Modular Software Design with Crosscutting Interfaces

William G. Griswold and Macneil Shonle, University of California, San Diego
Kevin Sullivan, Yuanyuan Song, Nishit Tewari, and Yuanfang Cai, University of Virginia
Hridesh Rajan, Iowa State University

Aspect-oriented programming (AOP) languages such as AspectJ offer new mechanisms and possibilities for decomposing systems into modules and composing modules into systems. The key mechanism in AspectJ is the advising of crosscutting sets of join points. An aspect module uses a pointcut descriptor (PCD) to declaratively specify sets of points in program executions (join points) where anonymous methods (advice) should run. An advice can run before, after, or around the specified join points. For example, the PCD in the DisplayUpdate aspect in figure 1a specifies all calls to methods in the FigureElement class hierarchy named moveBy or whose names start with set. The advice updates a graphical display after any such call completes.

We’ve devised a practical design approach that can significantly improve the modularity of programs written using AspectJ-style AOP. Our approach employs crosscut programming interfaces, or XPIs. XPIs are explicit, abstract interfaces that decouple aspects from details of advised code. Without limiting the possibilities for AO advising or requiring new programming languages or mechanisms, our approach better modularizes aspects and advised code. It allows for their separate and parallel evolution and produces a better correspondence between programs and designs.

The problem

Our approach emerged from an experiment using common AOP methods to improve the design of HyperCast, a 300-class, 50,000-LOC Java system for multicast overlay networks. The common approach for developing aspects is to write PCDs directly against the implementations of the code to be advised. Some AOP methodologists even argue that designers should be able to write programs without knowing aspect modules’ actual or potential integration, a goal called obliviousness. This idea has found currency in the practitioner-oriented press.

We found that such approaches led to programs that were unnecessarily hard to develop, understand, and change. First, our designers had to inspect all the code to identify the relevant join points for the PCDs to encompass. Second, these join points weren’t exposed consistently, so we needed complex PCDs and advice...
bodies to effect the desired advising. Moreover, apparently innocuous changes or extensions to the code base could then change the matched join points, violating assumptions the aspects made. Also, the resulting class and aspect abstractions didn’t reflect the underlying conceptual design adequately.

HyperCast’s protocols are state machines. We needed aspects to advise state transitions. Although the aspects did modularize policy decisions on how to respond to transitions, the state machines themselves weren’t exposed as explicit, design-level abstractions.

After some thought, we realized that a second category of crosscutting concerns existed, ingrained into HyperCast and distinct from those modularized in aspects—namely, HyperCast’s core protocols and other crosscutting abstract behaviors. We needed to expose these behaviors through interfaces against which we could write aspects. Benefits would include improvements to both abstraction (for example, the program would reflect key abstractions in the designers’ minds and conversations) and modularity (for example, parallel development, modular evolution, and modular reasoning).

To develop and test this idea, we repeated our experiment using abstract interfaces, called XPIs, that expose PCDs and impose contracts and design rules. We found that aspects were easier to develop, aspect code was separated better from the details of advised code, and the overall conceptual design was clearer.

Unlike our earlier, more theoretical work, in this article we show how to realize XPIs as syntactic constructs in AspectJ, with weakest precondition invariants defining the semantics.

An XPI has four elements: the XPI’s name, a scope over which the XPI abstracts join points, one or more sets of abstract join points, and a partial implementation. We express each abstract set of join points as:

- a PCD signature declaring a name and exposed parameters, and
- a semantic specification stating preconditions that must be satisfied at each point where an advice can run (called a provides clause) and postconditions that must be satisfied after an advice runs (called a requires clause).

The partial implementation comprises, for each set of abstract join points,

- a join-point pattern that matches the corresponding concrete join points;
- a before, an after, or an around designator; and
- a corresponding set of constraints (design rules).

The constraints prescribe how code must be written to ensure that all and only the desired points in program execution match the given pattern. The rest of the XPI implementation is in the code’s conformance to the stated design rules.

In AspectJ, these elements are declared in a stylized aspect. (In AspectJ, before, after, or around designators are associated not with PCDs per se but only with advice constructs that use PCDs.) Some invariants can be checked with separate pluggable aspects.

An XPI, like an API, abstracts changeable
Most research on improving program modularity through aspect-oriented mechanisms focuses on language models and expressiveness rather than software design methodology. Two recent developments that address design more directly are relevant here.

**Join-point scoping**

David Larochelle and his colleagues proposed a mechanism based on a pointcut descriptor (PCD) for hiding a crosscutting set of join points, thus preventing aspects from advising them. Daniel Dantas and David Walker’s AspectML provides advice access controls to a function definition’s parameters, hence modifying the join-point signature of calls on the function. Our crosscut programming interface (XPI) approach doesn’t provide a hiding mechanism; rather, it specifies the exposure of given abstract execution phenomena. Combining such approaches might produce interesting support mechanisms for software design.

Jonathan 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. It enables the exposure of join points such that a module state that’s intended to be hidden can’t be advised. Simply, a module must declare a pointcut to export join points on its private state. Thus, Open Modules let a module implementation evolve without reworking aspects. However, the resulting interface is limited to crosscutting the module’s implementation. Capturing the broadly crosscutting concepts in our HyperCast case study (see the main article) would be awkward at best. Also, Open Modules don’t make clear the constraints that would have to be observed in writing new code to avoid inadvertently compromising the advising PCD semantics.

**Aspect-aware interfaces**

Gregor Kiczales and Mira Mezini recognized the need to program against crosscutting interfaces. In aspect-aware interfaces (AAIs), aspects extend the interfaces of modules they advise. Specifically, this approach computes aspects’ dependences on a system’s join points and shows these dependences as annotations on the explicit interfaces of advised code. Revealing such dependences can support modular reasoning and change. A programmer can see how join points are being advised and avoid making changes that invalidate those uses. Before stable modular interfaces emerge (for example, in Extreme Programming-style development), AAIs can serve as a valuable substitute—they inform, even if they don’t decouple and abstract. Likewise, the cross-references that AAIs provide could help guide refactoring activities, perhaps resulting in XPIs.

Yet AAIs don’t clearly express concerns in conceptual design. Instead of textually distinct interface constructs, they merely consist of scattered annotations. Nor do they provide textually localized interface definitions where behavior contracts can be documented or against which clients can be programmed. In addition, modularity support is limited. The display of dependences between existing code and PCDs can’t tell developers how to shape new code to correctly expose behaviors to those PCDs or how to write new PCDs to capture the existing code’s desired behaviors.

**Related Work on Aspect-Oriented Mechanisms for Software Design**

and complex design decisions and operates as a decoupling contract between providers and users. Unlike an API, an XPI abstracts a crosscutting behavior rather than a localized procedure implementation. In the case of AspectJ-style AOP, an XPI abstracts advised join points, as we mentioned before. To paraphrase David Parnas, XPIs modularize crosscutting design decisions that are complex or likely to change. You then implement an XPI not by providing a procedure implementation, but by writing PCD patterns and shaping code to expose specified behaviors through join points matching the given patterns. Designers need not know about specific aspects, such as logging, but they must decide which abstractions to expose as XPIs to facilitate aspect development and evolution. The method that we found to work is to expose key domain abstractions that the class design doesn’t adequately capture. So, to design XPIs, you don’t need any explicit references to aspects, although you can check the XPIs’ usefulness against anticipated aspects.

For examples of other approaches to aspect-oriented mechanisms for software design, see the sidebar.

**References**


Two designs for a figure editor

Two designs for the classic figure editor example illustrate our approach and its potential benefits. We evaluate how well the designs manifest fundamental design concerns, abstract irrelevant details, and accommodate change.

A traditional AO design

Consider a simple tool for editing drawings comprising points and lines (figure elements), where a display depicts each figure element, always reflecting the figure elements’ current states. 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 all figure elements’ current states. The specification requires a call to update() whenever a figure element’s abstract state changes.

Researchers have used this example to illustrate 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 to the tangling of these calls into code concerned with FigureElement updating.

The Observer design pattern could remove the explicit calls by enabling a display manager to register for a callback to update() on a state change event. However, this approach still requires that event-related code be scattered and tangled in the FigureElement code and elsewhere. AOP provides an alternative to such preparation in support of display updating. The DisplayUpdate aspect (see figure 1a) satisfies the update specification.

We implemented this aspect using the common approach we described in the section “The problem.” We studied the FigureElement code to find points where changes in the FigureElement abstract state occur. We generalized and described this set of points in the form of a PCD, FigureElementStateTransition(). This PCD captures calls to mutator methods of Line and Point and to moveBy(), which moves a figure element by a certain offset. Figure 1b 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.

Such a design raises three concerns. First, we had to write the Point and Line implementations before we wrote the aspect, which limited the available parallelism in development. Second, the aspect implementer had to study the Point and Line implementations to be able to write the aspect correctly. The lack of an abstraction layer between the aspect and the advised code adds to the cognitive load on the aspect implementer. The aspect lets the FigureElement writer ignore display updating, but the aspect writer can’t ignore FigureElement’s low-level details. Third, the aspect’s correctness depends on unstable details of the Point and Line implementations. So, apparently innocuous changes could compromise the design.

An AO design with XPIs

By employing XPIs, the designer seeks to not only insulate aspects from the details of the code they advise but also constrain that code to expose specified behaviors in specified ways. In the process, important crosscutting concerns that were previously embedded in the implementation become manifest as XPIs in the program. In the figure-editing case, an XPI will separate the DisplayUpdate aspect from FigureElement details. Our XPI refines the concept a transition has occurred in a FigureElement’s abstract state. It provides simple PCDs by which aspects can advise all such actions without depending on the underlying source code. In addition, it constrains the system implementer to implement all and only all abstract state changes in a way that matches the PCD patterns.

The XPI’s syntactic part exposes two named PCDs: joinpoint() and topLevelJoinpoint(). The PCD signature (name and parameters) constitutes the abstract interface. The part of the PCD that matches points in the code is part of the XPI’s hidden implementation (see figure 2a). It’s only here that dependences on details of the underlying code arise.

We document the semantics informally in the following prose. The joinpoint() PCD exposes all FigureElement state transitions. 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 topLevelJoinpoint() PCD
public aspect XFigureElementChange {
   /*
   The purpose of the joinpoint() 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 topLevelJoinpoint() PCD exposes
   all and only “top level” transitions in the abstract states of compound FigureElement
   objects.
   */
   public pointcut joinpoint(FigureElement fe):
      target(fe)
      && (call(void FigureElement+.set*(..))
         || call(void FigureElement+.moveBy(..))
         || call(FigureElement+.new(..)));
   public pointcut topLevelJoinpoint(FigureElement fe):
      joinpoint(fe) && !cflowbelow(joinpoint(FigureElement));

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

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

(a)

/*
Checks the contracts for the XFigureElementChange XPI.
*/
public aspect FigureElementChangeContract {
   /*
   PROVIDES: XPI matches all calls and only calls to FigureElement mutators
   */
   declare error:
      (!XFigureElementChange.staticmethodsScope() 
       && 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(): adviceExecution();

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

(b)

Figure 2. Building a figure editor using crosscut programming interfaces (XPIs): (a) the XFigureElementChange XPI and (b) a separate contract-checking aspect.
exposes all and only changes to the states of compound FigureElement objects (such as Lines) but not changes to their components (namely, Points).

An XPI’s semantics 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 an advice from calling FigureElement mutators either directly or indirectly.

Like APIs, XPIs enable a degree of contract checking. When included in the program’s build, the aspect in figure 2b constrains developers to modify a FigureElement’s internal state from only within the FigureElement mutators. To a degree, it also ensures that the aspects using the XFigureElementChange XPI can’t modify any FigureElement’s abstract state. The aspect can’t, however, verify the programmer’s adherence to the naming requirements.

Figure 3 presents a DisplayUpdate aspect using this 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. These classes contribute to implementing the XFigureElementChange XPI by ensuring that method names match the given PCDs if they have the specified change semantics.

Analyzing the designs

To compare the two designs, we first change public data members to private ones, forcing updates to occur through advisable method calls. We then extend FigureElement to include Color.

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

```java
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 this implementation:

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

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 apparently innocuous change broke the aspect’s assumption about Line’s otherwise-hidden implementation.

The XPI approach avoids such problems by establishing interfaces that impose design rules. Aspects can assume that the rules are
followed, and code within the XPI’s scope must conform to its terms. The reverse also applies: aspects must conform to the XPI’s terms, and the code can assume that the rules are followed. It’s important that XPIs have both syntax, in the form of convenient abstract PCDs, and semantics. Our XPI specifies that the PCD must match join points that indicate a change in a FigureElement’s abstract state. Under this XPI, DisplayUpdate uses the provided convenient PCD (and promises not to inject changes into a FigureElement). Also, Line’s implementer will implement Line.moveBy() so that the PCD captures its join point.

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 a FigureElement update when a Line’s color changes.

In the non-XPI approach, one of two undesirable scenarios is necessary to ensure that the display updates properly. In one scenario, the Color implementer must be aware of the DisplayUpdating aspect and its PCD implementation to determine how to name the Color setter method so that the PCD will match it. In the other scenario, the aspect implementer must change the DisplayUpdating PCD to match whatever choice the Color implementer makes. As the number of aspects increases, these scenarios become increasingly problematic.

In the XPI case, the Color implementer need only be aware of the figure element state change XPI and its constraint that only a method whose name is moveBy or starts with set can change a state. The XPI’s presence thus guides the implementer in choosing names for methods and in making other decisions that can influence PCD matching. In this case, the implementer must name the method something such as setColor, rather than changeColor;

The advantage of using an aspect is that code changes can be localized to the aspect, even if their effects aren’t.
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.

Extending the new design
XPIs can facilitate adding a classic non-functional aspect, property enforcement. Adding a property and its implementation to a system is an important issue. To explore it in the context of XPIs, we add a feature that maintains a geometric invariant in the figure editor: Lines must not be degenerate. That is, the two points that define a line can’t have identical coordinates. Enforcing this invariant requires that no Line is degenerate when it’s 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.

Invariant enforcement essentially changes a Point’s originally specified behavior by conditioning a Point mutator’s effects on that 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 it does localize the code needed to effect the behavioral change in the Point class. However, it doesn’t necessarily obviate the need for changes elsewhere in the code to accommodate that change in behavior. With this observation in mind, we 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 reflect key abstractions in the conceptual design.

We assume that the designer will use an aspect module to implement the invariant enforcement. An appropriate XPI to write the aspect against doesn’t already exist. So, we need to determine the domain abstraction for decoupling the aspect’s development from the normal case’s development and then write that XPI. One abstraction we need is that of a change to a Point that is part of a Line.

Figure 4. The PointLineRelation aspect records the Line to which a Point belongs.

```
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.joinpoint(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;
    }
}
```

The precise invariant we seek for the given design is that a Line can’t have two end Points at the same coordinates. 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, no field in Point stores a containing Line. A subtlety is that some Points are part of a Line and some aren’t. (And in a real system, a point might be a part of many lines.)

So, 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 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 is
replaced. Although this aspect updates the parents of Points, it doesn’t violate the XFigureElementChange requires clause because the parent is part of the hidden state of FigureElements. In keeping with this XPI, this solution introduces no setParent method, 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 a Line’s endpoints: change in x coordinate, change in y coordinate, and change in both coordinates.

Having written this XPI, we can now straightforwardly write an aspect for invariant enforcement (not shown). Using around advice, the aspect advises changes in Points that are in Lines and lets them occur only if they preserve the invariant. The XPI abstracts changes to Points in Lines. The aspect separately abstracts the invariant and enforcement policy. Such separation is at the heart of our interface-oriented approach to AO design for improved modularity and abstraction. It permits reuse of the XPI for implementing other aspects and decouples those aspects from possible changes to the ways that Points and Lines may be modified.

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, this approach provides a principled alternative to the concept of oblivious design. In our discussions with best-practice AO programmers, we’ve found that some of them 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 that AO programming languages provide.

Our experience to date with XPIs is limited to two systems, HyperCast3 and the figure editor. We expect that integrated-development-environment support could aid programmers by showing the scope of an XPI’s applicability. Being nonhierarchical, XPIs can overlap in scope.

```
/*
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. Using an XPI to add an invariant property: (a) The XPointInLineChange XPI and (b) the resulting design.

(a)
Also, we haven’t yet investigated the promise of XPIs for AO languages with different mechanisms than AspectJ’s. An appealing aspect of our approach, however, is that it’s neutral with respect to a language’s join-point model. It forces specified behaviors to be revealed through interfaces implemented in terms of whatever join-point model a language supports.

Acknowledgments

US National Science Foundation CISE (Computer and Information Science and Engineering) grants FCA-0429947 and FCA-0429786 helped support this research.

References