Modular Aspect-Oriented Design with XPIs

KEVIN SULLIVAN$^{\phi}$, WILLIAM G. GRISWOLD$^{\lambda}$, HRIDESH RAJAN$^{\theta}$, YUANYUAN SONG$^{\phi}$, YUANFANG CAI$^{\beta}$, MACNEIL SHONLE$^{\lambda}$, and NISHIT TEWARI$^{\phi}$
$^{\phi}$University of Virginia, $^{\lambda}$University of California, San Diego,
$^{\theta}$Iowa State University, and $^{\beta}$Drexel University

The emergence of aspect-oriented programming (AOP) languages has provided software designers with new mechanisms and strategies for decomposing programs into modules and composing modules into systems. What we do not yet fully understand is how best to use such mechanisms consistent with common modularization objectives such as the comprehensibility of programming code, its parallel development, dependability, and ease of change. The main contribution of this work is a new form of information hiding interface for AOP that we call the crosscut programming interface, or XPI. XPIs abstract crosscutting behaviors and make these abstractions explicit. XPIs can be used, albeit with limited enforcement of interface rules, with existing AOP languages, such as AspectJ. To evaluate our notion of XPIs, we have applied our XPI-based design methodology to a medium-sized network overlay application called Hypercast. A qualitative and quantitative analysis of existing AO design methods and XPI-based design method shows that our approach produces improvements in program comprehensibility, in opportunities for parallel development, and in the ease with which code can be developed and changed.

Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design

General Terms: Design

Additional Key Words and Phrases: Aspect-oriented programming, design rules, options

1. INTRODUCTION

Over thirty years ago, seeing the opportunities afforded by the new module composition mechanisms of procedural programming and separate compilation, David Parnas asked the question, *by what criteria should systems be decomposed into modules?* [1972]. Targeting the objectives of ease of change, comprehensibility and opportunities for parallelizing the development process, he used a comparative analysis of modularization styles to argue that the prevailing criterion based on a functional decomposition performed poorly relative to

The work described in this article is the revised and extended version of papers presented at ESEC/FSE 2005 in Lisbon, Portugal and IEEE Software, Jan/Feb 2006 special issue on AOP. This research supported in part by NSF grants FCA-04-29947, CCF-06-13840, CCF-0613845, CNS-06-27354, and CNS-07-09217.

Authors' address: K. Sullivan, Y. Song, N. Tewari,{sullivan, ys8a, nt6x}@cs.virginia.edu, Computer Science, University of Virginia, Charlottesville, VA 22903. W. Griswold, M. Shonle, {wgg, mshonle}@cs.ucsd.edu, Computer Science and Engineering, UC San Diego, La Jolla, CA 92093. H. Rajan, hridesh@cs.iastate.edu, Computer Science, Iowa State University, Ames, IA 50010. Y. Cai, yfcai@cs.drexel.edu, Computer Science, Drexel University, Philadelphia, PA 19104.

his proposed new approach, *information hiding*. Under the information hiding approach, "one begins with a list of difficult design decisions or design decisions that are likely to change. Each module is then designed to hide such a design decision from the others" [Parnas 1972, p. 1058]. Parnas focused on changes to data representations and argued that one should design stable abstract interfaces—abstract data types—to hide decisions about concrete data representation that are expected to change, decoupling changes in them from design decisions throughout the rest of a system.

In this paper, seeing the opportunities afforded by the new mechanisms of *aspect-oriented* (AO) programming, we revisit Parnas's question, but with a twist: by what criteria should systems be decomposed into aspects? We address this question through a systematic study of a modern system, Liebeherr's HyperCast overlay network system [HyperCast]. Paralleling Parnas's study, we compare and contrast a well-known aspect-oriented decomposition strategy, called *obliviousness* [Filman and Friedman 2005a], with our proposal, which we call *crosscut programming interfaces*, or *XPIs* for short. XPIs are explicit, abstract interfaces that decouple aspects from details of advised code [Sullivan et al. 2005; Griswold et al. 2006]. Without limiting the possibilities for AO advising or requiring new programming languages or mechanisms, the XPI approach appears to better modularize aspects and advised code, allowing for their separate and parallel evolution, and producing a better correspondence between programs and designs.

The XPI approach in effect generalizes Parnas's notion of information hiding to the context of aspect-oriented languages. We exploit Baldwin and Clark's work on *design rules* [Baldwin and Clark 2000] to provide the perspective and a modeling notation for this work. In particular, design rules are imposed as interfaces between aspects and base code[1], constraining their subsequent development so as to decouple them. Our design rules govern how join points[2] are composed in base code and how aspects use these rules to ensure that specific join points are exposed in a way that enables the integration of separately implemented aspect modules. Concretely, our design rules specify (1) the types of join points used to expose specified behavior (e.g., field access versus method call); (2) naming conventions for join points; and (3) constraints on base code and aspect behaviors. The first guarantees that join points needed by aspects are visible. The first two together permit aspect designers to write pointcut descriptors[3] (PCDs) that do not have to change when the base code evolves. The last ensures that only intended effects occur across join points, between base code and aspects. These rules are documented in interface specifications that base code designers are constrained to "implement," and that aspect designers are licensed to depend upon.

One of the properties important to many designers of AOP languages is that the base code does not have to contain additional *hooks* for aspect code to advise. Our method does not require any auxiliary code to be added to base code (e.g., to signal events), or references from base code to aspects or other external code. Nor does it involve the introduction of any new programming mechanisms or semantics, although we discuss how the checking of design rules could be valuable in some cases. In particular, our approach does not constrain the join-point model (JPM), but rather generalizes to a range of possible JPMs.

[1] The base code in AO terms refers to the part of the application modularized using traditional OO mechanisms.

[2] A join point is a point in the dynamic execution of the program, e.g. method call, catching an exception, etc. For non-expert readers, we provide a more detailed background in Section 2.

[3] A pointcut descriptor (PCD) is a predicate that selects a subset of join points in a program. See Section 2.

The following section provides background material on aspect-oriented programming and the AspectJ model in particular. As an example we use the simple “figure element” system to demonstrate the capabilities of AspectJ. In Section 3 we show our solution based on XPIs and demonstrate their use in the figure editing system. This section also shows how these ideas can be used with current popular aspect-oriented languages, and with AspectJ, in particular. In Section 4 we compare and contrast our methodology with the oblivious design method to illustrate the advantages of our approach at a smaller scale. In Section 5 we introduce the HyperCast case study we conducted to compare the oblivious approach to our new information-hiding, XPI approach. In the case study we consider the design and development of HyperCast, and how it can be extended by utilizing XPIs. Then we compare the design structure of approaches and XPI approach, first qualitatively—from the designer’s perspective—and then quantitatively using an economic analysis based on options theory [Baldwin and Clark 2000]. In the process, the analysis shows that both AO techniques are better than the plain OO design of HyperCast. Section 6 discusses remaining design considerations when working with XPIs. Section 7 compares and contrasts our ideas with related work and Section 8 concludes.

2. BACKGROUND

In this section we briefly introduce aspect-oriented programming (AOP), in particular the AspectJ model. Large-scale industrial adoption of AOP is being reported [Bonér 2004; Colyer and Clement 2004; Colyer et al. 2006; Lesiecki 2005; Thomas 2005; Sabbah 2004] and a number of books have appeared [Pawlak et al. 2005; Gradecki and Lesiecki 2003; Laddad 2003; Jacobson and Ng 2005; Kisely 2002; Colyer et al. 2005; Filman and Friedman 2005b; Rashid 2004]. As an example, we use a figure editing system to introduce the concepts of crosscutting, scattering and tangling, and how AOP addresses these issues.

AO languages, such as AspectJ [Kiczales et al. 2001b] and the many languages that emulate it, offer new mechanisms and possibilities for decomposing systems into modules and composing modules into systems. In this paper we take AspectJ as the canonical exemplar of this class of languages. When we speak of AspectJ we mean any fundamentally similar language or set of mechanisms. One of the chief goals of such languages is to enable the modularization of concerns that cut across traditional abstraction (class and function) hierarchies. That is, aspects are meant to localize code that otherwise would have been scattered and tangled across the implicated classes and functions.

Canonical examples of crosscutting concerns include tracing, logging, transactionality, caching, and resource pooling. The ability to better separate these concerns is expected to ease software development and evolution through improvements in abstraction, comprehensibility, parallel development, reuse, and ease of change, among others. With such improvements should come reduced development costs, increased dependability and adaptability, and more value for software producers and consumers.

The most prominent AOP model today is based on the use of aspect modules that support quantified *advising* of selected concrete program execution events, called (*join points*), that are made visible for advising by the semantics of the given aspect language. In AspectJ, for example, an aspect module uses a declarative construct called a *pointcut descriptor* (PCD) to declaratively specify a set of join points at which anonymous-method-like constructs called (*advice*) [Teitelman 1966] should run. Aspects can specify advice to run before, after or around the specified join points. Advice have access to reflective information about the program state at join points. While these AO programming mechanisms were provided to enable the modularization of traditionally crosscutting concerns [Kiczales and Mezini 2005], they are quite general and do admit other uses.

To clarify these ideas we present a simple example from the literature: a *figure editing* system for editing drawings made up of point and line objects (the figure elements) [Kiczales and Mezini 2005]. Each figure element is to be depicted on a display, and the display is to be updated whenever any figure element changes its state.

The `FigureElement` class (not shown) 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 any figure element changes.

Here the crosscutting concern is the *display update policy*: "When the abstract state of a `FigureElement` changes, the `Display` must be updated." An OO Implementation of this policy leads to scattered `update` calls throughout `FigureElement` class hierarchy, and the tangling of these calls into code concerned with `FigureElement` updating.

The Observer pattern [Gamma et al. 1995] could remove the explicit calls from the `FigureElement` subclasses by providing a display manager with the ability to register a callback to `update` on a state change event. However, this approach still requires that event code be scattered and tangled in the `FigureElement` code and elsewhere. Nor does this approach enable declarative specification of the (possibly evolving) set of events with which the update callback should be registered.

```
public aspect DisplayUpdate {
 pointcut FigureElementStateTransition():
  call(* FigureElement+.set*(..)) || call(* FigureElement+.moveBy(..));
 after(FigureElement f): FigureElementStateTransition() && target(f) {
   Display.update(f);
 } }
```

Fig. 1. A Traditional Display-updating Aspect in AspectJ [Kiczales et al. 2001a].

AOP provides an alternative that avoids the need for such preparation of the base code or explicit enumeration of callback registrations in support of display updating. In AspectJ the display update specification is implemented in the `DisplayUpdate` aspect, shown in Figure 1. AspectJ [AspectJ; Kiczales et al. 2001b] extends Java with several complementary mechanisms, notably *join points* (JPs), *pointcut descriptors* (PCDs), *advice*, and *aspects*. Join points are dynamic events in a program's execution—such as when an object's method is called, when an object's field is set, or when a new object is created—that by definition of the programming language are subject to *advising.* Advising is the extension or overriding of the dynamic event at a join point by a CLOS-like [Steele 1990, Ch. 28] *before, after,* or *around* anonymous method called *advice.* A PCD is a declarative expression that matches a set of JPs. An advice extends or overrides the action at each join point matched by a given PCD. Because a PCD can select JPs that span unrelated classes, advice behaviors are crosscutting in their effects, yet are locally specified. Which dynamic events are subject to advising is defined by the *join-point model* (JPM). In AspectJ, the JPM is focused around exceptions and the pure OO features of Java.[4] Advice, pointcuts, and

[4] Lower-level events such as integer operations or loops are not subject to advising in AspectJ. Advising such events could conceivably be useful to, e.g., capture all integer operations that overflow, but integer operations are both common and do not have as rich of a context as OO operations do. Thus, expanding the JPM in this manner would likely harm efficiency by requiring too much instrumentation with no corresponding benefits.

ordinary data and method declarations are grouped into class-like modules called *aspects*.[5]

On line 1 of Figure 1 an aspect named `DisplayUpdate` is declared. On line 2, a PCD named `FigureElementStateTransition` is defined. That PCD is composed of all join points that match either calls to the `moveBy` method or any method whose name starts with **`set`**. The use of `FigureElement+` limits these join points only to matching calls that are in the `FigureElement` class hierarchy. On line 4 an **`after`** advice is declared; it states that after any of the events that match those in `FigureElementStateTransition` occur to make a call to the static `Display.update` method. Join points can also have context associated with them, for example the **`target`** of a method call (i.e., the object the method was being invoked on). This context can be bound to variables that can then used in advice bodies. For example, in the **`after`** advice body that starts on line 4 the variable '`f`' is the `FigureElement` object that just had a state transition, this binding is local to the advice body and can be used like any other variable.

3. ASPECT-ORIENTED PROGRAMMING WITH DESIGN RULES

The aspect-oriented design of the figure editor shown in Figure 1 was realized using the oblivious design method popularized in the research and practitioner literature on AOP. We will revisit this method in detail later, but in summary there are several reasons to be concerned about this design style. First, it promotes sequential development in the sense that base implementation (e.g. `Point` and `Line` classes) had to be written before the aspect could be written. Second, undue cognitive burden is places on the aspect implementor as they alone become responsible for the correctness of the PCD expressions. Last but not least, aspect code remains coupled to the unstable implementation details of the base code. Yet, if the base code developers were to be non-oblivious to the aspects, then the base code would become less comprehensible, tightly tangled with the aspects, and no longer separately developable from the the aspects. The situation when base code developers need to "prepare code for aspects" like this is known as *intimacy* [Elrad et al. 2001, p. 31].

The question, then, is if there is a third way, in which a new design method can realize the benefits of AO design and yet minimizes or eliminates the problems encountered with obliviousness or its alternative, intimacy. In the oblivious approach, pointcuts cannot be written and advice parameter lists cannot be formulated until the base code is written. If obliviousness is given up for intimacy, then the base code needs to be heavily revised when the aspect is developed [Elrad et al. 2001]. The one-direction coupling of each extreme provides the intuition that both the base code and the aspects need to be decoupled from each other. A short design phase that establishes symmetric separation of concerns between base designers and aspect designers could increase overall parallel development. In this section we describe our approach to aspect-oriented design based on this intuition. We first present necessary terms and then describe our proposed notion of the crosscut programming interface, a design construct that enables truly modular aspect-oriented designs. The figure editing example is used to illustrate the key benefits of our approach.

3.1 Terminology: Design Structure and Design Rules

Our analysis depends heavily on the notion of the *dependence* between elements in a software design, and especially on the *structure* of dependences in a given design. We say that

[5] Named pointcuts can be declared in classes, but other AspectJ constructs are restricted to aspects.

an element B *depends upon* an element A if and only if B makes assumptions about A that, if invalid, can cause B to fail to achieve its intended properties. Invalid assumptions can arise for many reasons, such as: A's developer miswrites A's specification; A's specification changes but A's developer is not aware of the change; B's developer misunderstands A's specification; B is unaware of changes in A on which B depends.

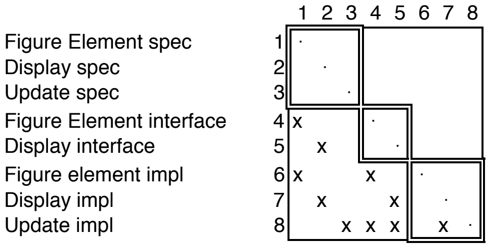
These design dependences are de facto dependences among the developers involved in the system design and evolution task. The depends-upon relation is critical for our analysis because changes to any property of A on which B depends create possible obligations for B to change. The structure of dependences on design elements largely determines the dynamics of the design task.

We say that a design or a part of a design is *modular* if its elements do not depend on each other (although they may depend on external contracts, or *design rules*, that serve to decouple them). The corresponding work in realizing the design is parallel in that these elements can be developed or changed independently, modulo their conformance to any prevailing contracts. A design and its corresponding implementation process are said to be *hierarchical* if elements are linked in a chain of dependences. In such a structure, upstream design elements are resolved before downstream decisions that depend on them. A design and its implementation process are called *coupled* if the elements depend on each other in some cyclic relation. Typically the different sub-elements within the same module demonstrate tight coupling to each other and thus must be considered together when either is modified. Thus, the presence of coupling alone does not exclude the potential for modular designs, but does affect what elements taken together should be considered as the same module.

The analysis of design structure is not limited only to describing the dependencies of software components, it can also be used to describe dependencies within processes. It is common for different dependence structures to prevail in different stages of the software process. In a waterfall process, for example, the relationship between the design-specification process and the implementation-design process is hierarchical, but the implementation process itself is modular, in that the identified modules can be developed independently subject to the prevailing rules (e.g., the syntax, behavioral, performance, and dependability specifications).

Following Baldwin and Clark [2000] and the earlier work of Sullivan et al. [2001], we present models of the coupling structures of design problems using Design Structure Matrices (DSMs) [Steward 1981]. DSMs present in a graphical form the pair-wise dependence structures of designs and of their corresponding development and evolution processes. Figure 2 is an example of a DSM.

The rows and columns of a DSM are labeled with *design variables*. These are the design elements or dimensions for which the designers must make decisions: e.g., the selection of an algorithm, the naming of a class, the formulation of an interface, or the choice of minimum acceptable reliability. For example, in Figure 2 `Display spec` (2) and `Display impl` (7) are examples of design elements. By `Display spec` (2) we mean the design variable `Display spec` shown in the second row and column in the figure. Cells in a DSM are marked to represent dependences among these decisions. For a given row (e.g., a design variable A), the marks in that row show which design decisions A depends upon. For a given column B, the marks in that column show which variables depend upon B. For example, in Figure 2 the design variable `Display interface`



		1	2	3	4	5	6	7	8
Figure Element spec	1	·							
Display spec	2		·						
Update spec	3			·					
Figure Element interface	4	x			·				
Display interface	5		x			·			
Figure element impl	6	x			x		·		
Display impl	7		x			x		·	
Update impl	8			x	x	x		x	·

Fig. 2. A design structure matrix representing the design of the figure editing system.

(5) depends only on the design variable `Display spec` (2), whereas the design choice for the variable `Update impl` (8) depends on the design choice for four other variables namely, `Update spec` (3), `Figure Element interface` (4), `Display interface` (5), and `Display impl` (7).

DSMs represent, in a readable albeit imprecise form, coupling structures in designs, design spaces and decision-making processes, and the paths of ripple effects of change. Any of these patterns can appear in sub-matrices of any overall DSM. An important pattern shows sequential coupling between large blocks but is modular within blocks. For example, one block may represent a set of decisions about interface designs. Implementation decisions then depend on the interfaces, but can be made independently of each other.

Carefully ordering and clustering the rows and columns of a DSM can reveal large- and small-scale dependence structures in patterns of marked cells. Consider as an example Figure 9, which we discuss in detail in section 5. At the outermost level of aggregation, the upper left part of the DSM represents abstract design concerns (i.e., the software specification), and the lower right, the code base that implements them. The code base comprises both interface and implementation elements. The marks below the concerns and to the left of the code base indicate a sequential structure: the code base depends on the abstract concerns. Similarly, within the code base, the implementation modules depend on the interfaces. The block diagonal structure of the implementation has a modular structure. The absence of off-diagonal marks indicates no dependences between the nested smaller boxes (each representing the interdependent design decisions within an implementation module). The dense marks in each implementation module reflect a coupled structure.

An *interface* is a statement of the properties of an element B upon which other elements, such as A, are permitted to depend upon, and that B's designer is obliged to produce in B. One of the central operations in a design activity is to create interfaces to decouple otherwise coupled design elements and their corresponding processes. For example, two elements can depend on an interface, but are made independent of each other. This property is what would commonly be called a modular design. However, by Parnas's criterion, a design exhibits information-hiding modularity only if the interfaces themselves are relatively immune to change. By modeling the external forces of change on a system as a special class of design variables that we call *environment variables*, it is possible to demonstrate whether the design is modular in Parnas's sense [Sullivan et al. 2001]. The interface cluster should not depend on the environment cluster; only implementation variables should depend on the environment variables. In Figure 9 the environment variables are modeled as elements of the system requirements specification (or concerns) that the HyperCast designers believe will change.

3.2 Crosscut Programming Interfaces

The key idea behind our design methodology is to introduce a short design phase before the design of base and aspect parts of the system. During this design phase, for each crosscutting concern a designer establishes a crosscutting design-rule interface to decouple the base design and the aspect design. The constraints imposed by a design rule govern two things: (1) which execution phenomena must be exposed as join points; (2) how the join points are exposed through the JPM of the given language (e.g., in AspectJ, rules could govern syntax, name, and stack shapes). These elements ensure two important properties: One, the PCDs required by aspects can be written once and will not have to change when the base code is evolved. Two, the state at a join point is the actual behavioral point in the execution of interest to the aspect. Our method is thus an instance of the information hiding approach, but for crosscutting join-point-based interfaces.

We call our crosscutting design rule interface a *crosscut programming interface*, or *XPI*. 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 behavior rather than a localized procedure implementation. In the case of AspectJ-style AOP, an XPI abstracts advised join points. To paraphrase Parnas [1972, p. 1058], XPIs modularize crosscutting design decisions that are complex or likely to change. One implements an XPI, then, 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 be aware of specific aspects, such as logging, but they do have to decide which abstractions to expose as XPIs to facilitate development and evolution of aspects. The method that we found to work is to expose, as XPIs, key domain abstractions not adequately captured in the class design. Thus, no explicit references to aspects are needed in designing XPIs, although the usefulness of the XPIs can be checked against anticipated aspects.

We realize XPIs as syntactic constructs in AspectJ, with semantics defined by weakest pre-condition invariants. An XPI is composed of several elements:

(1) a name

(2) a set of one or more abstract join points, each expressed as a PCD *signature* declaring a name and exposed parameters

(3) a scope over which it abstracts join points

(4) a partial *implementation* comprising a join point pattern; a before, after or around designator; and a corresponding set of constraints, or *design rules*, that 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 conformance of code to the stated design rules.)

In AspectJ these elements are declared in a stylized aspect.[6] Some invariants can be checked with separate pluggable aspects (discussed in detail in Section 3.4). In the next section, we describe an example XPI for the figure editor system.

[6] In AspectJ, before, after or around designators cannot be associated with PCDs, per se, but only with advice constructs that use PCDs.

3.3 The Figure Editing System with XPIs

In this section, we use the figure editing example introduced in Section 2 to show the XPI method in practice and what benefits can accrue from it.

```
public aspect XFigureElementChange {
 /** Informal specification of the join points exposed by this XPI. */
 /*@ exposes (* 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. *) */
  public pointcut joinpoint(FigureElement fe):
    target(fe) && (call(void FigureElement+.set*(..))
      || call(void FigureElement+.moveBy(..))
      || call(FigureElement+.new(..)));

 /*@ exposes (* all and only "top␣level" transitions in the
   @    abstract states of compound FigureElement objects. *) */
  public pointcut topLevelJoinpoint(FigureElement fe):
    joinpoint(fe) && !cflowbelow(joinpoint(FigureElement));

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

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

Fig. 3. An Example XPI, XFigureElementChange, for the Figure Editor.

By employing XPIs, the designer seeks to insulate aspects from the details of the code they advise, while constraining that code to expose specified behaviors in well-defined 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 example from Section 2, an XPI can serve to decouple the `DisplayUpdate` aspect from `FigureElement` details. One such XPI can do so by reifying the concept *'a transition has occurred in the abstract state of a `FigureElement`.'* Figure 3 is an example of an XPI written in AspectJ: 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 implementer to ensure that all abstract state changes (and only such state changes) are implemented in a way that matches the PCD patterns. By convention, aspects that specify XPIs begin with an "X" to distinguish them from non-interface aspects. Like an API, an XPI has a syntactic part, expressed in a programming language, and a specification, expressed informally in Figure 3 in a style similar to JML [Leavens et al. 2006], where informal contracts appear as **`exposes`** clause between `(* *)`.

The syntactic part of the XPI exposes two named PCDs, `joinpoint` and `topLevelJoinpoint` (highlighted in the figure). 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 3). It is only here that dependences on details of the underlying code arise.

The joinpoint PCD in Figure 3 exposes all FigureElement state transitions.[7] 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 exposes all and only changes to the states of compound FigureElement objects (such as Line) but not changes to their component elements (e.g. changes to Points embedded in Lines).

The scope of the XPI is also specified using two named PCDs staticscope and staticmethodscope (also highlighted in Figure 3). These together specify the types that are within the scope of this XPI and methods within these types. Only developers for modules that are within this scope need to adhere to the conventions specified by this XPI.

```
public aspect DisplayUpdate {
 after(FigureElement fe): XFigureElementChange.topLevelJoinpoint(fe) {
  Display.update(fe);
 }
}
```

Fig. 4. The Display Updating Aspect Using an XPI.

Figure 4 presents a DisplayUpdate aspect using this XPI. The aspect now depends only on the abstract, public PCD signatures of XFigureElementChange, not on implementation details of the Point and Line classes (in contrast with Figure 1). 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.

3.4 Behavioral Contracts for XPIs

The crosscutting interface between the base code and the aspect brings an additional advantage to aspect-oriented software development. This interface can now be used for specifying contracts between the base code and the aspects in a style similar to behavioral specification of APIs as found in languages like Eiffel [Meyer 1988; 1992], APP [Rosenblum 1995], Larch [Guttag and Horning 1993; Guttag et al. 1985], and JML [Leavens et al. 2006]. Such contracts establish preconditions and postconditions such that services are unaware of their clients, but are obliged to serve clients that meet the given preconditions; and clients need to only depend on the contract, and can assume the postconditions hold when they meet the preconditions. An example in a style similar to JML [Leavens et al. 2006] appears in Figure 5, where informal contracts appear between (* *).

The specifications for XPIs can govern constraints on behavior across join points (e.g., pre- and post-conditions for the execution of advice compositions). For each abstract join points exposed by the XPI, such specification could be expressed as:

—a *provides* clause: a semantic specification stating pre-conditions that must be satisfied at each point where advice can run, and

[7] We use the generic PCD name joinpoint as an analog of a generic "run" method name; the semantics are already suggested by the name of the containing XPI.

```
// 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(): adviceexecution();

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

Fig. 5. An Example Behavioral Contract Checking Aspect for the XFigureElementChange XPI

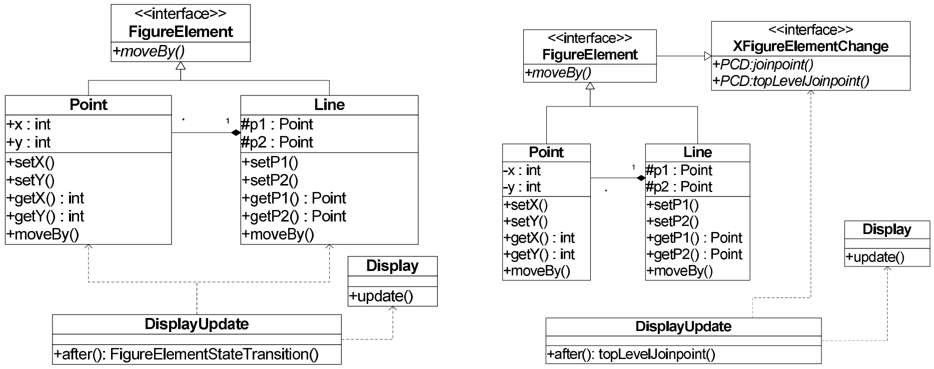
—a *requires* clause: a semantic specification stating post-conditions that must be satisfied after advice runs.

These two clauses ensure that the state at a join point is the actual behavioral point in the execution of interest to the aspect and that the system state after advice has returned is not compromised In other words, the semantics of XPIs can include behavioral constraints on aspects.

An example of such contract for the figure editor is shown in Figure 5. This contract, also expressed in the form of an aspect can be included in the program's build. 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. To a degree, it ensures that the aspects using the `XFigureElementChange` XPI are not able to modify the abstract state of any `FigureElement`. The contract also constrains developers to modify the internal state of a `FigureElement` from only within the `FigureElement` mutators. The aspect cannot, however, verify the programmer's adherence to the naming requirements.

4. COMPARATIVE ANALYSIS: TWO DESIGNS OF THE FIGURE EDITOR

In this section we compare and contrast our design methodology based on XPIs with an approach popularized as "oblivous method". We use the figure editing example for this analysis. We have considered three scenarios. Our first set of changes are inspired from the previous work of Kiczales and Mezini [2005], which simulates a change in an already advised part of the base code. Second scenario discussed in Section 4.2.2 simulates a change where new elements are introduced base code that may be subject to be advising by the aspects. This demonstrates that the notion of scope for XPIs reduces the analysis that must be conducted by the base code designer while making such a change. Instead of analyzing all aspects, the base code designer must only analyze the XPIs that are in the scope. The third scenario presented in Section 4.2.3 is an example of an integration relationship where components must behave together in a specified manner to fulfill system requirements. Our analysis illustrates that even in a smaller scale our design methodology offers significant benefits.



(a) Oblivious Design

(b) XPI-based Design

Fig. 6. Two Aspect-oriented Designs of Figure Editor.

4.1 Oblivious Design of the Figure Editor

To realize the AO design using the oblivious method popularized in the research and practitioner literature on AOP, we studied the FigureElement code to find the points where changes in a FigureElement's abstract-state occurs. 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 6(a) presents a UML model of this design. As is typical the oblivious method, the DisplayUpdate aspect depends on implementation details of the Point and Line classes. This dependence is shown as the dotted lines.

There are several reasons to be concerned about the design in Figure 6(a). 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 FigureElementStateTransition PCD 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 otherwise trivial changes in them.

As a result, the aspect code is tightly coupled to implementation details of the base code, a problem identified by Tourwé et al. [2003] as "the AOSD-Evolution Paradox." In the presence of primitive aspects, the oblivious design technique is non-modular, because it requires tight coupling of aspects to the base code implementations they advise. One active area of AOP programming language research is to define richer pointcut descriptor languages (e.g., those that are more declarative in nature) to reduce the dependence on implementation details, e.g. Model-based pointcuts [Kellens et al. 2006], test-based pointcuts [Sakurai and Masuhara 2007], and logic-based pointcuts [Rho et al. 2006].

Generally we found that in an aspect language like AspectJ the oblivious method led to programs that were hard to develop, understand, and change. Others have also pointed this out, e.g. [Clifton and Leavens 2003; Tourwé et al. 2003; Aldrich 2005; Altman et al.

2005; Dantas and Walker 2006; Ostermann 2008]. First, our designers had to inspect all the code to identify the relevant join points for the PCDs to encompass. Second, these join points were not exposed in a consistent way, so complex PCDs and advice bodies were needed to effect the desired advising. Moreover, apparently innocuous changes or extensions to the code base could then change the matched join points, violating assumptions made by the aspects. Nor did the resulting class and aspect abstractions reflect the underlying conceptual design adequately. For example, consider the display updates triggered by the state transitions in the figure element system. Although the aspects separated policy decisions on *how* to respond to transitions, the state transitions themselves were not exposed as explicit, design-level abstractions.

4.2 Abstraction and Evolvability of the Two Figure Editor Designs

We now compare the oblivous design with XPI-based design in terms of abstraction and evolvability. Figure 6(b) presents a UML model of this design. In this design, aspect only depends on the detail of the `XFigureElementChange` XPI, which is shown as a dotted arrow from aspect to the XPI. The base code, in particular, the `FigureElement` inheritance hierarchy is also responsible for implementing the protocols of the XPI, we show it using the **`implements`** arrow between `FigureElement` and the XPI. As Kiczales and Mezini [2005] did, we first change public data members to private, forcing updates to occur through advisable method calls (in section 4.2.1). We then extend the `FigureElement` class to include `Color` (in section 4.2.2). Finally, section 4.2.3 shows how adding a classic "non-functional" aspect—property enforcement—is facilitated by XPIs.

4.2.1 *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 leads the `Line.moveBy` coder 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 (Figure 1). 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 this apparently innocuous change [Aldrich 2005].

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, the `DisplayUpdate` aspect uses the provided PCD (and promises not to inject changes into `FigureElement`), and the implementor of `Line` would implement `Line.moveBy` so that its join point is captured by the PCD as required.

4.2.2 *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 a `FigureElement` update themselves 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. In the first scenario, 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. In the alternative scenario, 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.

4.2.3 *Property Enforcement.* We explore property enforcement in the context of XPIs by adding a feature that maintains the geometric invariant in the figure editor that: *`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` (for example, an exception may be thrown, an error logged, or the change can be prevented from transacting altogether). 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 reflect key abstractions in the conceptual design.

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 *all changes to `Points` that are 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 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, 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.[8]

```
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 changes, 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;
  }
}
```

Fig. 7. The PointLineRelation aspect.

Thus, the first part of our solution is an aspect that represents Point-Line relations. We use the aspect to introduce a *parent* field into the Point class, to record the Line a Point belongs in, if any (see Figure 7). The aspect uses the XFigureElementChange XPI, updating the parent field as appropriate when a Line is created or one of its Points replaced. Although the aspect updates the parents of Points, it does not violate the XFigureElementChange's requires clause because the parent is part of the hidden state of FigureElements. In keeping with the XFigureElementChange XPI, no setParent method is introduced, calls to which would inappropriately result in updating the Display.

Figure 8 presents the XPI and resulting design. The XPointInLineChange XPI exposes three events on the end-points of a line: changes in x-coordinate, changes in y-coordinate, and changes 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 for implementing other aspects, and decouples those aspects from possible changes to the ways that Points and Lines may be modified.

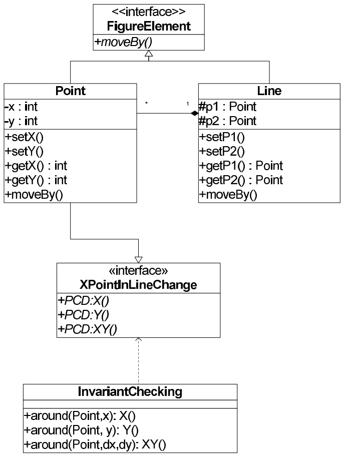
5. CASE STUDY: HYPERCAST

We validated our crosscut programming interface design methodology through a medium-sized case study [Sullivan et al. 2005]. We started with HyperCast [HyperCast; Liebeherr et al. 2003]—a pure Java application that consists of 686 classes and 48,984 lines of code—and did a comparative analysis of three different designs: (1) its actual OO design; (2) an

[8] And in a real system, a Point may be a part of many lines, a consideration we elide for simplicity. A more general solution could use sets of parents or provide canonical bindings with WeakHashMaps.

```
public aspect XPointInLineChange {

/*@ exposes (* changes to the x coordinate
 *@  of any point that belongs to a line *) */
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());
}
```

(a) The XPointInLineChange XPI



(b) Resulting UML Design

Fig. 8. The XPointInLineChange XPI and the resulting design.

AO design using the oblivious approach; and (3) an AO design utilizing XPIs. HyperCast is a real system developed independently of the analysis reported in this paper. It includes scattered code fragments for classic crosscutting concerns, including logging.

Section 5.1 provides an overview of the HyperCast system. Section 5.2 shows an AO design of HyperCast using the oblivious approach, and discusses the problems we encountered with such a design. In particular, the same problems encountered in the "toy" figure editing example recurred on a larger scale, such as tight coupling and the difficulty of writing correct PCDs. We found that the oblivious approach generally simplifies the task of the base code designer, but it can signficantly complicate overall system development due to the lack of an agreement between the base and aspect code developers about how their respective parts will be integrated. We categorize and discuss these problems in detail. Section 5.3 presents an AO design of HyperCast using XPIs, which decouples the modules and addresses the problems encountered with writing PCDs. Section 5.4 gives a qualitative and quantitative analysis of the different AO designs. The case study shows that it is practical to apply XPI-oriented AO technology on real software systems. Furthermore, it demonstrates that the XPI approach enables improved modularity for both aspects and advised code, which allows for their separate and parallel evolution.

5.1 HyperCast Overview

HyperCast, developed by the Multimedia and Networks Group at the University of Virginia, builds and maintains logical, self-organizing overlay networks between applications on the Internet. The version of HyperCast that we used in our study is a milestone build of HyperCast 3.0. HyperCast 3.0 supports overlay networks with 3 type of topologies: Hypercubes (HC) [Liebeherr and Beam 1999], Delaunay Triangulation (DT) [Liebeherr et al. 2002] and Spanning Trees (SPT). The HyperCast software is built on the notion of overlay sockets. An overlay socket is an endpoint for communication in an overlay network. It provides application programs an interface for communications over a logical overlay topology. An application specifies an overlay topology and a data transport protocol (TCP, UDP) when it instantiates an overlay socket. An overlay socket provides a variety of services (naming, reliability transportation, synchronization, etc). HyperCast also has some components that are external to an overlay socket but interact closely with an overlay socket (event notification, monitor and control, etc).

The key abstraction provided by HyperCast is the *overlay socket*. An overlay socket supports point-to-point and multicast communication in overlay networks. HyperCast integrates overlay sockets, viewed as nodes, into networks in a decentralized manner. It also offers network services including naming, reliable transport and network management.

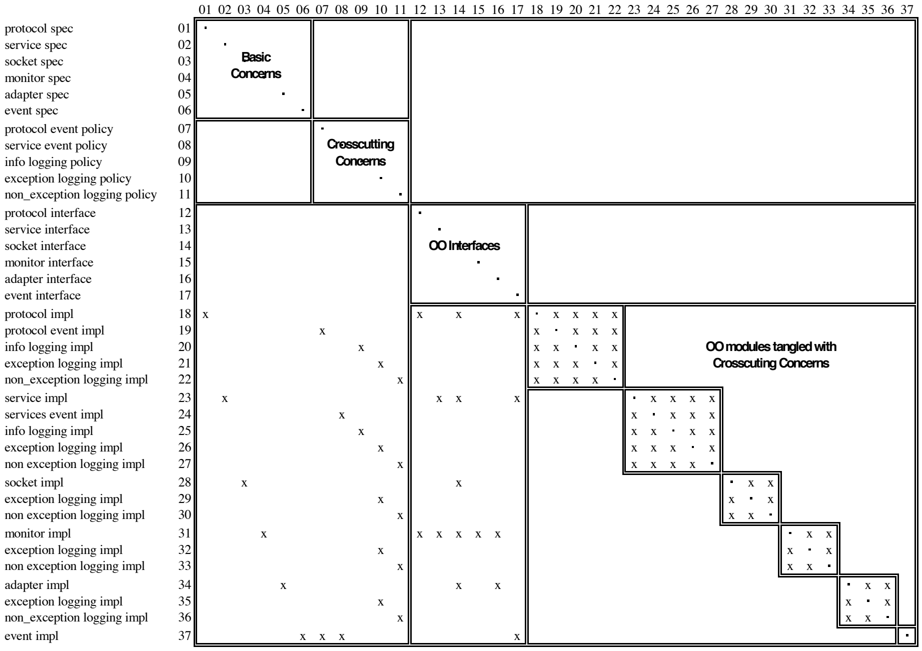
Key concerns in the design of HyperCast include the following: *Socket:* the design of the overlay socket API; *Protocol:* the protocols for maintaining various network topologies; *Monitor:* a HyperCast network management capability; *Service:* network mechanisms for end-to-end service; *Adapter:* a layer that virtualizes underlying networks; *Logging:* a mechanism to record selected events. These concerns map loosely coupled class in the implementation.

There are several areas in which scattering and tangling of code is evident. In this case study we address two of them: implicit invocation, and logging. The implicit invocation concerns in Hypercast are primarily integration concerns. The purpose of such concerns is to enable implementation of consistency policies. These policies ensure that the behavior of components in the system satisfies network-wide policies. Several HyperCast modules use implicit invocation to notify clients of key events. The protocol module, for example, announces events for some transitions in the state machine that implements the self-organizing behavior of HyperCast. The Service module announces end-to-end service-related events useful for maintaining desired quality-of-service (QOS).

The logging concerns in Hypercast are integral part of its core functionality. This is different from traditional AOP examples where logging/tracing typically refers to recording of desired join points in the execution of the program for such development purposes as debugging, bug reproduction, etc. These concerns, for example, help clients of Hypercast compute domain-specific metrics related to the performance of the overlay network. A careful study of the logging code in HyperCast revealed three sub-concerns: logging of informational messages, of raised exceptions, and of errors that do not raise exceptions.

We thus inferred the following additional concerns: *Information logging, Exception logging, Non-exception error logging, State machine events,* and *Service events.* Each concern leads to specification and corresponding implementation decisions, which we model as design variables. For example, one must specify a set of supported underlay networks. This information is passed to the Adapter developer who then produces an implementation. Figure 9 presents a DSM showing that design structure in terms of these design variables. We view the specifications as environment parameters. They are grouped as the first major module, in the block on the upper left. We distinguish the crosscutting concerns from non-crosscutting concerns. For example, design decisions corresponding to socket impl, exception logging impl and non-exception logging impl are distinguished from each other. Similarly, design decisions corresponding to monitor impl, exception logging impl, and non-exception logging impl are also distinguished. Implementation decisions are grouped in the lower right.

Figure 9 is a DSM that shows the design structure of HyperCast. In terms of these large blocks, we have a classic hierarchical (lower triangular) structure: Specification precedes implementation. The implementation block, by contrast, is classically modular (block



		01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
protocol spec	01	·																																				
service spec	02		·																																			
socket spec	03			·																																		
monitor spec	04				·																																	
adapter spec	05					·																																
event spec	06						·																															
protocol event policy	07							·																														
service event policy	08								·																													
info logging policy	09									·																												
exception logging policy	10										·																											
non_exception logging policy	11											·																										
protocol interface	12												·																									
service interface	13													·																								
socket interface	14														·																							
monitor interface	15															·																						
adapter interface	16																·																					
event interface	17																	·																				
protocol impl	18	x											x		x			x	·	x	x	x	x															
protocol event impl	19							x											x	·	x	x	x															
info logging impl	20									x									x	x	·	x	x															
exception logging impl	21										x								x	x	x	·	x															
non_exception logging impl	22											x							x	x	x	x	·															
service impl	23		x											x	x			x						·	x	x	x	x										
services event impl	24								x															x	·	x	x	x										
info logging impl	25									x														x	x	·	x	x										
exception logging impl	26										x													x	x	x	·	x										
non exception logging impl	27											x												x	x	x	x	·										
socket impl	28			x											x														·	x	x							
exception logging impl	29										x																		x	·	x							
non exception logging impl	30											x																	x	x	·							
monitor impl	31				x								x	x	x	x	x															·	x	x				
exception logging impl	32										x																					x	·	x				
non exception logging impl	33											x																				x	x	·				
adapter impl	34					x									x		x																		·	x	x	
exception logging impl	35										x																								x	·	x	
non_exception logging impl	36											x																							x	x	·	
event impl	37						x	x	x									x																				·

Fig. 9. Basic Object-Oriented Design of HyperCast.

diagonal). Once the specifications are fixed, the implementations can be developed and changed independently. However, each implementation module exhibits serious scattering and tangling internally due to the influences of the crosscutting concerns. Tangling is seen by reading rows. For example, the protocol implementation module depends on the protocol specification but also on that of protocol events and each of the logging concerns. Scattering is seen by reading columns. Changes in the exception logging policy (parameter 10), for example, would likely effect every implementation module in the system. We consider here that the crosscutting decisions identified by parameters 7–11 (i.e., the crosscutting concerns) to be relatively unstable and thus subject to change, making future changes harder. A more modular design would decouple these decisions from the base code. In Section 5.2 we consider reversing the direction of the dependencies, by using the oblivious method. In Section 5.3 we decouple the dependencies altogether, by using XPIs.

5.2 AO HyperCast using the Oblivious Method

What if the original HyperCast system was designed in an AO manner, using the oblivious method? To realize this possibility we refactored HyperCast into an oblivious aspect-oriented design. First we identified the scattered and tangled presence of the two crosscutting concerns (that is, logging and implicit invocation) and removed them from the code. Then, we re-introduced the concerns using aspects. We assume that the developers could *plausibly* have written essentially the same OO code, just leaving out the scattered parts that implemented the crosscutting concerns.[9] We localized the now-removed fragments (or functional equivalents) into these new aspect modules: *ao-protocol-events*, *ao-service-events*, *ao-info-logging*, *ao-exception-logging*, and *ao-non-exception-logging*. With implicit invocation now replaced by aspect-oriented advising, we also eliminated the entire OO *events* module. Writing aspects result in these fragments being woven back into the base code at the places from which it had been removed. In this way—by using AOP advicing—we are able to change the direction of the "knows about" relation, so that now the aspects are aware of the actions in the base code, but the case code is now oblivious to the functioning of the aspects.

In writing aspects that would result in these fragments being woven back into the base code we encountered a number of issues, which we describe below. We give code for each example in its original form to convey what behavior is required and where.

We studied the HyperCast code to try to characterize the join points that we would have to advise in order to weave the extracted code fragments back into the system. We found out that the context of these join points could be placed into one of the following four categories:

1. Private Join Points. In many cases, fragments had to be woven where there were no public join points (e.g., calls to public methods). In some cases, code had to be woven into nested switch- or if-statements, but join points were available, such as setting of a data member. To identify these join points, we often used the **`set`** and **`withincode`** pointcut designators. Figure 10 is an example. The pointcut works, but tightly couples the aspect to hidden details that the base code developer is free to change. If the developer changes the field name, `MyLogicalAddress`, the aspect is will not compile, and the aspect must be rewritten. In other words, the base code change is non-modular. Unless the base code developer can wait for the aspect developer to discover that the application no longer compiles, the base code developer will need to either make the change him or herself or notify the aspect developer. Here we highlight either the lack of obliviousness of the base code developer or the potential for high coordination costs, and chaotic, continual introduction of incompatibilities (bugs) into code.

```
pointcut HCLogicalAddrChanged(HC_Node node):
  set(I_LogicalAddress HC_Node.MyLogicalAddress)
  && (withincode(void HC_Node.messageArrivedFromAdapter(I_Message))
   || withincode(void HC_Node.timerExpired(Object))
   || withincode(void HC_Node.resetNeighborhood()))
  && target(node);
```

Fig. 10. Example pointcut that must know hidden details to function properly.

2. State/Point Separation. In many cases, the setting of a variable of interest and the join point at which weaving is needed are separated, and the given variable is not accessible to advice through the AspectJ join-point model. For example, we need to access an IP address error at certain place in order to construct a log message. The required value is stored in the local variable `addrStr`, which advice cannot access. It is computed earlier by an inlined block of code, which is neither governed by the same nesting of switch- and

[9] Under the set of all OO programs our assumption would likely be unsound, but we have high confidence in this assumption under the actual circumstances of the HyperCast system itself.

if-statements as the logging code, nor solely reserved for use by the logging concern (see Figure 11). There are a few possible ways to capture the IP address. Under one option a

```
String addrStr;
InetAddress ipAddr
if (AddrString != null) {
  addrStr = AddrString;
}
else {
  String addrType = config.getStringProperty(PROPERTY_NAME_PREFIX
    + ".PhyAddr", "INETV4AndTwoPorts");
  addrStr = config.getStringProperty(PROPERTY_NAME_PREFIX + "." + addrType
    + ".Address", ":0:0");
}
String[] paFields = addrStr.split(":");
...
for (int i = 0; i < paFields.length; i++) {
  paStr[i] = paFields[i].trim();
}
if (paFields.length > 3) {
  // logging code
  config.err("Str␣" + addrStr + ":␣invalid␣format␣for␣a␣physical␣address");
}
```

Fig. 11. An example of poor exposure in the base code of needed values.

two-stage advising sequence could be programmed, in which one advice advises calls to `config.getStringProperty` to save the IP address, and a separate advice advises the logging join point. Not only is this possibly computationally costly, but its complexity is sensitive to both whether the local variable computation dominates the logging statement and whether such a sequence could be nested (which would require a stack to capture all the relevant state). As another option the aspect developer could perhaps write a method in the aspect to compute the IP address from scratch. This would introduce unwanted scattering of IP address computations: in essence, base code is being copied into the aspect code. Finally, stepping outside the options available to the developer, the join point model of AspectJ could be extended for access to local variables. However, this approach would only exacerbate the coupling problems observed in category 1.

3. Inaccessible Join Point. This category includes join points within nested switch- and if-statements, where there is no proxy join point to advise. The check-and-branch sequence alone defines the join point.

Figure 12 is an example, where `notifyApplication` notifies the application of certain events. We want to replace the use of event notification with advising. The AspectJ JPM does not provide visibility into branches taken by a program. The solution of recreating these conditionals in the advice body only scatters the concern, would make advice complex and hard to understand, and might not always be feasible.

4. Quantification Failure. Many join points that have to be advised in the same way cannot be captured by a quantified PCD, e.g., using wild-card notations. A separate PCD is required for each join point. There were about 180 places in the base code where logging was required. Most of the join points do not follow a common pattern. Not only is there a lack of meaningful naming conventions across the set of join points, but also variation in syntax: method calls, field setting, etc. One failure mode here is that creating many PCDs is costly. More seriously, extensions to the base code that should be logged will require

```
switch (MyState) {
 case WaitforACK: {
  switch (e.getType()) {
    case FULL_E2E_ACK: {
      processAck(msg.getSourceAddress());
      if (ACKExpected.isEmpty()) {
        MyState = Done;
        MStore.setTimer(...);
        /* Notification concern - removed
        if (mylogicaladdress == root) {
          notifyApplication();
        }
        */
     }
    }
    break;
    case // ...
}
```

Fig. 12. An example of “invisible” branches that aspects cannot advice nor even describe.

the logging pointcut to be updated to capture the new logging points. If the pointcut is not updated, it will silently malfunction, as the non-advising of a join point does not manifest a syntax or type error that can be reported at compile time. In this case, then, it is imperative that the base code developer communicate the changes to the aspect developer.

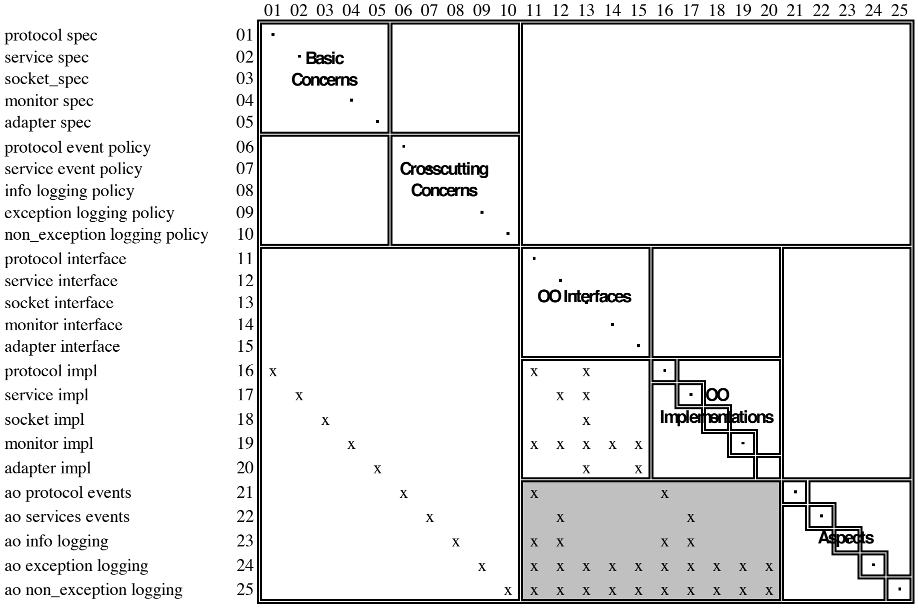
The common theme that runs through all of these problems is that the oblivious approach might simplify the task of the base code designer, but it can significantly complicate overall system development because there is no agreement between the base and aspect code developers about how their respective parts will be integrated. Rather, the base code developer proceeds to implement code, sometimes making arbitrary decisions that otherwise could have been better informed, and the resulting design decisions then dictate the conditions to which the aspect developers must conform.

In a nutshell, the oblivious design process is a hierarchical process (and thus must be sequentialized): Specifications are provided for base and aspect code modules. Then the base code is written. Finally the aspect code is written under the constraints imposed by both the external specifications and the coding decisions made by the base developer.

Figure 13 presents a DSM for the oblivious AO version of HyperCast and highlights this hierarchical design structure. The base code modules no longer depend on the crosscutting concerns, but the aspect modules do strongly depend on the base implementation modules. The highlighted region in this DSM show the strong dependence. The cell for (row 21, column 16), for example, indicates that *ao-protocol-event* depends on *protocol impl*. This dependence arises because the PCD of the aspect module depends on the form of the join points that signal protocol events in the base code, entirely under the control of an oblivious base code designer.

5.3 AO HyperCast using XPIs

In this section we provide an alternative AO design of HyperCast that uses the XPI-oriented approach instead of the oblivious approach. To achieve such a design we first started with the experimental version of HyperCast, which had the scattered and tangled logging and implicit-invocation event concerns removed, and then refactored this base code to follow the contracts of newly designed XPIs. Finally, the logging and event concerns were reintroduced as aspects that were clients of the XPIs. Section 5.3.1 discusses the process


Basic Concerns
Crosscutting Concerns
OO Interfaces
OO Implementations
Aspects

		01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
protocol spec	01	·																								
service spec	02		·																							
socket_spec	03																									
monitor spec	04				·																					
adapter spec	05					·																				
protocol event policy	06						·																			
service event policy	07																									
info logging policy	08																									
exception logging policy	09									·																
non_exception logging policy	10										·															
protocol interface	11											·														
service interface	12												·													
socket interface	13													·												
monitor interface	14														·											
adapter interface	15															·										
protocol impl	16	x										x		x			·									
service impl	17		x										x	x				·								
socket impl	18			x										x												
monitor impl	19				x							x	x	x	x	x				·						
adapter impl	20					x								x		x										
ao protocol events	21						x					x					x					·				
ao services events	22							x					x					x					·			
ao info logging	23								x			x	x				x	x								
ao exception logging	24									x		x	x	x	x	x	x	x	x	x	x				·	
ao non_exception logging	25										x	x	x	x	x	x	x	x	x	x	x					·

Fig. 13. Obliviousness Aspect-Oriented Design of HyperCast.

of identifying the crosscutting interfaces to expose in the base code and Section 5.3.2 discusses the process of making the base code achieve the obligations of those interface contracts.

5.3.1 *XPIs in HyperCast.* XPIs capture abstractions that are inherently crosscutting. In the design of XPIs for HyperCast, we focused on what abstractions would be useful to separate implementation details of the logging and event concerns while at the same time being relatively stable. The process was to identify several sets of "interesting states" of the HyperCast system. These are essential and stable states in that they represent some important abstract events in the base concerns.

The first set of interesting states we identified is the error states. Any system has abnormal states in execution and may use different mechanism to handle them. In HyperCast different components can cause different errors and are handled in different ways. For example, some errors lead to thrown exceptions, while some errors are handled internally. We defined a crosscutting interface, named `XErrorState`, to expose these error states (see Figure 14). The `Exceptions` PCD exposes all error states that cause thrown exceptions. The `NoExcptErrors` PCD exposes errors that are handled internally or ignored. This XPI governs join points over the whole system, as shown by the use of the top-level package name in its `staticscope` pointcut. Thus, this XPI abstracts all abnormal states throughout the system, against which clients of the XPI can code.

The key abstraction of any HyperCast protocol is "the overlay node runs a finite state machine which performs actions when timers expire and when messages are received" [HyperCast]. And the transition of the FSM is a function of 4-tuple: (node state, neighborhood,

```
public aspect XErrorState {
//@ exposes (* all abnormal states that throw exceptions. *)
 public pointcut Exceptions(Exception e): handler(Exception+) && args(e);

//@ exposes (* all abnormal states that dont throw exceptions but still get handled.*)
//@ requires (* all non-exception errors are handled by methods that match this PCD.*)
 public pointcut NoExcptErrors(int errno):
  execution (* *.*ErrorHandler(int,..)) && args(errno);

 public pointcut staticscope(): within(edu.virginia.mngroup.hypercast..*);
}
```

Fig. 14. The XErrorState XPI.

message type, time). Each overlay node has a state, maintains a neighborhood table that contains a list of its neighbors in the overlay network and listens to overlay messages from overlay adapters. Computing the right state transition is the main concern of HyperCast. The finite state machine is an abstraction of any possible concrete implementation of the protocols. The state transition, address shifting and topology management of the abstract protocol FSM is defined in the HyperCast specification and is independent of any implementation. We defined the `XProtocolFSM` XPI to expose all protocol statement transitions (see Figure 15).

The `XProtocolFSM` XPI exposes five PCDs. The first three PCDs expose join points related to state, address and neighborhood change events, which can trigger transition in the protocol finite state machine. The last two PCDs expose two important events: join overlay and leave overlay.

We also defined an XPI for services provided by the overlay socket. HyperCast provides a set of services, such as naming and acknowledgment services, which use events as a way to interact with client applications. The protocol events are not essential to the performances of the protocol, but some events are of interest to third-party clients. The events service is used to implement essential parts of other components. For example, a client of the HyperCast library can request a name translation for a remote node by calling a method exposed by a service API. The implementation of this method uses a HyperCast control message to request the translation from the remote node; the answer is returned internally by a control message in response; the result is then passed back to the HyperCast library client by an event notification. Thus, event notifications are used in this case not just to enable observers to monitor service execution, but as an essential part of the service API itself. The services implementation follows its own finite state machine. To properly abstract this model we should expose all service states, particularly the final states, where the event is notified in the original OO implementation. We also need to expose the result of a service. Because different type of services gives different type of results, we need to expose the results differently. Figure 16 shows the `XMessageStoreFSM` XPI that exposes these transitions.

5.3.2 *Implementing Crosscutting Concerns with XPIs.* In our experiment, we refactored the HyperCast base code to make it adhere to the contracts of all of the new XPIs. We modified 65 places in the protocol module to follow `XProtocolFSM`. Most modifications are within 6 classes, especially `MessageArrivedFromAdapter`. We modified 21 places in service module classes to follow `XMessageStoreFSM`. All modifications are simple, as the finite state machines were already implemented and we only used meth-

```
public aspect XProtocolFSM {
/*@ exposes (* all and only abstract state transitions of the protocol
  @   finite state machine. All I_Node state transitions be implemented
  @   by calls to corresponding setState(byte) method.*)
  @ requires (* advisors of this XPI may not change any states of any
  @   overlay nodes directly or indirectly. *) */
 public pointcut StateUpdate(byte s): call(void I_Node+.setState(byte)) && args(s);

/*@ exposes (* all and only logical address update of an overlay node.
  @    We require that all logical address be implemented by calls to
  @    corresponding setMyLogicalAddress(I_LogicalAddress) method. *)
  @ requires (* advisors of this XPI may not change any logical address
  @     of any overlay nodes directly or indirectly. *) */
 public pointcut LogicalAddressUpdate(I_LogicalAddress addr):
  call(void *.setMyLogicalAddress(I_LogicalAddress)) && args(addr);

/*@ exposes (* all and only neighborhood changes of an overlay node.
  @   We require that all logical address be implemented by calls to
  @   methods that match this PCD, and we assume that any such call
  @   causes changes to neighborhood information of overlay nodes. *)
  @ requires (* advisors of this XPI may not change any neighborhood
  @   information of any overlay nodes directly or indirectly. *) */
 public pointcut TopologyUpdate(): call(* *.NeighborHood*(..));

/*@ exposes (* leaving and joining overlay behaviors of any overlay node.
  @   We require that all nodes call method leaveOverlay to leave an
  @   overlay and joinOverlay method to join an overlay. *) */
    public pointcut LeaveOverlay(): call(void I_Node+.leaveOverlay());
    public pointcut JoinOverlay(): call(void I_Node+.joinOverlay());

    public pointcut staticscope():
      within(edu.virginia.mngroup.hypercast.DT..*)
      || within(edu.virginia.mngroup.hypercast.HC..*)
      || within(edu.virginia.mngroup.hypercast.SPT..*)
      || within(I_Node+);
  }
```

Fig. 15. The XProtocolFSM XPI.

```
public aspect XMessageStoreFSM {
/*@ exposes (* all and only abstract state transitions of the
  @   message store finite state machine. We require that all
  @   message store state transitions be implemented by calls
  @   to corresponding setState(byte) method. *)
  @ requires (* advisors of this XPI may not change any states
  @   of any message store directly or indirectly. *) */
 public pointcut StateUpdate(byte s):
  call(void I_MessageStoreFSM+.setState(byte)) && args(s);

 public pointcut staticscope():
  within(edu.virginia.mngroup.hypercast.I_MessageStore+);
}
```

Fig. 16. The XMessageStoreFSM XPI.

ods with naming convention to expose them. We also cleaned the old event notification method, which occurred in 41 places across 10 classes. For XErrorState, we first identified 14 error types according to the specification. We then created a template to implement error handler methods for non-exceptional errors. The refactoring involves 55 error occurrences across 18 classes.

After refactoring, we wrote aspects to implement logging and event notification and handling concerns. Table I shows their sub-concerns and corresponding XPIs they advised.

Crosscutting Concerns	**Pointcuts**
Exception Logging	`XErrorState.Exceptions`
Non-Exceptional Error Logging	`XErrorState.NoExceptErrors`
Info Logging and Tracing	`XProtocolFSM` (all PCDs)
Protocol Event Notification	`XProtocolFSM` (all PCDs)
Service Event Notification	`XMessageStoreFSM`

Table I. Crosscuttting concerns in HyperCast abstracted by XPIs and corresponding pointcuts.

5.3.3 *Extending HyperCast.* An important task in software evolution is introducing new features into a system. In this section we discuss a hypothetical extension to HyperCast to implement a traffic monitoring subsystem. Clearly the new feature should be designed against the abstraction of communication present in HyperCast. The communication concern crosscuts multiple modules in HyperCast and there are no existing XPIs provided for this abstraction. Thus, a new XPI that abstracts and exposes communication join points could serve to decouple the new traffic monitoring feature from the rest of the HyperCast system.

Although it is easy to identify the communication points it is not immediately obvious how to map this abstraction to the join points in the HyperCast base code. One challenge is that HyperCast provides a layered network system: An overlay socket provides communication primitives for applications; and an adapter abstracts physical network adapters with corresponding communication primitives. But from an implementation perspective all communication is implemented using Java sockets.

An overlay socket has two adapters: one for overlay messages and one for application messages. Although overlay messages are transparent to HyperCast clients, they are useful in traffic analysis and should be visible to a traffic monitor. The multiple-layered structure suggests multiple solutions in exposing the join points. The join points for communication between overlay sockets, adapters and Java sockets can be mapped into HyperCast communication abstractions. But each set of join points impose different constrains. Overlay socket communication provides the highest level of abstraction. So selecting this set of join points might achieve maximum information hiding. But as noted before, overlay messages are transparent to API clients and the overlay socket interface does not provide communication primitives for overlay messages. Overlay messages and their primitives also vary from protocol to protocol. It is hard to expose these points without revealing some protocol detail. However, only exposing application data communication may restrict possible clients of this abstraction. On the other hand, join points mapped to Java sockets depend only on the standard Java interface. This lowest-level mapping restricts the implementation of HyperCast by using only Java sockets. One solution is to abstract adapter communication, exposing the sending and receiving of application data and overlay messages. The `XHyperCastOverlayTraffic` XPI (Figure 17) exposes two events: the sending and receiving of overlay control messages. The `XHyperCastAppTraffic` XPI (Figure 18) exposes the corresponding events associated with application data messages.

These two XPIs provide an abstraction for clients to design aspects related to HyperCast network communication. In addition to a traffic monitoring subsystem, it can also be an accounting subsystem to calculate billing information based on network traffic of each

```
// Protocol message
public aspect XHyperCastOverlayTraffic {
/*@ exposes (* all protocol message sending. We require that all
  @    protocol messages are type of I_MultiProtocolMessage. We also
  @    require that all unicast messages are sent by adapters
  @    sendUnicastMessage method and all multicast messages are sent
  @    by adapters sendMulticastMessage method. *)
  @ requires (* advisors of this PCD are not allowed to change
  @    the message parameter. *) */
 public pointcut ProtocolMsgSend(I_Message msg):
  ProtocolMulticastSend(I_Message) && ProtocolUnicastSend(I_Message) && args(msg);

/*@ exposes (* all protocol message receiving. We require that all
  @    protocol messages are type of I_MultiProtocolMessage. We also
  @    require that an overlay node processes protocol message via
  @    callback method messageArrivedFromAdapter(). *)
  @ requires (* advisors of this PCD are not allowed to change the
  @    message parameter. *) */
 public pointcut ProtocolMsgReceive(I_Message msg):
  execution(* I_AdapterCallback+.messageArrivedFromAdapter(I_Message))
  && args(msg) && if(msg instanceof I_MultiProtocol_Message);

 public pointcut staticscope(): within(I_Adapter+);

 // private components of the XPI
 private pointcut ProtocolUnicastSend(I_Message msg):
  execution(* I_UnicastAdapter+.sendUnicastMessage(I_NetworkAddress,I_Message))
   && args(I_NetWorkAddress, msg)
   && if(msg instanceof I_MultiProtocol_Message);

 private pointcut ProtocolMulticastSend(I_Message msg):
  execution(* I_MulticastAdapter+.sendMulticastMessage(I_Message))
  && args(msg) && if(msg instanceof I_MultiProtocol_Message);
}
```

Fig. 17. The HyperCastOverlay Control Communication XPI.

client. These two XPIs constrain the HyperCast implementation through naming conventions for join points and through method calling conventions. For example, in HyperCast message arrival processing is handled by a callback from an adapter to an upper-layer node or overlay socket. The XPIs made this convention explicit through the new design rules they introduced. Information regarding exceptions is also made explicit in the XPIs. They are crucial to the semantics of exposed join point and may be important to clients if they distinguish successful sending/receiving and failures.

The traffic monitoring subsystem is easy design in the presence of the XPIs. A sample monitor in Figure 19 simply counts the number of packets and bytes for each kind of message. In this example, the aspect `TrafficMonitor` is defined as **`perthis`**, which is an aspect-instance qualifier. “If an aspect is defined perthis(Pointcut), then one object of the aspect is created for every object that is the executing object (i.e., "this") at any of the join points picked out by Pointcut [AspectJ]”. As a result, an instance of the `TrafficMonitor` is created for each instatiated adapter. This toy example shows that the XPIs serve their purpose and can guide a more advanced monitoring system (e.g., one with deeper analysis or a user interface).

5.4 Comparative Analysis of the Two HyperCast Designs

In this section we analyze the XPI-based AO HyperCast design and compare it to the oblivious design. We first present the XPI-based solution’s design structure and suggest

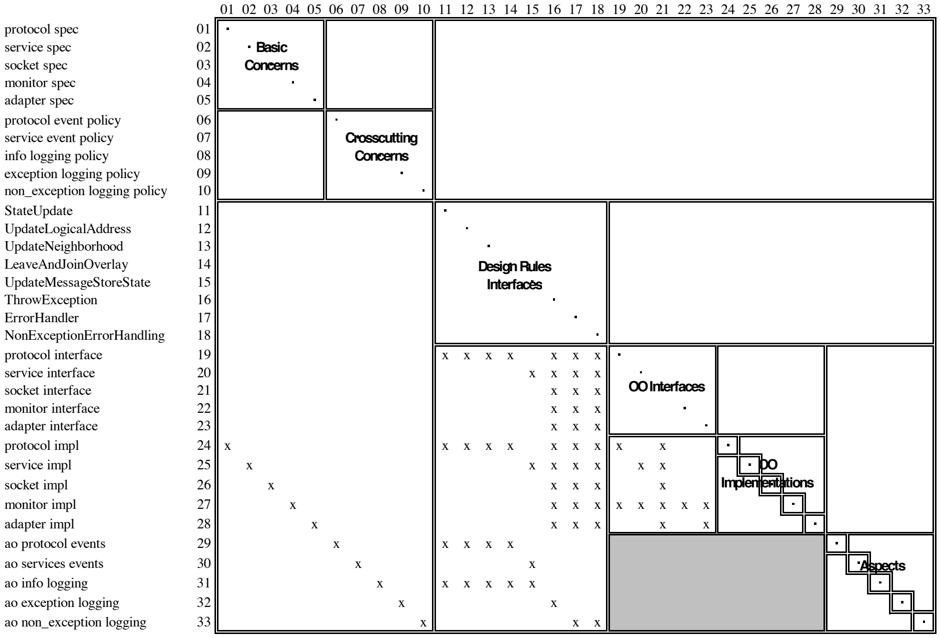
```
// Application message
public aspect XHyperCastAppTraffic {
/*@ exposes (* all application message sending. We require that
  @  all protocol messages are type of I_OverlayMessage. We also require
  @  that all unicast messages are sent by adapters sendUnicastMessage()
  @  method and all multicast messages are sent by adapters
  @  sendMulticastMessage() method. *)
  @ requires (* advisors of this PCD are not allowed to change the
  @  message parameter. *)
 public pointcut AppMsgSend(I_Message msg):
  AppUnicastSend(I_Message) && AppMulticastSend(I_Message) && args(msg);

/*@ exposes (* all application message receiving. We require
  @  that all protocol messages are type of I_OverlayMessage. We also
  @  require that a overlay socket processes application message via
  @  callback method messageArrivedFromAdapter(). *)
  @ requires (* Advisors of this PCD are
  @  not allowed to change the message parameter. *) */
 pointcut AppMsgReceive(I_Message msg):
  execution(* I_AdapterCallback+.messageArrivedFromAdapter(I_Message))
  && args (msg) && if(msg instanceof I_OverlayMessage);

 public pointcut staticscope(): within(I_Adapter+);

 // private components of the XPI
 private pointcut AppUnicastSend(I_Message msg):
  execution(* I_UnicastAdapter+.sendUnicastMessage(I_NetworkAddress,I_Message))
  && args(I_NetWorkAddress, msg) && if(msg instanceof I_OverlayMessage);

 private pointcut AppMulticastSend(I_Message msg):
  execution(* I_UnicastAdapter+.sendMulticastMessage(I_Message))
  && args(msg) && if(msg instanceof I_OverlayMessage);
}
```

Fig. 18. The HyperCast Application Communication XPI

```
public aspect TrafficMonitor perthis(adapterinit()) {
  int AppSendP, AppRevcP;
  int AppSendB, AppRevcB;
  int OverlaySendP, OverlayRevcP;
  int OverlaySendB, OverlayRecvB;

  pointcut adapterinit(): execution(I_Adapter.Adapter_Initialization());

  // only a successful send will be counted
  after(I_Message msg) returning: XHyperCastOverlayTraffic.ProtocolMsgSend(msg) {
    OverlaySendP++;
    OverlaySendB += msg.toByteArray().length;
  }

  // only a successful receive will be counted
  after(I_Message msg) returning: XHyperCastAppTraffic.AppMsgRecv(msg) {
    AppRevcP++;
    AppRevcB += msg.toByteArray().length;
  }

  //...
}
```

Fig. 19. A simple traffic monitoring system using the new XPIs.

intuitions to show that writing the logging and event notification aspects is more easily done with XPIs. Then we give a quantitative analysis computing the net option value to



		01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
protocol spec	01	·																																
service spec	02		·																															
socket spec	03			·																														
monitor spec	04				·																													
adapter spec	05					·																												
protocol event policy	06						·																											
service event policy	07							·																										
info logging policy	08								·																									
exception logging policy	09									·																								
non_exception logging policy	10										·																							
StateUpdate	11											·																						
UpdateLogicalAddress	12												·																					
UpdateNeighborhood	13													·																				
LeaveAndJoinOverlay	14														·																			
UpdateMessageStoreState	15															·																		
ThrowException	16																·																	
ErrorHandler	17																	·																
NonExceptionErrorHandling	18																		·															
protocol interface	19											x	x	x	x		x	x	x	·														
service interface	20															x	x	x	x		·													
socket interface	21																x	x	x			·												
monitor interface	22																x	x	x				·											
adapter interface	23																x	x	x					·										
protocol impl	24	x										x	x	x	x		x	x	x	x		x			·									
service impl	25		x													x	x	x	x		x	x				·								
socket impl	26			x													x	x	x			x					·							
monitor impl	27				x												x	x	x	x	x	x	x	x				·						
adapter impl	28					x											x	x	x			x		x					·					
ao protocol events	29						x					x	x	x	x															·				
ao services events	30							x								x															·			
ao info logging	31								x			x	x	x	x	x																·		
ao exception logging	32									x							x																·	
ao non_exception logging	33										x							x	x															·

Fig. 20. XPI-based Aspect-Oriented Design of HyperCast.

demonstrate that software systems can attain better modularity (than ad hoc AO and OO) using the XPI approach.

5.4.1 *Simple Comparison.* Figure 20 shows how the XPIs decouple the base code and the aspects from each other. Compared to the oblivious design's DSM in Figure 13, we observe that all dependences between the basic modules and aspect modules are removed. Instead, both the aspect modules and the basic modules now depend on the design rules dictated by the XPIs.

We now compare the aspects implemented over the two different versions of the base code. In the oblivious AO design, the aspect modules are an average 240 lines each in length, whereas for the XPI-based AO design the aspects average only 30 lines each for the same functionality. For example, Figure 21 shows the aspects for handling logical address events, using the two approaches. We observe that the aspect in the XPI-based design rules approach is much simpler because specific naming conventions were followed and interesting abstract states were implicitly exposed in all protocol modules. Without design rules, the aspect had to compute complex pointcuts by going through implementation details of the base code. Moreover, the design-rule pointcut can capture newly-coded address changes, because it is quantified and base code designers are constrained to write new code following the design rules.

5.4.2 *Quantitative Analysis with Net Option Value.* Sullivan et al. [2001] and Lopes and Bajracharya [2005], have previously used *net option value* analysis [Baldwin and Clark 2000] to quantitatively compare software designs modeled by DSMs. In this section, we

```
privileged aspect EventTest {
  // have to compute pointcuts for each protocols
  pointcut DTLogicalAddrChanged():
    execution(* DT_Neighborhood.DT_randomShiftMyCoordinates())
    || execution(* DT_Neighborhood.updateNodeAddress(DT_LogicalAddress));

  pointcut HCLogicalAddrChanged():
    set(I_LogicalAddress HC_Node.MyLogicalAddress)
    && (withincode(void HC_Node.messageArrivedFromAdapter(I_Message))
        || withincode(void HC_Node.timerExpired(Object))
        || withincode(void HC_Node.resetNeighborhood()));

  pointcut SPTLogicalAddrChanged():
   execution(* SPT_Node.setLogicalAddress(I_LogicalAddress));

  // advice
  after(): DTLogicalAddrChanged() || ... {
    // handle address changes here
  }

  // for other events...
}
```

(a)

```
privileged aspect DREventTest {
  pointcut logicalAddrChanged(I_LogicalAddress):
    execution(* I_Node+.setMyLogicalAddress(I_LogicalAddress));

  after(): logicalAddrChanged() {
    // handle address changes here
  }

  // for other events...
}
```

(b)

Fig. 21. Aspects for LogicalAddressChanged Events: without design rules (a), and with XPI-based design rules (b).

evaluate the HyperCast system similarly based on the DSMs introduced in previous sections.

Baldwin and Clark's theory is based on the idea that modularity provides a portfolio of options. They define a model for reasoning about the value added to a system by its modularity. Splitting a design into N modules increases its base value S_0 by a fraction obtained by summing the net option values (NOV_i) of the resulting options. NOV is the expected payoff of exercising a search and substitute option optimally, accounting for both the benefits and cost of exercising options:

$$\begin{aligned} V &= S_0 + NOV_1 + NOV_i + ... + NOV_m \\ NOV_i &= \max_{k_i}\{\sigma_i n_i^{1/2} Q(k_i) - C_i(n_i) \cdot k_i - Z_i\} \end{aligned}$$

For module i, $\sigma_i n_i^{1/2} Q(k_i)$ is the expected benefit to be gained by accepting the best positive-valued candidate generated by k_i independent experiments. $C_i(n_i) \cdot k_i$ is the cost to run k_i experiments as a function C_i of the module complexity n_i. $Z_i = \Sigma_{j\ \mathrm{sees}\ i} c n_j$ is the cost of ripple effects of changes due to module dependences. The max picks the experiment that maximizes the gain for module i. Details of the NOV model can be found in the literature [Baldwin and Clark 2000; Sullivan et al. 2001; Lopes and Bajracharya 2005]. The two most important parameters for NOV analysis are *technical potential*, σ,

Basic σ	1%	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%	99%
OO NOV	0	0	0	0.0016	0.077	0.17	0.31	0.50	0.78	1.09	1.26	1.402
Ob. AO NOV	0.537	0.483	0.423	0.375	0.375	0.41	0.49	0.62	0.83	1.10	1.255	1.391
% impr.	N/A	N/A	N/A	2179.5	386.8	141.18	58.06	24.00	6.41	0.92	-0.40	-0.78
XPI AO NOV	0.537	0.483	0.423	0.385	0.389	0.42	0.51	0.66	0.87	1.14	1.29	1.422
% impr.	N/A	N/A	N/A	2244.6	404.7	147.06	64.52	32.00	11.54	4.59	2.38	1.43

Table II. Net option values for the OO, oblivious AO, and XPI-based AO designs. Each basic column represents the assumption that the percentage of requested software improvements is confined to the base functionality (versus the aspect functionality) is σ.

and *complexity*, n.

Technical potential is the expected variance on the rate of return on an investment in producing a variant of a module implementation—that is, risk with commensurate rewards for success. On the assumption that the prevailing implementation of a module is adequate, the expected rate of return on investments is proportional to changes in requirements that drive the evolution of the module's specification. Consequently, a module whose specification does not change has low technical potential. For the evaluation of the benefits of the aspect-oriented designs, we operationalize technical potential as a stream of incoming change requests, with a given percentage of the changes impinging on the base modules, and the rest on the aspect modules. (We do not consider changes that crosscut between base functionality and the aspects.) We do not expect to see large advantages in the aspect-oriented designs, because only two crosscutting concerns have been modeled compared to 20 base concerns. Numerous other crosscutting concerns would need to be modeled for a considerable incremental value to be observed.

The standard method for estimating the complexity parameter for each module is to use the size of the artifact as a proportion of the overall system. Recognizing that lines of code (LOC) can include disproportionate inessential complexity for any given module, we decided to minimize this effect by using the LOC of the smallest version of each module amongst our three systems. (Recall that the oblivious aspects were on average eight times larger than the XPI aspects.) The XPI-based version of the modules in this system invariably produced this effect, so we used its sizes in computing the proportions.

With these two estimations in hand, we computed the NOV for each design for several scenarios, from a low change rate of the base functionality relative to the aspects to a high change rate relative to the aspects. Table II presents our results. The precise values in the table cannot be taken as absolute truth. It remains an open challenge to justify precise estimates for real options in software design. As discussed in our earlier work, estimating technical potential is difficult [Sullivan et al. 2001]. However, as a back-of-the envelope model, it provides ballpark figures and useful insights.

We first observe that the more likely it is for the crosscutting concerns to change (the left side of the table), the more relative value is added by either AO design. The value of the modularity in the OO design goes to zero because its modularity is unused. The value of the two AO designs similarly converges to the same value, because they only differ in how they relate to the base code.

Towards the right of the table, as the relative change rate of the base functionality goes above 90%, the oblivious design becomes even worse than the OO design. This is because the oblivious design's aspects are dependent on the base modules, which are changing at a high rate, thus producing a high Z.

In the middle to upper middle part of the table, presumably closer to where the relative base change rate is likely to lie, we see that the XPI-based design outperforms the oblivious design, entirely because its Z is zero when the base functionality evolves, due to the join point design rules. Both aspect-oriented designs outperform the OO design.

6. DISCUSSION

There is always some degree of crosscutting in any application, so the question is *how stable is the concern that crosscuts?* APIs are an example of high degrees of crosscutting in programs. For example, the API for the `String` class in Java likely crosscuts a significant portion of any large program. But we do not consider this crosscutting to be expensive, because the majority of the existing `String` API is expected to be stable.

Thus, it is not realistic to expect the elimination of all forms crosscutting (had we such expectations, we would need to eliminate all APIs as well). Instead, we should expect AOP technology to aid us in reducing the instable parts of what must crosscut, knowing at least that the parts that still crosscut will at least be less expensive, due to their stability. In the case of XPIs, it may well be the case that an XPI is no longer necessary, and all client aspects of that XPI can be deleted. Even when the XPI is removed, the base code will still follow the contracts dictated by the XPI (e.g., by following naming conventions), but there will no longer be the obligation to keep following those rules. This flexibility is similar to the case of no longer needing a special math library: if no module uses it, the library can be unlinked from the final build without cost. Similarly, when no aspects use an XPI, the XPI can be silently removed. Design rules create modularity by allowing only stable decisions to crosscut. But design rules are no panacea. The failure to identify crosscutting concerns and to establish appropriate XPIs during the design state is tantamount to depending on obliviousness.

6.1 XPI Design Guidelines

In determining design rules for new XPIs be sure the design rules are stable. Good opportunities for design rules are those that can work around weakness in the object model of the program. An XPI should result in easy-to-use join points that give base code designers considerable implementation freedom.

Similar to APIs, an XPI should have reasonably minimal assumptions on the base code and the XPI's clients. A good API lends itself to several different implementations and does not leak out inessential implementation details. A reusable API also does not limit itself to the particular needs of only one client when a similar level of abstraction could lend itself to more general use. For XPIs, then, we provide the dictum that the quantification of join points and the states at those points should be characterized in terms of the application's own concepts and abstractions, rather in terms of implementation-dependent aspect or base code details that are subject to change.

In HyperCast, for example, the design rules are best stated not in terms of the logging or notification aspects, but rather in terms of interesting abstract states and behaviors of the system, e.g., the abstract states of the finite state machine that tracks and manages the configuration and use of the overlay network. This reflective, application-centric view allows the base code designer to be oblivious to logging and other aspects, per se, and creates options for several possible aspect-oriented extensions to HyperCast such as mirroring and caching. Because XPIs are formulated in terms of the abstract system model, the resulting syntactic, naming and other constraints are consistent with base code's natural OO ontology. Although there is scattered work that has to be carried out to satisfy the rule, and the result is a crosscutting interface, we hypothesize that the result is practically indistinguishable from pure OO design.

7. RELATED WORK

Most of the 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. Some recent developments that address design more directly are relevant here.

Aspect-Aware Interfaces. Kiczales and Mezini recognized the need to introduce crosscutting interfaces between base-code and aspects. They defined a notion of *aspect-aware interfaces* (AAIs), in which aspects extend the interfaces of modules they advise [Kiczales and Mezini 2005]. Specifically, this approach computes aspects' dependencies on a system's join points and shows the dependencies as annotations on the explicit interfaces of advised code.

Revealing such dependences can support change and modular reasoning. A programmer can see how join points are being advised and prevent making changes that invalidate those uses. Prior to the emergence of stable modular interfaces (for example in Extreme Programming-style development [Beck 1999]), AAIs can serve as a valuable substitute—they inform, even if they do not decouple and abstract. Likewise, the cross-references that AAIs provide could help guide refactoring activities, perhaps resulting in XPIs.

Yet, AAIs do not 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. Also, support for modularity is limited. The display of dependences between existing code and PCDs cannot 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.

Open Modules. Aldrich, among others, has proposed language constructs—and by implication a design method—for module-based join point interfaces [Aldrich 2005]. Ongkingco *et al.* have since proposed an extension of the AspectJ language to support Open Modules [Ongkingco et al. 2006]. Open Modules expose only PCD-selected join points on private state. It enables 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, Open Modules let a module implementation evolve without requiring rework of aspects. However, the resulting interface is limited to crosscutting the module's implementation. It would be awkward to capture the broadly crosscutting concepts found in our HyperCast case study [Sullivan et al. 2005]. Also, Open Modules do not make clear the constraints that would have to be observed in writing new code to avoid inadvertently compromising the advising PCD semantics.

Quantified, Typed Events. Rajan and Leavens have recently proposed quantified, typed events as typed interfaces between the base code and the aspect code in their language Ptolemy [Rajan and Leavens 2008]. Event types in Ptolemy are declared separately from the base code and the aspect code and they make explicit the context information exchanged between the two. Events are explicitly and declaratively signalled without exposing the implementation details of the code. Furthermore, aspects could name event types in PCDs, which has the effect of quantifying over all events declared to be of that type, which solves the problem described above with Open Modules. The base and the aspect code can thus evolve independently.

Quantified, typed events are helpful towards anticipated evolution scenarios, where modification to the base code is permitted to announce events, however, they do not help much with scenarios where such modifications are not permitted. Language extensions and corresponding tool support is also needed, whereas we show that an XPI-based approach can be applied with existing languages.

Model-based Pointcuts. In Model-based Pointcuts [Kellens et al. 2006], PCDs are defined in terms of a conceptual model of the program. The advantage of this technique is that it is not coupled with lexical names, structures, or program traces. Their are two somewhat related disadvantages. First, developing a model of the program can be time consuming, second, as Kellens *et al.* also point out, keeping such model consistent with the source code can be challenging in the same manner as keeping artifacts such as requirements document, design document, etc consistent with the source code is known to be a challenging issue. On the other hand, XPIs are integral part of the source code and for the most part compilers verify their consistency during each build process.

Reasoning about AOP. A number of techniques have been proposed in the past decade that aim to facilitate change onto existing AOP systems by providing a reasoning framework. At a high-level objective often is to classify whether a change is innocuous. Earlier strand of such work can be seen in Clifton and Leavens's classification of aspects into what they call observers and assistants [Clifton and Leavens] followed by Krishnamurthi, Fishler, and Greenberg's work on verifying advice modularly [Krishnamurthi et al. 2004], Douence and Fradet and Südholt work on reasoning about stateful aspects [Douence et al. 2004], Dantas and Walker's harmless advice [Dantas and Walker 2006], Clifton, Leavens and Noble's type and effect system [Clifton et al.], Steimann's argument about the success of AOP [Steimann 2006], and most recently Ostermann's work on applying default logic to reason about aspects [Ostermann 2008]. Compared to these techniques, basic goals of our XPI-based strategy is to facilitate such decomposition of systems into aspects that is inherently easy to evolve.

Explicit Join Point Models. Another line of research that also aims to ease reasoning about aspects departs from the popular notions of AOP by explicitly labeling join points, with the hope that such labeled join points help with quantification and solve some problems that we have pointed out previous sections. This idea has appeared previously in SetPoint [Altman et al. 2005], in Model-based Pointcuts [Kellens et al. 2006], in Hoffman and Eugster's explicit join points [Hoffman and Eugster 2007], and in quantified, typed events [Rajan and Leavens 2008]. In SetPoint explicitly placed annotations are used for quantification. In Model-based Pointcuts, explictly created models, which express the relationship between names in the model and the program's structure, are used for quantification. Hoffman and Eugster allow explicit labeling of join points with Java annotation like syntax. Compared to these approaches, the novelty of our approach lies in: allowing the use of existing language models and features and a design strategy that has much of the similar benefits without any added cost of infrastructural development.

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 [Larochelle et al. 2003]. Dantas and Walker's AspectML provides advice access controls to a function definition's parameters, hence modifying the join-point signature of calls on the function [Dantas and Walker 2003]. As discussed in the next section, 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.

8. CONCLUSION

In his seminal paper on information hiding, Parnas showed the deleterious effects of scattered dependences on design decisions that are complex or likely to change [1972]. He showed that they make programs hard to understand, develop in parallel, and change at reasonable cost. These technical difficulties ultimately manifest themselves in ways that matter to people: poorer dependability, lower productivity, and fewer valuable choices. Parnas then showed designers how to do better using abstract interfaces to decouple decisions. In this paper, we have revisited Parnas's ideas in light of the addition of join point models and quantified (crosscutting) advising to the programmer's toolkit.

An AOP language like AspectJ implicitly publishes a vast array of join points for a given base program. Like the exposed data structures of Parnas's functional design, they create valuable opportunities for aspect designers. The problem is that, if not managed, they invite scattered dependences on unstable, complex design decisions. Oblivious aspect-oriented design, however, dictates that the base code designer should make no preparations for aspects. The difficulty, then, is that the aspect designer simply has to live with the arbitrary decisions of the base code designer, and cannot count on the availability, simplicity, or semantics of join points.

This paper provides a practical alternative criterion: Identify the crosscutting concerns that are likely to change; for each, define a crosscutting interface in the form of constraints on the exposure and semantics of certain join points. The evidence from our case study on the complex, modern HyperCast system supports the claim that this criterion provides qualitatively and quantitatively better modularity than obliviousness (or regular OO design). Crosscut programming interfaces *modularize* base and aspect code, decoupling them symmetrically. Oblivious design creates hierarchical dependence of aspect code on base code. We sacrifice *designer obliviousness*, but our application-centric approach to imposing design rules preserves *feature obliviousness*. By designer obliviousness, we mean that the base code developers can be oblivious to the presence of crosscutting in their applications [Sullivan et al. 2005, pp. 167]. By feature obliviousness, we mean that the base code developer are aware of crosscutting in their application, but unaware of the features that aspects implement [Sullivan et al. 2005, pp. 167].

Parnas's interfaces are procedural and hierarchical. Ours fall in the more general class of design rules: constraints that decouple otherwise coupled design processes. Our non-procedural and non-hierarchical crosscut programming interfaces modularize design decisions that would be scattered by OO design or subject to revision when the base code evolves in oblivious design. It is better for our interfaces to crosscut than the design decisions themselves. First, our interfaces are stable and simple; the design decisions are not. Second, our interfaces say only how to write code, not to write more code. No extra or tangled code is left in the base code.

Crosscutting interfaces are not wholly new. Designers of distributed and real-time systems have long provided programming rules to ensure that protocols, although not modularized, are written in a mutually consistent manner so as to achieve the desired result. The naming and coding conventions enforced on a project also function as crosscutting interfaces to an extent, aiding both comprehensibility and the use of tools to manipulate crosscutting code [Griswold 2001]. Our experience with HyperCast shows that the consistency enforced by our design rules have similar positive effects on comprehension. Approaches for checking design rules, e.g. PDL [Morgan et al. 2007] are likely to benefit further.

The need for aspects as a distinct abstraction mechanism from classes is a matter of on-going debate in the AOP community. Our comparative study shows that dichotomizing base functionality and aspects in design can be counterproductive. Rajan and Sullivan similarly showed that the base/aspect dichotomy in programming languages with advising has technical drawbacks [Rajan and Sullivan 2005], although it reportedly helped promote early adoption. Not all AOP approaches manifest a dichotomy, for example Eos [Rajan and Sullivan 2005] and the HyperSlice compile-time weaving approach [Tarr et al. 1999]. Without this a priori distinction, the question remains: how best to decouple concerns in the design process when given the unique mechanism of quantified advising over a join point model? Our design rules approach provides the basis for an answer.

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 do 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 [Sullivan et al. 2005] and figure elements. We expect that IDE support could aid programmers by showing the scope of an XPI's applicability. Being non-hierarchical, XPIs can overlap in scope. To date we have seen only simple examples. Nor have we 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 is neutral with respect to the join point model of a language, and rather works by forcing specified behaviors to be revealed through interfaces implemented in terms of whatever join point model a language supports.

REFERENCES

ALDRICH, J. 2005. Open modules: Modular reasoning about advice. In *2005 European Conference on Object-Oriented Programming (ECOOP'05)*. To Appear.

ALTMAN, R., CYMENT, A., AND KICILLOF, N. 2005. On the need for Setpoints. In *European Interactive Workshop on Aspects in Software*.

AspectJ. AspectJ project.
http://www.eclipse.org/aspectj/.

BALDWIN, C. Y. AND CLARK, K. B. 2000. *Design Rules: The Power of Modularity*. MIT Press, Cambridge, MA.

BECK, K. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA.

BONÉR, J. 2004. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *the 3rd international conference on Aspect-oriented software development (AOSD)*. 5–6.

CLIFTON, C. AND LEAVENS, G. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *FOAL 02*. 33–44.

CLIFTON, C. AND LEAVENS, G. T. 2003. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, Eds.

CLIFTON, C., LEAVENS, G. T., AND NOBLE, J. Ownership and effects for more effective reasoning about aspects. In *ECOOP '07, To appear*.

COLYER, A. AND CLEMENT, A. 2004. Large-scale aosd for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM Press, 56–65.

COLYER, A., CLEMENT, A., HARLEY, G., AND WEBSTER, M. 2005. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Pearson Education.

COLYER, A., HARROP, R., JOHNSON, R., VASSEUR, A., BEUCHE, D., AND BEUST, C. 2006. Point: Aop will see widespread adoption/counterpoint: Aop has yet to prove its value. *IEEE Software 23,* 1, 72–75.

DANTAS, D. S. AND WALKER, D. 2003. Aspects, information hiding and modularity. Tech. Rep. TR-696-04, Princeton University. Nov.

DANTAS, D. S. AND WALKER, D. 2006. Harmless advice. In *POPL 06*. 383–396.

DOUENCE, R., FRADET, P., AND SÜDHOLT, M. 2004. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 141–150.

ELRAD, T., FILMAN, R. E., AND BADER, A. 2001. Aspect-oriented programming. *Comm. ACM 44,* 10 (Oct.), 29–32.

FILMAN, R. E. AND FRIEDMAN, D. P. 2005a. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*. Addison-Wesley, 21–35.

FILMAN, R. E. AND FRIEDMAN, D. P. 2005b. Aspect-oriented programming is quantification and obliviousness. Addison-Wesley, Boston, 21–35.

GAMMA, E., HELM, R., VLISSIDES, J., AND JOHNSON, R. E. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

GRADECKI, J. D. AND LESIECKI, N. 2003. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley and Sons.

GRISWOLD, W. G. 2001. Coping with crosscutting software changes using information transparency. In *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. Kyoto, 250–265.

GRISWOLD, W. G., SULLIVAN, K., SONG, Y., SHONLE, M., TEWARI, N., CAI, Y., AND RAJAN, H. 2006. Modular software design with crosscutting interfaces. *IEEE Softw. 23,* 1, 51–60.

GUTTAG, J. V. AND HORNING, J. J. 1993. *Larch: languages and tools for formal specification*. Springer-Verlag.

GUTTAG, J. V., HORNING, J. J., AND WING, J. M. 1985. The larch family of specification languages. *IEEE Softw. 2,* 5, 24–36.

HOFFMAN, K. AND EUGSTER, P. 2007. Bridging java and aspectj through explicit join points. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*. ACM, New York, NY, USA, 63–72.

HyperCast. HyperCast project.
http://www.cs.virginia.edu/~mngroup/hypercast/.

JACOBSON, I. AND NG, P.-W. 2005. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley.

KELLENS, A., MENS, K., BRICHAU, J., AND GYBELS, K. 2006. Managing the evolution of aspect-oriented software with model-based pointcuts. In *European Conference on Object-oriented Programming*. 501 – 525.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. 2001a. Getting started with aspectj. *Commun. ACM 44,* 10, 59–65.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001b. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP 2001)*. 327–353.

KICZALES, G. AND MEZINI, M. 2005. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on software engineering*.

KISELY, I. 2002. *Aspect-Oriented Programming with AspectJ*. Sams.

KRISHNAMURTHI, S., FISLER, K., AND GREENBERG, M. 2004. Verifying aspect advice modularly. *SIGSOFT Softw. Eng. Notes 29,* 6, 137–146.

LAROCHELLE, D., SCHEIDT, K., SULLIVAN, K., WEI, Y., WINSTEAD, J., AND WOOD, A. 2003. Join point encapsulation. In *In Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT) at AOSD 2003*.

LADDAD, R. 2003. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning.

LEAVENS, G. T., BAKER, A. L., AND RUBY, C. 2006. Preliminary design of JML: A behavioral interface specification language for Java. *31,* 3 (March), 1–38.

LESIECKI, N. 2005. Applying AspectJ to J2EE application development. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. ACM Press.

LIEBEHERR, J. AND BEAM, T. K. 1999. Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Networked Group Communication*. 72–89.

LIEBEHERR, J., NAHAS, M., AND SI, W. 2002. Application-layer multicasting with delaunay triangulation overlays. *EEE Journal on Selected Areas in Communications 20,* 8 (oct).

LIEBEHERR, J., WANG, J., AND ZHANG, G. 2003. Programming overlay networks with overlay sockets. In *NGC 2003*.

LOPES, C. V. AND BAJRACHARYA, S. K. 2005. An analysis of modularity in aspect oriented design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. ACM Press, 15–26.

MEYER, B. 1988. Eiffel: A language and environment for software engineering. *The Journal of Systems and Software 8,* 3 (Jun), 199–246.

MEYER, B. 1992. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall.

MORGAN, C., VOLDER, K. D., AND WOHLSTADTER, E. 2007. A static aspect language for checking design rules. In *the 6th international conference on Aspect-oriented software development (AOSD)*. 63–72.

ONGKINGCO, N., AVGUSTINOV, P., TIBBLE, J., HENDREN, L., DE MOOR, O., AND SITTAMPALAM, G. 2006. Adding open modules to AspectJ. In *International Conference on Aspect-oriented Software Development*. 39–50.

OSTERMANN, K. 2008. Reasoning about aspects with common sense. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*. ACM, New York, NY, USA, 48–59.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15,* 12 (Dec.), 1053–1058.

PAWLAK, R., RETAILLÉ, J.-P., AND SEINTURIER, L. 2005. *Foundations of AOP for J2EE Development*. APress.

RAJAN, H. AND LEAVENS, G. T. 2008. Ptolemy: A language with quantified, typed events. In *ECOOP '08: 22nd European Conference on Object-Oriented Programming*.

RAJAN, H. AND SULLIVAN, K. J. 2005. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*. ACM Press, New York, NY, USA, 59–68.

RASHID, A. 2004. *Aspect-Oriented Database Systems*. Springer-Verlag.

RHO, T., KNIESL, G., AND APPELTAUER, M. 2006. Fine-grained generic aspects. In *FOAL*.

ROSENBLUM, D. S. 1995. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng. 21,* 1, 19–31.

SABBAH, D. 2004. Aspects: from promise to reality. In *the 3rd international conference on Aspect-oriented software development (AOSD)*. ACM Press, 1–2.

SAKURAI, K. AND MASUHARA, H. 2007. Test-based pointcuts: a robust pointcut mechanism based on unit test cases for software evolution. In *LATE*.

STEELE, G. 1990. *Common LISP: The Language*, 2nd ed. Digital Press.

STEIMANN, F. 2006. The paradoxical success of aspect-oriented programming. In *OOPSLA '06*. 481–497.

STEWARD, D. V. 1981. The design structure system: A method for managing the design of complex systems. *28,* 3, 71–84.

SULLIVAN, K. J., GRISWOLD, W. G., CAI, Y., AND HALLEN, B. 2001. The structure and value of modularity in software design. In *ESEC/FSE 2001*. Vol. 26. ACM Press, 99–108.

SULLIVAN, K. J., GRISWOLD, W. G., SONG, Y., CAI, Y., SHONLE, M., TEWARI, N., AND RAJAN, H. 2005. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE 2005*.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, S. M. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*. 107–119.

TEITELMAN, W. 1966. Pilot: A step toward man-computer symbiosis. Ph.D. thesis, MIT. AITR-221.

THOMAS, D. 2005. Transitioning aosd from research park to main street. In *KeyNote: AOSD '05 - 4th international conference on Aspect-oriented software development*. ACM Press.

TOURWÉ, T., BRICHAU, J., AND GYBELS, K. 2003. On the Existence of the AOSD-Evolution Paradox. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*.