Aspects and Exception Handling: the Case of Explicit Join Points

TECH REPORT ejp-200703-01

Kevin Hoffman
Purdue University
305 N. University Street
West Lafayette, IN 47907
kjhoffma@cs.purdue.edu

Patrick Eugster
Purdue University
305 N. University Street
West Lafayette, IN 47907
peugster@cs.purdue.edu

ABSTRACT

Several authors have debated the modularity and obliviousness of aspects in AOP and the links between these two notions, noting that obliviousness is not always desirable or achievable. Many proposals have appeared, mainly in the context of AspectJ, to mitigate these issues by restricting upfront, or "inferring" and documenting, where aspects can apply. As pointed out, sacrificing certain facets of obliviousness can not only increase safety but even increase modularity.

This paper presents and evaluates a simple extension to AspectJ, consisting in explicit join points (EJPs) which denote potential occurrences of aspects in the base code and enable information passing between base code and aspects. The evaluation takes place in the context of exception handling; by picking up on a recent study of the use of aspects for the same purpose, we *quantify* the benefits of our extensions for various common measures of code quality in the context of AOP, such as separation of concerns or coupling.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Aspect-Oriented Programming*; D.2.8 [**Software**]: Software Engineering—*Metrics*

1. INTRODUCTION

Aspect-oriented programming (AOP) [17] is slowly stepping out of its infant shoes. By offering programmers the possibility of dealing with *cross-cutting* 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, synchronization, or persistence. The ability to describe aspects both cleanly and safely is crucial to help developers think in terms of aspects at early development stages and thus represents a cornerstone towards aspect-oriented software development.

AspectJ. AspectJ [16] is an extension of Java that provides powerful new constructs to support aspect-oriented programming (AOP). In AspectJ, programs are decomposed along "functional" lines in an object-oriented fashion as facilitated by Java, but code to implement any cross-cutting concerns is isolated into aspects which are injected by *advice* into the base code at *join points*. AspectJ provides the *pointcut* construct, allowing for collections of join points to be specified based on lexical context, join point type, pattern matching, and even dynamic context (such as control flow or argument value).

Modularity and obliviousness. Various authors have recently debated the full implications of AOP for modularity and modular reasoning. In particular, the impact and feasibility of aspects' *obliviousness* with respect to the base code has been the topic of many recent publications, e.g. [26, 19, 7, 9]. Early work on AOP, and AspectJ in particular, have described modularity as a direct consequence of obliviousness. While 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 [14, 2, 18, 26] for *safety* purposes, Sullivan et al. [26] thoroughly decompose the notion of "obliviousness", pointing out that certain of its facets might even *reduce* modularity.

Contributions. This paper presents and evaluates a pragmatic extension of AspectJ, consisting in *explicit join points* (EJPs). In short, EJPs explicitly denote potential occurrences of aspects within the base code, allowing for the passing of information to and from aspects. *Scoped* EJPs are a generalization of any join points in the sense that they enable the advising of arbitrary code blocks, i.e., *several* subsequent statements, in contrast to common models which focus on advising single statements (or entire methods when distinguishing between callee- and caller-site). We present an overview of our new language features, comparing them to traditional methods, along with our implementation of EJPs based on the `abc` compiler [1].

While quite obviously sacrificing some obliviousness, we illustrate through an in-depth empirical study how EJPs *quantitatively* improve the quality of three industrial-strength

applications after refactoring the exception handling cross-cutting concern using EJPs. More precisely, we revisit a prior study [8] where these same applications were refactored for the same purpose, directly comparing our method to the AspectJ method and original code.We discuss the quantitative results of our study, considering the effect of EJPs on well-established metrics such as coupling, cohesion, size, and separation of concerns, and then discuss important qualitative issues. We also contribute the results of an extendibility study, exploring how EJPs improve the ability to implement future concerns obliviously. Finally, we position EJPs with respect to seminal work on modular aspect-oriented software development, pointing out how EJPs can be used to implement or complement these approaches.

Roadmap. Section 2 presents an overview of explicit join points. Section 3 presents the setting of our case study. Section 4 presents our findings, and Section 5 discusses various issues. Section 6 studies the impact of explicit join points on oblivious extendibility. Section 7 presents related work, and Section 8 summarizes our work.

2. EXPLICIT JOIN POINTS: A PRIMER

This section describes some of the existing challenges in aspect-oriented software development with AspectJ. We then present an overview of *explicit join points*, which we use in our case study to implement aspectized exception handling for major software packages.

2.1 Motivation

AOP as expressed in AspectJ has been proven to be very effective in separating cross-cutting concerns from the base code. However, while it is clear that implementations of such concerns in AspectJ are completely separated from the base code, the effects of this separation on other important factors such as modularity and coupling and the impact on the software development process remains a highly researched topic. Empirical case studies illustrating some of these effects have recently begun to emerge [8, 4, 26, 27, 12, 11].

One important problem highlighted by these studies is that even though couplings between the base code and the classes implementing the cross-cutting concern are removed, new couplings are introduced between the aspect and base code. This coupling is founded in the inherent complexity in aspects trying to describe the precise *join points* at which they should inject new logic because these join points are not explicitly named. Instead, these 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 so that they anchor cross-cutting logic precisely where needed, without unintentionally matching additional join points. When pointcuts are defined a tradeoff must inherently 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, meaning that there is greater possibility for it to stop matching what it semantically intends to advise as base code is refactored. This pointcut induced fragility couples the aspect to the base code, requiring the aspect's pointcuts to be revisited as the base code changes and thus reducing its modularity.

On the other hand, using pointcuts that are too general may cause advice to be applied unintentionally, which can be problematic when the concerns being implemented have non-trivial semantics (e.g. persistence or concurrency control). Finding the appropriate balance between precision and generality while optimizing the different software engineering factors is a non-trival challenge in aspect-oriented software development. Additionally, when base code is written without a priori planning for the aspectization of certain concerns (as is actually a goal promulgated by many AOP supporters), the resulting aspects can become tightly coupled to the base code.[8, 26]

Another challenge of AspectJ is the limited granularity of join points that can be advised. Because pointcuts pattern match against type names and identifiers in the base code, the granularity where you can apply *around advice* is limited – either around a primitive statement (field access, method call), an exception handler, or an entire method. This restriction makes it difficult to apply advice around a set of statements within a method, and thus limits how advice can affect control flow within a method.

This inability to advise arbitrary blocks of code can magnify the fragile pointcut problem [25], as programmers try to approximate advising blocks of code by using pointcuts that individually specify the field and method call accesses within that block of code. This technique actually encourages extremely fragile pointcuts, which can fail in two ways: (1) A call to a method that matches the pointcut targeting a block of code is added outside that block of code in the same method. In this case the programmer may attempt to exclude the new method call by increasing the precision of the pointcut (by matching based on properties of the target object or parameters instead of just the method name). (2) Doing so, however, increases the likelihood that the pointcut will stop matching the method call within the targeted block of code as that code changes. We again see the theme of trading between precision and pointcut stability, only magnified by the attempt to target a specific collection of join points instead of targeting a single join point.

Details of challenges created by join point anonymity along with illustrative examples are described in [26].

Beyond quantification difficulties, the one-sidedness of aspects obliviously advising base code creates limitations when the concerns being implemented may be semantically coupled in some fashion. Concerns of this nature can be classified as those with significant semantics (more than trivial logging or tracing semantics) that have the potential to fail or require complex contextual input from the base code. For example, an aspect implementing transactionalizing mechanisms may need to ensure that the base code (or at least some other error handling aspect) reacts to any fault encountered while committing a transaction. Using AspectJ, there is no mechanism for an aspect to require that a new type of checked exception is handled at certain join points (the

opposite functionality, exception softening is provided, further highlighting the one-sidedness of the advising methodology in the design of AspectJ). Additionally, there is no mechanism to capture the state of local variables, except as they are exposed as return values or arguments, limiting the amount of local state that can be captured by a single advice. This leads to complex (potentially fragile) multi-stage advising patterns and other problems, discussed as the state–point separation problem in [26].

In summary, while AspectJ provides mechanisms to effectively separate cross-cutting concerns from base code, these same mechanisms can introduce other problems affecting the effectiveness of the software engineering process as a whole.

2.2 Introducing Explicit Join Points

To address the above challenges we introduce a new type of join point into the quantification model – explicit join points (EJPs). Unlike anonymous join points that are exposed automatically by the AspectJ compiler, EJPs are explicitly declared by the programmer, given a unique name and signature, and referenced explicitly in the base code. Additionally, we allow EJPs to be scoped, empowering the advising of arbitrary blocks of code without increasing pointcut fragility. Finally, we define ways in which EJPs can enforce constraints, both on the base code referring to EJPs and on the aspects that advise EJPs. Evaluation of the concept will be reserved for future sections of the paper.

An explicit join point is a signature declared within an aspect, defining its properties, parameters, and constraints. In it's simplest form, an EJP looks similar to a method declaration in an interface and is named by an identifier, can return a value, has formals and a `throws` clause, and has a new construct – the *handles clause*. The syntax for declaring EJPs (simplified) is as follows:[1]

```
[scoped] joinpoint <return type> <name>(<formals>)
                            [handles ...] [throws ...];
```

The *scoped* modifier specifies that when the base code references the EJP it must be associated with a block of code, as will be shown below. The `handles` clause constrains aspect implementers of the EJP, requiring that for each type in the `handles` clause at least one aspect implements an around advice catching that exception type. This constraint can be enforced at compilation to ensure checked exception safety is preserved. In the base code use of a scoped EJP with a `handles` clause has the same semantics on checked exceptions as if the block of code in the scope of the EJP reference were contained inside a `try-catch` block with `catch` blocks for each type in the `handles` clause. Additional ways in which the EJP can be parameterized have been explored (e.g. *pointcut parameters*) but are outside the scope of this paper.

References to scoped EJPs in the base code are similar to static method invocations and anonymous class creation:

```
<AspectName>.<EJP name>(<formals>) {
   ...
};
```

[1]Full syntax definitions can be found in [15]

Finally, additional primitive pointcut designators are defined, `ejp` and `ejpscoped`, allowing pointcuts to match EJP references in a stable manner. For example:

```
pointcut needsMemoization():
     call(Value *.*(..))
  && cflow(execution(ejpscope(memoizeCalls)))
```

This pointcut matches calls to any function returning the type Value in the control flow of the memoizeCalls scoped EJP. This example sketches how the base code and aspect can cooperate to robustly advise arbitrary blocks of code.

To promote modularity, EJPs should be defined in abstract aspects separated from any concrete advice, thus becoming an explicit form of an abstract cross-cutting interface. All known principles of designing modular interfaces apply here, now keeping in mind the interface will be implemented by one or more aspects and the new power offered by scoped EJPs.

Figure 1 visualizes the difficulty in implementing a cross-cutting concern using AspectJ when that concern needs to access state only available across multiple (dissimilar) join points and how that compares to using EJPs. It also shows how scoped EJPs compare.

A well known alternative to advising arbitrary blocks of code is to extract the fragment of code that should be advised into a new method, exposing needed local variable values and allowing advising of that block of code. However, this has empirically been shown to decrease cohesion, sometimes significantly. [8] Additionally, such a technique causes unnecessary tangling in the base code when more than one concern requires the refactoring of methods. As the number of complex concerns increases, the base code can become splintered and hard to refactor. Another issue is that the signature of the newly created method is replicated each time a new fragment of code is extracted, making it more difficult if that signature must change. In contrast, EJPs are minimally intrusive into the base code and EJP signatures are only defined once. The benefits of EJPs vs. the traditional method are explored in our empirical case study.

2.3 Implementation

We implemented EJPs in AspectJ by extending the AspectBench research compiler (`abc`) [1]. To encourage industrial use and feedback, peer evaluation, and future research, and in agreement with the licensing style of `abc` and its dependent packages, our EJP extension with source code is freely available for download and distribution under the GNU Lesser General Public License (LGPL) [15].

Support for non-scoped EJPs was straightforward via abstract syntax tree rewriting and by extending the type system, while support for scoped EJPs proved to be more interesting. Our approach lifts the code within the scoped EJP and places it into a new inner class. Due to the limitations of Java, references to formals and local variables declared outside the EJP scope are converted into fields in the in-

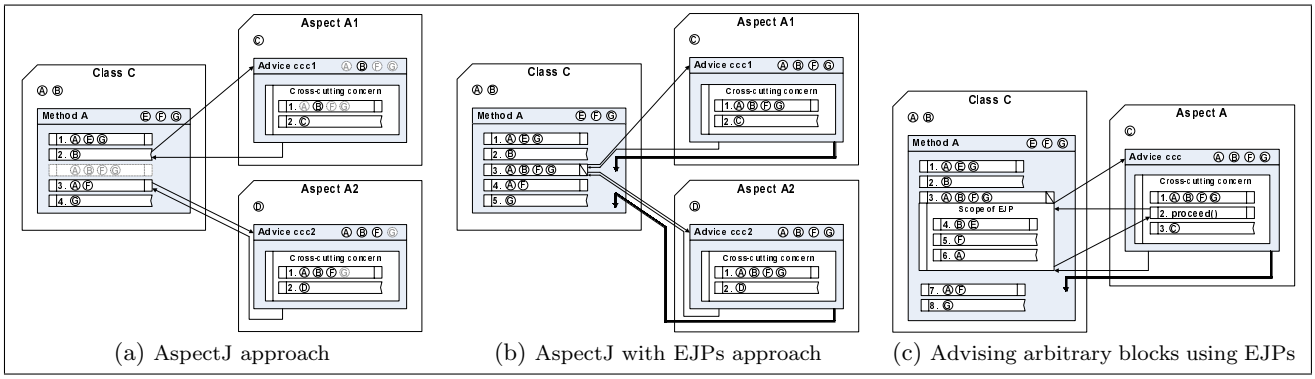


Figure 1: Visualization of the differences between traditional AspectJ join points and Explicit Join Points

ner class. Code is generated to instantiate the inner class, populate its fields, call the method with the lifted code, and copy changed field values back to local variables.

Although our technique requires a new inner class object to be instantiated every time a scoped EJP is entered, this runtime overhead is mitigated by fast allocation and escape-analysis optimizations in modern JVMs [13] (allowing heap allocation to be converted to stack allocation). Alternative implementation strategies, such as modifying the around weaver to treat EJP scopes as new dynamic contexts using mechanisms along the lines of [20] could facilitate additional compile-time optimization.

3. CASE STUDY SETTING

A recent study [8] (herein termed the AspectJ study) quantatatively compared the benefits and drawbacks of using AspectJ to implement exception handling in an oblivious manner. In their study four applications were refactored and then evaluated to see how cohesion, coupling, conciseness, and separation of concerns were affected. We build upon this study by refactoring three of the four applications using EJPs to implement exception handling, starting from source code provided by the authors of the AspectJ study. We then evaluate the effectiveness of EJPs using a superset of the metrics employed in the AspectJ study.

3.1 Technique

In the AspectJ 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. Advice were organized so that exception handling logic for either a single class, a single package, or a single concern were contained within a single aspect. Exception detection code was not aspectized.

Our strategy for refactoring closely follows that of the AspectJ study for both advice creation and organization, properly modified 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. The parameterization provided by EJPs allowed for modeling of many common exception handling patterns while still remaining generic and reuseable.

Rather than using a mixture of oblivious aspects and EJPs we used EJPs exclusively so the differences between the approaches would be highlighted, and this approach also preserves checked exceptions without requiring softening.

As in the AspectJ study, exception handlers were implemented using after advice when possible, reverting to around advice when exceptions had to be caught but not propagated. The following trivial example demonstrates the general pattern of our approach:

```
class C                                  aspect A {
  void m() throws ... {                    scoped joinpoint ejpH()
    try {/*body*/}                           handles E throws ...;
    catch(E e) {...}        ==>            void around() throws ...:
  }                                             call(ejpScope(ejpH)) {
}                                            try{ proceed(); }
                                             catch(E e) {...}
                                         }
                                       }
                                       class C {
                                         void m() throws ... {
                                           A.ejpH() {/*body*/}
                                         }
                                       }
```

In both our study and the AspectJ study the refactoring was semantics-preserving, such that 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 fair comparison between the different versions.

3.2 Target Applications

In our study we have refactored two object-oriented applications and one aspect-oriented application using EJPs. These same applications were refactored in the AspectJ study. They were chosen by the authors of the AspectJ study because they are representative of real-world applications that exhibit a variety of exception handling strategies, each with a different mixture of cross-cutting concerns.

The first application is a subset of Telestrada, a travelor information system originally implemented in Java (220+ classes and interfaces and about 3400 LOC).The second application is Java Pet Store – a demo application for the J2EE platform that showcases how to build robust enterprise applications (340+ classes and interfaces and 17800 LOC).

The final application is 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 (which were

Attributes	Metrics	Definition
Coupling	Coupling Between Modules	Number of modules declaring methods or fields potentially called or accessed by another module.
	Coupling on Intercepted Modules	Number of modules explicitly named within pointcuts.
Cohesion	Lack of Cohesion in Operations	Number of pairs of methods accessing different fields minus number of pairs of methods accessing common fields.
Size	Lines of Code	Number of uniformly formatted lines of code, excluding whitespace and comments.
	Concern Lines of Code	Subset of Lines of Code used to implement a specific concern.
	Number of Operations	Number of declared methods and advice.
Separation of Concerns	Concern Diffusion over Modules	Number of modules that implement a concern or reference one that does.
	Concern Diffusion over Operations	Number of operations that implement a concern or reference one that does.
	Concern Diffusion over LOC	Number of transitions between one concern to another across all lines of code.

Table 1: Metric definitions

converted to use EJPs). This application consists of 36 aspects, 96 classes and interfaces, and 6600 LOC.

3.3 Metrics Suite

Recently there has been increased interest in empirically evaluating aspect-oriented software. AOP specific metrics have been proposed in [23, 5, 28]and used within empirical case studies of aspect-oriented software such as [8, 4, 12, 11].

The metrics used in this study, are a superset of those used in the AspectJ study, being supplemented by metrics proposed in [5]. Table 1 overviews our primary metrics, focusing on coupling, cohesion, size, and separation of concerns.[2] The cohesion, coupling, and size metrics are variants of the well known CK metrics [6] extended to support AspectJ concepts.[23] We also use aspect-specific coupling metrics as proposed in [5] to better understand pointcut induced coupling. The separation of concern metrics are introduced in [23] and 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. Additionally, we introduce the concern lines of code metric to better understand code size for just exception handling.

The metrics were calculated for all three versions of each of the three target applications, primarily using the aopmetrics tool [24]. This tool has slightly different heuristics for some metrics (e.g. LOC) than in the AspectJ study, but any differences are minor, and the same trends and degrees of change can be observed within the data.

4. QUANTITATIVE RESULTS

This section presents the results of the empirical metrics, organized by attribute. The data are presented using stacked bar graphs, allowing for inspecting of the contribution of the base code, exception handling aspects, and other aspects (only in Health Watcher) to each metric. In addition to analyzing the total metric values, additional insight can be gained by inspecting the sub-totals for just the exception handling aspects. In all cases lower values are better.

Results for metrics are given in the following order: coupling and cohesion, size, and separation of concerns. From henceforth, metric results will always be listed in the following order: Telestrada, Java Pet Store, Health Watcher.

[2]Herein we use the term module to refer to interfaces, classes, and aspects.

4.1 Coupling and Cohesion Metrics

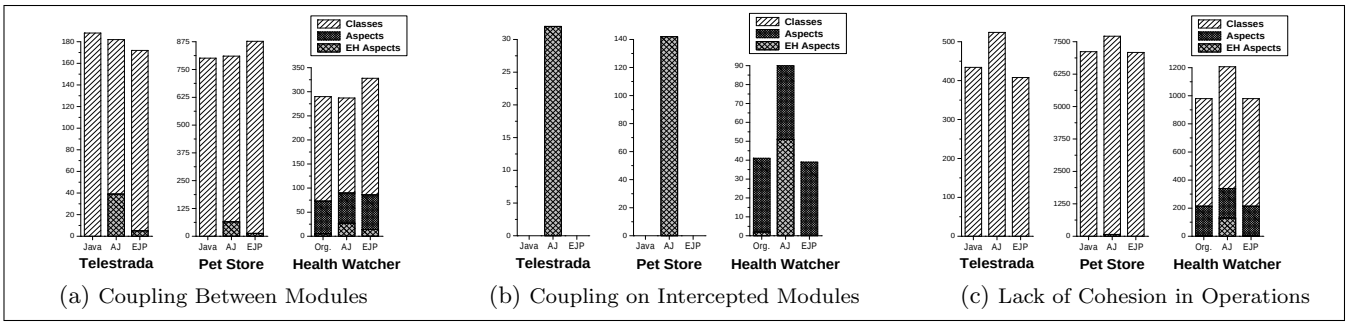
Figure 2 shows the results for the coupling and cohesion metrics. Careful inspection of the results show that in the EJP version there is significantly less coupling between the aspects and base code, and that there are significant improvements in cohesion in the EJP version over the AspectJ version. Overall the total coupling of a system can increase or decrease, depending upon factors discussed below.

The most significant indicator of the decrease in coupling between aspects and base code is the impact of EJPs on the Coupling on Intercepted Modules (CIM) metric. This metric counts the number of modules explicitly named in pointcuts.Compared to the AspectJ versions, the EJP versions have a reduction of 100%, 100%, and 57% in CIM.The exception handling aspects in the AspectJ version caused disproportionate increases in CIM. For example, these aspects in Health Watcher AspectJ caused a 120% increase in CIM but account for only 12% of the number of aspects.

For all AspectJ versions the exception handling aspects were almost all tightly coupled to their advisees, as indicated by the significant increases in their CIM values. Beyond referring explicitly to base code classes in pointcuts (increasing CIM), these pointcuts were sometimes quite complex, having to capture exact call sites or field accesses within specific methods, for example to deal with state–point separation scenarios [26]. The AspectJ study did take into account this particular metric, although they do mention a "hidden coupling" between the aspects and the base code. This metric clarifies the strength of that hidden coupling.

Just looking at the numbers, EJPs did not consistently perform well for total Coupling Between Modules (CBM) metric. Compared to the AspectJ versions the EJP versions differed by -5%, +8%, and +14%.Compared to the original versions, EJP versions differed by -9%, +9%, and +13%.

This metric counts the number of other modules coupled to a module through field accesses, method calls, or EJP references and was affected by three factors for the EJP versions. First, the use of parameterized EJPs facilitate generic exception handling logic (which are free from couplings), reducing the coupling within this logic significantly. For example, the CBM metric value for exception handling aspects for Java 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,



(a) Coupling Between Modules (b) Coupling on Intercepted Modules (c) Lack of Cohesion in Operations

Figure 2: Results for coupling and cohesion metrics (lower is better)

decreasing the CBM metric value in the base code (e.g. the reduction for Telestrada was 9%). Third, EJP references in base code increased the CBM metric, as each base code class is coupled to the aspect interface containing the EJP declarations. In Java Pet Store and Health Watcher, this factor outweighed the others, resulting in a higher total CBM. These results indicate the need to carefully design the EJP declarations to prevent unnecessary couplings.

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. 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, sometimes significantly (from 8% to 22%). The decreases in cohesion in the AspectJ version are caused by the need to extract new methods to expose advisable join points (e.g. `try-catch` blocks in loops, etc.). As dicussed in [8], these new methods are a negative byproduct of the AspectJ refactoring, and is one empirical indicator of the benefits of scoped EJPs.

4.2 Size Metrics

Figure 3 shows the results for the size metrics. Lines of code decreased for all applications for all versions, except compared to the AspectJ version for Health Watcher. Concern lines of code consistently decreased, usually significantly. Number of operations always increased vs the original version, but was always lower than the AspectJ version.

Compared to the original version, Lines of Code for the EJP versions differed by -6%, -4%, and -4%. Compared to the AspectJ version they differed by -4%, -4%, and +3%.

The Concern Lines of Code metric counts lines of code that implement the exception handling concern, including EJP references in base code. The EJP version performed significantly better than both the original and AspectJ versions for Telestrada and Java Pet Store (by 32% and 46%), while only a 6% reduction was observed for Health Watcher.

Two EJP induced factors affect the lines of code 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 showing EJPs performing better than AspectJ for this subtotal by 62.1%, 77%, and 61.9%). These significant reductions combined with the constraint that the semantics of the application remain exactly the same across all versions imply higher levels of reuse in the EJP versions. This increased level of reuse was made possible by the parameterization of EJPs. Second, true to our technique, in Health Watcher any oblivious exception handling aspects were converted to use EJPs, increasing lines of code due to EJP references.

The total Number of Operations for the EJP versions increased by 4%, 2%, and 6% compared to the original versions. The EJP versions were consistently better than the the AspectJ versions, with differences of -6%, -8%, and -5%.

The increase for the AspectJ and EJP versions vs the original versions are an expected byproduct of the refactoring, due to inlined code in `catch` blocks being extracted to advice.

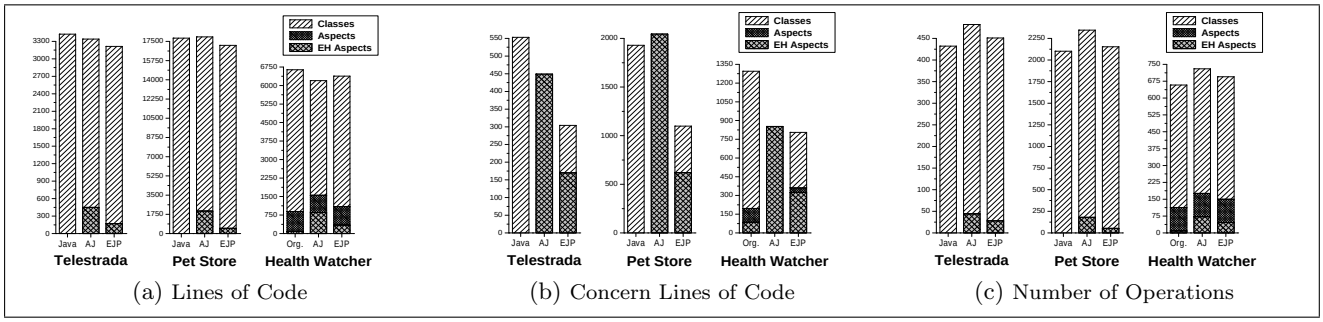
The improvement of the EJP versions vs the AspectJ versions was caused by (a) the higher level of reuse of exception handling logic (there were 36.4%, 71%, and 36.1% fewer handler operations for the EJP version), and (b) the use of scoped EJPs avoiding the creation of new methods to expose new join points (causing improvements of 2% to 3%).

4.3 Separation of Concerns Metrics

Figure 4 gives the results for the Separation of Concerns metrics. The results here are as expected, and clearly show that the use of Explicit Join Points does not facilitate pure obliviousness, as commonly defined by traditional AOP. We argue instead that EJPs provide a lesser level of obliviousness, feature obliviousness [26], and that these metrics are not reflective of this lesser level of obliviousness.

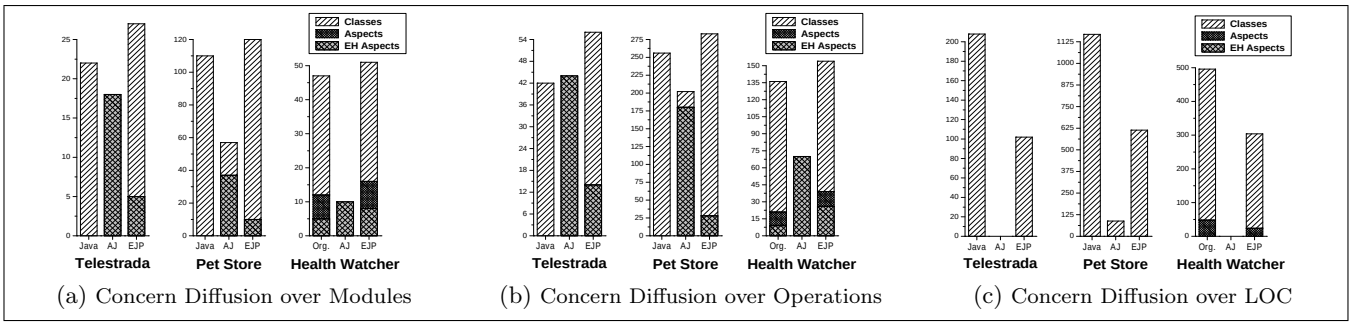
Concern Diffusion over Modules (CDoM) measures the number of modules that contain exception handling logic or a reference to a method or EJP that implements such logic. As all handler logic in the base code has been replaced with EJP references and then new aspects were added to implement the EJPs, it is expected that this metric differed by +23%, +9.1%, and +8.5% between the EJP and original versions. Additionally, the AspectJ versions showed an improvement over the EJP versions by 33%, 53%, and 80%.

Similarly and for the same reasons, the results for the Concern Diffusion over Operations (CDoO) differed by +33%, +11%, and +13% between the EJP and original versions and by +27%, +40%, and +120% between the EJP and AspectJ versions. Note that considering only handler aspects



(a) Lines of Code (b) Concern Lines of Code (c) Number of Operations

Figure 3: Results for size metrics (lower is better)



(a) Concern Diffusion over Modules (b) Concern Diffusion over Operations (c) Concern Diffusion over LOC

Figure 4: Results for separation of concerns metrics (lower is better)

for both the CDoM metric and the CDoO metric the differences between the EJP versions and the AspectJ versions actually decreased significantly, with the differences ranging from -20% to -72% for CDoM and -63% to -84% for CDoO.

The Concern Diffusion over Lines of Code (CDoLOC) metric counts the number of concern switches (between exception handling and some other concern). The differences between the EJP versions and the original versions were -51%, -47%, and -39%. The AspectJ versions showed an improvement over the EJP versions by 100%, 86%, and 100%.

In most modules 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 allowed a decrease greater than 50%.

While it is clear that the exception handling concern is still present in the base code, this presence has been greatly reduced, in fact to the point where only the existence of the concern and minimal contextual information remain. (Note that the size of the presence is not reflected in CDoM and CDoO.) In cases where required contextual information is not exposed by any one join point, we argue that it is less burdensome for the base code to explicitly provide it, rather than having the aspect reconstruct it (as discussed in [26]).

This decrease in pure obliviousness trades for benefits along other important dimensions, as discussed in this paper. Additionally, the level of traditional AOP obliviousness may not be desirable if the concern must constrain the base code (e.g. introduce new checked exceptions). EJPs instead provide feature obliviousness, as aspects implementing EJPs are oblivious to base code referring to the EJPs they match. This kind of obliviousness facilitates higher aspect modularity (i.e. decreased aspect–base code coupling) and less fragile pointcuts. Finally, adding EJPs as a language feature does not preclude using traditional, oblivious aspects, but rather adds flexibility within engineering tradeoffs.

5. QUALITATIVE RESULTS

Herein we present qualitative lessons learned, discussing safety, reusability, comprehensibility, and extendibility.

5.1 Safety

In the AspectJ version when exception handlers were moved to aspects any checked exceptions handled by the aspects had to be softened (wrapped by the AspectJ compiler in an unchecked exception) so that the base code could compile. When implementing exception handling aspects in AspectJ a tradeoff between generality and safety must be made.

On the one hand, very specific pointcuts can be created that precisely capture the exact method being advised (including both the method name and signature). These same pointcuts can be used to both soften exceptions and advise base code, ensuring that if the base code changes and handler advice no longer applies to a method, then exceptions are no longer softened, causing a compile error and preserving exception safety. However, this strategy creates a tight coupling between pointcuts and base code, requiring that as base code changes related pointcuts must change also, becoming a maintenance burden for the aspect programmers.

On the other hand, pointcuts can be more general and try to advise categories of exception handling scenarios (for example, whenever method A is called and throws exception B, throw exception C instead). This has the advantage that pointcuts are less coupled to the base code.However, if in an attempt to be more general the exception softening declarations are not precise (e.g. softening `Exception` instead of a specific subclass) or the pointcuts indicating when to soften exceptions are different from those indicating when to advise exceptions, then a checked exception can be softened by but not handled by an aspect. This hazard can be avoided by using more precise pointcuts, but the price is an increased coupling between the aspect and the base code.

With EJPs, this tradeoff between generality and safety is not required. The signature of each EJP explicitly states both which exceptions must be handled by *some* aspect advising the EJP and also indicates which checked exceptions could be thrown by implementers of the EJP. In this way the EJP models both the promises to and the requirements of code that references the EJP. The combination of generics and EJPs allow for parameterized exception handling that is both generic and safe. Examples EJP declarations are given in Figure 5.[3] EJPs like these provide for full checked exception safety while also allowing pointcuts to advise general exception handling patterns based solely on the EJP name.

```
scoped joinpoint <H>   printAndRethrow(String msg);
scoped joinpoint <H>   ignoreException() handles H;
scoped joinpoint <H,T> convertException()
                                handles H throws T;
```

Figure 5: Pseudo-code of example EJP declarations for generic exception handling

5.2 Reusability

Reusability is an important goal in well engineered systems, and was one of the original motivations of using aspects to implement exception handling [21]. 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. [8]

In contrast, the explicit presence of EJPs in the base code allow for parameterization of the handler logic implemented by aspects, allowing context and exception types to be explicitly communicated, as shown in Figure 5. The generic exception handling EJPs in our study were heavily reused, accounting for 53%, 71%, and 90% of all EJP references.

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.

Our study also shows supporting evidence for the feasibility of creating "EJP interfaces" that applications could depend upon even for complex concerns like exception handling. This is possible with AspectJ, but only for simple concerns need to advise single join points (not arbitrary blocks of code), unless you refactor methods to expose new join points. However, doing so decreases cohesion in the system and complicates tangling issues when aspectizing other cross-cutting concerns.[8] The EJP strategy sidesteps these issues, enhancing the power and modularity of generic aspect interfaces and libraries (such as exception handling or transactions). Even if a common set of EJPs cannot be agreed upon between two different libraries for the same concern, EJPs make it easier to write an adapter from one to the other, as the number of EJPs to adapt is relatively small.

Along these lines, EJPs can be viewed as a mechanism to explicitly model "feature contracts" in a feature oblivious system [26]. The principle of information hiding [22] can be invoked here, guiding EJP design so that hidden design decisions do not impact the EJP signature.Conversely, EJPs to explicitly expose points of extendibility, attaching specific semantics to EJPs and thereby allowing "plugin aspects" to flexibly modify or extend the behavior of base code. Similar concepts, such as extension points in Eclipse, have already been used in architectures with good results [3].

In summary, our experience has shown that EJPs enhance reusability for aspectized exception handling and that EJPs facilitate more powerful aspect interfaces and libraries.

5.3 Comprehensibility and Readability

This section discusses the tradeoffs between comprehensibility and readability in base code for each approach.

The Java approach provides good comprehensibility with weak readability, as handler logic is usually inlined within `catch` and `finally` blocks. This makes it easy to comprehend exactly how exceptional conditions will (or will not) be handled. However, it does distract the reader with unnecessary specifics about the handling of an exception. Also, in certain cases the handling concern was especially tangled (reminiscent of error handling in languages without exceptions), reducing readability and making refactoring more difficult.

In stark contrast, AspectJ provides good readability but weaker comprehensibility. The oblivious implementation of exception handling makes the primary functionality of the base code easy to read and comprehend. However, comprehending the details of exception handling logic becomes more complicated. Tools such as AJDT[4] assist in comprehending how join points in base code are advised. However, the overall big picture is less clear because the scope of a join point may not match handler scope. Also, as fragments are extracted into new methods readability is reduced. [8]

Finally, the EJP version provides a compromise between the two approaches. On the one hand, EJP references in the base code clearly express error handling semantics (i.e. logically attached to the EJP name) and the scope to which those semantics apply, providing good comprehensibility. On the other hand, the EJP implementation details are completely separate and not visible in the base code, providing

[3]The generic type variable declarations must actually follow the form `T extends Throwable` but are omitted for space.

[4]http://www.eclipse.org/ajdt/

good readability. Code implementing EJPs are free from application specific logic, increasing comprehensibility.

6. EXTENDIBILITY STUDY

One of the central themes of aspect-oriented programming is Filman and Friedman's oft-quoted statement: "Just program like always, and we'll be able to add the aspects later." [9] To gain insight into how EJPs affect the ability to extend functionality in this way (whether it becomes easier or more difficult), we performed an experiment where we enhanced the AspectJ and EJP versions of Java Pet Store to handle an additional exception-handling related concern. Herein we describe the new concern that was chosen and our techniques in implementing the new concern completely obliviously.[5] We then compare the two implementations, using metric measurements as material for discussion.

6.1 New Concern Definition

We postulated that because the exception handling logic had been separated from the main application it should be easier to change and enhance handler logic. One such enhancement could be integrating the application with a fault analysis engine [10], which would be desirable for an enterprise application. While the details of fault analysis and fault healing are beyond the scope of this paper, herein we can view a fault analysis engine as an interface that accepts information about thrown and handled exceptions. In addition to the exception itself, these engines require contextual information to classify faults and build fault models.

We thus defined our additional concern to be for the application to provide information on all handled exceptions and their context to this fault analysis engine interface. The contextual information was defined to include any log messages generated by the handler and also a flag indicating whether the handled exception would be rethrown. (Providing this information can facilitate insight into how the application reacted to the fault, when it was finally handled, and whether any action by an administrator might be required.) We note that this additional concern was not conceptualized until after all refactoring of the exception handlers had been finished, so that our initial refactoring would not be influenced by the additional concern.

6.2 Implementation Techniques

The new concern was implemented for both the AspectJ version and the EJP version in a purely oblivious fashion solely by creating new aspects to advise the handler base code. The new concerns requirement for fault contextual information required the new aspects to classify handler blocks as to whether or not a handled exception would be rethrown.

However, this classification was difficult because it is not possible to lexically pick out handler blocks within certain advice, even based on the types of the advice arguments, due to the limitations of AspectJ's `withincode` and `adviceexecution` primitive pointcuts (in the former case, it does not pick out advice execution, while in the latter case there is no pattern matching ability). Our workaround strategies were to either (a) pick out exception handlers and calls to print functions within advice based on the use of a Throwable object of a particular subtype within a target handler aspect, or (b) advise the same pointcuts as the advice in the handler base code with appropriate precedence so that thrown exceptions could be intercepted with first priority.When implementing the aspect for the EJP version we used an additional strategy where the aspect remembered the name of the last EJP that was exited (normally or via a thrown exception, on a per-thread basis), which simplified the code that classified printed messages in handlers. We applied our strategies with the goal that the resulting code be as simple as possible.

[5]To clarify the discussion, we refer to the aspects containing the exception handling logic as the handler base code.

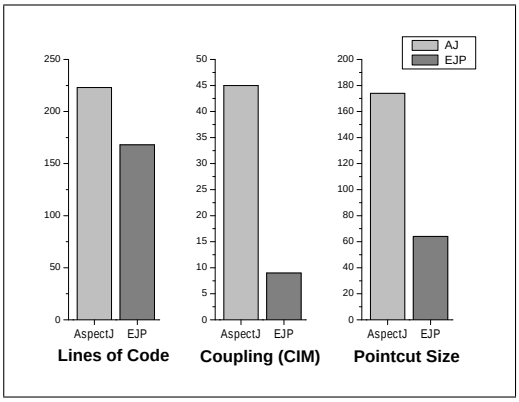


Figure 6: Metrics results for an aspect implementing a new concern in Java Pet Store (lower is better)

6.3 Results

Figure 6 show the results of the Lines of Code, Coupling on Intercepted Modules, and Pointcut Size metrics. Pointcut Size is defined as the total number of terms (primitive pointcuts or pointcut references) within all pointcuts, modeling potential pointcut complexity and fragility.

The oblivious aspect to implement the concern for the EJP version required 25% less Lines of Code (LOC), and its implementation primarily consisted of code to special case handler EJPs that conditionally handled exceptions based on a generic type parameter. In contrast, the implementation for the AspectJ version consisted mostly of complex pointcuts that picked out the different classifications of exception handlers for each exception handling aspect.

If additional functionality were added to Java Pet Store, we expect that the LOC for the new aspect for the AspectJ version would increase much faster than for the EJP version because the amount of code required in the AspectJ version is nearly directly related to the number of handler aspects. In contrast, the LOC for the new aspect in the EJP version were mainly used to implement advice for the generic exception handling EJPs, so additional code would only be needed for any new custom exception handling EJPs. In Java Pet Store, only 2% of all EJP references were due to custom exception handling EJPs, so it is expected that the number of new custom handlers required would be very low.

For the Coupling on Intercepted Modules metric, the aspect for the EJP version had 80% less dependent modules in its pointcuts. This was caused by the relatively few number of handler EJPs that had to be advised to implement the additional concern, whereas with the AspectJ version every custom Pet Store handler aspect had to be explicitly named, tightly coupling the new aspect to the handler base code.

Finally, Pointcut Size for the EJP version was 63% less than for the AspectJ version. Pointcuts for the AspectJ version were more complex because custom pointcuts were required for each handler aspect in order to determine whether an exception would be rethrown or not. For example, in one handler aspect all `catch` blocks would not rethrow the exception whereas in another aspect some handlers would rethrow and others would not. These detailed pointcuts in the AspectJ version are fragile and induce coupling because the above assumptions for a handler aspect could change as it is modified. In contrast, the EJP version pointcuts were structured around EJP names and would be robust against all but significant changes in their targeted EJP.

For both versions the aspects implementing the new concern are completely oblivious and have nearly identical Separation of Concerns metric values (the only significant difference being that the EJP version required two more operations to handle the concern). The results were very similar for the other metric values as well (CBM, LCO, NoO). The largest difference was that the EJP version had 25 NoO instead of 21 (two of those operations were generic).

Based on these results, we assert that using EJPs to implement cross-cutting concerns empowers oblivious implementation of unforeseen functionality. By giving up a certain amount of obliviousness up front through EJPs, oblivious programming in the future is more powerful and robust.

7. RELATED WORK

The relationship of obliviousness within aspect-oriented design, its definition, benefits, and drawbacks, are discussed in [26], and specific challenges of a design oblivious approach are detailed. They introduce crosscutting programming interfaces (XPIs) based on the principle of information hiding. [22] These interfaces represent design rules that constrain the structure of the base code, allowing stable pointcuts to be written solely on the structure provided by the design rules agreed upon a priori. Our work can be considered complementary, as we highlight with specific empirical evidence drawbacks to a design oblivious approach. EJPs provide a similar level of obliviousness and can be viewed as one possible mechanism to explicitly model XPIs, and additionally provide means to enforce semantic constraints upon base code and to advise arbitrary blocks of code.

In [14] the authors introduce a different kind of pointcut interface. They argue that aspects should not be tangled with the implementation details of classes, and thus pointcuts should be defined within the class itself. The named pointcuts, aggregated into pointcut interfaces, serve as contracts between the aspects and the base code, facilitating comprehensibility and independent development. EJP's pointcut arguments serve as pointcut interfaces, but provide a finer degree of granularity, and also allow the base code to expose local variables in arguments passed to the pointcuts. Additionally, they allow for scoping information finer than method boundaries. EJPs are complementary to pointcut interfaces, being appropriate where a semantic coupling between aspect and base code is needed to enforce safety, whereas pointcut interfaces provide more obliviousness and are appropriate where such coupling is not needed. We advise that both constructs be provided to facilitate well-engineered AspectJ programs.

Aldrich builds on the work of [14] by proposing *open modules* [2], defining the concept of a "package" that explicitly exports functions and pointcuts (representing internal events within that module), which become the only advisable join points. A formal model was developed within a limited AOP language that proved that the semantics of a package could be preserved even when its internal implementation changes. Whereas open modules allow for the hiding of implementation details from aspects, EJPs allow for the base code to specify advisable join points tied to specific semantics as defined by the programmer. With open modules either a join point is advisable by aspects or it is not, whereas EJP give you more flexibility and finer granularity in choosing *which* join points to expose and how far (via *scoped* EJPs and pointcut arguments). Also, with open modules you specify which join points are advisable statically, whereas using EJPs you can specify this information using dynamic context (via scoped EJPs and the `cflow` designator).

In [18], Mezini and Kiczales recognize the need to program against cross-cutting interfaces. The approach proposed can be viewed as a "reverse engineering" approach, where aspects' dependences on a system's join points are computed, and shown as annotations on the explicit interfaces of advised code.

8. CONCLUSIONS

We have proposed extending AspectJ in a fully backwards compatible manner with the explicit join point (EJP) construct, which facilitates semantically coupled cross-cutting concerns while only minimally reducing obliviousness. We have shown how EJPs facilitate finer granularity in how and where advice should be applied and have quantified these benefits through the case of exception handling.

Our implementation of EJPs, which includes more advanced features omitted in this paper for brevity, is based on an extension to the `abc` for AspectJ, and is freely available to encourage future collaboration, evaluation, and experimentation.

9. REFERENCES

[1] abc Project. *abc. The AspectBench Compiler.* `http://aspectbench.org`.

[2] J. Aldrich. Open Modules: Modular reasoning about advice. In *ECOOP'05*, pages 144–168, 2005.

[3] D. Birsan. On plug-ins and extensible architectures. *Queue*, 3(2):40–46, 2005.

[4] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena. Composing design patterns: a scalability study of aspect-oriented

programming. In *AOSD'06*, pages 109–121, 2006.

[5] M. Ceccato and P. Tonella. Measuring the effects of software aspectization. In *1st Workshop on Aspect Reverse Engineering*, Delft, The Netherlands, 2004.

[6] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[7] C. Clifton and G. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. 2003.

[8] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranho, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *FSE'06*, pages 152–162, 2006.

[9] R. Filman and D. Friedman. *Aspect-Oriented Programming Is Quantification and Obliviousness*, pages 21–35. Addison-Wesley, 2005.

[10] M. Fuad, D. Deb, and M. Oudshoorn. Adding self-healing capabilities into legacy object oriented application. In *ICAS'06*, pages 51–51, 2006.

[11] A. Garcia, C. Sant'Anna, C. Chavez, V. T. da Silva, C. J. de Lucena, and A. von Staa. *Software Engineering for Multi-Agent Systems II*, chapter Separation of Concerns in Multi-agent Systems: An Empirical Study, pages 49–72. Springer Berlin / Heidelberg, 2004.

[12] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD'05*, pages 3–14, 2005.

[13] S. Goldman, D. Detlefs, S. Dever, and K. Russell. New compiler optimizations in the java hotspot virtual machine. In *JavaOne 2006*, 2006.

[14] S. Gudmundson and G. Kiczales. Addressing practical software development issues in AspectJ with a pointcut interface. In *Workshop on Advanced Separation of Concerns of ECOOP'01*, 2001.

[15] K. Hoffman and P. Eugster. *EJP extension to The AspectBench Compiler*. `http://www.cs.purdue.edu/homes/kjhoffma/`.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *ECOOP'01*, pages 327–353, 2001.

[17] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, pages 220–242, 1997.

[18] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE'05*, pages 49–58, 2005.

[19] J. Kienzle and R. Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *ECOOP'02*, pages 37–61, 2002.

[20] S. Kuzins. Efficient implementation of around-advice for the AspectBench compiler. Master's thesis, Oxford University, 2004.

[21] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE'00*, pages 418–427, New York, NY, USA, 2000. ACM Press.

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

[23] C. N. Sant'Anna, A. F. Garcia, C. von Flach Garcia Chavez, C. J. P. de Lucena, and A. von Staa. On the reuse and maintenance of Aspect-Oriented software: An assessment framework. In *17th Brazilian Symposium on Software Engineering*, pages 19–34, October 2003.

[24] M. Stochmialek. *aopmetrics*. `http://aopmetrics.tigris.org/`.

[25] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM'05*, pages 653–656, 2005.

[26] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *FSE'05*, pages 166–175, 2005.

[27] P. Tonella and M. Ceccato. Refactoring the aspectizable interfaces: an empirical assessment. *IEEE Transactions on Software Engineering*, 31(10):819–832, Oct. 2005.

[28] J. Zhao. Measuring coupling in Aspect-Oriented systems. In *METRICS'04*, 2004.