for the Anonymous-Enumeration-Pointcut implementation. In the last two implementations the named pointcut also expresses this concern, so it is captured explicitly in three places.

The capture of the When-to-Refresh is implicit and non-localized in Straw-Man, GOFP and Annotation-Call. It is localized but implicit in Anonymous-Enumeration-Pointcut. No single place in these implementations explicitly says that execution of methods that change display state should cause a display refresh. In GOFP the scattered calls to Display.refresh() are implicitly about the fact that the affected methods change display state and so must refresh; but all they say explicitly is that the affected methods call Display.refresh(). The same is true for the scattered RefreshDisplay tags in Annotation-Call. In Anonymous-Enumeration-Pointcut, the pointcut localizes the description of what constitutes change, but because no name is given to it, the binding of when to refresh is not to a clear notion of on display state changes, but instead to an enumerated set of conditions. In the other implementations, this concern is explicit and localized in the after advice declarations, which say that any display state change should cause a refresh.

The capture of the What-Constitutes-Change is implicit and non-localized in Straw-Man, GOFP and Annotation-Call – no single place in these implementations explicitly says that execution of the four setter methods and the two moveBy methods changes display state. In Annotation-Property, the DisplayStateChange annotations capture this concern in an explicit, but non-localized way. In Anonymous-Enumeration-Pointcut, this concern is localized, but implicit. In the two named pointcut implementations this concern is localized and explicit. The Named-Pattern-Pointcut captures the decision about what methods change display state, as well as a rule for what methods are considered to change display state. The variation among the pointcut based implementations is discussed in more depth in Section 5.

Names Matter

The two annotation-based implementations differ only in the name of the annotation, but come out significantly different in our separation of concern analysis. Annotation-Call has the same properties as GOFP with regard to When-to-Refresh and What-Constitutes-Change. This should not be surprising since in Annotation-Call the annotation name makes it feel like alternate syntax for a procedure call, or a syntactic macro [6, 7]. So, like GOFP, Annotation-Call, is conflating these two concerns and simply saying to call refresh at certain points.

On the other hand, in Annotation-Property, When-to-Refresh is captured explicitly and in just one place in the code; What-Constitutes-Change is captured explicitly but is not localized. The different annotation name causes both concerns to be explicit. That names matter is not surprising to programmers, but it is important to note its significance in this case. We return to this issue in Section 5.

3.2 Ease of Evolution

This section analyzes the implementations in terms of how well they fare when performing a set of ten representative change tasks. Most tasks affect just a single concern, reflecting a good modularity in the concern model itself. The question we explore now is what must be done to the code to perform each task – how many edits and how localized are they. The analysis is summarized in the lower part of Table 1 by showing, for each change and each implementation, how many places in each implementation have to be visited and possibly edited by the programmer.

Double-buffering – changes the refresh implementation to use double buffering. So it is a change to just the Refresh-Implementation concern. In Straw-Man, the programmer must edit the refresh implementation code that appears in all the display state change methods. For GOFP and all other implementations only the Display.refresh() procedure must be edited. In Table 1, the Double-Buffering row shows ‘n’ in the first column and 1 in the remaining columns. This is one of the reasons we have learned to introduce a procedure in such cases.

Pass-Changed-Object – provides the actual shape that has changed to the refresh implementation, so that it can optimize refresh based on that information. This constitutes a change to both Refresh-Implementation and Context-to-Refresh. In Straw-Man, this change task involves editing all the state change methods. In GOFP it involves editing the procedure declaration and the call sites in all the state change methods. In the remaining implementations this involves editing the procedure, advice and pointcut declarations. The procedure is edited to accept the shape as an argument, the call sites are edited to pass the current object, and the pointcuts are edited to make the current object accessible.

Disable-Refresh – simply disables activation of display refresh when the state of shapes changes. So this is a change to just When-to-Refresh. In Straw-Man this change requires editing all the state change methods to delete the refresh implementation. GOFP and Annotation-Call require editing all the methods to remove the call to the refresh procedure or the refresh annotation respectively. In the last four implementations this change can be accomplished by removing the aspect containing the advice from the system, or by editing the aspect to delete the advice if for some reason the aspect should remain. The Disable-Refresh table row shows ‘n’ in the first 3 columns and ‘1’ in the last four.

One might argue that GOFP and Annotation-Call can accommodate Disable-Refresh more expeditiously – for GOFP, one could simply “comment out” the body of the refresh procedure declaration, and for Annotation-Call one could delete the advice declaration. But these alternatives are problematic. There may be other callers of the refresh procedure (or clients of the tag), since nothing has marked the procedure or the tag as particular to handling this kind of refresh activation. Even if there are no other callers, the expeditious changes make the code confusing – the reader sees a call to refresh (or the annotation), but must learn elsewhere that they do not do anything.

A programmer might deal with this by introducing an additional procedure, perhaps called Shape.fireDisplayStateChange(), and have that procedure call Display.refresh(). Then this change can be easily accommodated by making the body of the new procedure empty. This has the same effect of introducing the intermediate annotation, and has the same separation properties as Annotation-Property. Other more elaborate rendezvous mechanisms could be used as well. Having this extra procedure vs. not having it is similar to the difference between the two annotation-based implementations.

Reuse-What-Constitutes-Change adds logging of display state changes. So it reuses What-Constitutes-Change, but does not actually change any of the design concerns. In Straw-Man all the state change methods are edited to add logging code. In GOFP all the state change methods are edited to add a call to a logging operation. In Annotation-Call, each method gets an annotation and a new advice is defined. In the last four implementations, a new advice is defined; in Annotation-Property it references the @DisplayStateChange annotation, in the anonymous pointcut it duplicates the anonymous enumeration-based pointcut, and in the named pointcut implementations it references the displayStateChange pointcut. For all but Straw-Man the table includes an extra count assuming the logging operation must be defined as a procedure.

Again, one might argue that this can be accomplished more expeditiously in GOFP and Annotation-Call, simply by directly editing the refresh procedure or the advice to do the logging. This however, associates the logging with the activation of the refresh, rather than directly with the state changes.

Refresh-Top-Level-Changes-Only ensures that in recursive state change methods (e.g. moveBy on Line calls moveBy on Point, which calls setX and setY on Point) only the top-level display state change method causes a refresh. This prevents multiple refreshes for such methods. So it is a change to the When-to-Refresh concern. In Straw-Man and GOFP this change requires editing all the state change methods, to introduce some mechanism that can detect recursive state change method calls and prevent the sub-calls from calling refresh. A common pattern for doing this is to add a second parameter to all the state change methods, indicating whether they are part of a recursive call. Often a second overloaded method is introduced to handle this. In Java the programmer can use thread local state to do this in a more elegant way.

In the implementations that use pointcuts (all after GOFP), this can be done by editing the pointcut to use the cflowbelow primitive to filter out recursive calls; in the named pointcut implementations the AspectJ code for this would involve modifying the advice to be:

```
  after() returning: displayStateChange()
                     && !cflowbelow(displayStateChange()) {
    Display.refresh();
  }
```

which is read as saying to call refresh after any display state change that is not itself within the control flow of another display state change.

The next five changes all affect What-Constitutes-Change in different ways.

Add-Related-Class adds a new Circle subclass of Shape. The new class has setX, setY, setRadius and moveBy methods that constitute display state changes. This represents a modification of the What-Constitutes-Change concern. Straw-Man, GOFP and both annotation-based implementations each require that all the new state change methods be appropriately edited. The two enumeration-based pointcut implementations require that the pointcut be edited. The pattern-based pointcut does not need to be edited, but it must be at least examined to ensure that the new methods are covered by the pointcut.

The next two changes have the same implications for all implementations as Add-Related-Class. They are included nonetheless because they are typical changes to expect in such a system.

Add-Related-Method adds a new Line.setColor(Color) method that should be considered to change display state.

Table 1. Analysis of the seven implementations. The top part of the table shows many places in the code implement the concern, and whether the implementation is **E**xplicit or **I**mplicit; ‘n’ means each of the display state change methods. The bottom part of the table summarizes the change task analysis, showing the number of places each implementation must be edited for each change. The ‘n’ notation indicates that the number goes up as the number of shape classes increases, whereas other numbers are constant. The ‘*’ indicates that the code is only examined, not edited. In this part of the table the first column shows what concerns each tasks changes.

Implementations		Straw-Man	GOFP	Annotation-Call	Annotation-Property	Anonymous-Enumeration-Ptc.	Named-Enumeration-Ptc.	Named-Pattern-Ptc.
Design Concerns								
Refresh-Implementation		I, n	E, 1					
Context-to-Refresh		I, n	E, n+1	E, 2 / 3				
When-to-Refresh		I, n			E, 1	I, 1	E, 1	
What-Constitutes-Change		I, n			E, n	I, 1	E, 1	
Change Tasks	*Concerns*							
Double-Buffering	RI	n	1					
Pass-Changed-Object	RI, CtR	n	n+1	2			3	
Disable-Refresh	WtR	n			1			
Reuse-What-Constitutes-Change	WCC	n	n+1	n+2	2			
Refresh-Only-Top-Level-Changes	WtR	n		1		1 + 1		
Add-Related-Class	WCC	each new method						
Add-Related-Method	WCC	each new method				1		1*
Rename-Methods	WCC	0						
Add-Unrelated-Class	WCC	0						
Add-Unrelated-Method	WCC	0						1

Rename-Methods renames the Line.setP1(Point) and Line.setP2(Point) methods to Line.setEnd1(Point) and Line.setEnd(2).

Add-Unrelated-Class adds an entirely unrelated class to the system. It does not change any of the four concerns. None of the implementations require any editing or examination to perform this change.

Add-Unrelated-Method adds a new Shape.setOwner(Owner) method that has nothing at all to do with display state. This change also does not change any of the four concerns. The first six implementations require no editing, but the pattern-based pointcut must be edited to exclude the new setOwner method.

4 Uniform Characterization of Mechanisms

The above analysis suggests that one useful way to characterize the four mechanisms is as establishing different kinds of bindings along a path from points in a program to the implementation of an operation that must execute at those points. As shown in Figure 2, each mechanism introduces an explicit intermediate step along the path, and makes an explicit binding between those steps. These explicit steps and bindings work together to separate larger, higher-level concerns such as the four discussed here.

In these terms, a procedure call binds a point in the program to an operation – it says call this operation at this point in the program execution. A procedure declaration binds the operation to an implementation. So the effect of using a procedure – a declaration and one or more calls to it – is to introduce an explicit operation (the procedure), bindings from points in the program to the operation (calls), and a binding from the operation to the implementation (the declaration).[3] Annotations, pointcuts and advice introduce other explicit intermediate elements and bindings.

In discussing the relation between annotations and pointcuts, we use the following terminology: *Annotations* are the syntactic identifiers described by JSR-175 [4] that the programmer places in the program (i.e. @DisplayStateChange). Properties are the characteristics of points on which pointcuts can match, including class and method names, access modifiers etc. *Pointcut names* are the programmer defined names for pointcuts. We use the term *attribute* to include both annotations and pointcut names. In other words, attributes are user-defined names that can be attached to program points.

Annotations bind attributes to program points. An annotation such as @DisplayStateChange binds the DisplayStateChange attribute to the program point.

[3] We use the term *procedure declaration* to refer to a construct that defines both signature and implementation, such as a static method declaration within a class in Java, as opposed to a construct that just declares the procedure's signature.

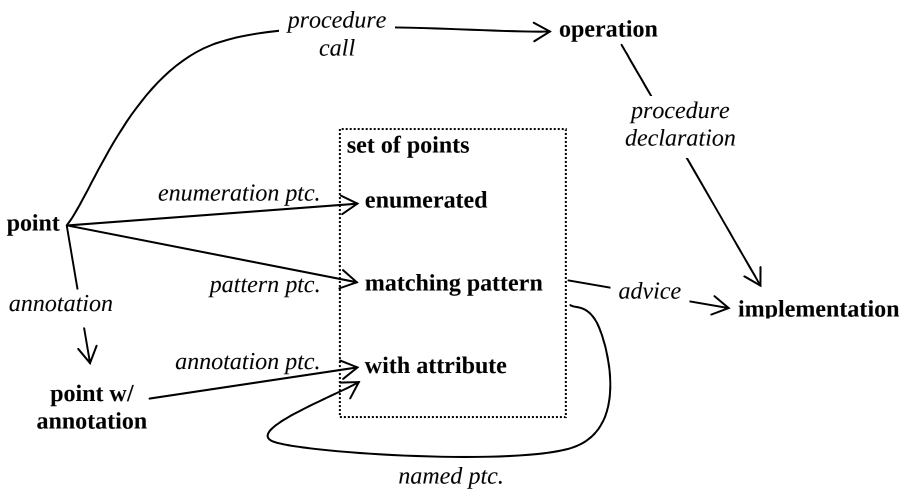

Figure 2. Intermediate elements and bindings established by the mechanisms. Elements are shown in boldface, the mechanisms are in italics.

There are several different kinds of pointcuts. Enumeration-based pointcuts make a set of points explicit, and establish a binding between the set and each of the points.

Pattern based pointcuts make a set of points and the fact that they conform to a common pattern explicit; they also establish a binding between the set and the points. Property-based pointcuts, such as 'execution(public com.acme.*.*(..))' do the same for properties instead of patterns. Annotation-based pointcuts do this for annotations.

Named pointcut declarations establish a binding between an attribute (the pointcut name) and a (possibly singleton) set of points.

Advice can be used with any kind of pointcut to bind between the intermediate step that pointcut makes explicit and the implementation of an operation to execute at those points.

This characterization provides an interesting perspective on one difference between AspectJ and AspectWerkz [5]. In AspectJ, the body of an advice is a code block. But in AspectWerkz, advice has no code block; instead it is written as a method, with an annotation that contains the kind of advice and the pointcut.[4] In terms of our model, this means that in AspectWerkz, the advice construct binds to an operation, whereas in AspectJ it binds to an operation implementation. So AspectWerkz provides an extra binding step. In AspectJ the programmer can achieve the extra binding step simply by having the advice body call a procedure.

5 Usage Guidelines

Our model of how the different mechanisms serve to separate concerns suggests a way to approach the process of deciding which mechanism(s) to use in a given

[4] AspectJ 5 includes both alternatives.

situation. The following guidelines are organized around the binding steps in Figure 2 and work to help the programmer decide which path through the figure is most appropriate in a given situation. For each guideline, we discuss how it is validated from by the study described above.

Procedures

If an operation is needed at a given point, then using a procedure (call and declaration) serves to make the operation explicit and local, and to make the binding from the point to the operation explicit. This can improve comprehensibility of both the operation and the context, enable reuse of the operation in other contexts, and facilitate later change to the operation.

Comparing Straw-Man to the subsequent implementations, we see that the use of a procedure makes Refresh-Implementation explicit and local. Separating this concern explicitly makes its implementation more clear, and also clarifies the contexts where the operation is invoked (e.g. the setX method). The refresh procedure can easily be called from other points (reused). When Refresh-Implementation changes in the Double-Buffering and Pass-Changed-Object tasks the implementations that use the procedure fare better. None of this is a surprise; we are all familiar with these properties of using procedures. We are elaborating this here only to show how this set of guidelines encompasses the familiar case of procedures and to lay a foundation for discussion guidelines regarding annotations, pointcuts and advice.

Advice and Pointcuts

If an operation is needed at a given set of points then using advice and pointcuts serves to make the binding from the set to the operation explicit and local, which can improve comprehensibility and evolvability in some cases. In particular, consider using advice and pointcut rather than multiple procedure calls if: (i) more than a small number of points must invoke the operation, (ii) the binding between the points and the operation may be disabled or otherwise be context-sensitive, or, (iii) the calling protocol to the operation may change.

All the implementations that use advice and pointcuts (Annotation-Property and on) make the calling protocol to Display.refresh explicit and localized. So they support part iii of this guideline.

But in this regard it is worth looking carefully at the way the implementations that use advice and pointcuts enhance the capture of When-to-Refresh (WtR) and What-Constitutes-Change (WCC). Annotation-Call does not improve WtR or WCC over GOFP. Annotation-Property makes WtR explicit and local and makes WCC explicit but non-local. With Anonymous-Enumeration-Pointcut both concerns are local, but are once again implicit. In the named pointcut implementations both concerns are local and back to being explicit. Since all these implementations use advice and pointcuts of some form, this suggests an interaction between using advice and the form of the pointcut used in the advice, which leads to the next guideline.

Attributes – Named Pointcuts or Annotations

If a set of points used in an advice has a common attribute, then using a named pointcut or an annotation can make that common attribute explicit. Using named pointcuts makes the attribute explicit and local, annotations make it explicit and non-local. When using named attributes, choose a name that describes what is true about the points, rather than describing what a particular advice will do at those points.

This guideline is supported by the Annotation-Property and the two named pointcut implementations. What-Constitutes-Change is made explicit in all three of these implementations. It is made local in the two named pointcut implementations. In each case, the capture of When-to-Refresh also benefits, which is the link to the previous guideline.

As with procedures, the motivation to make the additional bindings and intermediate steps explicit using advice and named attributes comes from comprehensibility, reuse, evolution and other considerations. Comprehensibility is subjective, but to our eye, Annotation-Property and the two named pointcut implementations are the easiest to understand because they make all the steps leading up to a refresh clear. They clearly say “there is an explicit concept of display state change”, “here are points that constitute such changes”; and “call refresh at those points”. Straw-Man, GOFP and Annotation-Call make it clear that refresh is happening, but not why. Anonymous-Enumeration-Pointcut makes it clear that there is a general condition that causes refresh to happen, but without a pointcut name the abstraction of the condition is not clear.

In terms of reusability, because Annotation-Property and the two named pointcut implementations make the (d/D)isplayStateChange attribute explicit, they make it easy to reuse What-Constitutes-Change in the change task.

In terms of evolution, making the binding from the (d/D)isplayStateChange attribute to the refresh signaling behavior explicit makes the Disable-Refresh change task easy.

The Annotation-Call and Annotation-Property implementations demonstrate the importance of choosing good annotation names. In Annotation-Call the name of the annotation is such that it fails to introduce the intermediate step and make clear why refresh is happening. A named pointcut with a similar name would have similar problems.

Introducing additional attribute names does not always add value. When writing procedural code, most programmers are unlikely to define a new onePlus procedure for the expression ‘x + 1’. They could, but in this case the primitive expression is sufficiently clear that it is usually left in line. Named abstraction has to stop at some point, or else programs would never reach primitives.

The same is true for attributes. The pointcut ‘execution(public com.acme.*.*(..))’ is sufficiently clear that it usually does not warrant a named pointcut. On the other hand ‘execution(* Shape+.set*(..))’ probably does warrant the displayStateChange named pointcut.

Enumeration, Property, Pattern-Based Pointcuts and Annotations

The previous guidelines leave open the question of what mechanism to use to establish the binding between the individual point(s) and the actual set of points. The choices are enumeration-based pointcuts, name-pattern based pointcuts, property-based pointcuts or annotations.

Prefer enumeration-based pointcuts when: (i) it is difficult to write a stable property-based pointcut to capture the members and (ii) the set of points is relatively small.

Prefer property- or pattern-based pointcuts when: (i) it is possible to write one that is stable or (ii) the set of points is relatively large (more than ten).

Use annotations to mark points when three things are true: (i) it is difficult to write a stable property-based pointcut to capture the points, (ii) the name of the annotation is unlikely to change, and (iii) the meaning of the annotation is an inherent to the points, rather than a context-dependent aspect of the points only true in some configurations.

In addition, lean towards annotations when the property that defines inclusion in the set is an inherent property of the points, and lean towards other pointcuts when the binding from points to the set might change non-locally, or come into existence non-locally.

The implementations after GOFP provide some support for these guidelines, but the example is too small to fully support them.

The difference between how Named-Enumeration-Pointcut and Named-Pattern-Pointcut fare for Add-Unrelated-Method both shows the concern about pattern-based pointcuts, and also shows that using stable patterns can mitigate that concern.[5] For example 'Shape+.set*(*)' means methods defined on Shape or a subtype of Shape, for which the name begins with set, and that have a single argument. This pattern has good stability both because it is restricted to a small part of the type hierarchy, and because it is based on a well-established Java naming convention. By contrast, 'set*(..)' is less stable, it covers any type of object, and methods with any number of arguments.

Once again, the difference between Annotation-Call and Annotation-Property supports the importance of annotation names. As formulated above, the guideline is intended to reduce the likelihood that the name will need to change, and will make it

[5] Practicing AspectJ developer report that the restrictions that come from the use of name patterns often benefits their code. The patterns force them to regularize the rules they use for naming, and that helps with overall system comprehensibility. Nonetheless, this issue is motivating a variety of important research in more powerful pointcut languages, that make it possible to express pointcuts in terms of properties that are more accurate and robust than name patterns [24, 35].

more natural to reference the same annotation in other aspects or in compositions of pointcuts based on the annotation. For example DisplayStateChange may be reasonable as an annotation. But MakesRemoteCall may not be, because it may depend on a particular deployment configuration rather than always being true of a method.

While the guidelines for preferring property and pattern-based pointcuts when the number of points is large and it is possible to write such pointcuts are not supported by this study, they seem fairly straightforward, although it would be valuable to validate them, and all the other above guidelines, in a larger case study.

6 Related Work

There have been a number of characterizations of aspect-oriented programming (AOP) mechanisms: as a means for modularizing crosscutting concerns [16, 17], in terms of obliviousness and quantification [10], in terms of a common join point model framework [25] and others. By contrast, the focus of this paper is on analyzing the separation of concern properties of annotations, pointcuts and advice, and describing those as binding mechanisms similar to procedures.

The work described in [10] and [24] is closer to this paper in that they characterize AOP mechanisms as a new step in “introducing non-locality in our programs” [10], specifically as a means of binding points in the execution space [24]. But, they do not consider annotations. They also do not focus on the way in which the mechanisms compare for separating different kinds of concern or provide guidelines for choosing among the mechanisms.

The discussion by Lopes et al. [22] shares with this paper the view that pointcuts act as a kind of referencing mechanism. The focus in [22] is more on motivating and speculating about future “more naturalistic” referencing mechanisms that go beyond current pointcut mechanisms. On the contrary, our focus is on characterizing and assessing state-of-the-art mainstream pointcut mechanisms and especially on providing guidelines for using them.

Rinard et al. [29] propose a classification and an analysis system for AOP programs that classifies interactions between aspects and methods to identify potentially problematic interactions (e.g., caused by the aspect and the method both writing the same field), and guide the developer’s attention to the causes of such interactions. Hence, their focus is different than ours. They also do not discuss annotations, and only indirectly suggest usage guidelines. To the extent they do suggest guidelines there appear to be no conflicts between their work and ours.

Baldwin and Clark have developed a general framework for assessing the value of modularity in technical systems [2]. Sullivan et al. [32] show how this framework can be applied to software systems. Lopes and Bajracharya [23] went on to apply the framework to AOP systems. The Baldwin and Clark framework is more heavy-weight than ours, and seems more suitable for architectural decision making than what we discuss here. But again, there does not appear to be any inherent conflict between the analyses. One interesting next experiment would be to see how the guidelines we develop interact with the analyses and net option value framework used by these researchers.

Our guidelines are ‘bottom-up’ or in-situ in nature. They are focused on how a developer makes isolated decisions about what mechanism to use guided by design goals. By contrast, Jacobsen and Ng have proposed a methodology for designing systems in an aspect-oriented style [15]. Again, there appears to be no contradictions between our guidelines and their methodology.

The work presented in [26] also involves an assessment of pointcut mechanisms with respect to how well programs using them fare in presence of change, as compared to equivalent OO programs that use method calls only. That assessment does not consider annotations, and is primarily on assessing the need for pointcut mechanisms that refer to more dynamic properties of join points than possible today. The design and implementation of such pointcuts is the main focus of their paper.

7 Future Work

The analysis and guidelines in this paper are based on first-principles analysis with a single small example. Based on this, there are several attractive avenues for future work.

One next step would be large-scale validation of these guidelines. There are (at least) two dimensions of improvement. First, they could be validated against a larger sample of code developed by experts. While attractive, at present there do not appear to be large bodies of suitable open source code to work with, although this appears to be changing rapidly.

A second line of work would be to validate these guidelines in some form of user study in which programmers are asked to work with the guidelines in a controlled experiment.

As discussed in Section 6, it would also be interesting to develop a detailed account of how the guidelines we propose interact with classifications such as in [29], architectural analyses such as in [2], and design methodologies such as in [15].

8 Summary

Metadata annotations, pointcuts and advice are useful techniques for separating concerns in source code. To better understand and be able to work with these mechanisms, we propose a characterization in which each is seen as making a different kind of binding: annotations bind attributes to program points; pointcuts create bindings between sets of points and descriptions of those sets; named pointcuts bind attributes to sets of points; and advice bind the implementation of an operation to sets of points.

This characterization yields insight into how the mechanisms relate and suggests areas for improvement. It also yields guidelines for how to choose among the mechanisms in the course of programming with them. The guidelines can be phrased in terms of deciding which kind of binding is appropriate in a given situation or they can be formulated in more prescriptive terms that may be more appropriate in some contexts.

The model and guidelines proposed here provide a good basis for further research and near-term development. We expect improvements to the model and guidelines as the combined use of annotations, pointcuts and advice grows.

Acknowledgements

The author would like to thank Mik Kersten and Gail Murphy for fruitful discussions on the ideas developed in this paper. Thanks also go to Andrew Eisenberg, Mik Kersten, Gail Murphy, Kevin Sullivan and Maria Tkatchenko who provided detailed comments on earlier drafts of the paper.

This work is partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), IBM Canada Ltd. and the European Network of Excellence in Aspect-Oriented Software Development (AOSD-Europe)

References

1. The Server Side Symposium: AOP Expert Panel, 2004, http://www.theserverside.com/news/thread.tss?thread_id=30564.
2. Baldwin, C.Y. and Clark, K.B. Design Rules: The Power of Modularity. MIT Press, 2000.
3. Bergmans, L. and Aksit, M. Principles and Design Rationale of Composition Filters. in Filman, R.E., Elrad, T., Aksit, M. and Clarke, S. eds. Aspect-Oriented Software Development, Addison Wesley Professional, 2004, 63 - 95.
4. Bloch, J. A Metadata Facility for the Java Programming Language, 2004.
5. Boner, J., AspectWerkz http://aspectwerkz.codehaus.org/.
6. Bryant, A., Catton, A., Volder, K.D. and Murphy, G.C., Explicit programming. Aspect-Oriented Software Development, 2002, ACM Press, 10-18.
7. Cheatham, T.E., JR., The introduction of definitional facilities into higher level programming languages. (AFIPS) Fall Joint Computer Conference, 1966, Spartan Books, 623-673.
8. Coady, Y., Kiczales, G., Feeley, M. and Smolyn, G., Using AspectC to improve the modularity of path-specific customization in operating system code. Foundations of Software Engineering (FSE), 2001, ACM Press, 88 - 98.
9. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K. and Ossher, H. Discussing aspects of AOP. COMMUNICATIONS OF THE ACM, 44 (10). 33-38.
10. Filman, R.E., Elrad, T., Aksit, M. and Clarke, S. (eds.). Aspect-Oriented Software Development. Addison Wesley Professional, 2004.
11. Gradecki, J. and Lesiecki, N. Mastering AspectJ: Aspect-oriented Programming in Java. Wiley, Indianapolis, Ind., 2003.
12. Group, G., Hype Cycle for Application Development, 2004, http://www4.gartner.com/DisplayDocument?doc_cd=120914.
13. Hirschfeld, R. AspectS - Aspect-oriented programming with squeak. Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, 2591. 216-232.

14. Jacobson, I. and Ng, P.-W. Aspect-Oriented Software Development with Use Cases. Addison-Wesley, 2003.
15. Jacobson, I. and Ng, P.-W. Aspect-Oriented Software Development with Use Cases. Addison Wesley Professional, 2004.
16. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ. European Conference on Object-Oriented Programming (ECOOP), 2001, Springer, 327-355.
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J., Aspect-oriented programming. European Conference on Object-Oriented Programming (ECOOP), 1997, 220-242.
18. Kiczales, G. and Mezini, M., Aspect-Oriented Programming and Modular Reasoning. ACM International Conference on Software Engineering, 2005 (to appear).
19. Krishnamurthi, S., Fisler, K. and Greenberg, M. Verifying aspect advice modularly. Foundations of Software Engineering (FSE). 137 - 146.
20. Laddad, R. AspectJ in action: practical aspect-oriented programming. Manning, Greenwich, CT, 2003.
21. Liberty, J. Programming C#. O'Reilly, Sebastopol, CA, 2003.
22. Lopes, C., Dourish, P., Lorenz, D. and Lieberherr, K. Beyond AOP: Toward naturalistic programming. ACM SIGPLAN NOTICES, 38 (12). 34-43.
23. Lopes, C.V. and Bajracharya, S., An Analysis of Modularity in Aspect-Oriented Design. Aspect-Oriented Software Development (AOSD'05), 2005 (to appear).
24. Masuhara, H. and Kawauchi, K., Dataflow Pointcut in Aspect-Oriented Programming. Asian Symposium on Programming Languages and Systems (APLAS), 2003, 105--121.
25. Masuhara, H. and Kiczales, G., Modeling crosscutting in aspect-oriented mechanisms. European Conference on Object-Oriented Programming (ECOOP), 2003, Springer, 2-28.
26. Ostermann, K., Mezini, M. and Bockisch, C., Expressive Pointcuts for Increased Modularity. In Proc. of European Conference on Object-Oriented Programming (ECOOP), 2005, Springer.
27. Project, A., AJDT Demonstration, 2004, http://eclipse.org/ajdt/demos/.
28. Rajan, H. and Sullivan, K., Eos: instance-level aspects for integrated system design. Foundations of Software Engineering (FSE), 2003, ACM Press, 297 - 306.
29. Rinard, M., Salcianu, A. and Suhabe, B., A Classification System and Analysis for Aspect-Oriented Programs. Foundations of Software Engineering (FSE), 2004, ACM Press, 147 - 158.
30. Schutter, K.D., What does aspect-oriented programming mean to Cobol? Aspect-Oriented Software Development, 2005, ACM Press, (to appear).
31. Spinczyk, O., Gal, A. and Schröder-Preikschat, W., AspectC++: an aspect-oriented extension to the C++ programming language. Fortieth International Confernece on Tools Pacific: Objects for internet, mobile and embedded applications, 2002, Australian Computer Society, 53 - 60.
32. Sullivan, K.J., Griswold, W.G., Cai, Y. and Hallen, B., The structure and value of modularity in software design. Foundations of Software Engineering, 2001, ACM Press, 99 - 108.
33. Teitelman, W. PILOT: A Step Toward Man-Computer Symbiosis Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1966.
34. Walker, D., Zdancewic, S. and Ligatti, J., A theory of aspects. International Conference on Functional Programming, 2003, ACM Press, 127 - 139.
35. Walker, R. and Viggers., K., Implementing protocols via declarative event patterns. ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE-12), 2004.