Aspect-Oriented Programming and Modular Reasoning

Gregor Kiczales
University of British Columbia
2366 Main Mall
Vancouver, BC, Canada
gregork@acm.org

Mira Mezini
Technische Universität Darmstadt
Hochschulstrasse 10
D-64289 Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

ABSTRACT
Aspects cut new interfaces through the primary decomposition of a system. This implies that in the presence of aspects, the complete interface of a module can only be determined once the complete configuration of modules in the system is known. While this may seem anti-modular, it is an inherent property of crosscutting concerns, and using aspect-oriented programming enables modular reasoning in the presence of such concerns.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, modules, packages.*

General Terms
Languages, Theory.

Keywords
Aspect-oriented programming, modularity, modular reasoning.

1. INTRODUCTION

Aspect-oriented programming (AOP) has been proposed as a mechanism that enables the modular implementation of crosscutting concerns [22]. It has proven popular [15, 23, 26] because it makes it possible for developers to write modular code for concerns such as synchronization [11, 14], error handling [29], persistence [37], certain design patterns [18] etc. Being able to code aspects cleanly is helping developers to think in terms of aspects at earlier stages of the lifecycle [16, 20, 34, 35].

An important dialogue has been raised about the full implications of AOP for modularity and modular reasoning [1, 8, 9]. This paper contributes an improved understanding of interfaces in the presence of AOP to that dialogue. We introduce the concept of *aspect-aware interfaces*, and show that a module's aspect-aware interface is not completely determined by the module, but rather depends in part on the other modules in the system – aspects cut new interfaces through the primary module structure. Using aspect-aware interfaces, we provide an argument based on first-principles that AOP supports modular reasoning in the presence of crosscutting concerns.

We show that some global knowledge is required as a precursor to modular reasoning with AOP. But, we also show that in the presence of crosscutting concerns – implemented with or without AOP – global knowledge is always required and that AOP makes this requirement more explicit and enables modular reasoning once the initial global analysis is complete.

The paper is structured as follows: Section 2 provides definitions of modularity and modular reasoning. Section 3 presents the example used in the paper. Section 4 presents the key properties of aspect-aware interfaces. Section 5 analyzes the modularity of the non-AOP and AOP implementations of the example. Section 6 outlines open research issues. Related work is discussed as appropriate throughout the paper (this crosscutting concern is not modularized).

2. DEFINITIONS

We say the code that implements a concern is *modular* if:

- it is textually local,
- there is a well-defined interface that describes how it interacts with the rest of the system,
- the interface is an abstraction of the implementation, in that it is possible to make material changes to the implementation without violating the interface,
- an automatic mechanism enforces that every module satisfies its own interface and respects the interface of all other modules, and
- the module can be automatically composed – by a compiler, loader, linker etc. – in various configurations with other modules to produce a complete system.

Modular reasoning means being able to make decisions about a module while looking only at its implementation, its interface and the interfaces of modules referenced in its implementation or interface. For example, the type-correctness of a method can be judged by looking at its implementation, its signature (i.e. interface), and the types (i.e. interfaces) of any other code called by the method.

Not all decisions are amendable to modular reasoning. Many program refactorings require more information for example [13]. *Expanded modular reasoning* means also consulting the implementations of referenced modules, and *global reasoning* means having to examine all the modules in the system or sub-system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-58113-963-2/05/0005...$5.00.

```
interface Shape {
  public moveBy(int dx, int dy);
}

class Point implements Shape {
  int x, y; //intentionally package public

  public int getX() { return x; }
  public int getY() { return y; }

  public void setX(int x) {
    this.x = x;
    Display.update();
  }
  public void setY(int y) {
    this.y = y;
    Display.update();
  }

  public void moveBy(int dx, int dy) {
    x += dx; y += dy;
    Display.udpate();
  }
}

class Line implements Shape {
  private Point p1, p2;

  public Point getP1() { return p1; }
  public Point getP2() { return p2; }

  public void moveBy(int dx, int dy) {
    p1.x += dx; p1.y += dy;
    p2.x += dx; p2.y += dy;
    Display.update
  }
}
```

```
interface Shape {
  public moveBy(int dx, int dy);
}

class Point implements Shape {
  int x, y; //intentionally package public

  public int getX() { return x; }
  public int getY() { return y; }

  public void setX(int x) {
    this.x = x;

  }
  public void setY(int y) {
    this.y = y;

  }

  public void moveBy(int dx, int dy) {
    x += dx; y += dy;  }

  }
}

class Line implements Shape {
  private Point p1, p2;

  public Point getP1() { return p1; }
  public Point getP2() { return p2; }

  public void moveBy(int dx, int dy) {
    p1.x += dx; p1.y += dy;
    p2.x += dx; p2.y += dy;

  }
}

aspect UpdateSignaling {
  pointcut change():
    execution(void Point.setX(int))
    || execution(void Point.setY(int))
    || execution(void Shape+.moveBy(int, int));

  after() returning: change() {
    Display.update();
  }
}
```

Figure 1. The Java and AspectJ implementations of the shape classes with display update signaling.

3. A RUNNING EXAMPLE

This section introduces the example that will be used throughout the paper.

The example involves a simple set of graphical shape classes, including Point and Line. Imagine that other shapes like Circle and Rectangle are also included. Also imagine a Display class that implements the drawing surface on which the shapes are displayed. This class has a static update method.[1] To save space these are not shown. The instances have state that determines their appearance on the display, e.g. `Point` objects have `x` and `y` coordinates. Finally there is code to signal the `Display` to update whenever a shape changes.

Figure 1 shows two implementations of this example: an ordinary object-oriented implementation in Java, and an aspect-oriented implementation in AspectJ.[2] The key difference between the implementations is that in the AOP version the update signaling behavior is implemented in an aspect, whereas in the non-AOP

[1] This code would be improved if update was an instance method of Display. But this requires using an additional feature of AspectJ, inter-type declarations, so for simplicity we use the less elegant approach.

[2] Examples in this paper are written in Java and the pointcut-and-advice part of AspectJ. For simplicity we focus on execution join points, execution pointcuts, and after returning advice. We also ignore examples where aspects advise aspects.

code it is scattered across the methods of Point and Line (and their siblings).

In the `UpdateSignaling` aspect, the first member declares a *pointcut* named `change()`. This pointcut identifies certain *join points* in the program's execution, specifically the execution of the `setX` and `setY` methods in `Point`, as well as `moveBy` methods defined on any sub-type of `Shape`.

The second member declares *after returning advice* that says that after returning from executing the join points identified by `change()`, the `Display.update()` static method should be called to signal the display to update.[3]

4. INTERFACES IN AOP SYSTEMS

This section presents the key properties of interfaces in AOP, also called aspect-aware interfaces. We start with one possible way of writing aspect-aware interfaces for the running example. This serves to give an intuition of how AOP changes the traditional notion of interface. Because our goal is to identify the general properties of aspect-aware interfaces, we then discuss some possible variations on that formulation. More significant variations and open issues are discussed in Section 6.

Figure 2 shows the aspect-aware extension of simple Java-style statically value-typed interfaces for the `Shape`, `Point` and `Line` classes. Much of the interface is traditional—it describes the type hierarchy, the public fields and methods defined on each class and gives result and argument types for each method. (For simplicity we ignore exceptions, constructors and non-public members, as well as the distinction between interfaces and classes.)

The interfaces in Figure 2 also describe how the aspects and non-aspects crosscut. The notation

```
: UpdateSignaling – after returning
                    UpdateSignaling.move()
```

following some methods says that: (i) the `UpdateSignaling` aspect has after returning advice that affects execution of the method, and (ii) the pointcut that advice refers to is `move()`, also defined in `UpdateSignaling`.

The interface of the `UpdateSignaling` aspect also has an entry for the advice, which includes inverse information about what methods it affects.

4.1 Interface Depends on Deployment

A key property of AOP, made explicit by aspect-aware interfaces, is that the interface of a module depends on the complete system into which it is deployed. Because aspects contribute to the interface of classes, and classes contribute to the interface of aspects, we cannot know the complete interfaces of modules in a system until we have a complete system configuration and run through the modules collecting aspects and analyzing the crosscutting.

This brings into focus what some authors have identified as a controversial property of AOP [1, 8, 9]. The concern is as follows: Prior to AOP modules had a "black-box" property – the interface of a module was defined in a single place, either part of the nodule or directly referred to by the module. So looking at the module was sufficient to know its interface. As a result, modular reasoning was possible with knowledge of only the module and the interfaces of the modules to which it explicitly refers. No knowledge of the rest of the system was required. This assumption does not hold for AOP.

These authors have generally sought to restrict the power of AOP in order to preserve existing black-box reasoning mechanisms.

In contrast, our goal is to show that the full power of AOP is compatible with modular reasoning, if we are willing to change some of our existing reasoning mechanisms. With aspect-aware interfaces we require a global analysis of the deployment configuration to determine module interfaces. But once that is done, modular reasoning is possible even for crosscutting concerns, as we will show in Section 5.

This phenomenon of interface depending on system configuration is similar to what is seen in other fields of systems engineering. In mechanical systems, key properties of a component with respect to composition depend on the whole system. Conductivity and corrosion resistance matter when a component is used in some systems but not others. Dynamic analysis requires knowing the whole system. Heat transfer behaves similarly. Recent research suggests that "compartmental systems" are not the only suitable modularities for understanding biological systems [24].

These aspects that force the analysis to consider the whole system – dynamics, corrosion, conductivity, chemical propagation etc. – are crosscutting concerns. They cut through the primary modularity boundaries and in doing so they act to define new module structures with which to analyze the system.

We observe an important difference between AOP and these other systems. In the physical systems, composition leads to new *crosscutting modules*. In mechanics, the modules involved in dynamic analysis are different than those in static analysis. The spring-damper-mass model of a simple system has a very different structure than the simple finite-element static model, even though they both describe the same system. The modules of the dynamic analysis may not even come into being until the system is composed, and the two sets of modules crosscut each other with respect to the physical artifact.

In AOP, the situation is different. Composition leads to new *crosscutting interfaces*, but the modules remain the same. From the perspective of traditional software interfaces, the idea that composition can lead to new interfaces may seem radical, but at least our situation is simpler than for some other engineers. We get new interfaces, but not new modules. And, once the composition (deployment configuration) is known, the interfaces can be identified, and, as we will show in Section 5, modular reasoning is possible.

[3] The semantics of this AspectJ code is that the advice body executes at the join points matched by the pointcut. The semantics are not to generate a new program in which the bodies of the advice have been woven into the methods, that is simply one possible implementation strategy. The AspectJ language design contemplates that weaving can happen at any time, even as late as in the interpreter, and implementations have been developed that weave at a range of times [3, 19, 21]. The aspect-aware interfaces are similar; they describe the semantics of the source AspectJ code, not its possible compilation into intermediate code.

```
Shape
  void moveBy(int, int) : UpdateSignaling – after returning UpdateSignaling.move();

Point implements Shape
  int x;
  int y;
  int getX();
  int getY();
  void setX(int)        : UpdateSignaling – after returning UpdateSignaling.move();
  void setY(int)        : UpdateSignaling – after returning UpdateSignaling.move();
  void moveBy(int, int) : UpdateSignaling – after returning UpdateSignaling.move();

Line implements Shape
  void moveBy(int, int) : UpdateSignaling – after returning UpdateSignaling.move();

UpdateSignaling
  after returning: UpdateSignaling.move(): Point.setX(int), Point.setY(int),
                                           Point.moveBy(int, int), Line.moveBy(int, int);
```

Figure 2 Interfaces in the AOP code.

4.2 Formulation of Aspect-Aware Interfaces

This section discusses some of the design decisions underlying the formulation of aspect-aware interfaces shown above. Our goal here is to identify the key properties of aspect-aware interfaces not to argue that the above formulation is ideal. A great deal of work remains to be done in refining aspect-aware interfaces, some of which is discussed in Section 6.

Intensional and extensional descriptions. One decision was whether to include the pointcut involved in an advice declaration in the interface. To be concrete, we could have written the following instead of what we have in Figure 2:

```
Line extends Shape
  void moveBy(int, int) :
     UpdateSignaling – after returning;
```

We include the pointcut because we feel it is key to understanding the interface abstraction. An AOP programmer thinks about advice being applicable at a group of join points with a common property. The emphasis is on the property more than the specific points, and the pointcut expresses that property.

The pointcut can be seen as the intensional definition of the interface. The marked set of methods is the extensional definition. For example, note that the pointcut is what the programmer should study when considering changes to the implementation of the class. Seeing the pointcut

```
execution(void Point.set*(*))
```

is different than seeing the pointcut

```
execution(void Point.setX(int))
|| execution(void Point.setY(int))
```

even if, as in this case, the same join points are identified.

Pointcut abstraction or reduction. Another decision was whether the interface should include the pointcut as it appears in the advice declaration or include its reduction (recursive inlining of the named pointcuts). We chose the former, because it reflects abstractions from the aspect. But clearly there are times when the programmer will want to see a partial or complete reduction of the pointcut.

We see this as analogous to a programmer sometimes wanting to see just a type name in an interface, and other times wanting to see more information about the type. As such, it seems amenable to being addressed as a tool issue.

Including advice kind. We also decided to include the kind of advice (before, after etc.) rather than just indicating the applicability of advice, without saying its kind. We feel that including the kind adds to the descriptive power of the interface, without overly restricting the implementation of the aspect. In practice, advice bodies change about as often as method bodies. But changing an advice from before to after is less common and more significant. Also, because advice declarations are not named, this helps the programmer know which advice is being referred to.[4]

Expressing extensional definition. A more complex decision had to do with deciding what methods to list as being affected by an advice. The answer we chose was to list those methods for which executing the body of the method might run the advice. In the subset of AspectJ we are considering (execution join points, execution pointcuts, after returning advice) this is clear enough. But once we allow call, get and set join points the issue becomes less clear. Should a method be listed as affected because it includes a call join point that is advised? Should a method be listed as affected because calls to it are advised? This is clearly an area for future work. One initial answer is to list any method for which the body lexically includes the *shadow* of an advised join point.[5]

Rather than marking each affected method, we could have marked just the enclosing classes with all the aspects that affect any of its

[4] In a system like AspectWerkz [4], where advice declarations associate a pointcut with a named method rather than an anonymous code block, the name of the method might also be included.

[5] The shadow of a dynamic join point is a code structure (expression, statement or block) that statically corresponds to execution of the dynamic join point. The shadow of a method execution join point is a method body; the shadow of a method call is a call expression etc.

Table 1. Analysis of modularity for non-AOP and AOP implementations of shape package.

		localized	interface	abstraction	enforced	composable
non AOP	display updating	no	n/a	n/a	n/a	n/a
	Point, Line	medium(1)	medium(2)	medium(2)	yes	yes
AOP	UpdateSignaling	high	high(3)	high	yes(5)	yes
	Point, Line	high(4)	high(3)(4)	high	yes(5)	yes

(1) Point and Line classes are contaminated with scattered and tangled display updating behavior.
(2) Except that the tangled display updating behavior is not a documented part of the interface.
(3) Using aspect-aware interfaces.
(4) Enhanced because display updating behavior is no longer tangled.
(5) Standard Java type checking extended to advice and advice parameters. In addition, assurance that advice is called when it should be and at no other times

methods. This would be a lower-granularity version of the interfaces we have here. Given this coarse-grained back link to the aspects, expanded modular reasoning could then be used to construct the more complete information in the interfaces we describe.

We chose not do this because it connotes the aspect applies to the whole class, which is often not the case. It is also less useful, because programmers will almost always have to go to the aspect implementation to find out exactly what methods are affected. And it fails to capture the crosscutting structure that is such an important part of AOP code.

5. MODULARITY ANALYSIS

We now analyze the AOP and non-AOP implementations. First we address the modularity criteria from Section 2; this is summarized in Table 1. Then we use a simple change scenario to analyze modular reasoning.

5.1 The Non-AOP Implementation

In the non-AOP code, the implementation of the display updating behavior fails to satisfy our modularity criteria. First, it is not localized. Since the additional modularity criteria build on locality and each other, they also fail: because there is no localized unit, there is nothing for there to be an interface to, and without an interface, we cannot ask whether it is an abstraction of the implementation. Similarly, the implementation cannot be composed independently; there is no automatic mechanism for producing a version of the shape classes without change signaling behavior.

The `Point` and `Line` classes meet our modularity criteria, but in a somewhat compromised form:

- They are textually local, but that boundary also includes the code for signaling the display to update.
- They have clearly defined interfaces, but those interfaces fail to say anything about the included display update signaling behavior.
- The interface is an abstraction of the implementation. The internal details of the classes could change in meaningful ways without changing the interface. The coordinates of a `Point` could be stored differently for example.
- The interfaces are enforced in that the Java type checker, loader and virtual machine ensure type safety.
- They can be composed automatically. The Java loader can load these with other classes in different configurations.

5.2 The AOP Implementation

In the AOP code, the `UpdateSignaling` aspect meets our criteria for a modular implementation of the display updating behavior: The `Point` and `Line` classes also meet our criteria, somewhat better than in the non-AOP implementation.

- Each is textually local. Locality is improved over the non-AOP implementation because the update signaling behavior is not tangled into the `Point` and `Line` classes.
- Each has a clear interface as shown in Figure 2. The interfaces are now a more accurate reflection of their behavior – update signaling is reflected in the interfaces as arising from the interaction between the aspects and the classes.
- In each case the interface is an abstraction of the implementation, in that there is room for material variation in how each is implemented. For example, a helper method could be called to do the signaling, or the signaling could be logged.
- The interfaces are enforced. Type checking works in the usual way, and the advice is called when it should be and at no other times. The advice calling enforcement is somewhat trivial – as with polymorphic dispatch a single advice declaration both declares the interface and defines the implementation.
- Each can be composed automatically with other modules – this is what the AspectJ weaver does.[6] For example, we can

[6] Since release 1.2, weaving can happen at compile-time, post compile-time on jar files, or at load time.

automatically produce a configuration that includes the shape classes but not the UpdateSignaling aspect.

5.3 Informal Reasoning about Change

In this section we consider a simple change scenario, and compare reasoning with traditional interfaces about the non-AOP code against reasoning with aspect-aware interfaces about the AOP code.

The example presented in Section 3 has a deliberately introduced weakness – the `x` and `y` fields of the `Point` class are public, not private. We now consider the scenario where a programmer decides to change the fields to being private. When doing this they must ensure the whole system continues to work as before.

We now walk through the reasoning and changes to the code that would most likely ensue for both the non-AOP and AOP code. The process starts out following the same path for both implementations. We nonetheless discuss the whole process, both to make the example realistic, and to stress the critical role modular reasoning can play as a sub-part of a larger, not entirely modular, reasoning process.

The programmer begins by asking what the implications of changing the fields are. Making the `x` and `y` fields private entails a change to the interface of the class. So reasoning shifts outside the class (outside the module), to clients of the `Point` interface, or more specifically clients of the `x` and `y` fields of the `Point` interface.

Unfortunately, global reasoning, in the form of a simple global search, is required to find all such clients. This is a typical consequence of interface changes. In this case, the programmer's attention next focuses on the `moveBy` method of the `Line` class:[7]

Reasoning in the non-AOP implementation. In the non-AOP implementation, the `moveBy` method of `Line` is originally:

```
public void moveBy(int dx, int dy) {
  p1.x += dx; p1.y += dy;
  p2.x += dx; p2.y += dy;
  Display.update();
}
```

To conform to the new interface of `Point`, this code must be revised to call accessor methods rather than access the fields directly. A straightforward revision of the code would be:

```
public void moveBy(int dx, int dy) {
  p1.setX(p1.getX() + dx);
  p1.setY(p1.getY() + dy);
  p2.setX(p2.getX() + dx);
  p2.setY(p2.getY() + dy);
  Display.update();
}
```

The programmer must now decide whether this change is reasonable. The answer is that it is not – it violates an important, but not explicit, invariant of the original code, which is that there should be a single display update for each top-level change to the state of a shape. In the revised code, a call to `moveBy` on a line object would produce 5 display updates. What we want to assess is what reasoning is required to reach this conclusion.

To discover the problem with this potential change, the programmer needs two pieces of information: a description of the invariant and enough of the structure of update signaling to infer that the invariant would be violated by the change.

Nothing in the implementation or interface of `Line` is likely to describe the invariant. But because of the explicit call to Display.update(), the programmer might choose to study the code for the `Display` class. We assume, optimistically, that the documentation for the update method includes a description of the one update per top-level change invariant.

At this point expanded modular reasoning with one step has led the programmer from a proposed change to the `moveBy` method to the invariant.

But the programmer still does not have enough information to be sure the proposed change is not problematic. They must also discover that the `setX` and `setY` methods call update, or, more generally, discover the existing *structure* of update signaling. This requires at least further expanded modular reasoning – to just find the calls from `setX` and setY; or global reasoning – to find all calls to update and discover the complete structure of display update signaling.

Once the programmer concludes, through expanded modular or global reasoning that the change to `moveBy` is incorrect, they are in a somewhat difficult situation. One solution is to add special non update-signaling setter methods to `Point`, and call those from `moveBy`. (This is when the programmer has the 'aha' realization of why they were package public in the first place, and perhaps gives up and leaves them that way.)

Summarizing the reasoning process in the non-AOP implementation, starting at the proposed changed to `Line's moveBy` method:

- One-step expanded modular reasoning may lead to documentation of the key invariant.
- Global reasoning is required to discover the complete structure of update signaling.
- Expanded modular reasoning discovers enough of the updates to handle this specific case.

Reasoning in the AOP Implementation. In the AOP code the change process proceeds along the same course as in the non-AOP code up to the point of considering the possible change to the `moveBy` method of Line. In the AOP code, the straightforward revision of `moveBy` is:

```
public void moveBy(int dx, int dy) {
  p1.setX(p1.getX() + dx);
  p1.setY(p1.getY() + dy);
  p2.setX(p2.getX() + dx);
  p2.setY(p2.getY() + dy);
}
```

As in the non-AOP case, this code is incorrect. It violates the update invariant in exactly the same way.

If we assume, with similar optimism, that the invariant is documented in UpdateSignaling then one-step expanded modular reasoning leads the programmer from the `moveBy` method to the invariant. If we are less optimistic, and only assume that the

[7] The programmer might feel that private fields should not be accessed directly even within a class, and so focus first on the moveBy method of Point, and then come to the moveBy method of Line later.

invariant is documented in `Display`, then two-step expanded modular reasoning is required.

The interface of `UpdateSignaling` includes the complete structure of what method executions will signal updates. So modular reasoning alone provides the programmer with this information.

Once the programmer understands that the simple change to `moveBy` is invalid, the situation is much simpler in the AOP case. In AspectJ and similar AOP languages, the proper fix is to use the `cflowbelow` primitive pointcut. Using this, the advice would be edited to be:

```
after() returning: change()
                   && !cflowbelow(change()) {
  Display.update();
}
```

The revised pointcut means only top-level changes are advised, and is read as "any join point matching change, unless that join point is in the control flow below a join point matching change".

Summarizing the reasoning process in the AOP implementation, starting at the proposed change to `Line's moveBy` method:

- One- or two-step expanded modular reasoning may lead to documentation of the key invariant.
- Modular reasoning leads to the complete structure of update signaling.
- A simple local change to the `UpdatingSignaling` aspect solves the problem, and results in the invariant being explicit. enforced and clearly reflected in the interfaces.

Comparison. In the first step of the process the two implementations perform similarly – global reasoning is required to find all the references to the x and y fields. Neither AOP nor traditional technologies prevent this.

With respect to documenting and allowing the programmer to discover the invariant, the two original implementations fare similarly. Under optimistic assumptions about the invariant being documented, the non-AOP implementation requires one-step expanded modular reasoning to discover the documentation. The AOP implementation requires one- or two-step expanded modular reasoning.

With respect to discovering the structure of update signaling the two implementations perform significantly differently. The non-AOP implementation requires expanded modular reasoning to discover the minimal structure required to reason about the change. It requires global reasoning to discover the complete structure. The AOP implementation requires only modular reasoning to discover the complete structure. In a more complex example the difference would be more dramatic.

Fundamentally, display update signaling is a crosscutting concern. With AOP, its interface cuts through the classes, and the structure of that interface is captured declaratively, and the actual implementation is modularized. Without AOP, the structure is implicit and the actual implementation is not modular.

The main cost of AOP, with respect to classical modular reasoning is that the interface of a module is context dependent. We must know the set of modules with which a given module will be deployed to know its interface. Without AOP, when reasoning about a change to a module we must ask whether the interface changes. With AOP, we must ask whether the interface for each deployment configuration changes. (Section 6.4 outlines an idea that can limit how many configurations are explicitly consulted.)

The main benefit of AOP is that once we accept the cost, we get the traditional benefits of modularity and modular reasoning for crosscutting concerns.

Without AOP, complete configuration information is not needed to determine a module's interface. But in such a world, modular reasoning fails for crosscutting concerns like display update signaling. A global search is required to discover the key invariant.

Our conclusion is that for crosscutting concerns programmers inherently have to pay the main cost of AOP – they have to know something about the total deployment configuration in order to do the global reasoning required to reason about crosscutting concerns. But using AOP, they get modular reasoning benefits back, whereas not using AOP they do not.

5.4 Automatic Reasoning

We have argued that AOP implies a new kind of interface, but that once those interface are computed, the power of modular reasoning is improved. In this section we point out three existence proofs of this claim.

Since version 1.2 AspectJ has supported incremental compilation and weaving for interactive development [19]. This works by having the weaver maintain a list of the aspects and classes in a deployment configuration, as well as a *weaving plan* data structure similar to the interfaces we describe (the weaving plan has more detailed information). When the weaver is called it first checks whether the weaving plan has changed. If not, only the code that has changed is re-compiled and re-woven. This is limited modular reasoning in the face of unchanging interfaces.

In [25] Krishnamurthi et. al. describe a similar scheme for incremental verification of AspectJ code.

The open modules work described in [1] provides a formal justification for our modular reasoning claim. The theorem developed in this work implies that once a module's aspect-aware interface is computed, we can prove functional correctness properties, and safely make changes to a module without affecting the rest of the program.

6. OPEN ISSUES

The key property of aspect-aware interfaces is that knowledge of the complete system configuration is required to compute how interfaces are cut through the primary decomposition. But the formulation and use of these interfaces can be extended in a variety of ways.

6.1 Other forms of AOP

A first task is to expand our concept of aspect-aware interfaces and the analysis here to full AspectJ, including the other kinds of dynamic join points, as well as inter-type declarations (aka introductions). A simpler task is to cover similar systems like Caesar [33] and AspectWerkz [4]. We expect that the generalized model of AOP presented in [32] will provide a basis for this.

A more interesting challenge is reconciling aspect-aware interfaces with aspect-oriented systems like MDSOC [38]. At first glance, our observation that aspect-aware interfaces show that in AOP the interfaces, but not the implementations crosscut, (Section 4.1) seems at odds with the conceptual account of MDSOC, in which code is explicitly copied into different modules (usually in different system configurations). But our account of aspect interfaces might enable a re-characterization of MDSOC that preserves the nice symmetrical properties, without having the code copying semantics.

6.2 Other Interface Technologies

The interfaces we describe are the aspect-aware version of standard Java interfaces. They support simple static value typing. But more sophisticated interface technologies have been developed for object-oriented and other languages. These include higher-order value typing like generic types, [5] state typing [10], behavioral specification [6, 27, 30] and others. One area of research is to explore the aspect-aware equivalent of these other kinds of interfaces. Our belief is that the basic idea of aspect-aware interfaces should carry-over to these interface styles.

Existing work adapting behavioral interfaces to AspectJ reinforces this belief [39]. But an experiment is needed to be sure. Part of this work would involve exploring what issues are better specified as behavioral specifications what issues are better addressed directly in pointcuts.

6.3 More expressive pointcuts

In Section 4.1 we said that the pointcuts represent the abstraction or intensional specification of the interface. More work is needed to increase the expressive power and abstraction of pointcuts.

The most common concern is that use of wildcarding risks unintended matches as the program evolves. This concern is valid concern, although the intentionally limited power of AspectJ pattern matching, together with the tool support for editing AspectJ code tends to mitigate this problem in practice.

Support for use of annotations as in C# [28] and Java JSR-175 [2] may be of some help, although the use of annotations violates the "obliviousness" property of AOP pointcuts, and requires scattering the annotations, and so has potential scaling and evolution problems.

Of more interest to us are mechanisms that allow the programmer to directly express the true intended semantics of the pointcut. The use of `cflowbelow` shows the power of making pointcuts more expressive this way. It makes it possible to express the structural invariant explicitly, and in a checked and enforced form.

We expect that it will be possible to do better than this. In the case of the change pointcut, what the programmer is thinking is that these are the methods that change state that affects the display. But what the programmer is doing in the pointcut is identifying those methods by name or name pattern. We would like to write a pointcut that directly says "the methods that change the state that affects the display". Computing the actual methods (the extensional description) would involve some sort of conservative control and data flow analysis. Several efforts are already underway to develop more expressive pointcuts [7, 12, 17, 31].

6.4 Interface Constraints

A number of researchers have expressed concern that aspects can advise classes without the class's "consent". They argue that classes should be able to prevent advice from affecting their methods. Most proposals allow classes to explicitly restrict aspects, or require classes to publish pointcuts, or even require that classes import explicitly import aspects [1, 8, 9]. All of these inherently limit the "obliviousness" property of AOP.

The identification of aspect-aware interfaces suggests a new possibility. Instead of associating aspect constraints directly with classes or packages, they could be associated with system configurations. System architects could define these constraints, and any aspects included in the configuration would have to respect them. This would make it possible to have different constraints for different configurations, and would reflect that reasoning about aspect interfaces requires prior knowledge of the configuration. It would not place any inherent limits on the obliviousness of classes with respect to aspects. A given configuration could have no constraints.

An additional issue for enforcement we see is that the way in which a join point is identified for advice is at least as important as what join points are identified. Consider advice using these two different pointcuts:

```
get(int Point.x) || get(int Point.y)
get(* Shape+.*)
```

With respect to the class `Point`, these two pointcuts match the same join points. But with respect to evolution and modularity, the two are quite different. The former hard codes exact names of private fields of the class. The latter identifies all the fields, regardless of their name. We believe that for many advice the latter is more comfortable than the former; the latter will evolve better. A means for enforcing aspect restrictions should be able to account for differences in how join points are identified.

Several researchers have noted that the nature of the advice is critical for enforcement [1, 8, 9]. The intuition is that advice that simply "observes" is less problematic than advice that has effect. Unfortunately, categorization of whether advice observes or effects appears difficult. What it means to observe depends on context – it is different on an application server than in real-time control code for example. In [36] Rinard et. al. describe an initial empirical analysis of advice behavior that we hope will prove helpful in better understanding this issue.

7. SUMMARY

AOP enables modular implementation of crosscutting concerns, and modular reasoning in the presence of crosscutting concerns. But it requires a change in how module interfaces are specified. With AOP interfaces are extended as aspects cut through the primary module structure. So a module's interface cannot be fully determined without a complete system configuration.

But crosscutting concerns inherently require global knowledge to support reasoning. Using AOP, programmers get modular reasoning benefits for crosscutting concerns whereas without AOP they do not.

ACKNOWLEDGEMENTS

We thank Jonathan Aldrich and Curtis Clifton for discussions about these topics, and for comments on the paper itself. Klaus Ostermann, Gail Murphy and Maria Tkatchenko also provided comments on drafts of the paper.

REFERENCES

[1] Aldrich, J., Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming, Carnegie Mellon Technical Report CMU-ISRI-04-108, 2004 (Earlier version appeared in Workshop on Foundations of Aspect-Oriented Languages.).

[2] Bloch, J. A Metadata Facility for the Java Programming Language, 2002.

[3] Bockisch, C., Haupt, M., Mezini, M. and Ostermann, K., Virtual Machine Support for Dynamic Join Points. International Conference on Aspect-oriented Software Development (AOSD), 2004, ACM Press, 83-92.

[4] Boner, J., AspectWerkz http://aspectwerkz.codehaus.org/.

[5] Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P., Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), 1998, 183-200.

[6] Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M. and Poll, E., An overview of JML tools and applications. Workshop on Formal Methods for Industrial Critical Systems (FMICS), 2003.

[7] Chiba, S. and Nakagawa, K., Josh: An Open AspectJ-like Language. International Conference on Aspect-oriented Software Development (AOSD), 2004, ACM Press, 102-111.

[8] Clifton, C. and Leavens, G., Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy, Iowa State University Technical Report, TR 03-15,

[9] Clifton, C. and Leavens, G., Observers and assistants: A proposal for modular aspect-oriented reasoning. Workshop on Foundations of Aspect-Oriented Languages (FOAL), 2002.

[10] DeLine, R. and Fähndrich, M., Typestates for Objects. European Conference on Object-Oriented Programming (ECOOP), 2004.

[11] Deng, X., Dwyer, M., Hatcliff, J. and Mizuno, M., SyncGen: An aspect-oriented framework for synchronization. Int'l Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2004, 158-162.

[12] Eichberg, M., Mezini, M. and Ostermann, K., First-Class Pointcuts as Queries. Asian Symposium on Programming Languages and Systems (APLAS), 2004, Springer Lecture Notes on Computer Science, to appear.

[13] Fowler, M. and Beck, K. Refactoring: improving the design of existing code. Addison-Wesley, Reading, MA, 1999.

[14] Furfaro, A., Nigro, L. and Pupo, F. Multimedia synchronization based on aspect oriented programming. Microprocessors and Microsystems, 8 (2). 47-56.

[15] Gradecki, J. and Lesiecki, N. Mastering AspectJ: Aspect-oriented Programming in Java. Wiley, Indianapolis, Ind., 2003.

[16] Grundy, J., Aspect-Oriented Requirements Engineering for Component-based Software Systems. International Symposium on Requirements Engineering, 1999, IEEE Computer Society Press, 84-91.

[17] Gybels, K. and Brichau, J., Arranging Language Features for More Robust Pattern--Based Crosscuts. International Conference on Aspect-Oriented Software Development (AOSD), 2003, ACM Press, 60-69.

[18] Hannemann, J. and Kiczales, G., Design pattern implementation in Java and AspectJ. Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), 2002, 161-173.

[19] Hilsdale, E. and Hugunin, J., Advice Weaving in AspectJ. International Conference on Aspect-Oriented Software Development (AOSD), 2004, ACM Press, 26-35.

[20] Jacobson, I. and Ng, P.-W. Aspect-Oriented Software Development with Use Cases. Addison-Wesley, 2003.

[21] 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.

[22] 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.

[23] Kiselev, I. Aspect-oriented programming using AspectJ. Sams, Indianapolis, Ind., 2003.

[24] Krakauer, D.C. Robustness in Biological Systems: a provisional taxonomy. in Complex Systems Science in Biomedicine, Kluwer, 2004.

[25] Krishnamurthi, S., Fisler, K. and Greenberg, M. Verifying aspect advice modularly. Foundations of Software Engineering (FSE). 137 - 146.

[26] Laddad, R. AspectJ in action: practical aspect-oriented programming. Manning, Greenwich, CT, 2003.

[27] Leavens, G., Cheon, Y., Clifton, C., Ruby, C. and Cok, D. How the design of JML accommodates both runtime assertion checking and formal verification. FORMAL METHODS FOR COMPONENTS AND OBJECTS, 2852. 262-284.

[28] Liberty, J. Programming C#. O'Reilly, Sebastopol, CA, 2003.

[29] Lippert, M. and Lopes, C.V., A Study on Exception Detection and Handling Using Aspect-Oriented Programming. International Conference on Software Engineering, 2002, ACM Press, 418-427.

[30] Liskov, B.H. and Wing, J.M. A Behavioral Notion of Subtyping. Transactions on Programming Languages and Systems (TOPLAS).

[31] Masuhara, H. and Kawauchi, K., Dataflow Pointcut in Aspect-Oriented Programming. Asian Symposium on Programming Languages and Systems (APLAS), 2003, 105--121.

[32] Masuhara, H. and Kiczales, G., Modeling crosscutting in aspect-oriented mechanisms. European Conference on Object-Oriented Programming (ECOOP), 2003, Springer, 2-28.

[33] Mezini, A.M. and Ostermann, A.K., Conquering aspects with Caesar. International Conference on Aspect-Oriented Software Development (AOSD), 2003, ACM Press, 90-100.

[34] Rashid, A., Moreira, A. and Araujo, J. Modularisation and composition of aspectual requirements International Conference on Aspect-oriented Software Development (AOSD), ACM Press, 2003, 11-20.

[35] Rashid, A., Sawyer, P., Moreira, A. and Araujo, J. Early Aspects: A Model for Aspect-Oriented Requirements Engineering International Conference on Requirements Engineering, IEEE Computer Society Press, 2002, 199--202.

[36] 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.

[37] Soares, S., Laureano, E. and Borba, P., Implementing distribution and persistence aspects with AspectJ. Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), 2002, 174-190.

[38] Tarr, P., Ossher, H., Harrison, W. and Sutton, S.M., N degrees of separation: multi-dimensional separation of concerns. International Conference on Software Engineering (ICSE), 1999, IEEE Computer Society Press, 107-119.

[39] Zhao, J. and Rinard, M., Pipa: A behavioral interface specification language for AspectJ. Fundamental Approaches to Software Engineering (FASE), 2003, Springer, 150-165.