highlighter/.style = highlightercolor, line width =
Trading Obliviousness for Modularity with Cooperative Aspect-oriented Programming\textsuperscript{1}

KEVIN HOFFMAN and PATRICK EUGSTER
Purdue University

The potential of aspect-oriented programming to adequately capture crosscutting concerns has yet to be fully realized. For example, authors have detailed significant challenges in creating reusable aspect component libraries. One proposed solution is to introduce explicit join points (EJPs) to increase modularity by reducing obliviousness, enabling a cooperative aspect-oriented programming (Co-AOP) methodology where base code and aspects synergistically collaborate.

This paper explores the tradeoffs between obliviousness and modularity. We briefly introduce EJPs and Co-AOP, and hypothesize how to balance obliviousness and modularity using Co-AOP. We build upon a prior empirical study to refactor three real-life Java applications to implement the exception handling concern using three distinct strategies: (1) using fully oblivious aspects in AspectJ, (2) using EJPs in a fully explicit fashion, and (3) using EJPs while following the Co-AOP methodology. We study other crosscutting concerns by refactoring a fourth application, JHotDraw. The differences in terms of common code metrics are analyzed, and the impact on modularity is assessed using design structure matrices. Results indicate that the Co-AOP methodology can in many cases significantly improve code quality attributes versus fully oblivious or fully explicit approaches. We conclude with guiding principles on the proper use of EJPs within the Co-AOP methodology.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques—modules and interfaces; D.2.13 [Software Engineering]: Reusable Software; D.2.11 [Software Engineering]: Software Architectures—information hiding

General Terms: Design, Languages, Measurement
Additional Key Words and Phrases: Aspect-oriented programming, design structure matrix, explicit join points, modularity, obliviousness

1. INTRODUCTION

Aspect-oriented programming (AOP) [Kiczales et al. 1997] is slowly stepping out of its infant shoes. By offering programmers the possibility of dealing with crosscutting concerns once and for all in the form of aspects alongside the primary application logic, AOP strives for an increased modularity in applications, and a break-down of development efforts. Typical show-cases for aspects include concerns such as security, distribution, or persistence.

The ability to not only separate crosscutting concerns from the primary application, but to actually modularize them into reusable components is crucial for effective aspect-oriented software development. However, this goal has yet to be effectively realized. Various authors have debated the full implications of AOP for modularity [Baldwin and Clark 1999; Griswold et al. 2006; Parnas 1972] and modular reasoning [Aldrich 2005; Clifton and Leavens 2005; Kiczales and Mezini 2005]. In particular, the impact and feasibility of aspects’ obliviousness with respect to the

\textsuperscript{1}This article is a revised and extended version of a paper, “Towards Reusable Components with Aspects: An Empirical Study on Modularity and Obliviousness,” presented at ICSE 2008.

base code has been the topic of recent publications: Kienzle and Guerraoui [Kienzle and Guerraoui 2002] discuss the challenges of using AOP to implement transactions. Clifton and Leavens [Kienzle and Guerraoui 2002] concisely articulate the issues involved with modular reasoning in AspectJ due to obliviousness. Early work on AOP, and AspectJ [Kiczales et al. 2001] in particular, have conjectured modularity to be a direct consequence of obliviousness. Several authors have proposed extensions to AOP models and languages to restrict or to infer and document the occurrence of aspects within the base code in order to enhance modularity while preserving absolute obliviousness [Gudmundson and Kiczales 2001; Aldrich 2005; Kiczales and Mezini 2005; Sullivan et al. 2010]. However, Sullivan et al. [Sullivan et al. 2005] thoroughly decompose the notion of obliviousness, pointing out that certain of its facets reduce modularity.

Cooperative aspect-oriented programming (Co-AOP) [Hoffman and Eugster 2009] has been proposed as a methodology to lessen obliviousness in return for increased modularity. In the Co-AOP methodology, the base code is aware of the presence of aspects, where appropriate, and guides the application of aspects to the base code. Co-AOP allows for a balanced mix of obliviousness and explicitness, with the goal that the benefits of obliviousness can be gained where appropriate, while avoiding or lessening obliviousness where it causes problems. We explore the benefits and drawbacks of the Co-AOP methodology in the context of a previously proposed language construct known as explicit join points (EJPs) [Hoffman and Eugster 2007]. EJPs provide a mechanism for the base code to explicitly guide the application of aspects onto either specific statements or onto specific blocks of code. EJPs have been proposed as a fully backwards compatible language extension to AspectJ, and thus allows the use of either the oblivious AOP features of AspectJ or the explicit techniques of EJPs. In this paper we use EJPs to fully remove obliviousness from an aspect-oriented implementation of certain crosscutting concerns, and then study the resulting effects on software size, coupling, separation of concerns, and modularity. This reveals insights into how to strike a favorable balance between obliviousness and the modularity of base code and aspects. We present hypotheses governing the proper use of Co-AOP, which state when obliviousness or explicitness should be employed to improve attributes of code quality, such as size, coupling, cohesion, the separation of concerns, and reusability. We also compare the fully explicit approach and the fully oblivious approach to the Co-AOP methodology where explicitness was only used where it was likely to improve code quality, and use this data to evaluate the hypotheses.

In summary, this paper seeks to increase understanding of aspect modularity and how AOP can be adjusted to facilitate highly reusable aspect components, and we seek to better understand the following key hypothesis:

**Appropriately employing mechanisms for base code and aspects to cooperatively implement crosscutting concerns causes software modularity and reusability to increase, despite a reduction in obliviousness.**

To accomplish the goals of this study, this paper contributes the following:

---

Three industrial strength Java applications are refactored so that the exception handling crosscutting concern is implemented with aspects involving various lev-
els of EJPs: (a) without EJPs, (b) with EJPs employed using a fully explicit strategy, (c) with EJPs referenced either explicitly or obliviously as appropriate. More precisely, we expand upon a prior study [Castor Filho et al. 2006] in which these same applications had their exception handlers aspectized without EJPs.

— Additionally, the JHotDraw Java application is refactored to aspectize the following crosscutting concerns to use EJPs (both following a fully explicit strategy and the more balanced Co-AOP methodology): contract enforcement, undo/redo, persistence, and selection changed notification (an instance of the observer pattern).

— We directly compare the EJP methods (fully explicit and Co-AOP) to the AspectJ method and to the original code. Metrics targeting software quality attributes such as coupling, cohesion, complexity, the separation of concerns, and reusability are used as the basis for our multidimensional assessment of EJPs.

— We study the modularity properties of one of the applications in particular using design structure matrices (DSMs). We analyze this data to better understand the overarching connections between obliviousness and modularity, aiding in the understanding of the tradeoffs that exist between these two concepts.

— We conclude with general guiding principles to maximize the benefits of Co-AOP.

Roadmap. Section 2 discusses challenges to aspect modularity and possible solutions. Section 3 overviews Cooperative AOP and explicit join points, while Section 4 presents the setting of our study. Section 5 presents and discusses our results. An in-depth modularity analysis using design structure matrices is presented in Section 6. Section 7 enumerates threats to internal and external validity, and discusses our controls to mitigate these threats. Section 8 discusses related studies, and Section 9 concludes our work.

2. ASPECTS AND MODULARITY

This section overviews known challenges in creating reusable aspect components and discusses prior proposals from the research literature to tackle these challenges.

2.1 Challenges

AOP as expressed in AspectJ has been proven to be effective in lexically separating crosscutting concerns from the base code. Nevertheless, the effects that this separation has on other key factors such as modularity, coupling, and software development processes is not fully understood. Empirical studies analyzing some of these effects have recently begun to emerge: Garcia et al. [Garcia et al. 2004] have studied the separation of concerns in multi-agent systems. Garcia et al. [Garcia et al. 2005] and Cacho et al. [Cacho et al. 2006] have studied the impact on modularity when following AOP-specific design patterns. Sullivan et al. [Sullivan et al. 2005] have studied how obliviousness affects software modularity and have proposed crosscutting programming interfaces to improve modularity. Tonella and Ceccato [Tonella and Ceccato 2005] have studied the impact on software quality when certain crosscutting concerns (as exhibited in the code by methods belonging to the same crosscutting concern are scattered across several interfaces) are implemented with aspects. Castor Filho et al. [Castor Filho et al. 2006] have studied
how the aspectization of exception handling affects software quality. Greenwood et al. [Greenwood et al. 2007] have studied design stability in evolving applications in the presence of aspectual decompositions.

One problem highlighted by these studies is that even though the aspectization of crosscutting concerns couplings can remove lexical dependencies between classes within base code, new lexical dependencies are introduced between aspects and the base code. This coupling is due to the complexity inherent in precisely describing sets of join points (program locations) when these join points have no explicit name. Instead, the descriptions of which join points to advise, termed pointcuts, must rely on matching against join point type (method call, field access, etc.) and pattern matching against type names and identifiers.

The anonymity of join points makes it difficult to define pointcuts such that they anchor crosscutting logic precisely where needed without unintentionally matching additional join points. When pointcuts are defined, an inherent tradeoff must be made between precision and pointcut stability:

—On the one hand, the pointcut description language of AspectJ has been shown to be reasonably powerful in picking out very specific join points. However, as pointcut precision increases, so does its potential for fragility [Sullivan et al. 2005]. This means that it becomes more likely for the semantics of a pointcut to change as the base code evolves [Stoerzer and Graf 2005]. This pointcut induced fragility couples aspects to base code, requiring the aspects’ pointcuts to be revisited as the base code changes and thereby reducing the aspects’ modularity. One recent study [Greenwood et al. 2007] observed that pointcut fragility is one of the foremost causes of so called “ripple effects” and recommended more expressive and semantic-based pointcuts.

—On the other hand, using pointcuts that are too general may cause advice to be applied unintentionally and thus break program semantics. Finding the appropriate balance between precision and generality while also strengthening other software quality factors is a non-trivial challenge in aspect-oriented software development. Additionally, when base code is developed without a priori planning for the aspectualization of certain concerns, the resulting aspects can become tightly coupled to the base code and even tangled with other concerns [Sullivan et al. 2005; Castor Filho et al. 2006; Greenwood et al. 2007].

The fragile pointcut problem is magnified in AspectJ, in which multiple join points cannot be advised as an atomic unit, giving rise to the state–point separation problem [Sullivan et al. 2005]. This problem describes the situation where aspects need to access dynamic state scattered across multiple join points in the base code, resulting in “multi-stage” advice where aspects store captured state in a thread-local cache until the final join point is reached, at which all required dynamic state is available in the cache or from the advised join point. Another, related, problem is that arbitrary blocks of code cannot be advised, which results in artificial method refactorings that decrease cohesion [Castor Filho et al. 2006].

The relationship between aspect inheritance and pointcut definitions as expressed within AspectJ also limits aspect modularity. AspectJ allows for abstract aspects to declare abstract pointcuts without specifying concrete expressions indicating which join points the abstract pointcuts match. Concrete aspects then extend the
abstract aspect and specify concrete expressions for the abstract pointcuts. The abstract pointcut effectively becomes the disjunction of all of the expressions in the concrete aspects that extended the abstract aspect. However, abstract pointcuts cannot be shared between different abstract aspects, as they can only be shared through inheritance (an aspect cannot refer to an abstract pointcut in another aspect). This results in either unnecessary coupling between unrelated aspects or the redefinition of similar pointcuts [Hoffman and Eugster 2007], aggravating the fragile pointcut problem. This precludes the creation of abstract pointcut interfaces that are shared by different crosscutting concerns (or different implementations of the same crosscutting concern). Consequently, the aspectual logic cannot be decoupled from the pointcuts describing where to apply that logic, preventing the creation of interchangeable, reusable aspect libraries in the case of libraries that employ non-homogeneous pointcuts.

2.2 Known Strategies and Limitations

Primarily, the proposed strategies for mitigating the above challenges within AspectJ focus on ways in which obliviousness can be preserved while aspect modularity can be increased. Gudmundson et al. [Gudmundson and Kiczales 2001] propose pointcut interfaces and allow pointcuts to be defined closer to or even within base code, moving fragile pointcuts close to the code they are describing, although not actually decreasing fragility. In contrast, Aldrich proposes open modules [Aldrich 2005], where the only advisable join points are those exported by base code. If base code only exports robust pointcuts, the fragile pointcut problem can be mitigated to some degree, although it does not address the state–point separation problem. A similar approach is proposed by Kiczales and Mezini [Kiczales and Mezini 2005] where whole program analysis is used to infer and document crosscutting interfaces. These interfaces help programmers to understand the implications of changing base code or aspects with respect to the semantics of crosscutting concerns. While the above approaches mitigate the fragile pointcut problem to some degree, they do not address the inability in AspectJ for pointcuts in one aspect to refer to abstract pointcuts in other aspects, limiting pointcut expression reusability; pluggable, reusable aspect components are still a challenge to build.

In XPIs [Sullivan et al. 2010; Griswold et al. 2006], abstract design rules are specified and developers conform base code to these design rules. In this way, stable pointcuts can be written, owing to the structure provided by the design rules in the base code. However, not all design rules can be enforced in AspectJ, relying instead on developers to correctly follow the design rules when creating and refactoring code. HyperJ [Ossher and Tarr 2000] provides for on-demand remodularization through multidimensional separation of concerns. Multiple dimensions of concerns (simultaneously different decompositions) are explicitly identified, encapsulated, and integrated. Reusable concern implementations are represented as hyperslice packages. Hypermodules then describe how to integrate these concerns with each other. CaesarJ [Aricic et al. 2006] combines ideas from aspect-oriented and feature-oriented programming to facilitate modularity. Collaboration interfaces are introduced in order to decouple aspect implementation from aspect binding. Along similar lines, Classpects [Rajan and Sullivan 2005] also unifies the notion of aspects and classes but goes one step further by removing anonymity from advice
3. EXPLICIT JOIN POINTS AND COOPERATIVE AOP

This section introduces and gives examples for explicit join points (EJPs) and cooperative aspect-oriented programming (Co-AOP). As a thorough discussion on the motivations behind and specifics of EJPs and Co-AOP has been previously published [Hoffman and Eugster 2009], herein we only present those aspects of EJPs and Co-AOP that are most relevant to this paper in evaluating the tradeoffs between obliviousness and modularity. In this section we also propose guiding principles governing the proper use of EJPs and discuss how EJPs and Co-AOP simplify multi-stage advice.

3.1 Explicit Join Points (EJPs)

Herein we give a brief introduction and overview to explicit join points (EJPs) with examples, and refer to prior work [Hoffman and Eugster 2009] for comprehensive treatment of the motivations, design principles, formal language syntax, and semantics of EJPs. EJPs extend AspectJ such that new join points can be declared within aspect interfaces, using syntax similar to an abstract method declaration. These interfaces model the interactions between base code and aspects in terms of the expected semantics of the crosscutting concerns as well as state information required by these concerns. Base code explicitly references explicit join points where crosscutting concerns should apply. EJPs allow for the base code to shape aspects in new ways: advice can be parameterized by value and type, arbitrary blocks of code can be advised, and pointcuts can be defined piece-wise within base code using new constructs that further reduce pointcut fragility.

The core idea of EJPs is to represent crosscutting concerns via explicit join point interfaces that act as mediators between aspects and base code, as depicted in Figure 1. Each interface representing a crosscutting concern needs to be carefully designed such that it minimally embodies the required interaction, as dictated by the information hiding principle [Parnas 1972]. Interfaces can also represent more general advising patterns that many crosscutting concerns may be interested in advising.
aspect ExampleAspectInterface {
    // Syntax for EJP declarations:
    [JavaModifiers] [scoped] joinpoint
    ["<" GenericTypeParameters ">"]
    ReturnType EJPName ["(" Formals ")"]
    pointcutargs ["(" DefaultPCArgument [""," DefaultPCArgument "]")] 
    handles ExceptionTypeList
    throws ExceptionTypeList
    ["=" DefaultValueExpression "]
    // Syntax of DefaultPCArgument:
    ArgumentName ["(" Formals ")"]
}
aspect ExampleAspect {
    // Examples of ejp and ejpscope pointcut modifiers:
    pointcut pc1(...): call ( ejp( EJPName )) && args (...);
    pointcut pc2(): cflow( call( ejpscope( ScopedEJPName )));
}
class ExampleClass {
    void exampleMethod () {
        // Syntax for using EJPs in base code:
        AspectName ["." [GenericTypeArguments] EJPName ["(" Arguments ")"]
        pointcutargs PCArgument [""," PCArgument "]]
        // Syntax for using scoped EJPs in base code:
        AspectName ["." [GenericTypeArguments] EJPName ["(" Arguments ")"]
        pointcutargs PCArgument [""," PCArgument "]]
        // block of code to be advised
        }
        // Syntax of GenericTypeArguments:
        ["<" GenericTypeArgument ["," GenericTypeArgument ">"]
        // Syntax of PCArgument:
        ArgumentName ["(" Formals ")"]
    }
}

Fig. 2. Overview of syntax for the EJP language extension of AspectJ: keywords are bolded, other concrete syntax is in quotes, syntax productions are underlined, optional constructions are bracketed, * designates that the prior construct can be repeated zero or more times. Adapted from [Hoffman and Eugster 2009].

EJPs introduce a new join point type into the quantification model of AspectJ. Unlike anonymous join points that are exposed automatically by the AspectJ compiler, EJPs are explicitly declared by the programmer, are given a unique combination of name and type signature (allowing for overloading), and are referenced explicitly in the base code or within aspects advising base code.

Figure 2 summarizes the syntax extensions that EJPs add to AspectJ, including syntax for non-scoped and scoped EJP declarations, EJP pointcut modifiers, referencing EJPs in base code, and other advanced features such as generics and pointcut arguments. Note that ExampleAspectInterface, ExampleAspect, ExampleClass, and exampleMethod demonstrate the required context for the specified syntax, but are not part of the syntax specification and are formatted as code rather than in the syntax format. A comprehensive introduction to these features can be found in prior work [Hoffman and Eugster 2009]; we give an example below to concretely introduce the EJP features that are most relevant to this paper.

Figure 3 gives an example of an EJP interface EH for exception handling with two EJP declarations discardAll (line 4) and catchAllAndPrint (line 6), the use of these EJPs in base code, and an aspect Handler that implements that EJP interface. The example is based on common exception handling patterns found in ACM Journal Name, Vol. V, No. N, Month 20YY.
Trading Obliviousness for Modularity with Cooperative Aspect-oriented Program.m.

1 // Example EJP interface
2 aspect EH {
3    // Exceptions in the control flow of the block of the EJP ref will be ignored.
4    public scoped joinpoint void discardAll() handles Throwable; //Exceptions within the scope of the block of the EJP ref will be printed.
5    public scoped joinpoint void catchAllAndPrint(String context) handles Throwable;
6 }
7 // Example base code that references an EJP
8 class C1 {
9    public void M1() {
10       EH.discardAll(); // Any method calls in the cflow of this block (the block surrounded by
11       // the curly braces associated with the EJP) will discard all exceptions
12      );
13       EH.catchAllAndPrint("Caught within M1") { // Any exception thrown in this block will be caught and printed
14          );
15    }
16 }
17 }
18 // Example aspect implementing the example EJP interface
19 aspect Handler { Object around(): call(*,..(),) & & cflow(call(ejpscope(EH.discardAll))) { try {
20       return proceed();
21    } catch(Throwable e) { }
22    }
23 Object around(String msg): call(ejpscope(EH.catchAllAndPrint)) & & args(msg) { try {
24       return proceed(msg);
25    } catch(Throwable e) { System.out.println(msg + ": "+e); }
26    }
27 }
28 // Other aspects implementing the EJP interface could also be used.
29 // For example, an aspect that logs all discarded exceptions.

Fig. 3. Introductory EJP example based on common exception handling patterns found in the Java Pet Store application

the Java Pet Store application, which is one of the applications in the refactoring study elaborated further on in this paper. The discardAll EJP allows all exceptions thrown by method calls to be ignored and not propagated within the control flow of regions of the program designated by base code. The catchAllAndPrint EJP provides an exception handler that catches all exceptions and prints them. This EJP is an example of how base code can easily parameterize the behavior of aspects. In this case the contextual log message can be specified in the base code, as shown by "Caught within M1" (line 15).

EJPs can thus be referenced or “instantiated” in the base code using syntax similar to a static method invocation. Adding the scoped attribute to an EJP signature indicates that references in the base code to this EJP can further attach blocks of code that should be advised. The syntax for associating such a block of code with an EJP is then similar to an anonymous class declaration, as shown by EH.discardAll() {...} or EH.catchAllAndPrint(...) {...} in method M1 of class C1 on lines 11 and 15 respectively in the example.

An EJP declaration allows for specifying that data will be returned to base code from the crosscutting concern through the return type of the EJP. In the case of a non-void return type a default value can be specified to handle the case where the EJP is not advised (otherwise null is used in that case if no default value is
specified). Two types of exception handling constraints can also be specified within
an EJP declaration: First, EJP declarations can use a throws list to constrain
code referencing EJPs that they must handle (or further propagate) certain checked
exceptions. Second, EJP declarations can use a handles list as seen in the example
to specify that for each checked exception in the list, at least one advice must
catch that exception; the handles list also acts as a promise to code referencing
the scoped EJP that any checked exceptions in the list will be properly handled.
In the example in Figure 3, the EJP interface is promising to handle exceptions
within its scope of any type: the discardAll EJP simply handles all throwables
by discarding them, and the catchAllAndPrint EJP prints out all throwables.
Thus these two EJP declarations on lines 4 and 6 are defined to handle the type
Throwable. The EJP interface could have been more specific with the type, or
could have parameterized this type using Java generics, allowing the base code
to specify the type of the exception to be handled within the scope of each EJP
reference.

EJPs with a handles clause thus have the same exception handling semantics in
base code as a catch statement, while offering the advantage of being abstracted
from the handler implementation. Through the EJP return value and handles
clause features, the EJP declaration becomes more than a constraint on the base
code; it also becomes a constraint on the advising aspect, requiring the advising
aspect to return an object of a specified type or to handle one or more checked
exceptions.

Pointcuts can then capture explicit join points by using the ejp and ejpscope
designators that specifically match EJP references and the blocks of code attached
to EJP references, respectively. In the example, the advice in Handler for the
discardAll joinpoint (line 22) triggers upon all method calls within the control
flow of that scoped joinpoint. In contrast, the advice for catchAllAndPrint
(line 27) only handles exceptions propagating all the way up to the level of the
joinpoint scope due to the absence of the cflow designator.

Having two designators as opposed to just one allows for precisely indicating
whether or not it is the actual EJP reference (epj designator) or the block of
code attached to the EJP reference (ejpscope designator) that should be advised.
These differing behaviors for the two different designators apply when used within
pointcuts associated with after and around advice. Additionally, the ejpscope
designator will only match EJP references whose corresponding EJP signature con-
tains the scoped attribute, whereas the ejp designator will match either scoped
or non-scoped EJP references. Advice inject logic before, around, or after EJP
references or the blocks of code associated with EJP references by combining the
ejp and ejpscope designators with the call and execution designators. The
only difference between using call and execution is with respect to advice prece-
dence, following the same precedence rules for call and execution as in AspectJ.
EJP pointcut designators are also fully compatible with complex pointcuts such as
cflow, as seen in the example. Lexical pointcut designators, such as withincode,
can be used with the ejpscope designator to match code lexically within the blocks
of code associated with EJP references. The arguments of an EJP are accessible
via the `args` designator as shown in the `Handler` aspect for the advice applied to `catchAllAndPrint`.

*Pointcut parameters* are an advanced feature of EJPs that allow for pointcuts to be defined piece-wise when EJPs are referenced in base code; remaining pointcuts that must be fragile can thus be moved closer to the join points being advised. As this feature has not been used in the programs studied in the article, we refer the reader to an earlier paper for further discussion and an example use of pointcut parameters [Hoffman and Eugster 2009].

### 3.2 Cooperative AOP

As discussed in Section 2, there are significant challenges in using aspect-oriented programming to create modular, reusable aspect components. *Cooperative aspect-oriented programming* (Co-AOP) is a programming methodology motivated by these challenges, with the hypothesis that reduced obliviousness in the base code can be traded for increased aspect modularity and reusability *in certain scenarios*. In Co-AOP, a complete separation of concerns is not taken as an absolute requirement. The base code and aspects cooperate together to implement crosscutting concerns. Crosscutting concerns are modeled as abstract interfaces (obliviously or explicitly), and programmers specify where and how these interfaces interact with base code either through code conformance to design rules (e.g., XPIs) or through explicit lexical references (e.g., EJPs) or via oblivious injection of explicit references to interfaces.

Co-AOP intuitively gains the following benefits from the use of interfaces:

1. The interface becomes a point of design stability between aspects and base code such that each can evolve independently. In this way direct coupling between aspects and base code is greatly reduced.

2. Aspects implementing the interfaces are freely interchangeable and can be chosen at compile or load time, facilitating reusable aspect component libraries.

3. One interface can easily be adapted to another interface, by using an aspect to obliviously adapt references to one interface to conform to some other interface. A single aspect can implement an interface adaptation without requiring any knowledge about how the interface being adapted is used within base code. In contrast, without an interface all related join points in base code would have to be adapted instead, thus making such an adaptation fragile to changes or additions within the base code.

While Co-AOP advocates a reduction in the obliviousness of base code, this does not necessarily require an explicit presence of the concerns in the base code. For example, crosscut programming interfaces (XPIs) [Sullivan et al. 2010] have been proposed wherein design rules form abstract interfaces that both base code and aspects adhere to. This cooperation (at a design level) to implement crosscutting concerns is thus a form of Co-AOP. XPIs remove design obliviousness, but preserve lexical obliviousness. However, AspectJ is not able to fully enforce conformance to all design rules at compile time; instead some design rules must be enforced at runtime through the use of the `cflow` construct [Sullivan et al. 2010]. Additionally, XPIs are constrained by the limitations of the AspectJ join point model, and
thus in order to expose state information for complex crosscutting concerns, code blocks must be extracted into new methods, which has been shown previously to decrease software cohesion [Castor Filho et al. 2006]. For this reason, where there is significant interaction or coupling between the aspects and the base code, we advocate that interfaces are represented more explicitly within base code than as is done with XPIs.

Co-AOP can be practiced within the unextended AspectJ language using the following approach: Aspect interfaces are modeled by classes containing static methods that have empty method bodies. Base code then references these aspect interfaces (and passes state information) by using standard Java syntax to call static methods within aspect interfaces. Advice then implement aspect interfaces by using the \texttt{call} pointcut designator to match calls to static methods within aspect interfaces.

While this approach does allow base code to participate in the implementation of crosscutting concerns by referencing aspect interfaces, the features of the EJP language extension provide more powerful tools for both the base code and aspects to cooperate. Specifically, EJPs allow base code and aspects to cooperate in ways that were previously not possible: Arbitrary blocks of code can be attached to EJP references in base code, allowing base code to assign crosscutting semantics to these blocks. Examples are contract enforcement, undo/redo logic, transaction handling—a case whose solvability with pure AspectJ has been debated [Kienzle and Guerraoui 2002]—or exception handling. Of these examples this study considers the exception handling, contract enforcement, and undo/redo crosscutting concerns. The signature of the EJP allows for parameterization of advice by value, and the combination of EJPs with generics allows advice to be parameterized by type. Value parameterization mitigates the state–point separation problem, while type parameterization allows advice to be more reusable. Additionally, EJPs allow base code to guide aspects in where to apply advice through pointcut parameters, and also allow aspects to statically enforce new design constraints upon the base code that they advise.

These features, combined with the ability to use the full power of aspect quantification with EJPs, empower the full application of the Co-AOP methodology. These features are also key distinctions between Co-AOP and the traditional OO and event-style programming methodologies.

In summary, the Co-AOP methodology encompasses a spectrum of techniques wherein obliviousness is reduced in return for increased modularity. Co-AOP implemented using design abstractions but not lexical explicitness of interfaces (e.g., XPIs) is best suited where there is not significant semantic interaction or coupling between the aspects and the base code, or where the weaving of the crosscutting concerns are more coarse-grained (e.g., method-level) and can effectively be captured through the existing quantification features of AspectJ. XPIs and EJPs are fully compatible with each other, and thus each can be used where most appropriate, to maximize code quality. This study focuses on the impact in code quality for the fully explicit Co-AOP techniques, as the impact of less explicit techniques have already been documented previously [Sullivan et al. 2010].
It is important to note that Co-AOP does not advocate following a fully explicit strategy, but rather only introducing as much explicitness in the base code that is necessary in order to improve modularity and reduce unnecessary dependencies. In the next section, we present hypotheses stating where it is likely to be more effective to employ fully explicit techniques (e.g., EJPs) when following Co-AOP. These hypotheses are then evaluated and substantiated in the remainder of the paper.

3.3 Proper Use of EJPs for Cooperative AOP

When employing EJPs to implement crosscutting concerns, the base code explicitly references the EJPs related to the crosscutting concerns that need to be implemented. This induces coupling between the base code and the EJP declarations. For this reason, EJPs are not intended as an unconditional replacement of oblivious aspects or XPIs. Certainly many aspects can be effectively implemented using broad-ranging pointcuts that succinctly capture tens, hundreds, or even thousands of join points while remaining robust to base code evolution. In the Co-AOP methodology, EJPs are intended to be used only for the situations where there is significant interaction or coupling between the aspects and the base code. We propose the following hypotheses that govern the appropriate application of EJPs:

Hypothesis 3.3.1 Code modularity, reuse, or complexity is improved when employing EJPs where the pointcuts are complex and specific [Stoerzer and Graf 2005], including cases of multi-stage advice.

Hypothesis 3.3.2 Code modularity, reuse, or complexity is improved when employing EJPs where arbitrary blocks of code need to be advised (e.g., exception handler blocks or a code block for only one branch of a conditional inside of a method with many conditional blocks).

Hypothesis 3.3.3 Code modularity, reuse, or complexity is improved when employing EJPs where the advice are not harmless [Dantas and Walker 2006].

Hypothesis 3.3.4 Code modularity, reuse, or complexity is improved when employing EJPs where there are interactions of multiple crosscutting concerns [Sanen et al. 2006], such as when advice apply to the same joinpoint, or rely on or even modify the same state information in base code.

To further clarify Hypothesis 3.3.4, Dantas and Walker define an advice as harmless if it does not influence the final value of the computation in the base code. Note that harmless advice may produce additional values (e.g., perform I/O inside the aspect), affect the execution timing, or change the termination behavior of the base code (it may cause a particular method to never return a value). The goal of harmless advice is to allow local correctness reasoning on the base code without knowledge of all of the advice that may be applied (now or in the future) to the base code. When an advice is not harmless, any correctness reasoning must have knowledge of the advice in addition to the base code. The motivation behind Hypothesis 3.3.4 is that using an EJP in the base code models the anticipated semantic effects of any applicable advice and thereby restores the ability to perform local correctness reasoning on the base code without a knowledge of all advice.
public aspect TimestampAspectHealthWatcher extends AbstractTimestampAspect {
  private long searchTimestamp(String tableName, String id) {
    Statement stmt = null;
    ResultSet resultSet = null;
    long answer = 0;
    try {
      String sql = "SELECT TIMESTAMP FROM " + tableName + " where codigo=" + id;
      PersistenceMechanismRDBMS pm = PersistenceMechanismRDBMS.getInstance();
      stmt = (Statement) pm.getCommunicationChannel();
      resultSet = stmt.executeQuery(sql);
      if (resultSet.next()) {
        answer = resultSet.getLong(1);
      } else {
        throw new RuntimeException("...");
      }
      return answer;
    } catch (Exception ex) {
      throw new SoftException(ex);
    } finally {
      try {
        if (resultSet != null) resultSet.close();
        if (stmt != null) stmt.close();
      } catch (Exception ex) {  
        throw new SoftException(ex);  
      }
    }
  }
}

The motivation behind Hypothesis 3.3.5 comes from observations from a prior study by Castor Filho et al. [Castor Filho et al. 2006], wherein in the presence of multiple complex concerns, additional tangling and code complexity (as defined by the study metrics) was observed. This increased tangling was caused by the need to refactor code fragments to new methods to expose join points needed by one concern but not by the other(s). The explicit presence in the base code of each concern through EJP references prevents the need to refactor code fragments into new methods, thereby allowing multiple concerns to be composed more cleanly in the base code.

3.4 Multi-stage Advice and Complex Pointcuts Example

A key principle in properly applying the Co-AOP methodology is recognizing where there is complex or fragile interaction between oblivious aspects and the base code, and then simplifying and making these interactions more robust by introducing explicitness into the base code, for example through EJP references. To this end, in this section we provide a detailed example from one of the applications (Health Watcher) from our study of Section 4. In this example, the goal is to aspectize the catch and finally code blocks.

Figure 4 lists the original base code before aspectization of exception handling. The finally handler in this example ensures that the Statement and ResultSet

```
Fig. 4. Complex advice example: Original base code before aspectizing the finally handler, taken from the Health Watcher application; note that this is also an example where multiple crosscutting concerns interact—Health Watcher implements the persistence crosscutting concern using an aspect, and this aspect itself contains a finally handler and intersects with the exception handling crosscutting concern; thus, in the context of the exception handling concern, the TimestampAspectHealthWatcher aspect is termed base code.
```
public aspect TimestampAspectHealthWatcher extends AbstractTimestampAspect {
    ...

    private long searchTimestamp(String tableName, String id) {
        Statement stmt = null;
        ResultSet resultSet = null;
        long answer = 0;
        String sql = "SELECT TIMESTAMP FROM " + tableName + " where codigo=" + id;
        PersistenceMechanismRDBMS pm = PersistenceMechanismRDBMS.getInstance();
        stmt = (Statement) pm.getCommunicationChannel();
        resultSet = stmt.executeQuery(sql);
        if (resultSet.next()) {
            answer = resultSet.getLong(1);
        } else {
            throw new RuntimeException("...");
        }

        return answer;
    }

    privileged aspect OtherAspectsExceptionHandlingAspect {

        Map statement = new HashMap();
        Map resultSet = new HashMap();

        // Part 1 of multi-stage advice, save return value
        // of getCommunicationChannel call on line 9
        after() returning(Statement st) :
            call(Object PersistenceMechanismRDBMS.getCommunicationChannel()) &&
            withincode(\* aspects..TimestampAspectHealthWatcher.searchTimestamp())
        { 
            // Save inner method variable to thread-local value
            statement.put(String.valueOf(Thread.currentThread().getId()), st);
        }

        // Part 2 of multi-stage advice, save return value
        // of executeQuery call on line 10
        after() returning(ResultSet rs) :
            call(ResultSet Statement.executeQuery(..)) &&
            withincode(\* aspects..TimestampAspectHealthWatcher.searchTimestamp())
        {
            // Save inner method variable to local(multi-thread)
            resultSet.put(String.valueOf(Thread.currentThread().getId()), rs);
        }

        // Final part of multi-stage advice, implement
        // catch/finally logic using thread-local data
        Object around() :
            execution(\* aspects..TimestampAspectHealthWatcher.searchTimestamp(..))
        {
            try {
                return proceed();
            } catch (Exception ex) {
                throw new SoftException(ex);
            }

        try {
            String curThread = String.valueOf(Thread.currentThread().getId());
            Statement st = (Statement) statement.get(curThread);
            ResultSet rs = (ResultSet) resultSet.get(curThread);
            if (rs != null) rs.close();
            if (st != null) st.close();
        } catch (Exception ex) {
            throw new SoftException(ex);
        }

        finally {
            try {
                String curThread = String.valueOf(Thread.currentThread().getId());
                Statement st = (Statement) statement.get(curThread);
                ResultSet rs = (ResultSet) resultSet.get(curThread);
                if (rs != null) rs.close();
                if (st != null) st.close();
            } catch (Exception ex) {
                throw new SoftException(ex);
            }

        }
    }
}

Fig. 5. Complex advice example: Aspectization of the finally handler using AspectJ

ACM Journal Name, Vol. V, No. N, Month 20YY.
//Generic EJP interface and its implementation, can easily be reused in other apps

public aspect SQLCleanupExceptionHandling {

// Makes sure that the sql objects passed are closed before the scope exits
public scoped joinpoint ejpCleanupSQLObjects(
    Statement[] stmt, ResultSet[] resultSet);

// Implementation of the above EJP declaration (EJP interface)
after(Statement[] stmt, ResultSet[] resultSet) {
    for (ResultSet r : resultSet){
        if (r != null) r.close();
    }
    for (Statement s : stmt){
        if (s != null) s.close();
    }
}

//Base code that uses the EJP interface
public aspect TimestampAspectHealthWatcher extends AbstractTimestampAspect {
    ...
    private long searchTimestamp(String tableName, String id) {
        Statement[] stmt = new Statement[] {null};
        ResultSet[] resultSet = new ResultSet[] {null};
        long answer = 0;
        ExceptionUtil.ejbSoftenException(Exception.class, SoftException.class) {
            SQLCleanupExceptionHandling.ejbCleanupSQLObjects(stmt,resultSet) {
                String sql = "SELECT TIMESTAMP FROM " + tableName + " where codigo=" + id;
                PersistenceMechanismRDBMS pm = PersistenceMechanismRDBMS.getInstance();
                stmt[0] = (Statement) pm.getCommunicationChannel();
                resultSet[0] = stmt[0].executeQuery(sql);
                if (resultSet[0].next()) {
                    answer = resultSet[0].getLong(1);
                } else {
                    throw new RuntimeException("...");
                }
            } return answer;
        }
    }

Fig. 6. Complex advice example: Aspectization of the finally handler by applying Co-AOP via EJPs

objects (obtained on lines 9 and 10) are always closed whether or not the rest of the method succeeds.

Figure 5 shows the aspectization of the finally handler as published in a prior study by Castor Filho et al. [Castor Filho et al. 2006]. Aspectizing this finally handler using AspectJ proves to be a challenge, because the aspect needs access to multiple data variables not exposed by any one join point. Thus, the aspect must use a complex and fragile “multi-stage” advice where the earlier stages capture the needed data variables in thread-local storage, and then the final stage implements the actual logic. The pointcuts on lines 25 and 34 are also quite specific, having to use the call primitive pointcut to match the specific call within the method that produced the value needed later, and also having to use withincode to narrow the scope of the pointcut to the desired method containing the original finally handler. If the base code evolves, then these types of pointcuts are quite fragile and might also have to be changed. For example, if the method was changed to add another call to executeQuery there would potentially be a different ResultSet instance that would need to be tracked and closed, but the current aspect code would only close the last ResultSet instance created within the method.
Figure 6 shows the aspectization of the finally handler using the Co-AOP methodology and EJPs. An aspect SQLCleanupExceptionHandling is defined to model the EJP interface and implementation. The scoped EJP declaration on line 5 provides the explicit means whereby base code can declare they want certain objects to be cleaned up at the end of each block of code associated with that EJP reference. The EJP takes arrays of Statement and ResultSet objects so that the base code can specify multiple values at once, and more importantly for the base code to effectively pass certain local variables by reference (in Java arrays are objects and are passed by reference). This allows the base code to declare the variables that need to be cleaned up and reference the EJP and enter a new scope (as we do in the example on line 23) before the values for these variables are actually known. In the example the variables are declared on lines 19 and 20 before the EJP reference, but then the values are populated after the EJP reference on lines 26 and 27. Passing by reference in this way is not required and should only be used if it simplifies the overall code.

The EJP interface in this case is implemented within the same aspect, by the advice on line 8. Note that the required pointcut here refers only to the one EJP declaration inside the EJP interface, and does not reference anything specific to the application. It thus eliminates the complexity and fragility of the AspectJ method, while also facilitating reuse. The base code now explicitly refers to the EJP interface through the scoped EJP reference on line 23. The base code explicitly passes those values required by the interface, thereby avoiding the need for complex multi-stage advice to “extract” the proper values from the base code.

If we needed to adapt the ejpCleanupSQLObjects EJP declaration to conform to some other EJP interface (e.g., an EJP declaration that took the resultSet first and then the stmt), this could be accomplished by using an around advice matching the ejpCleanupSQLObjects EJP, and within the around advice referencing the EJP declaration in the new interface. Without EJPs, such adaptation is likely to be more intrusive, because the pointcuts of the interface being adapted to must be modified, in addition to any adaptation logic.

This example demonstrates how obliviousness can introduce significant complexity and fragility, and how this could be mitigated by applying Co-AOP using EJPs. Although the example does provide supporting arguments for Hypothesis 3.3.1, Hypothesis 3.3.2, and Hypothesis 3.3.5, this example does not substantiate how Co-AOP affects the quality of the entire application or how these effects would generalize to other applications. To this end, we perform a more in-depth study of multiple applications in the next section.

4. STUDY SETTING

A prior study [Castor Filho et al. 2006] (herein termed the “Castor Filho study”) quantitatively compared the benefits and drawbacks of using AspectJ to implement exception handling in an oblivious manner. In this static-analysis study [Zelkowitz and Wallace 1998] four applications were refactored and then evaluated to see how cohesion, coupling, conciseness, complexity, and separation of concerns were affected.
We build upon the Castor Filho study by refactoring three of the four applications using EJPs to implement exception handling, starting from source code provided by the authors of the Castor Filho study. We implement exception handling in the EJP version without any oblivious aspects in order to highlight the differences between the oblivious and explicit approach. We then use a superset of the metrics employed in the Castor Filho study to gain insight into the relationship between modularity and obliviousness.

Additionally, we study the JHotDraw application [Riehle 2000] by implementing four crosscutting concerns using EJPs, comparing the above mentioned code quality attributes to both the Java version and also a previously published version where AspectJ was used to implement these same crosscutting concerns [van Deursen et al. 2005; Marin et al. 2009].

4.1 Technique

In the Castor Filho study, exception handling code was refactored according to the following strategy: new advice were created for each catch or finally block, and advice were then combined where reuse was possible. Advices were organized so that exception handling logic for either a single class (Telestrada), a single package (Pet Store), or a single concern (Health Watcher) were contained within a single aspect. Code to detect and throw exceptions was not aspectized.

Our strategy for refactoring closely follows that of the Castor Filho study for both advice creation and organization. We modified the strategy to exploit the benefits of EJPs by also creating a generic exception handling aspect, free of application-specific types and code, and using EJPs from that aspect whenever possible. We consider two methodologies in refactoring crosscutting concerns using EJPs:

—EJP: In this method, EJPs are used exclusively in order to highlight the benefits and drawbacks of a fully explicit approach
—Co-AOP: In this method, a mixture of oblivious aspects and EJPs are used, as governed by the guidelines proposed in Section 3.3.

These versions of the code will be compared to the original code version as well as a version where crosscutting concerns were refactored using fully oblivious aspects with AspectJ.

As in the Castor Filho study, exception handlers were implemented using after advice when possible, reverting to around advice when exceptions had to be caught but not propagated. Figure 7 demonstrates the general pattern of the Castor Filho study as well as the EJP approach. Note that in the AspectJ version, in order for the code to compile without errors, checked exceptions had to be softened [Kiczales et al. 2001] using the declare soft construct, whereas in the EJP version checked exception semantics are fully preserved due to the handles and throws EJP language features.

In both this study and the Castor Filho study the before and after versions of the code will behave the same way and produce the same output across all possible executions (including those having exceptional conditions). This constraint ensures a more fair comparison between the different versions.

With respect to our technique for JHotDraw, the existing refactoring in AspectJ, AJHotDraw, employed a combination of inter-type declarationss, pointcuts, and ad-
Fig. 7. Examples demonstrating how exception handling in Java is aspectized using either AspectJ or EJPs.

vice to implement the contract enforcement, undo/redo, persistence, and selection changed notification crosscutting concerns, as discussed in detail by van Deursen et al. and Marin et al. [van Deursen et al. 2005; Marin et al. 2009]. To create the fully explicit EJP refactoring of JHotDraw, we re-used the same intertype declarations, and then used an EJP reference everywhere in the base code that was advised by an aspect in AJHotDraw.

To create the Co-AOP versions of the four applications, we started with the fully explicit EJP versions, and then replaced some of the explicit EJP references using pointcuts and advice when all of the following conditions were met: (a) there was none or very little advice parameterization (e.g., no context-dependant error codes or messages), (b) the pointcuts required to inject the references could be considered both general (not tied to many concrete classes) and stable (single-stage advice not tied to intricate details of individual method implementations), and (c) more than two explicit EJP references could be eliminated by introducing the aspect to inject EJP references.

4.2 Generic EJP Exception Handling

Figures 8, 9, and 10 depict parts of the generic EJP exception handling interface, its implementation, and its use in the base code. The convert EJP represents a
aspect EH {
  // Promises to convert the type of an exception. 
  // Note the type parameterization via Java generics.
  public scoped joinpoint <FromType extends Throwable, ToType extends Throwable> 
    void convert(Class<FromType> from, Class<ToType> to) 
      handles FromType throws ToType;
  // Parameterized with a String, allowing base code to specify extra message to log
  public scoped joinpoint <ExceptionType extends Throwable> 
    void logAndRethrow(Class<ExceptionType> t, String msg)
      handles ExceptionType throws ExceptionType;
}

Fig. 8. Example of generic EJP interface

public static boolean executeSQLStatement(...) throws PopulateException {
  EH.convert(SQLException.class, PopulateException.class) {
    EH.logAndRethrow(SQLException.class, "Error executing statement: ") {
      // SQLExceptions will be logged and then converted to PopulateException 
      // and rethrown. The original body of method goes here, including return 
      // statement(s).
    }
  }
}

Fig. 9. Example from Pet Store application of using the generic EJP interface

common pattern in our target applications – converting the type of an exception 
to a more general type.

The EJP declaration in Figure 8 uses the FromType and ToType type variables to 
represent the type being converted from and to respectively. The use of variables of 
type Class are required so that advising aspects can check for and instantiate the 
appropriate exception types. Reflection could be avoided if AspectJ was enhanced 
to allow aspects access to the type variables of the join points they are advising. The 
use of handles and throws allows base code referencing the EJP to appropriately 
affect the checked exception handling constraints to match the defined semantics 
of the EJP. Note that the EJP declaration requires a return type, allowing aspects 
to communicate values back to the base code if necessary.

Figure 9 demonstrates how a generic EJP is referenced within base code. This 
example is based on an actual use of an EJP within the Pet Store application. The 
types being converted from and to are explicitly passed as arguments. We found 
that another useful parameter was a String description allowing the application to 
specify English descriptions of context when errors occur. The requirement for the 
aspects to obliviously use precise contextual messages was a source of significant 
fragility and hindered aspect reuse in the AspectJ versions of our target applications.

Type parameterization allows the implementation aspect in Figure 10 to remain 
free from any application-specific types. The advice matches when the block of 
code associated with the convert EJP exits due to any exception being thrown. 
The advice checks to see if the thrown exception matches the exception type that 
needs to be converted, and if so it instantiates and throws the new exception type. 
Combining generics with EJPs facilitated high levels of reuse, as discussed further 
in Section 5.5.

ACM Journal Name, Vol. V, No. N, Month 20YY.
aspect Handler {
  after(Class<Throwable> from, Class<Throwable> to) throwing(Throwable e):
    call(ejpscope(EH.convert)) && args(from, to)
    if (from.isAssignableFrom(e.getClass())){
      Util.unsafeThrow(to.getConstructor().newInstance());
    }
  after(Class<Throwable> exc, String msg) throwing(Throwable e):
    call(ejpscope(EH.logAndRethrow)) && args(exc, msg)
    if (exc.isAssignableFrom(e.getClass())){
      System.err.println(msg + e);
      Util.unsafeThrow(e);
    }
}

class Util {
  //AspectJ does not fully integrate generics in that advice cannot be parameterized
  //with generic type variables. In order for aspects implementing exception handling
  //EJPs to compile properly with checked exceptions, we exploited a Java loophole
  //to throw a checked exception without listing it in the advice's throws clause.
  public static void unsafeThrow(Throwable e) throws RuntimeException {
    Util.<RuntimeException>unsafeThrowWorker(e);
  }
  private static <T extends Throwable> void unsafeThrowWorker(Throwable e) throws T {
    throw (T)e;
  }
}

Fig. 10. Example of a generic EJP implementation and supporting utility code; the imple-
mentation of unsafeThrowWorker leverages type erasure in the Java compiler implemen-
tation of Java generics in order to throw a checked exception without affecting checked exception
method type constraints; the compiler is checking exception constraints against the generic type
(RuntimeException, which is an unchecked exception), but the statement throw (T)e is
compiled so that e is cast to the “erased” type Throwable; further details have been documented
by Anders Nordás at http://www.javaworld.com/community/node/1127

4.3 Addressing Prior Challenges with EJPs

The use of EJPs combined with generics can address some of the challenges with
aspectizing exception handling in AspectJ as found in a prior study by Castor Filho
et al. [Castor Filho et al. 2007]. An important finding of this study is that if an
“application is non-uniform, strongly context-dependent, or too complex, ad hoc as-
pectization might not be possible,” worsen code quality, or affect the system design.
The study classifies exception handlers according to five characteristics, and then
analyzes which combinations of characteristics are the most challenging to aspec-
tize. The classes of handlers that they found are the most difficult to aspectize are:
(a) block handlers, (b) context-dependent handlers, (c) nested context-dependent
handlers, (d) context-affecting handlers, and (e) loop iteration handlers. We discuss
in turn how EJPs can effectively aspectize each of these classes of handlers.

Block handlers are characterized by tangled try-catch blocks that mask excep-
tions, and then have statements following the masking handler. As noted in the
study by Castor Filho et al., in AspectJ normally these types of handlers must be
refactored into new methods in order to preserve program control flow in all cases.
EJPs can implement block handlers without refactoring into methods through the
use of scoped EJP references in the base code.

ACM Journal Name, Vol. V, No. N, Month 20YY.
Context-dependent handlers rely on the value of local variables. An AspectJ handler must access this state by either (a) refactoring the local variable into a field, (b) use a multi-stage advising pattern, (c) extract the code into a new method, passing the value of the local variable as a parameter, or (d) extending exception classes to store values of local variables. The paper details why each of these strategies reduce code quality or design quality. EJPs avoid these less effective strategies by allowing base code to explicitly pass local variable values as parameters to the EJP references.

The paper gives even stronger reasons why it is harmful in AspectJ to aspectize context-affecting handlers that assign new values to local variables. With EJPs, assignments to local variables are cleanly supported by passing parameters to EJP references (note that primitive types must be boxed, e.g., integer values must be passed as an Integer object and not int, because the value must be passed by reference).

Loop iteration handlers that include break or continue statements are very challenging to aspectize in AspectJ because it is not possible for advice to issue the break or continue, and as such involve a major redesign of the original loop code. EJPs can implement this type of handler by impacting the code much less, although the handler in this case is not cleanly encapsulated. An aspect implementation of a loop iteration EJP cannot directly include the break or continue statement, just as with AspectJ; however, the EJP implementation can cleanly communicate back to the base code that an exceptional condition was raised and that the base code should execute a break or continue. Thus, all handler implementation logic except for the actual break or continue statement can be refactored into the EJP implementation code inside an aspect, and the residue left in the base code is the scoped EJP reference, the parameter passed to that EJP reference, and the conditional test after the EJP reference on the parameter value to see if a break or continue statement should be executed. While not completely ideal, this is much cleaner than in the AspectJ case, where it is difficult to implement a tangled context-affecting handler.

For all of these classes of exception handlers, the key challenge for AspectJ is to robustly inject the handler into the precise place into the base code completely obliviously. The explicit presence of EJPs allows the base code to indicate precisely where the handler behaviors should be applied and what context information should be read or modified, and thereby addressing the key challenge.

4.4 Target Applications

In our study we refactored the exception handling concern in two object-oriented applications and one aspect-oriented application using EJPs. This refactoring was performed twice: once where EJP references are always explicit in the base code, and once where some EJP references are implemented using oblivious aspects. These same applications were refactored in the Castor Filho study. They were chosen by the authors of the Castor Filho study because they are representative of real-world applications that exhibit a variety of exception handling strategies, each with a different mixture of other crosscutting concerns.

The three applications we consider from the Castor Filho study are as follows:
(1) A subset of Telestrada (the same subset that was studied in the Castor Filho study). Telestrada is a traveler information system originally implemented in Java. The subset consists of 220+ classes and interfaces and 3400 lines of code.

(2) Java Pet Store, a demo application for the J2EE platform that showcases how to build robust enterprise applications (340+ classes and interfaces and 17800 lines of code).

(3) Health Watcher, a web-based information system. This application was originally implemented in AspectJ and has aspects for concurrency control, distribution, persistence, and some exception handling. It has 36 aspects, 96 classes and interfaces, and 6600 lines of code.

In this study we also consider the refactoring of four crosscutting concerns in the JHotDraw application, an application framework for drawing applications written by Erich Gamma that was patterned after HotDraw [Johnson 1992] (a similar framework written in Smalltalk). A detailed description of the architecture of JHotDraw is presented by Riehle [Riehle 2000]. In prior work four crosscutting concerns were refactored using AspectJ to create AJHotDraw [van Deursen et al. 2005; Marin et al. 2009]. The four concerns implemented by aspects in AJHotDraw are contract enforcement, undo/redo, persistence, and selection changed notification (an instance of the observer pattern). A significant portion of these four crosscutting concerns were implemented using AspectJ intertype declarations, supported by advice and pointcuts. We refactored AJHotDraw to use EJPs according to the two different refactoring strategies described above. In addition to analyzing the differences between the two EJP versions of AJHotDraw, this study is the first that employs the metrics listed in Section 4.5 to analyze the differences between AJHotDraw and JHotDraw. JHotDraw consists of 350 classes, 48 interfaces, and 22700 lines of code.

4.5 Metrics Suite

The metrics used in this study are a superset of those used in the Castor Filho study, relying on metrics proposed by Sant’Anna et al. [Sant’Anna et al. 2003] being supplemented by metrics proposed by Ceccato and Tonella [Ceccato and Tonella 2004]. Table I overviews our primary metrics, focusing on coupling, cohesion, size, complexity, and separation of concerns.2 The cohesion, coupling, and size metrics are variants of the well known Chidamber-Kemerer metrics [Chidamber and Kemerer 1994], extended to support AspectJ concepts as defined previously [Sant’Anna et al. 2003; Ceccato and Tonella 2004; Zhao 2004].

Of particular interest is the “Coupling Between Modules” (CBM) metric, defined as the number of modules referenced through field accesses or method calls. This metric is fully symmetrical in the sense that if an advice makes a field access or method call to an object whose class is defined in the base code, this is counted in the CBM metric just as it is when the base code makes a field access or method call to another module. Thus, the CBM is not expected to be non-zero for aspects and advice. While CBM measures lexical coupling between modules for field and method references within advice, this metric does not measure the dependence of

---

2Herein we use the term module to refer to interfaces, classes, and aspects.
Table I. Metric definitions [Sant’Anna et al. 2003; Ceccato and Tonella 2004; Zhao 2004]

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Metrics</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coupling</td>
<td>Coupling Between Modules</td>
<td>Within a given module, counts the number of modules referenced through field accesses or method calls. In the case of polymorphic method calls, only the reference to the static type of the object in the source is counted.</td>
</tr>
<tr>
<td></td>
<td>Coupling on Intercepted Modules</td>
<td>Within a given module, counts the number of modules explicitly named within pointcuts.</td>
</tr>
<tr>
<td></td>
<td>Depth of Inheritance Tree</td>
<td>Counts the depth of a class or aspect from its root in the inheritance tree.</td>
</tr>
<tr>
<td>Cohesion</td>
<td>Lack of Cohesion in Operations</td>
<td>Within a given module, counts the number of pairs of methods accessing different fields minus number of pairs of methods accessing common fields.</td>
</tr>
<tr>
<td>Size</td>
<td>Lines of Code</td>
<td>Within a given module, counts the number of uniformly formatted lines of code, excluding whitespace and comments.</td>
</tr>
<tr>
<td></td>
<td>Concern Lines of Code</td>
<td>Within a given module, counts the subset of Lines of Code used to implement a specific concern.</td>
</tr>
<tr>
<td></td>
<td>Number of Operations</td>
<td>Within a given module, counts the number of declared methods and advice.</td>
</tr>
<tr>
<td></td>
<td>Number of Fields</td>
<td>Within a given module, counts the number of declared fields.</td>
</tr>
<tr>
<td>Separation of Concerns</td>
<td>Concern Diffusion over Modules</td>
<td>Counts the number of modules that implement a concern or reference one that does.</td>
</tr>
<tr>
<td></td>
<td>Concern Diffusion over Operations</td>
<td>Counts the number of operations that implement a concern or reference one that does.</td>
</tr>
<tr>
<td></td>
<td>Concern Diffusion over LOC</td>
<td>Counts the number of transitions between one concern to another across all lines of code.</td>
</tr>
</tbody>
</table>

aspects upon specific modules due to specific references to other modules within pointcuts. To measure this dependence, we rely on the “Coupling on Intercepted Modules” (CIM) metric, as previously proposed by Ceccato [Ceccato and Tonella 2004].

We also use aspect-specific coupling metrics as proposed by Ceccato and Tonella [Ceccato and Tonella 2004] to better understand pointcut induced coupling. The separation of concern metrics [Sant’Anna et al. 2003] model the scattering of a concern (e.g., exception handling) across modules and operations, and also model the interleaving of a concern across lines of code (tangling). Additionally, we introduce the “Concern Lines of Code” (CLoC) metric to better understand code size for just the crosscutting concerns of interest.

The metrics were calculated for all four versions (Java, AspectJ, explicit EJP, CoAOP EJP) of each of the four target applications, primarily using the aopmetrics tool [Stochmialek 2005]. This tool has slightly different heuristics for the lines of code metric than in the Castor Filho study (e.g., differing in how lines are counted for single-line if statements, for curly braces starting new basic blocks, etc.), but the differences are minor and the data still exhibits the same trends and degrees of change.

ACM Journal Name, Vol. V, No. N, Month 20YY.
5. QUANTITATIVE ANALYSIS

This section presents the results of the software quality metrics, organized by attribute, along with our corresponding analysis. Herein we quantitatively evaluate whether code modularity, reuse, and complexity are improved when EJPs are employed according to the guidelines set forth in the hypotheses set forth in Section 3.3. The presentation of the metric results are organized by metric category, and with each category we introduce hypotheses that are evaluated against the data. Each metric category provides additional insight into the broader hypotheses of Section 3.3. Section 5.2 presents coupling and cohesion metric results, analyzing code modularity. Section 5.3 presents conciseness and complexity metric results, analyzing code complexity. Section 5.4 presents separation of concerns metric results, analyzing code modularity and complexity. Section 5.5 presents reusability metric results, analyzing code reusability.

5.1 Overview

The data are presented using stacked bar graphs, allowing for inspecting of the contribution of the base code, the aspects related to the refactored concern(s), and other unrelated aspects (only in Health Watcher) to each metric. Note that because Health Watcher was originally written in AspectJ, with other concerns (such as distribution and persistence) being written with aspects, in the figures the original version of Health Watcher is labeled “Original” (“Org.”) instead of “Java.” All data is also available for download and review. In addition to analyzing the total metric values, additional insight can be gained by inspecting the sub-totals for just the aspects for the refactored concern(s) (e.g., exception handling). In all cases lower values are better.

Our results and analysis for the metrics are presented in the following order: coupling and cohesion, size, and separation of concerns. From henceforth, metric results for the applications will be ordered as follows: Telestrada, Pet Store, Health Watcher, and HotDraw. Finally, as Co-AOP is largely motivated by the current difficulties in developing reusable aspect libraries, we evaluate reusability.

In short, the results indicate that separation of concerns can decrease with EJPs due to the explicit references to join points introduced, but that code reuse, which Co-AOP primarily sets out to achieve, indeed increases. As expected, the explicitness of EJP references increase coupling between modules, although this increase is mitigated by (and in one application completely counteracted by) decreased coupling in exception handling logic, due to parameterized exception handling. Significant improvements are observed for coupling from intercepted modules with EJPs and Co-AOP compared to fully oblivious aspects. Code complexity and conciseness remain largely stable. The overall number of operations and amount of code required to implement the exception handling concern typically decrease in the EJP versions due to increased reusability.

5.2 Coupling and Cohesion

Using coupling and cohesion metrics we evaluate the following hypotheses:

http://www.kevinjhoffman.com/tosem2012/
Hypothesis 5.2.1  Coupling of base code to aspects and EJP interfaces should significantly increase in a fully explicit methodology, as compared to a fully oblivious methodology.

Hypothesis 5.2.2  Coupling in the opposite direction, coupling of aspects to base code, should decrease in a fully explicit methodology.

Hypothesis 5.2.3  Use of EJPs in a Co-AOP methodology instead of a fully explicit methodology should decrease overall coupling, as compared to a fully explicit methodology.

Hypothesis 5.2.4  Cohesion should increase in the presence of EJPs.

Figure 11 shows the results for the coupling and cohesion metrics.

Summary of Data. Careful inspection of the results show that in the EJP version there is generally more coupling of base code to aspects, confirming Hypothesis 5.2.1, with notable exceptions caused by parameterization being discussed below. Additionally, there are significant improvements in cohesion in the EJP version vs the AspectJ version for the exception handling concern, confirming Hypothesis 5.2.4. While Hypothesis 5.2.2 generally held true, one notable aberration is that when obliviousness was removed in the EJP case, sometimes the system-wide coupling (coupling of base code to aspects and vice versa) actually decreased, which is surprising because one of the main motivations of obliviousness was to reduce overall coupling. Insights into why this can happen are detailed below. The data also supports Hypothesis 5.2.3 except for the case of Telestrada, where overall coupling increased slightly in the Co-AOP version vs the EJP version.

Discussion of Coupling Between Modules. The Coupling Between Modules (CBM) metric counts for a given module the number of other modules it depends upon explicitly through field accesses, method calls, or EJP references. Surprisingly, removing obliviousness via EJPs affected overall CBM both positively and negatively. Compared to the AspectJ versions the EJP versions differed by -5%, +8%, +14%, and +1%. Compared to the original versions, EJP versions differed by -9%, +9%, +13%, and +5%.

In general the data supported Hypothesis 5.2.1, although to a lesser degree than expected. It was anticipated that removing obliviousness would result in greatly increased CBM for the EJP versions compared to the AspectJ versions due to the explicit presence of EJPs in base code. However, two factors significantly reduced coupling and so counterbalanced the increase in CBM due to EJP references: First, parameterized EJPs facilitated generic exception handling logic, which is free from application-specific couplings, and also avoided couplings caused by application-specific advice. For example, the CBM metric value for exception handling aspects for Pet Store for the EJP version was reduced by 80% compared to the AspectJ version. Second, parameterized EJPs allow base code to customize the generic exception handling EJPs without coupling links due to removed handler logic. This decreases the CBM metric value in the base code (e.g., the reduction for Telestrada was 9%). These factors provide supporting evidence for Hypothesis 3.3.2 and Hypothesis 3.3.4.

In all cases except for Telestrada, overall CBM decreased slightly in the Co-AOP versions, due to fewer modules having references to EJP declarations, supporting
Fig. 11. Results for coupling and cohesion metrics (lower is better)
Hypothesis 5.2.3. In Telestrada the additional aspects required to obliviously inject the EJP references caused CBM to increase by 1% vs the explicit EJP version, while still being lower than the AspectJ version by 4%, supporting Hypothesis 3.3.1.

The results for CBM indicate the need to carefully design the EJP declarations to prevent unnecessary couplings.

Discussion of Coupling on Intercepted Modules. The most significant indicator of the decrease in coupling of aspects to base code is the impact of EJPs on the Coupling on Intercepted Modules (CIM) metric. This metric counts for a given module the number of other modules explicitly named in pointcuts. Compared to the AspectJ versions, the EJP versions have a reduction of 100%, 100%, 57%, and 65% in CIM. For Co-AOP the reductions in CIM vs the AspectJ versions are 88%, 93%, 53%, and 52%, strongly supporting Hypothesis 3.3.1.

These results confirm Hypothesis 5.2.2, and indicate that the pointcuts in the EJP version were more robust with respect to evolution of the base code than in the AspectJ version. Looking deeper, exception handling aspects in Health Watcher AspectJ caused a 120% increase in CIM but account for only 12% of the aspects in the system. This disproportionate increase indicates that the exception handling concern inherently led to more fragile points and exhibited the state-point separation problem more often than any other concern in the application. See Section 3.4 for one example from Health Watcher AspectJ that is prototypical of the cause of this increase in CIM. The EJP version avoided this unnecessary complexity due to EJP parameterization and this reduction of complexity was one of the major benefits gained by trading obliviousness. The Health Watcher and HotDraw EJP versions exhibited non-zero CIM values due to either non-exception handling related aspects (in Health Watcher) or due to intertype declarations (in HotDraw).

The impact of obliviously injecting EJP references is small yet visible in all CIM values. The number of explicitly referenced modules in pointcuts increased between 5 and 10 for the Co-AOP versions vs the EJP versions (representing an increase between 4% and 20% vs the CIM values for their corresponding AspectJ versions), yet still in each case having much lower CIM than in the AspectJ versions. This increase in CIM for the Co-AOP versions was a tradeoff in return for a reduced number of explicit EJP references. In the EJP versions, there were 67, 323, 240, and 56 EJP references. The Co-AOP versions had 60 (-10%), 262 (-19%), 100 (-58%), and 24 (-57%) EJP references. This beneficial tradeoff is a result of obliviously injecting EJPs only where appropriate, as described in Section 4.1.

The smaller number of explicit EJP references in the Co-AOP versions reduced CBM in the base code in all cases, and also more than compensates for increased CIM values in the Co-AOP version in all cases, thus resulting in a decreased level of overall coupling and supporting Hypothesis 5.2.3. The one exception to this decrease in overall coupling is Telestrada, where increased CBM in the aspects in the Co-AOP versions caused the overall coupling for the Co-AOP version to be slightly higher than for the EJP version. If Co-AOP is improperly applied, a negative tradeoff between CIM and the number of EJP references would be expected.

This CIM data also supports Hypothesis 3.3.1 in that code modularity and code complexity are improved only when using explicit EJP references where pointcuts
Trading Obliviousness for Modularity with Cooperative Aspect-oriented Programming.

<table>
<thead>
<tr>
<th></th>
<th>Java / Original</th>
<th>AspectJ</th>
<th>EJP</th>
<th>Co-AOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telestrada</td>
<td>186</td>
<td>188</td>
<td>186</td>
<td>186</td>
</tr>
<tr>
<td>Pet Store</td>
<td>245</td>
<td>258</td>
<td>245</td>
<td>245</td>
</tr>
<tr>
<td>Health Watcher</td>
<td>178</td>
<td>178</td>
<td>178</td>
<td></td>
</tr>
<tr>
<td>HotDraw</td>
<td>523</td>
<td>523</td>
<td>523</td>
<td>523</td>
</tr>
</tbody>
</table>

Table II. Depth of Inheritance Tree (DIT) metric values

are specific and complex—where pointcuts were broad or simple, the Co-AOP version used an oblivious aspect to inject the EJP references and resulted in improved modularity and complexity.

Discussion of Lack of Cohesion of Operations. In all cases the Lack of Cohesion of Operations (LCO) for the EJP version compared to the original version either improved or remained the same, in support of Hypothesis 5.2.4. The improvement in cohesion by 6% in the EJP version vs the original version for Telestrada results from the removal of fields in base code no longer needed to implement exception handling.

Compared to the AspectJ version, the EJP version improved cohesion in all cases for the exception handling concern, sometimes significantly (from 8% to 22%), supporting Hypothesis 5.2.4. The decreases in cohesion in the AspectJ version are caused by the need to artificially extract new methods to expose advisable join points (e.g., try-catch blocks in loops). As discussed by Castor Filho et al. [Castor Filho et al. 2006], these new methods are a negative byproduct of the AspectJ refactoring and represent an empirical indication of the benefits that can come when base code explicitly cooperates with “fine-grained” aspects.

In HotDraw, cohesion was improved for all aspectized versions vs the Java version, because of the use of intertype declarations to separate unrelated code. In HotDraw cohesion did not improve in the EJP versions vs the AspectJ version, weakening support for Hypothesis 5.2.4, because the join points needed to be captured by the four concerns could be cleanly captured in AspectJ without having to split methods, indicating that these four concerns tend to be more “coarse-grained” than exception handling.

Discussion of Depth of Inheritance Tree. Results for the Depth of Inheritance Tree (DIT) metric are presented in Table II. EJP and Co-AOP did not change the metric results vs the original versions. In a few cases AspectJ increased this metric value due to subclassed aspects of abstract aspects. These were not needed in the EJP or Co-AOP case because the EJP interface serves as a mechanism for reusing advice in aspects rather than the use of aspect inheritance in combination with abstract pointcuts.

5.3 Conciseness and Complexity

Using conciseness and complexity metrics we evaluate the following hypotheses:

Hypothesis 5.3.1 When parameterization of crosscutting concerns is possible, the use of EJPs should reduce code size.
Hypothesis 5.3.2 The Co-AOP EJP versions should reduce code size vs the fully explicit EJP versions.

Hypothesis 5.3.3 The number of operations should be the smallest in the original versions and the greatest in the AspectJ versions.

Figure 12 shows the results for the size metrics.

Summary of Data. For the exception handling concern, Lines of Code decreased for all applications in all EJP versions, except compared to Health Watcher AspectJ. Concern Lines of Code consistently decreased for this concern for the EJP versions, usually significantly, due to the ability to parameterize the exception handling types over a small set of common exception handling patterns. For HotDraw, Lines of Code and Concern Lines of Code increased slightly vs the AspectJ version, which in turn had increased slightly vs the Java version. This data supports Hypothesis 5.3.1, showing decreased code size when parameterization is possible.

The use of Co-AOP was not able to significantly reduce code size vs the fully explicit EJP versions, having approximately no change (within 50 lines) for Telestrada and HotDraw. Co-AOP marginally reduced code size for Pet Store (around 100 lines). Co-AOP was able to significantly reduce the concern code size for Health Weather by nearly a third. Thus, Hypothesis 5.3.2 only holds in the case of Health Watcher. We found that in the other cases for Co-AOP the reduction in code size from fewer EJP references was counterbalanced by the code for the additional pointcuts to obliviously inject the EJP references back onto the base code.

As anticipated in Hypothesis 5.3.3 the Number of Operations was always the smallest in the original versions. Also, the Number of Operations was always greatest in the AspectJ versions, except in the case of HotDraw where the EJP versions nearly identical (0.3% higher).

Discussion of Lines of Code and Concern Lines of Code. Compared to the original version, Lines of Code for the EJP versions differed by -6%, -4%, -4%, and +1%. Compared to the AspectJ version they differed by -4%, -4%, +3%, and +0.5%. As the logic related to our refactored crosscutting concerns is only a small part of these applications, we introduced a new metric–Concern Lines of Code—to better understand how the size of just these concerns was affected. The Concern Lines of Code (CLoC) metric counts lines of code that implement the concern(s) that were refactored to use EJPs, including EJP references in base code. The EJP version performed significantly better than both the original and AspectJ versions for Telestrada and Pet Store (by 32% and 46%), while only a 6% reduction was observed for Health Watcher. CLoC actually increased for the AspectJ and EJP versions of HotDraw vs the Java version, having an increase of 8% and 12% respectively.

For the exception handling concern, two EJP induced factors affect these metrics: First, greater code reuse facilitated by generic exception handling EJPs reduced the number of lines of code, especially for the exception handling aspects (the data showed EJPs performed better than AspectJ for this subtotal by 62.1%, 77%, and 61.9%). Increased levels of code reuse were facilitated in the EJP versions by the ability to parameterize (e.g., context-specific error messages), supporting Hypothesis 5.3.1. Second, in the Health Watcher application most exception handler
Trading Obliviousness for Modularity with Cooperative Aspect-oriented Programming.

Fig. 12. Results for size metrics (lower is better)
locations could be matched by broad, stable pointcut definitions. Additionally, in Health Watcher exception handlers did not need additional per-join point context information, such as English descriptions of errors. This allowed the oblivious version to use a few pointcuts to apply exception handling logic to the appropriate places. True to our technique, any oblivious exception handling aspects were converted to use EJPs, increasing Lines of Code due to EJP references. Notwithstanding, in the EJP version the reduction in lines of code due to the above factors was more significant, resulting in an overall decrease in Lines of Code.

This indicates that the obliviousness technique should be applied where stable pointcuts are possible and where parameterization is not required, supporting Hypothesis 3.3.1. Obliviousness should be maximized when possible using known strategies as mentioned in Section 2.2, and then obliviousness can be reduced or removed to avoid fragility or when parameterization is needed.

For the HotDraw application, the four concerns inherently exhibited very little potential for re-use. Most of the refactoring (for both the AspectJ and EJP versions) was moving sections of code into aspects, and then having added to that the glue code needed to weave the application back together. There was nearly a one-to-one correspondence between advice and the join point where that advice should be applied. The main exception to this was the contract enforcement concern, which could be represented with an advice that attached to many join points in the base code. For these reasons both the AspectJ and EJP versions exhibited larger CLoC vs the Java version. Additionally, because there was very little re-use of crosscutting implementations, there was no effect to counterbalance the increase in code size due to the increased EJP references, and so the EJP versions had larger CLoC than the AspectJ version.

For Pet Store and Health Watcher, Co-AOP reduced both Lines of Code and CLoC, because the removal of explicit EJP references was greater than the new pointcuts and advice needed to obliviously inject the EJP references. This was not true for Telestrada and HotDraw, which remained nearly the same, because the number of EJP references injected obliviously per advice were much lower on average. The most significant impact in CLoC by Co-AOP was observed for the Health Watcher application, where Co-AOP decreased CLoC by 31% vs the fully explicit EJP version. Thus the data generally supports Hypothesis 5.3.2, although in many cases to a lesser degree than expected.

Discussion of Number of Operations. The total Number of Operations for the EJP versions increased by 4%, 2%, 6%, and 1% compared to the original versions. The EJP versions for the exception handling concern were consistently better than the AspectJ versions, with differences of -6%, -8%, and -5%. For HotDraw the Number of Operations increased slightly vs the AspectJ version by 0.3%. Thus, with the exception of HotDraw where the Number of Operations remained nearly the same across all versions, the data strongly supports Hypothesis 5.3.3.

For the exception handling concern, this increase for the AspectJ and EJP versions vs the original versions is an expected byproduct of the refactoring, due to inlined code in catch blocks being extracted to advice. The improvement of the EJP versions vs the AspectJ versions was caused by (a) the higher level of reuse of exception handling logic (there were 36.4%, 71%, and 36.1% fewer handler op-
Trading Obliviousness for Modularity with Cooperative Aspect-oriented Programm.

<table>
<thead>
<tr>
<th></th>
<th>Java / Original</th>
<th>AspectJ</th>
<th>EJP</th>
<th>Co-AOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telestrada</td>
<td>230</td>
<td>221</td>
<td>221</td>
<td>221</td>
</tr>
<tr>
<td>Pet Store</td>
<td>1118</td>
<td>1124</td>
<td>1119</td>
<td>1120</td>
</tr>
<tr>
<td>Health Watcher</td>
<td>234</td>
<td>237</td>
<td>234</td>
<td>234</td>
</tr>
<tr>
<td>HotDraw</td>
<td>725</td>
<td>722</td>
<td>719</td>
<td>722</td>
</tr>
</tbody>
</table>

Table III. Number of Attributes metric values

erations for the EJP version), and (b) scoped EJPs eliminating the need to create methods to expose new join points (causing improvements of 2% to 3%), reducing code complexity and supporting Hypothesis 3.3.2. This again underscores the benefits of reducing or removing obliviousness where state-point separation issues can hinder reuse in an oblivious approach, and supports Hypothesis 3.3.4.

For HotDraw, due to the almost non-existant reuse of code within the implementation of the crosscutting concerns, extra operations were needed to implement the glue code, in the form of advice and EJP declarations.

Discussion of Number of Attributes. Results for the Number of Attributes metric are presented in Table III. For the most part neither AspectJ nor EJPs nor Co-AOP had much affect on this metric. The aspectization of exception handling in Telestrada removed nine exception handling fields from the base code, resulting in a 3.9% decrease for the AspectJ, EJP, and Co-AOP versions. The AspectJ version of Health Watcher added three new fields (+1.3%) in the implementation of a complex multi-stage advice. In HotDraw aspects were able to eliminate three fields (decrease of 0.4%), and the EJP version eliminated an additional three more fields due to parameterization (total decrease of 0.8%).

5.4 Separation of Concerns

Using separation of concerns metrics we evaluate the following hypotheses:

Hypothesis 5.4.1 The logic for crosscutting concerns should be contained in the fewest modules and fewest operations in the oblivious AspectJ versions.

Hypothesis 5.4.2 Co-AOP should reduce the number of modules and operations that have knowledge of crosscutting concerns.

Hypothesis 5.4.3 The AspectJ versions should exhibit the least amount of lexical concern tangling, with both the fully explicit and Co-AOP EJP versions exhibiting less concern tangling than the original versions.

Hypothesis 5.4.4 Co-AOP should reduce lexical concern tangling vs fully explicit EJP.

Figure 13 gives the results for the separation of concerns metrics, measuring both the scattering of a concern over modules and operations, and the tangling of a concern with other concerns or base code.

Summary of Data. Hypothesis 5.4.1 is mostly supported by the data, with the AspectJ versions always having the fewest modules and operations relating to the crosscutting concerns, with the only exception being the number of concern-related
Fig. 13. Results for separation of concerns metrics (lower is better)
operations for Telestrada. In this case, the aspectization of exception handling required more operations (in this case all advice) than the number of methods referencing exception handling in the Java version. The fully explicit and Co-AOP EJP versions exhibited the largest number of modules and operations having knowledge of the crosscutting concerns, as expected; however, as measured earlier by the CLOC metric, the actual amount of code with knowledge of the crosscutting concerns in the base code is significantly smaller, ranging from 4 times to more than 10 times less concern-related code in the base code.

Co-AOP reduced the number of modules and operations that have knowledge of crosscutting concerns, usually significantly, in all cases but one, supporting Hypothesis 5.4.2. The exception was Telestrada, where Co-AOP caused several operations to be fully oblivious to exception handling, but never fully eliminated exception handling from any modules in the base code and had additional modules with knowledge of exception handling for the aspects to obliviously inject the EJP references.

Lexical concern tangling, as measured by Concern Diffusion over LOC, was the lowest in all cases for the AspectJ versions, supporting Hypothesis 5.4.3. Additionally, all EJP versions significantly reduced concern tangling, typically by a factor of 2 or more, further supporting Hypothesis 5.4.3. In all cases Co-AOP further reduced lexical concern tangling by a minimum of 10%, and in some cases reduced tangling by a factor of 2, supporting Hypothesis 5.4.4.

Discussion of Concern Diffusion over Modules. Concern Diffusion over Modules (CDoM) measures the number of modules that contain exception handling logic or a reference to a method or EJP that implements such logic. This is a coarse metric and as such is a more direct measure of obliviousness, it measures the scattering of a concern over application modules. As the technique for one of the EJP strategies was to use a fully explicit approach, these numbers represent the worst case scenarios for the fully explicit EJP versions. This metric differed by +23%, +9.1%, +8.5%, and -13% between the EJP and original versions. The AspectJ versions showed an improvement vs the EJP versions by 33%, 53%, 80%, and 40%, supporting Hypothesis 5.4.1. For the exception handling concern, it increased in all cases for the EJP versions because each module in the base code still referenced the exception handling concern (in the form of an explicit EJP reference) while also adding new modules for the aspects implementing the exception handling EJPs. For HotDraw, CDoM actually decreased vs the Java version because the use of intertype declarations completely removed certain crosscutting concern code from the base code, compensating for some of the explicit EJP references in the base code, and resulting in an overall decrease of CDoM.

Co-AOP improved CDoM vs the EJP versions in every case except for Telestrada, resulting in changes of +4%, -11%, -14%, and -7%, supporting Hypothesis 5.4.2. Co-AOP was usually able to completely remove the presence of a crosscutting concern from one or more modules, resulting in a decrease vs the EJP versions. In Telestrada the use of Co-AOP did reduce the number of EJP references (and the number of operations referencing exception-handling, as seen below), but did not completely remove exception handling from any one module; this combined with
the new module (aspect) that was required to obliviously inject some of the EJPs caused the increase in CDoM in this case.

Discussion of Concern Diffusion over Operations. The Concern Diffusion over Operations (CDoO) metric is more fine-grained and measures obliviousness and concern scattering on a per-method level. These numbers thus also represent the worst possible values for the fully explicit EJP version, since all obliviousness has been removed through the use of EJP references. CDoO differed by +33%, +12%, +13%, and -4% between the EJP and original versions and by +27%, +40%, +120%, and +119% between the EJP and AspectJ versions, supporting Hypothesis 5.4.1.

Interestingly, for the exception handling concern, considering only handler aspects for both the CDoM metric and the CDoO metric, the differences between the EJP and the AspectJ versions actually decreased significantly, with the differences ranging from -20% to -72% for CDoM and -63% to -84% for CDoO. This was caused by the parameterization of advice via EJPs, resulting in the removal of application-specific code from the exception handler implementations and higher levels of handler reuse. Attempts to separate this application-specific code into other aspects in AspectJ would be hampered by the advice-reuse problem and the state-point separation problem, as described in Section 2.1. Obliviousness can thus be traded for increased separation of concerns within the aspect implementation for cases where these problems apply. This technique can be seen as a tradeoff between modularity and obliviousness, where the aspect can be semantically isolated from the base code concern at the cost of the crosscutting concern not being completely removed from the base code.

For HotDraw, the small decrease in CDoO vs the Java version is caused by the slightly improved code re-use in the implementation of the contract enforcement concern, resulting in fewer overall methods referencing or implementing the concerns. This reduction was not enough to offset increase vs the AspectJ version due to the explicit EJP references remaining in the base code, causing the EJP version to be about 19% higher than the AspectJ version.

The application of Co-AOP consistently decreased CDoO vs the fully explicit EJP version in every case, although to varying degrees of effectiveness. The differences in CDoO for the Co-AOP versions vs the fully explicit EJP versions are -9%, -20%, -33%, and -2%, supporting Hypothesis 5.4.2. The decrease was the smallest for HotDraw because a significant portion of the crosscutting concerns were implemented with intertype declarations, such that the proportional effect of the reduction of explicit EJP references was less than it would have been without the intertype declarations.

Discussion of Concern Diffusion over Lines of Code. The Concern Diffusion over Lines of Code (CDoLOC) metric counts the number of concern switches (between any refactored concern and the base code or another concern) and serves as a measure of tangling. Note that while this measure counts how many concern switches there are, it does not reflect the size of the code implementing the concern in between the concern switches. Thus a large amount of implementation code for a crosscutting concern could be abstracted away or removed from the base code and this metric value might not significantly change. For a measure of the actual code
size of the crosscutting concerns within the base code see the CLOC metric results presented earlier. The EJP versions and original versions differed by -51%, -47%, -39%, and -57%. The AspectJ versions showed an improvement over the EJP versions by 100%, 86%, 100%, and 92%. This data strongly supports Hypothesis 5.4.3. This is not surprising because the goal of the AspectJ version was to be completely oblivious, which would be indicated by a CDoLOC value of 0, whereas in the EJP versions some measure of explicitness is part of the technique. Note that the AspectJ version of Pet Store and HotDraw did have a small amount of concern intermingling (non-zero CDoLOC), due to the introduction of a new concern in the base code to catch and handle exceptions softened within aspects.

In most modules for the exception handling concern, refactoring the original version with EJPs caused this metric to be decreased by exactly one half, as each try-catch block (usually contributing four switches) was converted into one EJP reference (contributing two switches). In some modules the decrease was less than 50% because of the contiguous alignment of catch blocks belonging to try blocks that started at different places. In the case of Telestrada, the removal of exception handling code existing outside catch blocks provided a decrease greater than 50%.

For HotDraw, the abstraction of some of the implementation level details of the concerns allowed for references to the concerns to be less common and more focused. This along with the reduction due to the intertype declarations provided a significant decrease of CDoLOC vs the Java version.

While these metrics indicate that the base code was lexically aware of the exception handling concern, albeit to a lesser degree, these metrics do not effectively measure the degree to which the concerns are separated “semantically” (in other words, the degree to which their implementation details are intertwined within the same module). The Concern Lines of Code (CLoC) metric can provide additional insight here. Looking at the CLoC values for just the base code portion of the application, we can discover the “size” of the presence of the crosscutting concern in the base code. Taken together with the CDoLOC these two metrics give a better overall picture of how much the implementation of the crosscutting concerns has been separated, although a better metric is still needed. The CLoC values for the explicit EJP versions for just the base code part of the application vs the original versions are -76%, -67%, -60%, and -57%. Taken together with the consistently lower values of CDoLOC for the EJP versions vs the original versions, it is clear that the amount of implementation related code for the crosscutting concerns has been significantly reduced in the EJP versions, and thus provides evidence for Hypothesis 3.3.4.

A clear pattern of the effects of Co-AOP on CDoLOC can be seen. In all cases Co-AOP reduced CDoLOC, due to the significantly reduced number of EJP references in the base code (between 10% and 58% fewer EJP references), supporting Hypothesis 5.4.4. Refer to the section discussing the CIM metric results for precise numbers on exactly how many references were removed by Co-AOP.

Moreover, the separation of concern metric results taken together highlight the need to carefully design EJP interfaces so that (a) the implementation details of a crosscutting concern induced by EJP references are minimized, and (b) EJPs do not unnecessarily require parameterization so that they can be applied obliviously in as
many places as possible. More generally, always following an oblivious methodology may cause a heavy presence of other concerns within aspect implementations, which can be mitigated through an intermediary interface.

5.5 Reusability

Reusability is an important goal in well engineered systems and was one of the original motivations of using aspects to implement exception handling [Lippert and Lopes 2000]. However, in systems with complex, application-specific handler logic, the realized level of reuse is lower than anticipated, being hindered by application-specific context (e.g., error messages), exception types, and control flow differences [Castor Filho et al. 2006]. In contrast, the explicit presence of EJPs in the base code allows for parameterization of the handler logic implemented by aspects, allowing context (e.g., log messages) and exception types to be explicitly communicated.

Figure 14 presents the percentage of exception handlers that were implemented by either abstract aspects with several concrete subaspects (in the AspectJ case) or by generic EJP interfaces. The data confirms that significant increases in reusability can be gained by applying an explicit approach where appropriate.

A balance must be maintained between the separation of concerns and the increased reusability that comes from explicit parameterization. Reusability is also enhanced when there is an interface acting as mediator between both user and implementor. Where crosscutting concerns are more coarse-grained, oblivious aspects can be used to link the base code to the crosscutting concern interfaces.

Additionally, the generic exception handling EJPs and the advice implementing them were free from any references to application-specific logic (implied by their CIM of 0) and could be used in other applications without modification. In contrast, in the AspectJ version only abstract aspects were decoupled from the target applications, and were responsible for handler logic only 16%, 0%, and 10% of the time.

Co-AOP resulted in an increased proportion of references to reusable EJPs for the Telestrada (75% vs 72%) and Health Watcher (63% vs 53%) applications, and
a slightly lower proportion in the Pet Store application (88% vs 90%). The reason for the increase in Telestrada and Health Watcher is that the aspects obliviously injecting EJP references were able to remove many more of the application-specific explicit EJP references vs the number of generic EJP references that were removed. In these applications the application-specific exception handlers followed a structured pattern and were less likely to require parameterization. Conversely in Pet Store the opposite was true, and there were slightly more generic EJP references that were candidates for oblivious EJP reference injection.

Reusability is hardly applicable to the HotDraw application as expressed in the AJHotDraw refactoring—all of the concerns that were aspectized in AJHotDraw were tightly connected to very specific logic and were not designed to be reusable (apart from these application-specific types) in other applications. Certainly some of the code for these concerns could be further abstracted and made into reusable components; however, in order to avoid bias we chose to keep AJHotDraw unmodified for the purposes of refactoring to use EJPs and for comparison purposes.

6. DESIGN STRUCTURE MATRIX ANALYSIS

While the software quality metrics utilized in Section 5 are effective in capturing different aspects of overall code quality attributes, they are limited in that the few numbers used cannot of course capture all properties for all components within a software system. They are thereby less precise for evaluating whether or not certain code follows well-known design principles, such as the dependency inversion principle [Martin 1996], and the acyclic dependency and open–closed principles [Meyer 1997; Martin 2000], and if not, then where and how a given program fails to do so.

6.1 Design Structure Analysis Background and Study Setting

To assess adherence to these principles and to further analyze differences between the versions of the software (Java, AspectJ, EJPs (fully explicit), EJPs (Co-AOP methodology)), we utilize design structure matrices (DSMs) [Steward 1981; Baldwin and Clark 1999; Lopes and Bajracharya 2005]. Below, we present DSMs for all versions of the Pet Store application, starting from reverse-engineered DSMs generated by Lattix [Sangal et al. 2005]. Pet Store has been chosen for this discourse as (1) it was specifically created to be an example of good design, (2) was one of the larger and more complex applications compared to the others in this study, and (3) accentuated the general trends. The design of Pet Store can be summarized as follows: Pet Store implements business logic in several reusable components, follows the model-view-controller pattern for web applications, and divides different sections of the store front into four application packages. Pet Store also relies on a set of utility packages and the packages for exception handling.

To gain additional insight into the layering of crosscutting concerns and ripple effects, we used a version of Pet Store that was enhanced with an additional exception monitoring concern, as described in more detail by Hoffman and Eugster [Hoffman and Eugster 2009]. To summarize, exception monitoring is useful in order to implement a fault analysis engine [Fuad et al. 2006], which would be desirable for an e-commerce application such as Java Pet Store. For the purposes of this paper, a fault analysis engine can be viewed as an interface that accepts information...
about thrown and handled exceptions; additionally, contextual information about exceptions is also required in order to classify faults and build fault models.

Thus more precisely stated, the exception monitoring concern is for the application to provide information on all handled exceptions and their context to this fault analysis engine interface. Additionally, the concern defines the contextual information to include any log messages generated by the handler, as well as a flag indicating if the handler rethrows the exception, which is useful to understand when faults reach their final handlers in the propagation chain. Fault analysis engines can use this information to provide insights into how the application reacted to a fault, when it was finally handled, and the probability of whether administrative action is required. We note that this additional concern was not conceptualized until after the exception handling concern in the original application had been aspectized. In this way the initial refactoring was not biased towards either AspectJ, EJPs, or Co-AOP when we implemented the additional concern.

An important reason for selecting fault-analysis as the additional crosscutting concern is that it cross-cuts the base application in similar ways as exception handling, but the logic to be woven at these points is very different (in terms of both behavior as well as information required). This allows the paper to explore how well the software is open for extension but closed for modification for a related yet distinct crosscutting concern.

6.2 DSM Overview and Construction Techniques

DSMs visually represent the pair-wise dependencies between design structures, interfaces, and components within an application in matrix form. The rows and columns are labeled with these various design variables. Cells are marked to indicate that the design variable for that row depends on the design variable for that column. One design variable (e.g., A) is said to depend upon another (e.g., B) if changes in design variable B could possibly require changes to variable A as well. The cells across the diagonal are nominally marked, as a design variable is always said to depend on itself.

When reverse engineering DSMs as in this case, design variables are chosen to represent Java packages. Dependencies between design variables are introduced due to method calls, field accesses, inheritance relationships, object instantiation, references to explicit join points, type dependencies, and explicit references to specific method names or EJP names in pointcuts. DSMs can also model more abstract design variables, such as choices in application architecture or algorithms [Baldwin and Clark 1999] or environmental constraints through augmented constraint networks [?, ?]. Modeling these higher-level design variables can provide insight into the architectural-level design quality of an application. In this study we included only concrete design variables in the DSMs, as deeply exploring the algorithmic, architectural, and environmental factors of Pet Store are beyond the scope of this paper.

Clustering algorithms [Browning 2001] can be applied to DSMs such that design variables that tightly depend on each other are grouped closer together. Module boundaries containing tightly interrelated and independently reusable collections of design variables can then be drawn across the diagonal. Clustering also orders design variables and modules such that design variables and modules are placed...
below any variables or modules that they are dependent upon. When ordered in this way, any mark above the diagonal represents a cyclic dependency, and indicates that the two design variables should likely be placed within the same module, or at least have the dependency analyzed to ensure the dependency does not easily propagate ripple effects. The DSMs in this paper are clustered hierarchically by Lattix where outermost modules (as depicted by the drawn boxes) are clustered, and then the contents of each module is then recursively clustered further.

Figures 15–17 present the DSMs for the AspectJ, fully explicit EJP, and Co-AOP EJP versions of the Pet Store application.

<table>
<thead>
<tr>
<th>DSM</th>
<th>com_supplier</th>
<th>com_petstore</th>
<th>com_admin</th>
<th>com_opc</th>
<th>com_waf_view</th>
<th>com_waf_controller</th>
<th>com_components</th>
<th>com_util</th>
<th>com_ga</th>
<th>com钋sub</th>
<th>com钋subp</th>
<th>com钋supp</th>
<th>com钋sign</th>
<th>com钋ve</th>
<th>com钋e</th>
<th>com钋cart</th>
<th>com钋pur</th>
<th>com钋cat</th>
<th>com钋con</th>
<th>com钋asyn</th>
<th>com钋serv</th>
<th>com钋gen</th>
<th>com钋abstract</th>
<th>com钋tracer</th>
<th>com钋xml</th>
<th>com钋mail</th>
<th>com钋uidgen</th>
<th>com钋signon</th>
<th>com钋processmanager</th>
<th>com钋supplier</th>
<th>com钋creditcard</th>
<th>com钋lineitem</th>
<th>com钋address</th>
<th>com钋lineitem2</th>
<th>com钋creditcard2</th>
<th>com钋address2</th>
</tr>
</thead>
<tbody>
<tr>
<td>com钋main</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>com Suppliers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Fig. 15. DSM for Pet Store AspectJ

### 6.3 Coupling Analysis

These DSMs confirm that in the AspectJ approach the base code is completely oblivious to exception handling – we see that the main application components (1–24) have no dependencies on either the exception handlers (components 25–43) or upon exception monitoring (44–45). The exception handlers in the AspectJ version, however, can be seen to be tightly coupled to the rest of the application (being coupled to 21 out of 24 design variables for the application). Consequently, any change in the main application can potentially induce changes into the exception handlers. This kind of pairwise tangling between application base code and crosscutting concerns is something that aspect-oriented programming seeks to eliminate, but in this case we see that an oblivious methodology has only reversed the dependencies.
Fig. 16. DSM for Pet Store EJP (fully explicit)

Fig. 17. DSM for Pet Store EJP (Co-AOP)
These dependencies also imply that these exception handlers in the AspectJ version are not easily reused within another application, supporting Hypothesis 3.3.1. The only handler component that is not coupled to a Pet Store specific component is the \texttt{generic\_abstract\_handlers} component, which was shown in Section 5.5 to only implement 10\% of exception handling within Pet Store.

In contrast, in both EJP versions much of the main application is coupled to the generic exception handling EJP interface. As these dependencies are void of any implementation or application-specific detail, the \texttt{generic\_EJP\_handler\_interface} component in the EJP DSM can be seen to represent the exception handling specification of the application. Given that most of the application requires some sort of exception handling, it is not surprising that most application components are coupled to the exception handling specification.

The EJP versions did require a few application-specific handlers, but in all cases except for \texttt{util\_tracer\_handlers} these handlers were coupled exclusively to one application component. Each application component and its related handler component can then be viewed as a small module that should not be separated (and could be clustered next to each other in the DSM). Note that the Co-AOP version required three more application-specific handler modules than in the fully explicit EJP version (10 modules for Co-AOP vs 7 for fully explicit EJP). This increase in the number of modules was a tradeoff in order to reduce overall coupling between modules across the application, as studied in Section 5.2 earlier.

6.4 Acyclic Dependency Principle Conformity

The Acyclic Dependency Principle [Martin 2000] states that “the dependencies between packages must not form cycles,” because the package is the unit of software release. Thus, any application-specific handler EJPs and their implementations should be moved inside the related application-specific package so that the code will conform to this design principle. The reason that they were not moved into the same package for our study is that in the Castor Filho study the chosen technique for Pet Store was to place the exception handlers for each application package into a separate package, and in order to avoid bias in our results we followed the design decisions of the Castor Filho study as closely as possible, which meant using such separate packages. Note that for the AspectJ case the application-specific exception handlers could be placed into separate packages while still avoiding cyclic dependencies because the base code is oblivious to the exception handlers, thus the dependency is only unilateral.

The presence and nature of cyclic dependencies in the DSMs in the EJP versions also supports the notion that each application component and its related application-specific handler component should be viewed as a small module. In the DSMs for the EJP versions, there are five (fully explicit version) and six (Co-AOP version) cyclic dependencies. Each of these cyclic dependencies is caused by an application component depending on application-specific exception handler EJPs and by the implementation of these EJPs depending on the application-specific component (e.g., because it depends on the related exception types).

These cyclic dependencies caused by EJP references have very low potential to cause ripple effects or otherwise hinder software maintenance in hard to predict ways for the following reason: except for the component that depends on its as-
associated EJP handler component, there are no other components that depend on each EJP handler component. For example, the $\text{components\_uidgen}$ component depends on the $\text{components\_uidgen\_handlers}$ component (and vice-versa), but no other component depends on $\text{components\_uidgen\_handlers}$. Thus, if one of these “handler components” with a cyclic dependency changes (e.g., $\text{components\_uidgen\_handlers}$), then this change only has potential to impact one other component — the corresponding “non-handler component” (e.g., $\text{components\_uidgen}$). Additionally, changes made in the non-handler component to adjust for changes in its corresponding handler component are unlikely to propagate further. This is because in these cases the code that uses the handler component is internal to the implementation details of that component, and changes to this code will not usually affect publicly visible attributes of the non-handler component (such as method signatures) that could cause changes to ripple farther out.

6.5 Dependency Inversion Principle Conformity

The dependency inversion principle [Martin 1996] states that: first, high-level (policy) modules should not depend on low-level (implementation) modules, and second, abstractions should not depend upon details. The AspectJ DSM indicates that the implementation of the exception handlers has violated the dependency inversion principle in two ways. First, the exception handlers themselves are coupled to application-specific implementation details, rather than to a higher abstract interface modeling exception handling requirements. Second, the additional exception monitoring concern implementation is very tightly coupled to the implementation details of the exception handlers.

These violations of the dependency inversion principle can cause ripple effects as the base code changes, which are magnified as the number of crosscutting concerns increases. For example, suppose that in the AspectJ version the $\text{waf\_view}$ (20) component changes. From the perspective of the application, this is an isolated change, as no other application components or design rules depend on $\text{waf\_view}$. However, changing $\text{waf\_view}$ also affects $\text{waf\_view\_handlers}$, which in turn can affect $\text{exception\_monitor\_impl}$. In contrast, in the EJP versions, changing $\text{waf\_view}$ (28) causes no further necessary changes, supporting Hypothesis 3.3.1 and Hypothesis 3.3.2.

Another factor that mitigates ripple effects in the EJP versions is that the application specific handlers are implemented via dependencies on the abstract exception handling specification via $\text{generic\_EJP\_handler\_interface}$. This allows the exception monitoring concern to be unaware of any application-specific handlers, and allows to limit its coupling to the generic exception handling specification via $\text{generic\_EJP\_handler\_interface}$.

In general, in the AspectJ version, changing application components affect the application-specific handlers, which ripple outward to also affect the exception monitoring concern. In the EJP versions changing application components affects related application-specific handlers (if any), but does not ripple outward. Also note that in the EJP versions only 7 (fully explicit) and 10 (Co-AOP) application-specific handler components were required (7–9, 26, and 29–31), as opposed to 18 components in the AspectJ version (26–43), further reducing potential rippling effects. This data supports Hypothesis 3.3.5.
The conformity of the EJP versions to the dependency inversion principle also provides for both the generic exception handling implementation (modeled in the DSM by `generic_EJP_handler_impl`) and the exception monitoring implementation (modeled by `generic_EJP_handler_impl`) to be packaged into reusable, pluggable aspect libraries. Other aspect-library implementations conforming to the `generic_EJP_handler_interface` and `exception_monitor_interface` interfaces could be employed instead at compile-time or application load-time. Such use of generic aspect-libraries is not viable in the AspectJ version, as the exception monitoring implementation is directly coupled to many application-specific design rules.

6.6 Open–Closed Principle Conformity

The open–closed principle (OCP) states that a module should be open for extension but closed for modification [Meyer 1997]. In this section we present a qualitative analysis on conformity of the application versions to this principle. Quantitative techniques to assess this have also been proposed, such as by Cai et al. [Cai et al. 2007], wherein a pair-wise dependency relation is used to explicitly measure conformity to the OCP.

To analyze OCP conformity, for each version (AspectJ, fully-explicit EJP, or Co-AOP EJP) we consider the differences between two consequent versions of the DSM: the DSM of the original application without the exception monitoring concern, and the DSM for the application extended with the exception monitoring concern. We study based on the differences between these DSMs the degree to which existing modules had to be modified in order to accommodate the extension. For the sake of brevity, we do not present the DSMs for the versions without the exception monitoring concern, but they can be easily derived from the DSMs in Figures 15 and 16 by removing the rows and columns for the `exception_monitor_interface` and `exception_monitor_impl` design variables. Both the AspectJ and the EJP versions conform to the open–closed principle with respect to the exception monitoring concern, as can be seen by observing that none of the other design variables depend on the design variables representing the new exception monitoring concern.

6.7 Analysis Summary

The DSMs confirm that in the AspectJ version the application is indeed completely oblivious to exception handling, but this comes at the cost of minimal handler re-use and increased application-specific handlers and application-specific coupling. The non-conformity of the aspect handlers to the dependency inversion principle magnifies ripple effects in the presence of multiple related crosscutting concerns (such as exception monitoring), and also restricts packaging the exception monitoring implementation into a separate reusable aspect library.

The DSMs also confirm that in the EJP versions the application is coupled to both an abstract exception handling interface as well as a few application-specific handler EJPs. The application-specific EJPs in turn also depend upon the abstract exception handling interface. This conformity to the dependency inversion principle allows for the implementation of exception handling to be well-encapsulated within a reusable, pluggable aspect-library. This conformity also facilitates the addition of new concerns after the initial application has been developed (e.g., ex-

ACM Journal Name, Vol. V, No. N, Month 20YY.
ception monitoring). The EJP versions are thereby open for extension but closed for modification (conforming to the open–closed principle).

We note that the Java DSM is nearly identical to the fully explicit EJP DSM except that:

— There are no handler components present.
— There is no generic EJP interface for exception handling.
— Any component that requires exception handling depends on the exception monitoring interface.

The design qualities of the Java version mainly differ vs the fully explicit and Co-AOP EJP versions in that extensibility is reduced (as there is no design rule modeling the exception handling specification) and that there is no reusability for exception handling code.

7. LIMITATIONS OF THIS STUDY

In this section we discuss identified threats to internal and external validity as well as our controls to mitigate these threats.

7.1 Threats to Internal Validity

The following lists the threats to internal validity that we have identified, and also describes our controls to mitigate these threats:

— Our metrics collection tools were different than those used in the prior Castor Filho study [Castor Filho et al. 2006], so these new tools have the possibility of being inaccurate or biased in the way the metrics are calculated. By first replicating the results of the prior study, we validated that our chosen metrics collection tools and techniques were accurate and consistent with those of the Castor Filho study. Our study also considers metrics not considered by those in the Castor Filho study. These metrics were previously proposed and implemented by other authors [Ceccato and Tonella 2004], reducing the likelihood that these metrics are biased for the purposes of our study. We also compared the output of the tool on small code samples to manually calculated metric values to ensure that the values were being calculated as we expected them to be.

— Our selection of metrics is arguably a limitation, as there are certainly other metrics proposed in the literature not presented herein. Whereas a study cannot consider all possible metrics and measures, it is inevitable that some selection must be made. Additionally, even if all metrics were considered, metrics may not effectively evaluate all possible aspects of software quality. We focused on the metrics in Section 4.5 as they have been shown to be effective in many previous case studies [Garcia et al. 2004; Garcia et al. 2005; Cacho et al. 2006; Castor Filho et al. 2006; Greenwood et al. 2007]. We also selected other metrics previously proposed specifically for studying aspect-oriented software that haven’t yet been widely used in empirical studies, although some preliminary analysis on these newer metrics have been conducted [Shen and Zhao 2007]. Our study serves as a data point in order to better understand these relatively new metrics, especially as future studies that include these metrics emerge.
Our choice of methodology in how code was refactored with the different techniques could bias the results in favor of one technique or another. We address this in two ways. First, we employ applications that already had the concerns of interest refactored from Java to AspectJ in previously published studies. The goal of the authors of these studies was to create high-quality refactorings from Java to AspectJ to facilitate meaningful comparisons between the two versions. Re-using these refactorings allows us to compare against previously published baselines. Second, we defined a consistent methodology for performing the refactoring (as described in Section 4.1) and then strictly followed this methodology during the refactoring. While this cannot guarantee the absence of bias, it does provide more confidence than having performed the refactorings in an ad hoc manner.

In order to increase confidence in the accuracy of our refactoring and metrics calculations as well as in our adherence to our refactoring methodology, we are making available all versions of the source code as well as the full raw empirical data that was calculated. This should be sufficient information to allow others to reproduce the results of our study and confirm that our described methodology was correctly followed.

7.2 Threats to External Validity

The greatest threat to the generalizability of our results is that the number of applications (four) and number of crosscutting concerns considered (five) are not large enough for the observed differences to be statistically significant. A current challenge for any study of aspect-oriented software is that there is not yet many large, publicly available AspectJ applications that can be studied, compared, and evaluated, as noted elsewhere [Störzer 2007]. While admittedly not ideal, our current set of applications does provide a broad range of applications: Pet Store is a Java Blueprint for the Java Platform Enterprise Edition (Java EE), and is intended to be a model of good software design and proficient use of design patterns. Telestrada is a corporate information system developed for a Brazilian national highway administrator. JHotDraw was originally designed as a showcase of how to effectively use the Gang-of-Four object-oriented design patterns in Java. AJHotDraw is a refactoring of JHotDraw where many of the concerns have been implemented using aspects. Health Watcher is a web information system designed using the model-view-controller pattern, and also makes use of Java servlets, both characteristics being prototypical of many Java backend web applications.

Additionally, while our study does have a heavy focus on the exception handling concern, with three out of the four applications focusing on this concern, the fourth application considers an entirely different set of four crosscutting concerns, namely: contract enforcement, undo/redo, persistence, and selection changed notification. All such concerns considered in this study can be classified as dynamic, advanced, heterogeneous concerns, as defined in [Apel and Batory 2008; Apel et al. 2008; Colyer et al. 2004; Mezini and Ostermann 2004]. Given that these types of concerns are the most likely to exhibit the challenges discussed in Section 2.1, it is not unreasonable to postulate that EJPs and Co-AOP will be the most effective when applied to this particular type of concern. While further data is needed to make a
more definitive statement, this study should provide the insights needed to properly apply EJPs and Co-AOP to other similarly classified concerns.

This study only considers two aspect-oriented languages (AspectJ and AspectJ with EJPs), whereas many others have been proposed in the literature. Other AO languages with more flexible or powerful join point models than AspectJ may not experience all of the same problems as seen in AspectJ, or at least to the same degree. The purpose of this study is to better understand potential tradeoffs between obliviousness and modularity. AspectJ was designed to implement aspects in a completely oblivious fashion, making it a suitable language choice to implement crosscutting concerns in a completely oblivious fashion. The EJP language extension was included in the study because it was designed to facilitate implementations of crosscutting concerns in a completely explicit fashion, providing the greatest contrast for this study.

Another threat to generalizability is how much in-depth knowledge of an application is required in order to effectively apply Co-AOP techniques. A key point in making Co-AOP effective is selecting the proper join points to make explicit in the base code. Although none who refactored code in this study were familiar with the applications previously, all of the participants were graduate students and had prior experience with large software systems. However, metrics for determining fragile pointcuts, such as Coupling on Intercepted Modules, can effectively identify the most likely pointcuts where applying explicitness should be considered. Additionally, the principles discussed in this paper should further guide proper application of explicitness to reap the found benefits of Co-AOP.

8. RELATED STUDIES

AOP and its accompanying conjectures of improving software development (e.g., by increasing modularity or via the separation of concerns) have been the subject of various empirical studies. We overview the most closely related ones.

Early studies mostly focused on the overall applicability of aspects. Pace and Campo [Pace and Campo 2001] compare different ways of putting aspects to work other than through linguistic constructs as in AspectJ or HyperJ—namely through frameworks or by architectural design. A main conclusion is that the design of applications following some aspectual decomposition is more important than the actual mechanism used to put aspects to work, since the application has to be designed to match the constructs at hand. This work provides indication that using explicit join point interfaces within the aspectual decomposition can provide more room for future design and implementation choices due to semantic decoupling.

Baniassad et al. [Baniassad et al. 2002] show evidence for the presence of crosscutting concerns in practice and how they are implemented without AOP. Three main types of such crosscutting concerns are identified as well as implementation strategies. The study represents a cornerstone for the assessment of AOP.

Tsang et al. [Tsang et al. 2004] present an initial study illustrating the tradeoffs involved in increasing modularity. They empirically compare AO vs OO solutions in the context of Real Time Java, illustrating that modularity is improved with AOP but that in terms of other factors such as maintainability, reusability and
testability, the original object-oriented programming (OOP) solutions are often favorable. Based on the results of our study we note that reducing obliviousness via EJPs in appropriate places may help these other factors remain more favorable.

Lopes and Bajracharya [Lopes and Bajracharya 2005] employ design structure matrices, modular operators, and Net Options Value (NOV) to assess the impact of aspects on modularity and design evolution within the context of a web services application. Their results validated the use of DSMs to analyze aspect-oriented software, found that aspectization was a variant of the inversion operator, and in their case study found that aspects improved the Net Options Value of the case study’s design. They recommended specifying design rules for aspects, which implies making join points more explicit. Sullivan et. al. [Sullivan et al. 2005] study design rules for aspects in more detail, proposing crosscutting interfaces that are evaluated through DSMs and NOV. Our study focuses on how both code quality attributes and application design are affected when joinpoints are fully explicit.

Garcia et al. [Garcia et al. 2005] implement the 23 Gang-of-Four patterns using aspects and present the results of an empirical study comparing the AO implementation to the OO implementation. Results indicated that while aspects usually improved separation of concerns, for some patterns aspects increased coupling, complexity, and size. Additionally, significant reuse was only exhibited within 4 patterns. Our study further explores factors affecting aspect reusability.

Li et al. [Li et al. 2006] study the applicability of AOP to implement, maintain, and evolve commercial-off-the-shelf components. They conclude that aspects help mainly for homogenous crosscutting concerns, i.e., concerns which “have a consistent application of the same or very similar policy in multiple places.” For more heterogeneous, scattered concerns, their study shows no improvement. Our study seeks to better understand how improvement for this case can be obtained through Co-AOP.

Castor Filho et al. [Castor Filho et al. 2006] investigate the way aspects affect coupling, conciseness, cohesion, and the separation of concerns in the context of exception handling by an extensive test suite. They rightly point out that most applications employ complex exception handling, as opposed to more structured exception handling patterns occurring in frameworks (as studied previously in [Lippert and Lopes 2000]). We build on their study as a starting point for our investigation of obliviousness and modularity.

Cacho et al. [Cacho et al. 2008] propose EJFlow, a language extension to AspectJ to improve modular and oblivious handling of exceptions. The authors empirically analyze the effects on code quality attributes for a mobile phone application. Their results show that in the case of their application of study the EJFlow extension facilitated significant improvements in code quality metrics. EJFlow differs from implementing exception handling via EJPs in primarily two ways: (a) the EJFlow language extension is specifically for exception handling, whereas EJPs are designed for any concern requiring tight interaction with the base code, (b) the design goal in EJFlow is to allow for oblivious modeling of exception handling behavior in an end-to-end fashion, and allows for reusable handling behaviors to be applied based on matches over a combination of the raise site and the exception propagation chain within the source code (including handler sites); in contrast in this work EJP
declarations model reusable handler behaviors, and then allow the base code to apply these behaviors at the point where exceptions must be handled.

Greenwood et al. [Greenwood et al. 2007] compare pure OOP and AOP with respect to software evolvability. As expected, changes to crosscutting concerns are better handled in AOP software, while changes of a more fundamental and architectural nature can often be better assimilated by OOP solutions. The authors state that this is due to the observation that AOP narrows the boundaries of concern dependencies, but tightens their interaction. We note that explicit join point interfaces as studied in this work can become a focal point to guide and control this interaction.

9. CONCLUSIONS

We have explored the relationship between obliviousness and modularity of aspects as expressed within AspectJ, as well as the effectiveness of the explicit join point construct and the Cooperative AOP methodology. The use of well-established software quality metrics were used as a basis for this discussion and our evaluation of explicit join points and Cooperative AOP. This is also the first study that considers coupling metrics to measure both \([\text{base code} \rightarrow \text{aspect}]\) coupling and \([\text{aspect} \rightarrow \text{base code}]\) coupling. This combined perspective allows us to more fully understand the modularity implications of obliviousness and the ways in which obliviousness and modularity can both be maximized. Additionally, the design properties of the AspectJ and EJP versions were analyzed using Design Structure Matrices to discover conformity to well-established design principles.

The primary conclusions of the study, subject to the limitations discussed in Section 7, are as follows:

— The use of an explicit join point interface to model crosscutting concerns facilitates the creation of reusable aspect libraries.

— The parameterization of aspects made possible by these explicit join point interfaces can increase code reuse and reduce pointcut complexity.

— Explicit join point interfaces must be carefully designed to be as minimal as possible or overall application modularity may actually decrease.

— Explicit join point interfaces enhance conformity to both the dependency inversion principle and the open–closed principle.

— When pointcuts can be written in a stable fashion and advice parameterization is minimal, obliviousness is favored over explicitness in the base code, but in the remaining cases leveraging EJPs results in improved code quality attributes, as modeled by the considered metrics. This conclusion is consistent with the design rule theory in [Sullivan et al. 2005], in which the interfaces between the base code and aspect should be stable. EJPs can be used to effectively and explicitly model these stable interfaces.

Overall the study shows that by explicitly modeling crosscutting concerns and appropriately applying aspect-oriented techniques, Cooperative AOP makes it possible to improve separation of concerns (vs an object-oriented approach) while also achieving reduced coupling between aspects and the base code (vs an AspectJ approach) and facilitating reusable aspect components.
10. ACKNOWLEDGEMENTS

We would like to thank the authors of the study built upon in this paper [Castor Filho et al. 2006] for providing code and help in reproducing their results, and the anonymous reviewers for their guidance and detailed comments.

REFERENCES

ACM Journal Name, Vol. V, No. N, Month 20YY.


Received Month Year; revised Month Year; accepted Month Year