Information Hiding Interfaces for Aspect-Oriented Design*

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

[ϕ]Computer Science
University of Virginia
Charlottesville, VA 22903
fsullivan,ys8a,yc7a,nt6x,hr2jg@cs.virginia.edu

[λ]Computer Science & Engineering
UC San Diego
La Jolla, CA 92093-0114
fwgg,mshonleg@cs.ucsd.edu

ABSTRACT

The growing popularity of aspect-oriented languages, such as AspectJ, and of corresponding design approaches, makes it important to learn how best to modularize programs in which aspect-oriented composition mechanisms are used. We contribute an approach to information hiding modularity in programs that use *quantified advising* as a module composition mechanism. Our approach rests on a new kind of interface: one that abstracts a crosscutting behavior, decouples the design of code that advises such a behavior from the design of the code to be advised, and that can stipulate behavioral contracts. Our interfaces establish *design rules* that govern how specific points in program execution are exposed through a given join point model and how conforming code on either side should behave. In a case study of the HyperCast overlay network middleware system, including a real options analysis, we compare the widely cited *oblivious* design approach with our own, showing significant weaknesses in the former and benefits in the latter.

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

General Terms: Design

Keywords: Aspect-oriented programming, design rules, options

1. INTRODUCTION

Aspect-oriented (AO) programming languages aim to improve the ability of designers to modularize concerns that cannot be modularized using traditional procedural or object-oriented (OO) methods. Examples of *crosscutting concerns* include tracing, logging, transactionality, caching and resource pooling. The ability to modularize such concerns is expected to improve comprehensibility, parallel development, reuse and ease of change, reducing development costs, increasing dependability and adaptability and ultimately creating more value for producers and consumers alike.

The most prominent AOP model today is that of AspectJ [2, 12]. AspectJ extends Java with several complementary mechanisms, notably *join points* (JPs), *pointcut descriptors* (PCDs), *advice* and *aspects*. JPs are points in a concrete program execution, such as method calls and executions, that, by the definition of the join point model of the AO language, are subject to *advising.* Advising extends or overrides the action at a join point with a CLOS-like [22, Ch. 28] *before, after,* or *around* anonymous method called an *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, an advice can have effects that cut across a class hierarchy. Advice, pointcuts and ordinary data members and methods are grouped into class-like modules called *aspects*.[1] Aspects are intended to support the modular representation of crosscutting concerns [13], although they admit other uses.

Thirty-three 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 [19]? Taking comprehensibility, parallel development and ease of change as goals, he used a comparative analysis to argue that the prevailing criterion for modularizing systems according to stages of processing in their flow charts performed poorly relative to his new approach, *information hiding*. Under this 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 [19, p. 1058]." The idea is that stable interfaces should abstract and decouple such decisions. As an example Parnas cites the choice of a data representation, the hiding of which is accomplished by an abstract data type interface. In this paper, seeing opportunities afforded by the new mechanisms of AO programming, we revisit Parnas's question with a twist: by what criteria should systems be decomposed into aspects?

A widely cited method of aspect-oriented design, popularly called *obliviousness*, advocates that the designers and developers of *base* functionality[2] need not be aware of, anticipate or design code to be advised by *aspects.* Filman and Friedman say, "Just program like always, and we'll be able to add the aspects later [7, p. 31]." This idea is widely taken as partially defining, and as a design process and architectural style for, AO design: first separate base and crosscutting concerns; next implement base concerns in an OO style ignoring crosscutting concerns; finally implement the crosscutting concerns as aspects that advise the base code directly. Is this straightforward AO modularization criterion the best to be found?

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

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal

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

[2]The term *base* in the AOP community refers to the advised, usually non-aspect-oriented elements of the system. *Base code* more specifically refers to the actual program text, either as implementation or design.

Aided by a case study of a modern OO system—the HyperCast overlay network middleware implementation [10]—we address this question by comparing the oblivious method to a new one based on *design rules* [3] as generalized information hiding interfaces. We find that although the oblivious method allows base code designers to ignore aspect design, it leaves important abstractions implicit in program details and does not decouple design decisions in the other direction. In particular, the freedom afforded to base code designers comes at considerable cost to aspect designers, who are thwarted by a lack of constraints on the code they need to advise. Results include unnecessarily complex aspects (especially pointcut descriptors); in some cases the inability to integrate aspect and base code at all without changes to the base code; the implicit representation of important abstractions; and tight coupling of aspect code to complex and changeable details of base code. AO language researchers have sought solutions in extensions to [20] and constraints on [1] join point models and advising. However, providing new language features is costly and incurs risks in semantic complications, runtime costs and ultimately lack of adoption.

Our software engineering approach, by contrast, is to impose design rules as abstract, design-decoupling interfaces between aspects and advised code. These interfaces govern how code has to be written to reveal specified execution points through the join point model of a language and how aspects can use these interaces. Concretely, our rules specify (1) the behaviors to be exposed; (2) constraints on program implementation to ensure that these behaviors are exposed by join points with specific properties, including type (e.g., field access versus method call) and signature (e.g., method names); and (3) behavioral contracts subject to enforcement at the discretion of the engineer. The first point ensures that the join points that aspects need are visible. The first two constrain base code designers and inform aspect designers so that PCDs can be written that remain stable as base code evolves. The last allows the engineer to trade between behavioral assurances at interfaces and the openness of programs to new advising relationships.

We document these rules in interface specifications that base code designers "implement" and that aspects may depend upon. Once these interfaces are defined, designers can develop aspect and base code independently and concurrently, with each side aware of the shared rules but "oblivious" to each others' detailed decisions. We find that formulating rules in terms of application-specific abstract state and behavior, not just in terms of concrete execution events (e.g., method calls), helps to reveal important abstractions, better separate concerns and improve the resilience of designs.

Unlike some alternative approaches, ours does not generally require any of the following: that auxiliary code be added to base code (e.g., to signal events); references from base code to aspect or other code; new programming languages, mechanisms or semantics (although tools for design rule checking are sometimes valuable); either extensions to or constrains on join point models (JPMs); or prior constraints on the ability to advise join points.

The rest of this paper is organized as follows. The next section provides background on the concept of obliviousness and on the use of design structure matrices and design rules in modeling modularity in design. Next we introduce the HyperCast case study, including its design considerations and crosscutting concerns. Then we consider designing, developing, and extending HyperCast through the oblivious aspect-oriented design approach. We then repeat the exercise with the information-hiding design-rules approach. Finally, we compare the two approaches, first qualitatively, from the designer's perspective, then quantitatively using an economic analysis based on the theory of real options [3].

2. BACKGROUND

2.1 Obliviousness

Obliviousness as a distinguishing characteristic of AOP was first proposed by Filman and Friedman [6, p. 2]:

> *AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.*

The above-cited paper, whose purpose is to justify this definition, has at least 50 citations in the literature. The vast majority use this paper's definition of AOP, while a few distinguish their work from this definition.

Quantification needs to be understood before obliviousness can be discussed. One distinguishing feature of AspectJ is the ability to select sets of join points declaratively. For example, the PCD, `call(* *State(..))`, refers to all calls to methods with names ending in `State` and having any parameter list. Not only do PCDs save programming effort, but to the extent that they exploit coding conventions they automatically match desired join points in new code that follows the conventions. Some have referred to this intensional property of properly quantified aspects as *shyness* [18]. In the terminology of obliviousness, such aspects are oblivious of base code.

Quantification's link to obliviousness is that more expressiveness in the quantification language—for example extending the join point model and pointcut language of AspectJ—provides for more obliviousness. That is, more power in the hands of aspect designers means that less help is required from base code developers to add aspects to a system. As often is the case, greater power can be a double-edged sword, as we discuss in Section 4. The desire for greater expressiveness in the quantification language and its attendant difficulties have been drivers of aspect-oriented programming language research.

Many variants and degrees of obliviousness can be found in the literature, each of which carries different implications. We list the definitions most relevant to the current paper in an approximate hierarchy from weakest to strongest, and give them appropriate names:

Language-level obliviousness is what is allowed when advising constructs are introduced to a programming language. Filman and Friedman provide an apt definition, arguing that "...the distinguishing characteristic of aspect-oriented programming (AOP) languages is that they allow quantified programmatic assertions over programs that lack local notation indicating the invocation of these assertions" [7, p. 21]. Specifically, the language enables the base code developer to write code without needing to use callback hooks or macros.

Feature obliviousness is when the base code developer is unaware of the features that aspects implement. A base code designer can prepare code for aspects, e.g., with event hooks, sacrificing language-level obliviousness but retaining feature obliviousness. From an information-hiding perspective [19], feature obliviousness matches the classic notion of obliviousness: services are unaware of their clients, but are obliged to serve clients that meet the given preconditions. Our approach provides for what amounts to a combination of both language-level and feature obliviousness as well as shyness through naming, syntactic, procedure calling and similar conventions. One result is that aspects can be more "base code shy" and hence not have to change when the base code evolves.

Designer obliviousness is when the designers of the base code can be oblivious to the existence of aspects; in particular, not designing any differently than they normally would. As Filman and

Friedman say "For true AOP, we want our system to work with oblivious programmers—ones who don't have to expend any additional effort to make the AOP mechanism work" [6, p. 2]. In addressing the properties of AOP that aid decoupling (i.e., separation of concerns), Erad, Filman, and Bader note [5, p. 31]:

> *This includes obliviousness, whether the writer of the main code be aware that aspects will be applied to it; intimacy, what the programmer has to do to prepare code for aspects; and globality versus locality, whether aspects apply to the program as a whole or only parts of it.*

Intimacy, as used here, is at the other end of the spectrum from obliviousness: the oblivious base code developer would not *prepare code for aspects.* To satisfy designer obliviousness, however, the developer has to meet the stronger criterion of not even needing to be *aware that aspects will be applied to it*. This would generally rule out communication with aspect designers.

The adoption of this definition of AOP has reached the popular press. In Laddad's widely acclaimed book on programming with AspectJ, he says of AspectJ, "AOP modularizes the individual aspects and makes core modules oblivious to the aspects. Adding a new functionality is now a matter of including a new aspect and requires no change to the core modules" [14, p. 28].

Pure obliviousness is the limit where perfect obliviousness and shyness are achieved, allowing for total, symmetric separation of concerns. Filman and Friedman laid it out as the ultimate goal to be sought by AOP researchers [7, p. 31]:

> *It's a really nice bumper sticker to say "Just program like always, and we'll be able to add the aspects later." (And change policies later, and we'll painlessly transform the code for that, too.)*[3]

Such a degree of obliviousness is viewed as a far-off ideal by others [4]. In the presence of "harmful aspects" [11], however, complete obliviousness might not be desirable.

2.2 Design Structure

The analysis in this paper depends heavily on the notion of *dependences* between elements in a software design, and especially on the *structure* of dependences in a given design. We say that an element B *depends upon* an element A iff B makes assumptions about A that, if invalid, can cause B not to satisfy its specification. Invalid assumptions can arise for many reasons, including the following: A's developer misimplements A's specification; A's specification changes but A's developer is not aware of the change; B's developer misunderstands A's specification; or B is unaware of changes in properties of 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 *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 rules. 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 decisions have to be resolved before downstream decisions that depend on them. A design and its implementation process are called *coupled* if its elements depend on each other in cyclic relations.

It is common for different dependence structures to prevail in different stages of a software process and in different parts or at different levels of a design. In a waterfall process, for example, the relationship between the architectural design and implementation processes is hierarchical, but the implementation task 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 [3] and the subsequent work of Sullivan et al. [24], we represent design dependence structures using Design Structure Matrices (DSMs) [23]. DSMs present in a graphical form the pair-wise dependence structures of designs and and of their corresponding development and evolution process.

Figures 1, 2, and 4 in the following sections are examples. 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 design 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. 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. The choices made for, or changes to, those design variables influence the best choice for A. In Figure 1, for example, variable 28, socket_impl, depends on 3, 14, 29, and 30 (socket_spec, socket_interface, exception_logging_impl and non_exception_logging_impl, respectively). For a given column for B, the marks show which variables depend upon B.

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 1, which we discuss in detail in the next section. At the outermost level of aggregation, the upper left part of the DSM represents abstract design concerns, 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 hierarchical 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 reveals 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 internally coupled structures.

An *interface* is an agreement about properties that an element B should have and that other elements, such as A, may depend upon. An interface thus comprises a set of design rules to which B is subject and which A may assume to be followed. One of the central operations in a design activity is to create interfaces to decouple otherwise coupled design elements and corresponding processes. Both elements come to 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 literally see whether the design is modular in this sense [24]. The interface cluster should not depend on the environment cluster; only implementation variables should depend on the environment variables.

[3] A similar quotation can be found in their 2000 paper [6, p. 6].

In Figure 1 the environment variables are modeled as elements of the system requirements specification (or concerns) that the Hypercast designers believe will change.

3. THE HYPERCAST CASE STUDY

This paper centers around a comparative analysis of three designs for HyperCast, a scalable, self-organizing overlay network system [10, 15, 16]. The three designs are (1) its actual OO design, (2) an oblivious AO design that we produced by moving scattered code into aspects using oblivious design, and (3) one based on the design rules approach. 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. The placement of each fragment indicates objectively and precisely where aspects need to advise in order to factor concern-related code into aspects using an oblivious approach.

3.1 What HyperCast Does

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 to loosely coupled classes in the implementation.

There are several areas in which scattering and tangling of code is evident. In this case study we address two. The first is logging. 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. Second, 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. We thus inferred the following additional concerns: *Information logging, Exception logging, Non-exception error logging, State machine events,* and *Service events.*

3.2 The Design Structure of HyperCast

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 1 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. Implementation decisions are grouped in the lower right.

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 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 con-

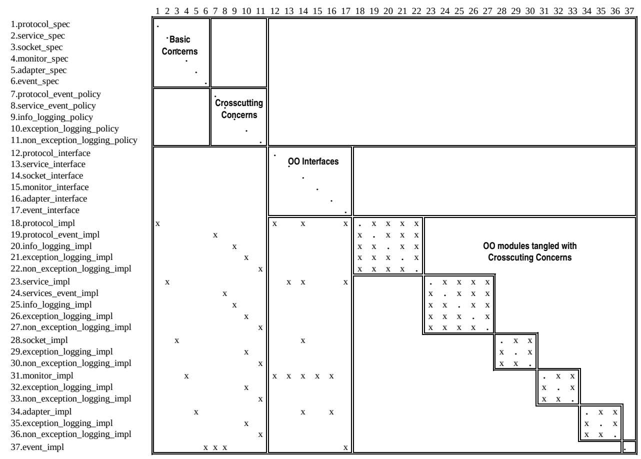

Figure 1: Basic Object-Oriented Design of HyperCast.

cerns. 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. AOP is meant to enable improved modularity.

3.3 Comparative Methodology

We compare the actual HyperCast design with two AO alternatives: an oblivious design and one produced using the design rule method. We produced and tested both alternatives by refactoring HyperCast. We produced an oblivious design by assuming that, had the developers been able to ignore crosscutting concerns, they would have written the same code but with code for the crosscutting concerns left out. The results is a perfectly reasonable design for the base concerns. We then wrote aspects that advise this code in precisely the places where scattered fragments appear in the original design.

It was easy to find scattered code for the concerns of interest: they made explicit calls to methods for logging or event notification. We refactored the system to move such code (or functional equivalents) into aspects using AspectJ 1.2 [2]. The aspects had to advise the new base code with PCDs that caused the given fragments, now in the form of advice bodies, to be executed at the points from which they had been removed.

The design rules approach is different: Here we asked the question, what constraints on the code would shape it to make it relatively easy to write the aspects at hand, as well as support future aspects? We thus not only moved fragments from the original code to aspects, but refactored the original code in ways dictated by the design rule interfaces that we designed. In the following sections, we compare the results in order to gain insights into the properties of the respective design methods and resulting designs.

The phenomena we sought to understand include the difficulty of writing the aspects in the first place, their sensitivity to changes in the base code, and the value of the oblivious approach relative to the design rule interfaces method. For our design rule method, we also ask how intrusive it is into the developer's practice and the resulting code.

4. THE OBLIVIOUS APPROACH

We first explore the benefits and costs of a commitment to the notion of oblivious design. Following the method outlined in the previous section, we refactored HyperCast into an oblivious aspect-oriented design. We assume—reasonably, we believe—that the developers could plausibly have written essentially the same OO code, just leaving out the scattered parts that implement crosscutting concerns. We removed the scattered fragments and localized them in 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 whole OO *events* module.

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'd have to advise to weave the extracted code fragments back into the system. We found out that the context of these join points was in 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. Below is an example.

```
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);
```

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 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 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.

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 if and switch statements as the logging code, nor solely reserved for use by the logging concern:

```
 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("String" + addrStr + "
    has wrong format for a physical address.");
 }
```

There are a few possible ways to capture the IP address. First, a 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). Two, 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 in 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. Here is an example:

```
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 {
      ...
}
```

Here `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 at best 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

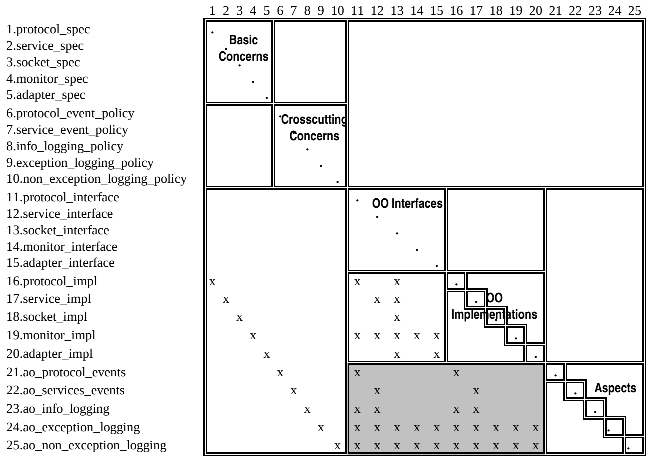

Figure 2: Obliviousness Aspect-Oriented Design of HyperCast.

PCDs is costly. More seriously, extensions to the base code that should be logged will require 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 blithely implement "as he would anyway," 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. 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 2, which presents a DSM for the oblivious AO version of HyperCast, highlights this design structure. The base code modules no longer depend on the crosscutting concerns, but the aspect modules now strongly depend on the base implementation modules. 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. THE DESIGN RULES APPROACH

The oblivious design process assumes that it's bad for base developers to be aware of aspects, and that intimacy—the explicit preparation of code for advising by aspects—is even worse. Why? For one, the comprehensibility of the base code could be compromised: it might not manifest a design that matches the designer's conceptions of the base functionality. Two, the base code could contain tangled concern code, compromising ease of change. Similarly, explicit measures in the base code to anticipate certain extensions might effectively exclude others. Three, parallel development could be compromised, in that the base code designers would have to coordinate with the aspect developers on the proper way to prepare the base code for aspects.

Yet, as the previous section highlights, obliviousness in the design of the base code can cause numerous problems. The question, then, is whether a new method can be devised that realizes the benefits of AO design and yet minimizes or eliminates these problems. The first two issues could be addressed with a method that molds base code in ways that help aspect designers but without the need for any auxiliary code, such as event notifications, method calls, or tags. The OO design would be uncompromised, and there would be no tangling. The third problem in fact highlights an opportunity: the freedom afforded to base code designers by obliviousness delays aspect design because it flows precisely from the hierarchical nature of the design process. In the oblivious approach, pointcuts cannot be written and advice parameter lists cannot be formulated until the base code is written. A short design phase that establishes symmetric separation of concerns between base designers and aspect designers could actually increase overall parallel development.

The design rules methodology provides the basis for such a method. An essential idea is that for each crosscutting concern, a crosscutting design rule interface is established to decouple the base design and the aspect design. The constraints imposed by a design rule govern three things: (1) which execution phenomena must be exposed as join points, (2) how they are exposed through the JPM of the given language (e.g., in AspectJ, rules could govern syntax, name, and stack shapes), (3) constraints on behavior across join points (e.g., pre- and post-conditions for the execution of advice compositions). These elements ensure two important properties. One, the PCDs required by aspects can be constructed 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, and that the system state after advice has returned is not compromised. Our method is thus an instance of the information hiding approach, but for crosscutting join-point-based interfaces.

A base/aspect design rule should result in easy-to-use join points that give base designers considerable implementation freedom. For ADT design, the key to an easy-to-use interface that provides freedom in implementation is that the interface should place reasonably minimal assumptions (constraints) on both the implementation and the clients. For example, an associative memory abstraction should not have an accessor method that reports the average collision chain length. Nor should the interface make assumptions about clients' intents, say by calling it a Dictionary and limiting use to short string inputs and outputs. For base/aspect design rules, 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 design rules are formulated in terms of the abstract system model, the resulting syntactic, naming and other constraints are consistent with the 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. The following is a description and comparative evaluation of our method through our case study application.

5.1 Experimental application to HyperCast

To create interfaces for the two crosscutting concerns of logging and notification, we formulated eight major design rules. They constitute five interfaces (with rules 1–4 comprising one interface).

1. **State Update.** DR1: HyperCast's functionality is driven by the abstract state transitions of the protocol's finite state machine (FSM). This rule ensures that these transitions are visible to clients of HyperCast and it prohibits clients from interfering with HyperCast's core FSM behavior.

2. **Update Logical Address.** DR2: The changing of a Node's logical address is essential to the transition function. This rules ensures that the changes of logical address are visible, and it prohibits interference with base behavior.

3. **Update Neighborhood.** DR3: The changing of overlay topology is essential to the transition function. This rule ensures that changes of topology are visible to HyperCast clients, and it prohibits them from interfering with base protocol behavior.

4. **Join and Leave Overlay.** DR4: Expose the leaving and joining of the overlay

5. **Update Message Store State.** DR5: HyperCast's services are driven by abstract state transitions in a component called the MessageStore FSM. This rule ensures that these transitions are visible to clients of HyperCast.

6. **Throw Exception.** DR6: Expose exceptions with context information. This rule ensures that context information for exceptions is available to clients, and it prohibits clients from interfering with the base code behavior.

7. **Error Handler.** DR7: This design rule provides a unified error handling approach in HyperCast.

8. **Non-Exception Error Handling.** DR8: This rule ensures that error handling is done by the approach defined in DR7. Thus all error states are exposed to HyperCast clients.

Figure 3 presents the resulting structure, with base code on the left, design rules in the middle, aspects on the right, and dependences of base and aspect codes on DRs indicated by arcs.

We refactored HyperCast to implement these rules. We modified 65 places in the Protocol module to follow DRs 1–4, mostly in six classes, especially in MessageArrivedFromAdapter. We modified 21 places in the service classes to follow DR 5. All the modifications are simple, as the finite state machines were already implemented and we only used methods with naming conventions to expose them. We also removed the old event notification method, which occurred in 41 places across 10 classes. For DRs 7–8, we first identified 14 error types according to the specification. We then create a template to implement DR 7 and DR 8. The refactoring involved 55 error occurrences across 18 classes.

Such design rules would be used extensively by the aspect designers. Unlike APIs, our interfaces do not have any explicit representation in the source code. Rather, they simply constrain the manner in which the code is written. We document these interfaces in a style reminiscent of design patterns [8], as shown in Figure 5.

Figure 4 shows how the design rules decouple the base and aspect elements. Compared to the oblivious design's DSM in Figure 2, we observe that all dependences between the basic modules

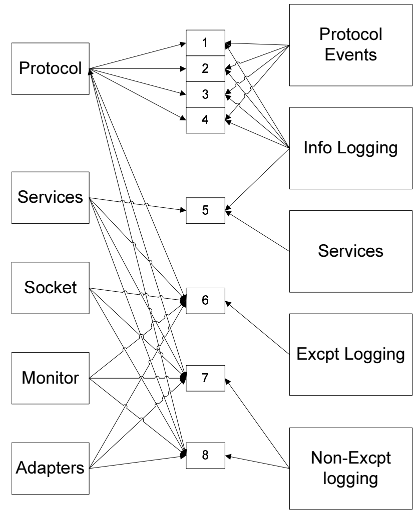

Figure 3: The crosscutting interface in HyperCast. The dependences of base and aspects on DRs is indicated by the arcs. Base functionality is on the left, design rules in the middle, and aspects on the right.

and aspect modules are removed. Instead, both the aspect modules and the basic modules now depend on the design rules.

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 design-rules AO design the aspects average only 30 lines each for the same functionality. For example, Figure 6 shows the aspects for handling logical address events, using the two approaches. We observe that the aspect in 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 lots of details of base code. Moreover, the design-rule pointcut will capture newly-coded address changes, because it is quantified and base code designers are constrained to write new code following the design rule.

6. QUANTITATIVE ANALYSIS WITH NET OPTION VALUE

Sullivan et al. [24] and Lopes [17], have previously used the *net option value* model of Baldwin and Clark [3] to characterize and quantitatively compare modular software designs modeled by DSMs. In this section, we evaluate HyperCast similarly based on the DSMs introduced in previous sections.

Given a product, the visible properties of which have a market value of S_0, the model estimates the additional value due to modularity in its hidden design. The idea is that modularity creates a portfolio of valuable *real options*, one per module—options to replace existing modules with ones that capture new sources of value. The model treats the cost of attaining modularity as sunk. The model thus gives a partial picture—of the value of the flexibilty created by modularity but not of the costs of designing a modularization. The value of a product including the value of m options embedded in its design is modeled as follows.

$$V = S_0 + NOV_1 + \ldots + NOV_m, \text{ where}$$
$$NOV_i = max_{k_i}\{\sigma_i n_i^{1/2} Q(k_i) - C_i(n_i)k_i - Z_i\}$$

The value of modularity is modeled as the sum of the net options

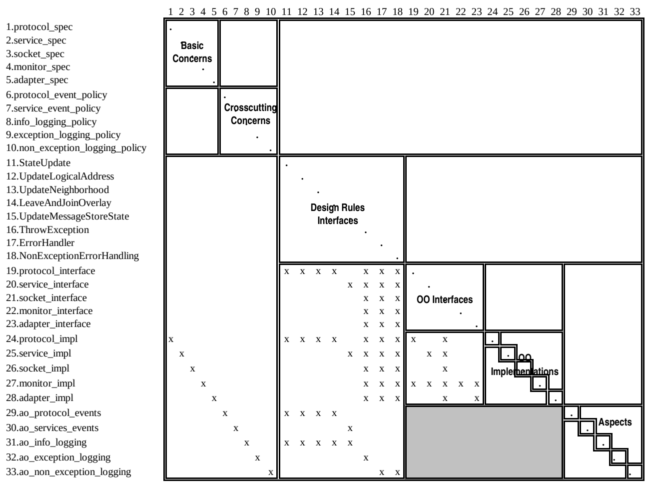

Figure 4: Design Rules Aspect-Oriented Design of HyperCast.

Name: State Update

Rationale: HyperCast's functionality is driven by the abstract state transitions of the protocol FSM. This design rule ensures that these transitions are visible to clients of HyperCast and alert clients that they may not interfere with HyperCast's code behavior.

Depends upon: none

Base code scope: implements `edu.virginia.cs.mng.hypercast.I_Node+`

Design Rule: *Provides:* Call to `void setState(byte)` at the conclusion of performing a state transition. *Requires:* No changes to the trace of `edu.virginia.cs.mng.hypercast.I_Node+`

Example: A pointcut for advising all state transitions might be:

```
pointcut NodeStateChanged () :
  call (void I_Node+.setState(*));
```

Figure 5: Specification of DR 1, Update State.

values ($NOV_i, i = 1 \ldots m$) of the modules. The NOV of a module is defined as the payoff on a research and development (R&D) investment that, for an optimal choice of k explores k candidate replacements for the current module and that replaces it with the best provided that a better one is found. The NOV formula assumes that as k increases, the R&D costs increase linearly but that the benefits diminish. In detail, the model assumes a normal distribution, centered at 0, on the values of the k R&D experimental results. $Q(k)$ is the expected maximum positive of k draws from a standard normal distribution. For module i, $\sigma_i n_i^{1/2} Q(k_i)$ is the expected value of the best of k positive-valued new candidates relative to the value of the current module. The standard deviation, $\sigma_i n_i^{1/2}$, models the *richness* of the design space around the current module as the product of a factor, the *technical potential*, σ_i, of the module, and the square root of its complexity, n_i, viewed as the number of major design decisions in the module. $C_i(n_i)k_i$ models the cost to run k_i experiments as a function C_i of the module's complexity n_i. $Z_i = \Sigma_{j\text{sees}i} cn_j$ models the cost to substitute in a module given

```
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)

Figure 6: Aspect for LogicalAddressChanged Events, (a) without design rules, and (b) with design rules.

the dependences on the current module. Finally, the max selects the value of k_i that maximizes the gain for module i. In a nutshell, this is an options pricing formula meant to estimate the value of the real option created by a module in a design.

The two key parameters are *technical potential*, σ, and *complexity*, n. In this paper, we operationalize technical potential as a stream of change demands, with a percentage of demands impinging on base modules and the rest on the aspect modules. A choice of σ thus models a judgment as to where value can be created by changing modules: in the base code or in the aspects.

We estimate complexity using lines of code (LOC) relative to the LOC of the overall system as a proxy. Recognizing that code can have siginificant inessential complexity, we used the LOC of the smallest version of each module amongst our three systems. Oblivious aspects were, on average, eight times larger than design rule aspects. We thus used the design-rules version sizes to estimate essential complexity.

We computed the value of the embedded options for each design, varying σ, the change demands impinging on the base concerns relative to crosscutting concerns, from low to high. Table 1 presents our results. The OO row presents the summed NOV "value of modularity" as a fraction of the base system value S_0 for each value of σ. The next two rows present value of modularity for the oblivious and design rule designs, respectively (the upper number in each row), and this number as a percentage of the corresponding number for the base OO design.

The values cannot be taken as economic truths but only as indicators. The development and validation of Baldwin and Clark's model is still at an early stage, even in the economics community, and it remains an open problem to justify precise estimates for σ, both in general and for software design [24], in particular. The model does provide for back-of-the envelope exploration of the consequences of various assumptions, plausibly valid ordinal comparisons 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 design-rule design outperforms oblivious design, chiefly 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 in the value of flexibility.

7. CONCLUSION

In his seminal paper on information hiding [19], Parnas showed the deleterious effects of dependences on design decisions that are complex or likely to change. They make programs hard to understand, develop in parallel, and change at a reasonable, predictable cost. Parnas then showed designers how to do better using abstract interfaces to decouple such design decisions. In this paper, we revisited Parnas's ideas in light of the addition of join points and quantified advising to the programmer's toolkit. The question we have addressed is straightforward: what form of information hiding interface is needed to expose crosscutting abstract behaviors to AO quantified advising while hiding the complex, changeable and inessential design decisions on either side?

An AOP language like AspectJ implicitly publishes a vast array of join points for any given program. Unless their use is managed carefully, complex integration problems and proliferating dependences of PCDs on unstable, complex and arbitrary implementation decisions result. The oblivious approach dictates that designers make no preparations for aspects. The problem is that aspect code then has to conform to the arbitrary decisions of the base code designer and cannot count on the presence, regularity or semantics of join points.

Our approach provides an alternative design criterion. Identify the important, abstract crosscutting behaviors whose implementations are complex or likely to change. For each, define an interface that at once constrains the exposure of join points and the behavior expected and allowed by the interface. Our study of HyperCast suggests this approach provides qualitatively and quantitatively better modularity than obliviousness (or regular OO design). Our interfaces *modularize* base and aspect code, decoupling them symmetrically. Oblivious design creates hierarhical dependence of aspect code on base code. Our interfaces also make important crosscutting concerns explicit, that remain implicit in oblivious and OO design. Our approach abandons design obliviousness but preserves feature obliviousness. Base code designers must be aware of and adapt their code to crosscutting interfaces, but need not be aware of the aspects that use these interfaces.

Of course our approach is not a panacea. No approach on the horizon is likely to accommodate unplanned changes at a predictably reasonable cost. It is thus not surprising that our study does not support the obliviousness proposition, that AOP fulfills a promise that at a predictable and reasonable cost one can "Just program like always, and . . . add the aspects later." Nor will our approach. Rather, ours promises benefits when relevant crosscutting behaviors are anticipated and when new code, anticipated or not, can be written against existing interfaces.

If existing interfaces are inadequate, the designer will, as always, have no choice but to dig into existing code to determine how best to integrate the new code. Oblivious AO design provides an option to integrate new code without changing existing code, provided that the relevant behaviors are exposed through the join point model (which is not assured in general). However, integration can be complex and the resulting code subject to disruption by seeming innocuous changes in implementation. Our approach would have the designer develop a new interface and refactor the code to bring it into conformance. In the face of unanticipated change we thus trade non-invasiveness for much simpler aspect code (PCDs) and the preservation of a modular design with abstract interfaces. Of course, our approach does not preclude an oblivious approach.

Parnas's interfaces are procedural and hierarchical. Ours fall in the more general class of design rules: constraints that serve to decouple otherwise coupled design decisions. Our interfaces are non-procedural and non-hierarchical. They modularize decisions that generally would be scattered by OO design or subject to change when base code changes in oblivious design. It's 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.

Recently, in a paper on the nature of crosscutting interfaces and modularity in AO design, Kiczales and Mezini wrote [13]:

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

This definition supposes that aspects change the interfaces of advised modules by the join points they use. What Kiczales and Mezini really compute and document are *dependences* on join points in a given system configuration. In the absence of agreement on the assumptions that aspect designers *may* make about join points, revealing those they *do* make is an important enabler of modular reasoning and change. With our approach, sets of permissible assumptions are specified explicitly as interfaces. There is no need for a concept of unstable interfaces inferred *ex post* from system configurations.

Aldrich recently proposed scoping constructs and a formal semantics that relate to our design rules. The Open Modules system focuses on the exposure of join points such that module state that is intended to be hidden cannot be advised [1]. A module has to declare a pointcut to export join points on private state. This approach permits the evolution of module implementations without rework of aspects. The semantics also supports checking these properties. However, the resulting interfaces pertain to individual modules and are not crosscutting, so Open Modules do not directly support ex-

σ	1%	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%	99%
OO	0	0	0	0.00164	0.077	0.17	0.31	0.50	0.78	1.09	1.26	1.402
Oblivious	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	2179.55	386.89	141.18	58.06	24.00	6.41	0.92	-0.40	-0.78
Design Rules	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	2244.62	404.78	147.06	64.52	32.00	11.54	4.59	2.38	1.43

Table 1: Net option values for the base OO, oblivious, and design rule designs as a function of the fraction of change demand impinging on base as opposed to crosscutting concerns.

posure of join points across non-hierarchical parts of a system.

Today, prominent industrial development teams are reporting successes with AOP. How can this be if the oblivious approach has such problems? Our informal discussions with insiders suggest that some do formulate and follow coding rules that aspect designers depend upon. Our approach recognizes such good practice, raises it to the level of a principle, and has the benefit of producing explicit abstractions of important crosscutting concerns behaviors against which aspects are written.

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 consistent manner so as to achieve the desired result. Naming and coding rules enforced on a project also serve as implicit crosscutting interfaces, aiding comprehension and the use of tools to manipulate crosscutting code [9]. Our work with HyperCast suggests that the consistency imposed by our designs rules can have similar positive effects on comprehensibility.

The need for the *aspect* as distinct from the *class* is a matter of on-going debate in the AOP community. Our study shows that distinguishing between base and aspect code can be counterproductive. Rajan and Sullivan similarly showed that in the programming language context, the dichotomy might have helped promote early adoption of AOP but it also has drawbacks [21]. Not all approaches manifest a dichotomy: e.g., Eos [21], HyperSlices [25]. In either case, the question remains: how to modularize designs when advising over join points is available as a programming mechanism? Our work provides a basis for an answer.

Open questions remain, constituting our future research agenda. With what notations and semantics should our interfaces be specified? Section 5 suggests a possibility, but it has not been fleshed out or tested. To what extent and how can our interfaces be checked? AspectJ's `declare error` construct can be used to check some rules, especially prohibitions. Can environments such as Eclipse's AJDT make crosscutting rules more visible and ease their maintenance? AJDT has an effective model for exposing advising; what about rules governing advising, if written as pointcut descriptors? Forthcoming work will address syntactic and behavioral contract specification and checking, in particular.

8. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *2005 European Conference on Object-Oriented Programming (ECOOP'05, to appear)*, July 2005.

[2] AspectJ project. http://www.eclipse.org/aspectj/.

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

[4] C. Constantinides and T. Skotiniotis. Reasoning about a classification of cross-cutting concerns in object-oriented systems. In P. Costanza, G. Kniesel, K. Mehner, E. Pulvermüller, and A. Speck, editors, *Second Workshop on Aspect-Oriented Software Development of the German Information Society*. Institut für Informatik III, Universität Bonn, Feb. 2002. Technical report IAI-TR-2002-1.

[5] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[6] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

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

[8] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

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

[10] Hypercast project. http://www.cs.virginia.edu/ mngroup/hypercast/.

[11] S. Katz. Diagnosis of harmful aspects using regression verification. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, Mar. 2004.

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

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

[14] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

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

[16] J. Liebeherr, M. Nahas, and W. Si. Application-layer multicasting with delaunay triangulation overlays. *EEE Journal on Selected Areas in Communications*, 20(8), oct 2002.

[17] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26. ACM Press, 2005.

[18] J. Marshall, D. Orleans, and K. J. Lieberherr. DJ: Dynamic structure-shy traversal in pure Java. Technical report, Northeastern University, May 1999.

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

[20] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *the Fourth International Conference on Aspect-Oriented Software Development (AOSD 2005)*, March 2005.

[21] H. Rajan and K. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In *Proceedings of the 27th International Conference on Software Engineering ICSE 2005)*, page To appear, May 2005.

[22] G. Steele. *Common LISP: The Language*. Digital Press, 2nd edition, 1990.

[23] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28(3):71–84, 1981.

[24] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. *SIGSOFT Softw. Eng. Notes*, 26(5):99–108, 2001.

[25] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119, May 1999.