Modular Software Design with Crosscutting Interfaces*

William G. Griswold[λ] Kevin Sullivan[φ] Yuanyuan Song[φ] Macneil Shonle[λ]
Nishit Tewari[φ] Yuanfang Cai[φ] Hridesh Rajan[φ]

[φ]Computer Science
University of Virginia
Charlottesville, VA 22903
{sullivan,ys8a,yc7a,nt6x,hr2j}@cs.virginia.edu

[λ]Computer Science & Engineering
UC San Diego
La Jolla, CA 92093-0114
{wgg,mshonle}@cs.ucsd.edu

Abstract

Aspect-oriented programming languages such as AspectJ offer new mechanisms for decomposing systems into modules and composing modules into systems. This paper introduces *crosscut programming interfaces (XPIs)* as a practical approach to improving the modular designs of programs written using AspectJ-style AOP. It does not limit existing aspect-oriented mechanisms or require new ones.

Categories and subject descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques—modules and interfaces; D.3.3 [Programming Languages]: Language Constructs and Features; D.2.11 [Software Engineering]: Software Architectures—information hiding; D.2.7 [Software Engineering]; Distribution, Maintenance, and Enhancement.

General terms: Design, Languages.

Keywords: aspect-oriented, preconditions, postconditions.

1 Introduction

Aspect-oriented programming languages such as AspectJ [1] offer new mechanisms and possibilities for decomposing systems into modules and composing modules into systems. AspectJ-style design is based on the use of aspect modules that support quantified advising of dynamic points in program execution: namely the invocation of aspect-defined advice methods at *join points* matched by *pointcut descriptors*, or PCDs. In practice, PCDs are written directly in terms of lexical properties of advised code.

*This research supported in part by NSF CISE grants FCA-0429947 and FCA-0429786.

The contribution of this paper is the presentation of a practical approach to improve the modular designs of programs written using AspectJ-style AOP. Our approach is based on the use of what we call *crosscut programming interfaces*, or *XPIs*, to decouple aspects from the complex and changeable details of the code that they advise [2]. Without limiting the possibilities for AO advising or requiring new programming languages or mechanisms, our approach appears to better modularize aspects and advised code, allowing for their separate and parallel evolution, and producing a better alignment between programs and designs.

1.1 Problem

Our approach emerged from an experiment using AOP to improve the design of HyperCast, a 300-class, 50 KLOC Java system for multicast and other communications in self-organizing overlay networks [3]. HyperCast's logging concern and client-facing event notification behaviors were implemented by scattered code fragments, making enhancement of their limited behaviors difficult. Using AspectJ aspects to improve the design seemed a natural idea.

The oblivious design was obtained by assuming that had the developers been able to ignore crosscutting concerns, they would have written the same code, just leaving out the code for the crosscutting concerns. Aspects then advise the code in precisely the places where scattered fragments appear in the original design.

We started by using aspects to modularize a logging concern and a few concerns implemented with an implicit invocation mechanism. We changed HyperCast as little as possible, just encapsulating scattered code in aspects, resulting in a design of the base functionality that was essentially the same as before, absent the crosscutting concerns. The AOP notion of *obliviousness* presents such a

design as an ideal: designers should not have to consider crosscutting concerns or aspects that implement them [4]. In comparison to the original object-oriented design, our economic analysis showed an increased *net options value of modularity* in this straightforward AO design [2]. However, we encountered costly problems that largely offset the net value improvement.

First, aspect design and evolution were complicated. The designers had to inspect all the code to identify relevant join points. Since join points were not exposed in a consistent way, complex PCD expressions and advice were needed. Innocuous changes to code could change the matched join points, violating assumptions made by the advising aspects.

Second, the resulting class and aspect abstractions did not reflect the conceptual design. In the designers' minds, HyperCast's protocols are implemented by abstract state machines. Event notifications expose key state machine transitions so that aspect-like clients can react to them. Our traditionally derived AO design had no separate, explicit representation of the state machines as interfaces. The PCDs of the aspects were candidates, but they fell short; each aspect defined its own PCDs for its own purpose.

One way to make the state machine abstraction clearer in the design would have been to rename the pointcut descriptors used in the aspects after the state machine concepts they were implicitly using. However, as reconceptualized, renamed, and used, these pointcuts would have no longer belonged to their respective aspects, *per se*. Yet there was no appropriate place for these pointcuts in a HyperCast class, either, as the advised join points were not centralized in a class or meaningful subclass hierarchy. The pointcuts identified true crosscutting behaviors.

1.2 Approach

The above line of reasoning exposed important crosscutting concerns distinct from those modularized in aspects. The "new" concerns were HyperCast's core protocols and other crosscutting abstract behaviors. These behaviors needed to be exposed through interfaces, against which aspects could be implemented. Benefits would be obtained in abstraction (e.g., the program structure reflecting the key abstraction in the designers' minds and conversations), and modularity (e.g., parallel development and modular evolution).

We thus repeated our experiment with HyperCast, using *Design Rules* as [5] conceptual interface constructs, which stabilized and syntactically regularized the relevant join points in the code, and also constrained the aspects to not corrupt the state machine [2]. The resulting design better mirrors the conceptual design and provides far better separation of aspect code from the details of the advised code. Our net option value analysis showed greater value of modularity over the first design, while the complexity of the aspects and overall coupling was significantly reduced.

The primary contribution of this paper is to realize these conceptual crosscutting interfaces as syntactic crosscutting interfaces in AspectJ, with semantics defined by weakest precondition invariants. A crosscut programming interface, or XPI, has several elements:

- a name;
- a static scope over which it abstracts join points;
- one or more abstracted sets of join points, each expressed as:
  - a concrete, named PCD;
  - preconditions that must be satisfied at each join point where advice can run, called a *provides* clause; and
  - postconditions that must be satisfied after advice may have run, called a *requires* clause.
- a set of constraints, or *design rules*, prescribing how code must be written to expose all and only the specified points in program execution through the given PCDs.

In AspectJ these elements are declared in an aspect. Checkable invariants can be checked with a separate pluggable aspect.

An XPI, like an API, abstracts changeable and complex design decisions and operates as a decoupling contract between providers and users. Unlike an API, an XPI abstracts a crosscutting concern rather than a localized concern. In the case of AspectJ-style design, an XPI abstracts advised join points. To paraphrase Parnas [6, p. 1058], XPIs should modularize crosscutting design decisions that are complex or likely to change. In turn, the implementor implements an XPI, not by providing code to simulate a specified behavior, but by shaping code to expose specified behaviors through join points matching designated PCDs. With respect to obliviousness, a designer does not necessarily anticipate particular aspects, such as logging, but does anticipate the need for domain-relevant abstractions that facilitate development and evolution of aspects expressible in terms of provided XPIs. The design method that we have found to work is to identify domain abstractions that are not adequately captured in the class design,

and to express them as XPIs. Thus, no explicit reference is made to future aspects in their design, although the usefulness of the XPIs can be checked against anticipated aspects.

In the following, we use a detailed comparative example to show how to apply the XPI method and what benefits can accrue from it.

2 Two Designs for a Figure Editor

In this section we present two designs for the classic figure editor example [1]. The first is a traditional AOP modularization; the second, a design with XPIs. We evaluate the designs in terms of how well they manifest fundamental design concerns, abstract irrelevant details, and accommodate change.

2.1 A Traditional AO Design

Consider a simple figure editing tool for editing drawings comprising points and lines (figure elements), where each figure element is depicted on a display, and where the display always reflects the current states of figure elements. The FigureElement class provides an interface for the concrete subclasses, Point and Line. The Display class manages the display and provides a method, update(), to display the current states of all figure elements. The specification requires a call to update() whenever the abstract state of a figure element changes.

The figure editing example has been used to introduce the ideas of crosscutting concerns, scattering and tangling, and how AOP addresses these issues. The crosscutting concern in this case is the policy that states *when the abstract state of a FigureElement changes, the Display must be updated.* Implementing this policy in an OO style leads to scattered update() calls throughout FigureElement subclass implementations, and the tangling of these calls into code concerned with FigureElement updating.

The Observer pattern could be used to remove the explicit calls, but it still requires that event-related code be scattered and tangled in the FigureElement code and elsewhere. The approach has been criticized on the grounds that it requires that the author of the FigureElement code anticipate extensions. AOP provides an alternative that avoids the need for such preparation in support of display updating. The DisplayUpdate aspect, shown in Figure 1, satisfies the update specification.

We implemented this aspect in the manner popularized in the research and practitioner literature on AOP. We studied the FigureElement code to find points where

```
public aspect DisplayUpdate {
  pointcut FigureElementStateTransition():
    call (* FigureElement+.set*(..))
    || call (* FigureElement+.moveBy(..));

  after(FigureElement f):
    FigureElementStateTransition(f) && target(f) {
    Display.update(f);
  }
}
```

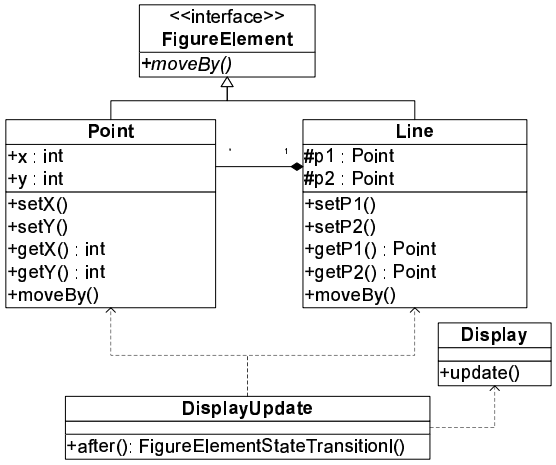

Figure 1: A traditional display updating aspect and the resulting aspect-oriented design of figure editor.

changes in FigureElement abstract state occur. We generalized and described this set of points in the form of a PCD, FigureElementStateTransition(), which captures calls to mutator methods of Line and Point and calls to moveBy(), which moves a figure element by a certain offset. Figure 1 presents a UML model of this design. As is typical in straightforward AOP, the DisplayUpdate aspect depends on implementation details of the Point and Line classes. A more sophisticated AO design could remove these dependences.

There are several reasons to be concerned about such a design. First, the Point and Line implementations had to be written before the aspect could be written, limiting the available parallelism in development. Second, the aspect implementor had to study the Point and Line *implementations* in order to be able to write the aspect (pointcut descriptors) correctly. The lack of an abstraction layer between the aspect and the advised code adds to the cognitive load on the aspect implementor. The aspect lets the FigureElement writer ignore display updating, but the aspect writer cannot ignore low-level details of FigureElement. Third, the correctness of the aspect depends on unstable details of the Point and Line implementations. The design is thus subject to be compromised by apparently innocuous changes in them.

2.2 An AO Design with XPIs

By employing XPIs, the designer seeks to insulate aspects from the details of the code they advise, while constraining that code to expose certain behaviors in specified ways. In the process, important but previously submerged crosscutting concerns become manifest as XPIs in the program. In the figure editing case, an XPI will separate the DisplayUpdate aspect from FigureElement details. Our XPI reifies the concept *a transition has occurred in the abstract state of a FigureElement.* It provides simple PCDs by which aspects can advise all such actions without having to depend on the underlying source code, while constraining the system implementor to ensure that all abstract state changes (and only such state changes) are implemented in a way that matches the PCDs.

The syntactic part of the XPI exposes two named PCDs, point() and topLevelPoint() . The PCD signature (name and parameters) constitute the abstract interface; the part of the PCD that matches points in the code is part of the hidden implementation of the XPI (see Figure 2). It is only here that dependences on details of the underlying code arise.

We document the semantics informally in the following prose. The point() PCD exposes all FigureElement state transitions.[1] This abstraction is implemented, in a sense, by the pattern that matches calls to FigureElement mutators. The system designer is constrained to ensure that the PCD pattern matches all and only such FigureElement mutator calls and that state transitions occur only as a result of such calls. The topLevelPoint() PCD exposes all and only changes to the states of compound FigureElement objects (such as Line) but not changes to their component elements (namely Point).

The semantics of XPIs can include behavioral constraints on aspects. In our example, we require that no advisor of this XPI cause a side-effect on a FigureElement object. This constraint in effect prohibits advice from calling FigureElement mutators either directly or indirectly.

Like APIs, XPIs enable a degree of contract checking [7]. When included in the program's build, the aspect shown below the XPI in Figure 2 constrains developers to modify the internal state of a FigureElement from only within the FigureElement mutators. To a degree, it also ensures that the aspects using the XFigureElementChange XPI are not able to modify the abstract state of any FigureElement. The aspect cannot, however, verify the programmer's adherence to the naming requirements.

Figure 3 presents a DisplayUpdate aspect using this

```
public aspect XFigureElementChange {
/*
 The purpose of the point() PCD is to expose all and
 only FigureElement abstract state transitions. We
 require that all such transitions be implemented by
 calls to FigureElement mutators with names that match
 the PCDs of this XPI, and we assume that any such call
 causes such a state transition. Advisors of this XPI
 may not change the state of any FigureElement
 directly or indirectly. The topLevelPoint() PCD
 exposes all and only "top level" transitions in the
 abstract states of compound FigureElement objects.
*/
  public pointcut point(FigureElement fe):
    target(fe)
    && (call(void FigureElement+.set*(..))
        || call(void FigureElement+.moveBy(..))
        || call(FigureElement+.new(..)));

  public pointcut topLevelPoint(FigureElement fe):
    point(fe) && !cflowbelow(point(FigureElement));

  protected pointcut staticscope():
    within(FigureElement+);

  protected pointcut staticmethodscope():
    withincode (void FigureElement+.set*(..))
    || withincode(void FigureElement+.moveBy(..))
    || withincode (FigureElement+.new(..));
}

/*
 Checks the contracts for the XFigureElementChange XPI.
 */
public aspect FigureElementChangeContract {
  /*
   PROVIDES: XPI matches all calls and only
   calls to FigureElement mutators
   */
  declare error:
    (!XFigureElementChange.staticmethodscope()
     && set(int FigureElement+.*)):
    "Contract violation: must set FigureElement"
    + " field inside setter method!";

  /*
   REQUIRES: advisers of this XPI must not change
   the abstract state of any FigureElement object
   */
  private pointcut advisingXPI():
    within(XFigureElementChange+) && adviceexecution();

  before(): cflow(advisingXPI())
    && XFigureElementChange.point(FigureElement) {
    ErrorHandling.signalFatal("Contract violation:"
      + " advisor of XFigureElementChange cannot"
      + " change FigureElement instances");
  }
}
```

Figure 2: The XFigureElementChange XPI and separate contract-checking aspect.

[1] We use the generic PCD name, *point()*, not to be confused with the Point class, as an analog of a generic "run" method name; the semantics are already suggested by the name of the containing XPI.

```
public aspect DisplayUpdate {
  after():
    XFigureElementChange.topLevelPoint(FigureElement) {
    updateDisplay();
  }

  public void updateDisplay() {
    Display.update();
  }
}
```

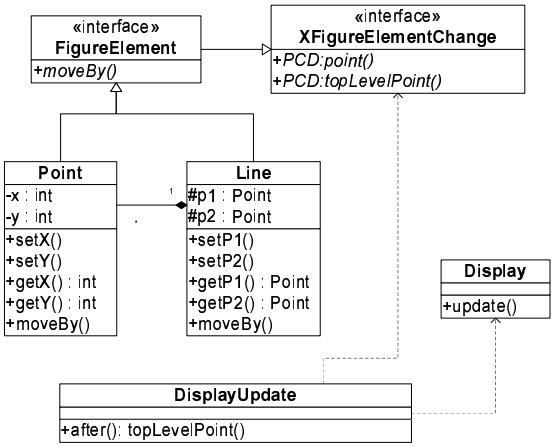

Figure 3: Display updating aspect using an XPI, and the resulting aspect-oriented design.

XPI and the resulting UML model. The aspect now depends only on the abstract, public PCD signatures of XFigureElementChange, not on implementation details of the Point and Line classes. Classes Point and Line contribute to implementing the XFigureElementChange XPI by ensuring that method names match the given PCDs if and only if they have the specified change semantics.

3 Analysis of the Designs

We now compare the designs in terms of abstraction and evolvability. As Kiczales and Mezini did [8], we first change public data members to private, forcing updates to occur through advisable method calls. We then extend FigureElement to include Color. The next section shows how adding a classic "non-functional" aspect, property enforcement, is facilitated by XPIs.

3.1 Data Member Access

In our original design, the coordinates in the Point class were public, permitting this implementation of Line.moveBy():

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

Making the fields private drives the Line.moveBy() designer to change to the implementation:

```
public void moveBy(int dx, int dy) {
  p1.moveBy(dx, dy);
  p2.moveBy(dx, dy);
}
```

Now consider the DisplayUpdate aspect implemented without the XPI. When Line.moveBy() is invoked, the advice is invoked three times: once for the call to Line.moveBy() and once for each call to Point.moveBy() in the body of Line.moveBy(). The assumption by the aspect about Line's otherwise-hidden implementation was broken by the apparently innocuous change [9].

The XPI approach avoids such problems by establishing an interface in the form of design rules, where aspects can assume that the rules are followed, and code within the scope of the XPI is required to conform to its terms, and vice versa. It is important that XPIs have both syntax, in the form of convenient abstract PCDs, and semantics. Our XPI specifies that the PCD must match a join point if and only if it indicates a change in the abstract state of a FigureElement. Under this XPI, DisplayUpdate uses the provided convenient PCD (and promises not to inject changes into FigureElement), and the implementor of Line would implement Line.moveBy() so that it's join point is captured by the PCD.

3.2 Adding Color to Figure Elements

The second change is behavioral, adding Color as a Line attribute with getter and setter methods, with the requirement that all observers of figure element update when a Line's color changes. In the non-XPI approach, one of two undesirable scenarios are required to ensure that the display updates properly. One, the Color implementer must be aware of the DisplayUpdating aspect and its PCD implementation to figure out how to name the Color setter method so the PCD will match it. Two, the aspect implementor must change the DisplayUpdating PCD to match whatever choice the Color implementor makes. With more aspects involved, these scenarios become less appealing.

In the XPI case, the Color implementor need only be aware of the figure element state-change XPI and its constraint that a state can be changed only by a method named moveBy or a name starting with set. The presence of an XPI thus guides the implementor in choosing names for methods and in making other decisions that can influence

PCD matching. In this case, the implementor must name the method something like setColor, rather than changeColor; and merely doing so exposes color changes as abstract state changes through the XPI. To our knowledge, no prior work clearly guides programmers to design code for ease of advising.

4 Extending the New Design

Adding a property and its implementation to a system is an important issue. We explore it in the context of XPIs by adding a feature that maintains a geometric invariant in the figure editor: Lines may not be degenerate. That is, the two points that define a line cannot have identical coordinates. Enforcing this invariant requires that no Line be degenerate when first created and that no change to a Point in a Line makes it degenerate. This is an instance of the more general problem of maintaining invariants for compound structures under changes to their respective parts.

Note that invariant enforcement essentially changes the originally specified behavior of Points by conditioning the effects of a Point mutator on a Point's participation in a Line. Such a change could require broad changes in the software's implementation; the advantage of using an aspect is that the code changes can be localized to the aspect, even if their effects are not. With this observation in mind, we now argue that the use of XPIs, while not a panacea, can improve a designer's ability to express and use abstractions that both manage these complex effects and are central to designers' thinking.

We assume that the designer has decided to use an aspect module to implement the invariant enforcement. Since an appropriate XPI to write the aspect against does not already exist, we need to determine the domain abstraction for decoupling development of the aspect from development of the normal case, and then write that XPI. The behavioral abstraction we need is *change to a Point that is part of a Line*. Given this, the aspect can then implement the policy *prevent changes to Points in Lines that would create any degeneracies.*

The precise invariant we seek for the given design is that a Line cannot have two end Points at the same coordinate. Modifying a Line by calling method Line.setP1(Point) or Line.setP2(Point) can violate this invariant. So can directly modifying the coordinates of a Point that belongs to a Line, without direct reference to the Line. However, a key concept absent from the original system is the relation between Points and Lines. For instance, there is no field in Point that stores a containing Line. A subtlety is that some Points are part of a Line, and some are not.[2]

```
public aspect PointLineRelation {
  private Line Point.parent;

  public boolean Point.partOfLine() {
    return parent != null;
  }

  public Line Point.getParent() {
    return parent;
  }

  /*
   When a Line's Point is possibly set,
   reestablish the parent of the Line's Points.
   */
  private pointcut changePoint(Line l):
    target(l)
    && XFigureElementChange.point(FigureElement);

  before(Line l): changePoint(l) {
    l.getP1().parent = null;
    l.getP2().parent = null;
  }

  after(Line l): changePoint(l) {
    l.getP1().parent = l;
    l.getP2().parent = l;
  }
}
```

Figure 4: The PointLineRelation aspect.

Thus, the first part of our solution creates a representation of a new Point-Line relation. We use an aspect to introduce a *parent* field into the Point, to record the Line to which a point belongs, if any (see Figure 4). The aspect uses the XFigureElementChange XPI, updating the parent field as appropriate when a Line is created or one of its Points replaced. Note that although this aspect updates the parents of Points, it does not violate the XFigureElementChange Requires contract because the parent is part of the hidden state of FigureElements. In keeping with this XPI, no setParent method is introduced, calls to which would inappropriately result in updating the Display.

Figure 5 presents the XPI and resulting design. The XPointInLineChange XPI exposes three events on the end-points of a line: change in X coordinate, change in Y coordinate, and change in both coordinates.

Having written this XPI, it is now straightforward to write an aspect for invariant enforcement (not shown): using *around* advice, it advises changes in Points that are in Lines and allows them to occur only if they preserve the invariant. The XPI abstracts changes to Points in Lines. The aspect separately abstracts the invariant and enforcement policy. This kind of separation is at the heart of our interface-oriented approach to AO design for improved modularity and abstraction. It permits reuse of the XPI

[2] And in a real system, a point may be a part of many lines.

```
/*
 The X() PCD exposes changes to the x coordinate
 of any point that belongs to a line (similarly
 for Y() and XY().
 */
public aspect XPointInLineChange {
  public pointcut X(Point p, int x):
    call(void Point+.setX(int))
    && target(p) && args(x) && if(p.partOfLine());

  public pointcut Y(Point p, int y):
    call(void Point+.setY(int))
    && target(p) && args(y) && if(p.partOfLine());

  public pointcut XY(Point p, int dx, int dy):
    call(void Point+.moveBy(int,int))
    && target(p) && args(dx, dy)
    && if(p.partOfLine());
}
```

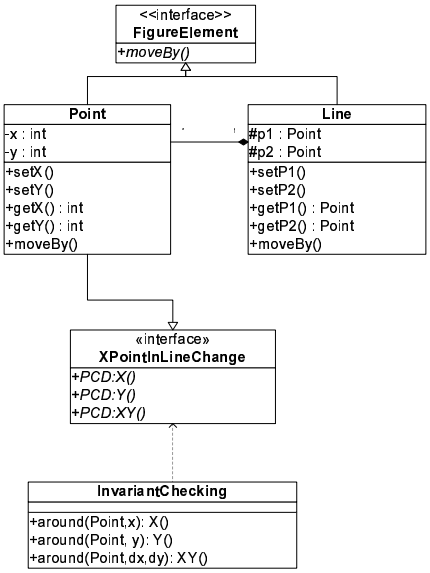

Figure 5: The XPointInLineChange XPI and the resulting design.

for implementing other aspects, and decouples those aspects from possible changes to the ways that Points and Lines may be modified.

5 Related Work

Most work that aims to improve program modularity under the use of AO mechanisms focuses on language models and expressiveness, rather than on software design methodology. Two recent developments that address design more directly are relevant here.

Join point scoping. Larochelle et al. proposed a PCD-based mechanism for hiding a crosscutting set of join points, thus preventing their being advised by aspects [10]. Dantas and Walker's AspectML provides advice access controls to the parameters of a function definition, hence modifying the join point signature of calls on the function [11]. The XPI approach does not provide a hiding mechanism, rather it specifies the exposure of given abstract execution phenomena. Combining these approaches might produce an interesting point in the space of design methods and supporting mechanisms.

Aldrich, among others, has proposed language constructs—and by implication a design method—for module-based join point interfaces. Open Modules expose only PCD-selected join points on private state [9]. The Open Modules system provides for the exposure of join points such that a module state that is intended to be hidden cannot be advised. Simply, a module has to declare a pointcut in order to export join points on its private state. Thus, it permits the evolution of a module implementation without requiring rework of aspects. Because the resulting interface is limited to crosscutting the module's implementation, it would be at best awkward to capture the crosscutting concepts found in our HyperCast case study [2].

Aspect-aware interfaces. Kiczales and Mezini recognized the need to program against crosscutting interfaces. They defined a notion of *aspect-aware interfaces* (AAIs), in which aspects extend the interfaces of modules they advise [8]. Specifically, dependences of aspects on join points are computed for a system and are shown as annotations on the explicit interfaces of advised code.

Revealing such dependences is an enabler of modular reasoning and change. A programmer can see how join points are being advised and avoid making changes that invalidate those uses. Prior to the emergence of stable modular interfaces (for example in XP-style development [12]), AAIs can serve as a valuable substitute—they inform, even if they do not decouple and abstract. Likewise, the cross-references provided by AAIs could prove useful in guiding refactoring activities, perhaps resulting in XPIs.

Yet, AAIs do not clearly manifest conceptual design concerns. There is no textually distinct interface construct, but instead a set of annotations. There is no construct for attaching contracts or programming against. Also, support for modularity is limited. The display of existing dependences between existing code and PCDs cannot tell developers how to shape new code to correctly expose behaviors to existing PCDs, nor how to write new PCDs to capture the desired behaviors of existing code.

6 Conclusion

The XPI approach decouples aspect code from the unstable details of advised code without compromising the expressiveness of existing AO languages or requiring new ones. By extending well-understood notions of module interfaces to crosscutting interfaces, it provides a principled alternative to the concept of oblivious design. In our discussions with best-practice AO programmers, we have found that some indeed design and develop in stylized ways that are consistent with the XPI approach. It thus has the potential to ground, regularize, and disseminate best software engineering practices using the new mechanisms provided by AO programming languages.

Our experience to date with XPIs is limited to two systems, HyperCast [2] and figure element. We expect that IDE support could aid programmers by showing the scope of an XPI's applicability. Being non-hierarchical, XPIs could overlap, but we have not seen this possibly complex interaction. We have yet to investigate the promise of XPIs for AO languages with different mechanisms than AspectJ's.

References

[1] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP 2001)*, pages 327–353, June 2001.

[2] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE 2005*, page To Appear, 2005.

[3] Jorg Liebeherr and Tyler K. Beam. Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Networked Group Communication*, pages 72–89, 1999.

[4] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, 2005.

[5] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity*. MIT Press, Cambridge, MA, 2000.

[6] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[7] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[8] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on software engineering*, 2005.

[9] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *2005 European Conference on Object-Oriented Programming (ECOOP'05)*, page To Appear, July 2005.

[10] David Larochelle, Karl Scheidt, Kevin Sullivan, Yuan Wei, Joel Winstead, and Anthony Wood. Join point encapsulation. In *In Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT) at AOSD 2003*, March 2003.

[11] Daniel S. Dantas and David Walker. Aspects, information hiding and modularity. Technical Report TR-696-04, Princeton University, November 2003.

[12] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999.