Technical Report EJP-200712-1
Cooperative Aspect-Oriented Programming

Kevin Hoffman Patrick Eugster

Purdue University
305 N. University Street
West Lafayette, IN 47907

Abstract

Aspect-oriented programming (AOP) seeks to improve software modularity via the separation of cross-cutting concerns. AOP proponents often advocate a development strategy where programmers write the main application (base code), ignoring cross-cutting concerns, and then aspect programmers, domain experts in their specific concerns, weave in the logic for these more specialized cross-cutting concerns. This purely oblivious strategy, however, has empirically been shown to tightly couple aspects to base code in many cases, hindering aspect modularity and reuse. In essence, the more intricate the weaving between the cross-cutting concern and the base code (lexically and/or semantically), the harder it becomes to: (a) robustly specify how to weave the aspects in at the required points, (b) capture interactions between aspects and base code, and (c) preserve the correct weaving as the base code evolves.

We propose an alternate methodology, termed *cooperative aspect-oriented programming* (Co-AOP), where complete lexical separation of concerns is not taken as an absolute requirement. Instead, cross-cutting concerns are explicitly modeled as abstract interfaces through *explicit join points* (EJPs). Programmers specify where these interfaces interact with base code either through explicit lexical references or via traditional oblivious aspects. This explicit awareness allows base code and aspects to cooperate in ways that were previously not possible: arbitrary blocks of code can be advised, advice can be explicitly parameterized, base code can guide aspects in where to apply advice, and aspects can statically enforce new constraints upon the base code that they advise. These new techniques allow aspect modularity and program safety to increase, and bring us towards a cooperative AOP paradigm.

We illustrate our methodology via an example on transactions, and also give an initial evaluation of cooperative AOP through an empirical study on program extensibility comparing both the traditional and cooperative AOP methodologies. Initial results show that cooperative AOP techniques result in code that is less complex with lower overall coupling, facilitating extensibility.

Key words: Aspect-oriented programming, extensibility, join point, modularity

1 Introduction

The principles of *aspect-oriented programming* (AOP) [34] are becoming increasingly popular in software development. As languages, tools, and development environments providing AOP have begun to mature, increasing numbers of projects are considering using AOP. Java programmers are offered the benefits of AOP through AspectJ [33], the most prominent of all aspect languages.

The lure of AOP comes from its promise to help alleviate one of the most challenging facets of software development – the detangling of code through the separation of *cross-cutting concerns*. Beyond increasing modularity due to this separation, additional promises include increased programmer specialization, increased parallel development, and improved application debugging, introspection, and reconfiguration. Disregarding cross-cutting concerns is an attractive approach in that it allows the use of domain-specific experts to implement cross-cutting concerns. Indeed, this notion of complete obliviousness is often taken as a defining tenant of AOP where one would always say, "Just program like always, and we'll be able to add the aspects later [15]."

However, the adoption of AOP is not as widespread as might be expected given these promises. While many factors certainly affect the adoption of a new methodology and its associated languages [12], the slow adoption rate is one indicator that there may be "strings" attached to these promises in that AOP brings with it a new set of challenges that have not yet been resolved. There is an increasing amount of research that details these challenges, their underlying causes, and potential solutions [2,11,14,26,35,36,45,46].

Specifically, the desirability and even feasibility of absolute obliviousness as outlined above has recently been questioned. Oblivious programming strategies have been shown to cause quantification-related problems [46], reduce aspect modularity [14,30], and cause safety and feasibility problems [36].

To address these challenges authors have proposed ways in which to plan for, restrict, infer, or document aspects' interaction with base code [2,26,35,46]. With the exception of [35] these proposals argue for a reduced level of obliviousness and a shift in design strategy where the presence of aspects are planned for a priori and aspect and base code programmers actively cooperate. However, these proposals do not fundamentally increase the power of the AOP

Email addresses: `kjhoffma@cs.purdue.edu` (Kevin Hoffman), `peugster@cs.purdue.edu` (Patrick Eugster).

models on which they are built, but rather seek to adjust, guide, or constrain systems such that modularity is improved.

Contributions. In this paper we present the case that modularity and safety are improved when aspects and base code can communicate *explicitly.* By removing the premise that *all* base code should be oblivious to aspects and by allowing for explicit interaction instead through *explicit join points* (EJPs), we show how the above challenges are addressed and that the power of AspectJ is fundamentally increased. By providing mechanisms for base code and aspects to explicitly work together in new and interesting ways we empower a new *cooperative aspect-oriented programming* (Co-AOP) paradigm.

This paper presents the full explicit join point concept, the ramifications of their advanced features, and EJP's impact on AspectJ and AOP for the general case. More precisely this work uniquely contributes the following: [1]

- A discussion on the design principles backing the EJP extensions, giving insights into challenges faced by oblivious design as expressed through AspectJ.
- The fully developed EJP language extensions with corresponding syntax and an exploration of their advanced features, including pointcut arguments and policy enforcement. We show how these advanced features fundamentally increase the power of AspectJ for AOP.
- A running example showing the fundamental and practical benefits of EJPs with all their advanced features by using EJPs to implement the transactions cross-cutting concern.
- The architecture of the EJP extension to the AspectBench Compiler for AspectJ [3].
- An empirical study wherein we compare two versions of the known Java Pet Store application — one based on AspectJ and one based on EJPs — in order to evaluate oblivious extensibility. Both versions were extended by adding a new exception-monitoring concern after exception handling had already been aspectized. The study shows that EJPs increase extensibility of aspect code with respect to pure AspectJ.

Roadmap. Section 2 presents background and discusses related work. Section 3 discusses design principles leading up to our language extensions. Section 4 introduces explicit join points and show how they can implement transactions, and Section 5 presents advanced EJP features, including pointcut parameters and policy enforcement. Section 6 details the implementation of EJPs within the `abc` compiler. Section 7 presents our aspect extensibility empirical study, and Section 8 summarizes our work and concludes.

[1] A prior version of this paper appeared in [29], which contained shortened versions of the first three contributions.

2 Background

This section first overviews AspectJ and then we briefly survey related work highlighting challenges to aspect-oriented software development and previously proposed solutions, setting the stage for the contributions of the paper.

2.1 AspectJ fundamentals

AOP seeks to improve modularity through two well-known concepts termed *quantification* and *obliviousness*: aspects make quantified statements over programs about where and how new logic should be injected, and then these quantified statements are applied obliviously to programs. [15] These two concepts are so fundamental that some authors accept the assertion that AOP *is* quantification and obliviousness.

AspectJ [33] was the first industrial-strength AOP language and remains the most prominent and popular AOP extension of Java to date. Every language extension that is part of AspectJ can be classified as either directly supporting quantification or obliviousness, and the language itself was designed under the assumption of absolute obliviousness. In AspectJ obliviousness dictates that code to implement cross-cutting concerns is completely separated from the code implementing the primary concern (termed the *base code*) and is moved into *aspects*. Key structural elements of the base code, such as method calls and field accesses, are exposed as *join points*, representing the points at which logic to implement cross-cutting concerns can be woven in. A quantification language is defined to allow for collections of join points, termed *pointcuts*, to be defined in terms of syntactic patterns, types, and even dynamic state and program control flow. Logic to implement cross-cutting concerns is placed within *advice*, which then use pointcuts to inject this logic *before*, *after*, or *around* join points. Advice and pointcuts are grouped into *aspects*, the basic reusable unit in AspectJ. Aspects are similar in appearance to classes and can inherit from classes or other aspects, whose lifetimes are singletons or tied to the lifetime of entities in the base code.

AspectJ has developed quite significantly since its inception. It has added generics-related features introduced in Java 5, and also merged with the AspectWerkz project [5] to gain new annotation-style syntax and load-time weaving capabilities. Compiler efficiency has also improved greatly, in relation to both the time required to perform aspect weaving and the runtime efficiency of the generated Java bytecode. Enhancements for popular development environments to facilitate AOP have also become available and have matured, providing advanced features such as aspect visualization to aid aspect programmers. All of these development efforts have made AspectJ a viable and robust platform for application development.

2.2 Related work

While the benefits of AOP and AspectJ are many and have been clearly demonstrated in the literature, researchers have also highlighted significant challenges faced by AOP as modeled within AspectJ. Herein we present those challenges and proposed solutions that are most relevant to our work.

Case studies based on empirical software quality metrics have begun to emerge [8,14,30,19,18,46,48]. One empirical case study [14] explored how metrics were affected when exception handling was implemented using AspectJ instead of Java. While certain aspects of software quality were improved (viz. separation of concerns), the overall system complexity increased, cohesion decreased, and the modularity of the aspectized exception handling code was much less than anticipated, showing low levels of handler code reuse.

One reason code reuse was limited was that the exception handling code was slightly different for each case (e.g., error messages, logging behavior, etc.). Thus, even though relatively few exception handling patterns were exhibited in the code and could be captured by a collection of pointcuts, the desired behavior at each join point was slightly different. Because AspectJ has no mechanism for pointcuts to communicate explicit parameter values (values separate and apart from those exhibited in the base code, such as a string representing an error message) they were forced to use separate advice for each slightly customized behavior, unnecessarily increasing system complexity. This is akin to writing a new method several times over instead of adding a new parameter. These particular issues are explored in more detail in [30].

The issues involved in obliviously using aspects to add transactional semantics to base code were studied in [36]. This particular cross-cutting concern is relatively complex and requires intricate interaction with program state exposed over many distinct join points. The authors found significant challenges in placing the transactionalizing logic solely within aspects, noting that transaction scope, compensation, and isolation level are better determined by the base code in most cases. Also, implementing this concern raises correctness concerns, as there is no way for aspects to enforce constraints upon the base code being advised. For example, certain actions executed within a transaction might require explicit compensation if the transaction is rolled back, and the transactionalizing aspect has no mechanism of statically enforcing that these actions do indeed have compensation associated with them. (The transactionalizing aspect could *specify* in its design that some other aspect would provide compensation logic, but would have no means of *enforcing* that specification.)

In [26] the authors introduce the idea of a pointcut interface, proposing that pointcuts be defined in a hierarchical manner, moving complex pointcut definitions closer to the base code they match against, even defining certain pointcuts in classes. While this helps to separate aspects from implementation details, the modularity of the pointcut interface is limited because aspects cannot use other abstract aspect's pointcuts, as discussed in Section 3.6. This limitation implies that pointcuts that are reuseable between aspects must be defined within a single aspect, and this single aspect becomes directly coupled to all base code it advises.

In [46] Sullivan et al. demonstrate deficiencies in modularity due to an oblivious development methodology, using a large OO system (HyperCast) as a case study. They propose that base code be structured according to design rules contained within *cross-cutting programming interfaces* (XPIs), extending the ideas of [26]. Pointcuts are then written according to patterns specified by the design rules, and in this way pointcuts are made more robust. In their follow-on work [25] they describe how to represent XPIs using AspectJ pointcut descriptors. However, there is no mechanism to ensure that base code conforms to the design rules, and this technique also suffers from the same modularity issues as [26] for the same reasons.

Aldrich takes a different approach by proposing *open modules* [2]. Aspects are only allowed to advise the join points explicitly exported by a module. If a module carefully defines visible join points to avoid internal implementation details, the resulting pointcuts in aspects are robust to future changes in that module. While this technique prevents aspects from writing fragile pointcuts it does so by removing aspects' power to advise fragile points, which might be necessary to implement a cross-cutting concern, which, as was demonstrated in [46], is often the case.

A similar approach to improving modularity is taken in [35], but instead of relying on modules explicitly declaring aspect interfaces these aspect interfaces are automatically generated. Changes to base code can be compared against these generated interfaces to discover unwanted side effects. While this helps mitigate unwanted changes in how aspects advise base code, it does not actually reduce coupling nor increase pointcut robustness.

The general direction of prior research is to involve aspects early in the design process and shape base code to accommodate them. We argue that if obliviousness is already lessened to this degree at this higher level then the step to introduce explicit communication between base code and aspects is not a big one, especially if these explicit constructs are only used where necessary and with care. We show herein that by allowing programmers the possibility of explicitly cooperating with aspects the power of AspectJ is increased and that aspect modularity and safety can be increased.

3 Motivation and design

In this section we discuss design principles underlying the development of our EJP extensions. We briefly overview previous ways in which cross-cutting concerns are implemented in Java and AspectJ, explaining specific weaknesses of each approach. We visualize these approaches and also introduce a running example based on transactions. Later we use this pedagogical framework to clarify the presentation of explicit join points.

3.1 Visualization framework

Figures 1-3 abstractly depict the structure, control flow, and data flow for the different approaches to implement cross-cutting concerns. The labeled circles represent fields, formal parameters, or local variables. For example, in Figure 1, class C contains four fields (A, B, C, and D), and method A has 3 formals (E, F, and G). The numbered boxes represent statements – method call or field access – and thus represent advisable join points. The labeled circles inside each statement represent the variables used within that statement that are accessible to pointcuts within AspectJ. Control flow between blocks of code or methods is depicted by arrows. A double-lined arrow represents the control flow after an exception is raised.

3.2 Inlined approach

Figure 1 depicts how the code would be structured in an inlined approach, where code to implement a cross-cutting concern is written in place as needed. The use of variables A, B, F, and G within the cross-cutting concern depict that it shares data with the base code and that data may be exchanged. Variables C and D are used exclusively to implement the cross-cutting concern. Note that the cross-cutting concern may trigger exceptions that must be handled.

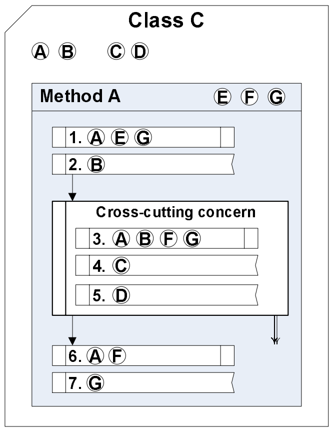


Fig. 1. Inlined approach: the cross-cutting concern is integrated at the "call-site"

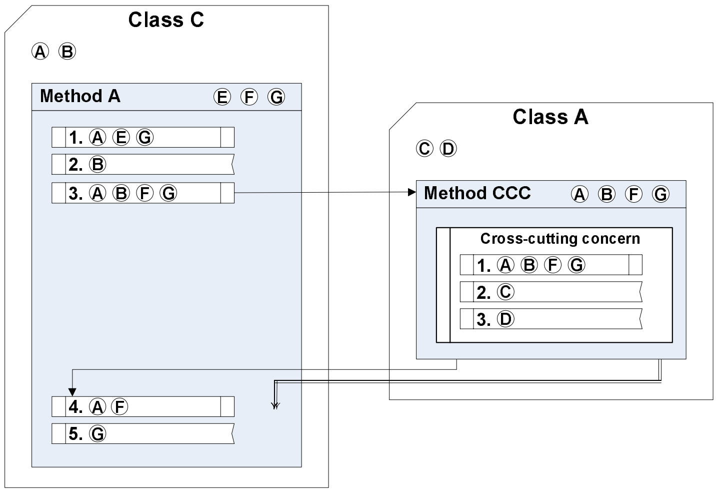


Fig. 2. Refactoring approach: the cross-cutting concern is moved to another class

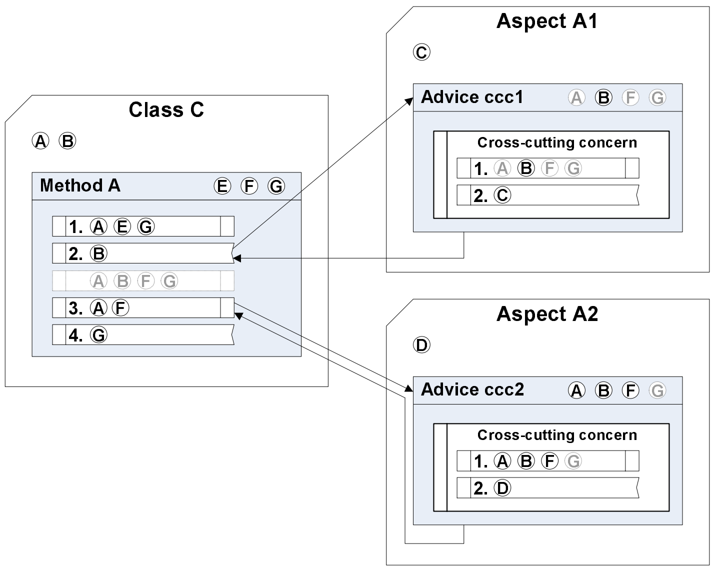


Fig. 3. AspectJ approach: the cross-cutting concern is enclosed in an aspect

3.3 Refactoring approach

Inlining the full implementation of cross-cutting concerns wherever they're needed produces redundant and tangled code. Thus, in Figure 2 we refactor the cross-cutting concern into another class. Any data within base code required by the cross-cutting concern's implementation is passed as an argument to the implementation method in class A. This approach is functionally identical, reduces redundancy, and preserves constraints. However, the base code is still tied to the specific implementation of the cross-cutting concern. Class C must specifically instantiate or store a reference to class A, and if the implementation was changed or refactored the base code would probably need to be modified. In this way the base code is unnecessarily coupled to the cross-cutting concern, hindering modularity and flexibility.

3.4 AspectJ approach

Eliminating these issues is the main motivation behind AOP and AspectJ. Figure 3 depicts how the concern would be implemented with AspectJ. The base code is completely oblivious to the aspect, depicted by the removal of the explicit call to the other class. This allows for the details of the implementation of the cross-cutting concern to change freely without concerning the base code. In this figure we depict the possibility that the cross-cutting concern be implemented by two aspects. The details of how the implementation was factored into two aspects is not important here – just that it is possible for more than one aspect to implement the cross-cutting concern, and that the aspect(s) may require all or some of the state that the non-AspectJ implementation would have required.

Note that in this instance the concern (and thus the aspect) requires specific state from the base code. Because AspectJ is limited to either advising a single statement or an entire method body, it is not possible for a *single* pointcut to capture all of the state required for proper execution of the cross-cutting logic. If the aspect advised the entire execution of method A then it would have access to variables B (through `target`), E, F, and G, but it would not be able to inject the cross-cutting code at the proper location, nor could it access the new value of variable G if statement 1 in Figure 3 changed variable G (and G was a scalar). If the aspect chose to weave logic immediately after statement 2, it would only be able to access variable B. Similarly, if the aspect wove logic immediately before statement 3 it would only have access to variables A, B, and F. In this way advice in AspectJ have limited access to state within base code.

Additionally, aspects that need to inject logic at very specific points within a method have to use very detailed (and therefore fragile) pointcut patterns. For example, for aspect A2 to inject logic at the desired location, it uses a pointcut similar to `withincode(* C.A(..)) && call(mpattern)` where `mpattern` matches the method call at statement 3. The pointcut expression would have to be even more specific and complex if the method called by statement 3 was called elsewhere within method A – the pointcut expression would then have to pick out the specific location by argument type or value. If the method calls had identical arguments (or no argument) it would not even be possible to match the exact location, and the programmer would probably choose to refactor the base code to accommodate the aspect. Thus, in addition to state accessibility issues, implementing cross-cutting concerns in this fashion aggravates the *fragile pointcut problem* due to unilateral pattern matching in pointcuts, tightly coupling aspects to base code and hindering base code modification or refactorization [45].

Another problem with this approach is that the aspect can no longer directly affect the control flow of the base code beyond the scope of a single join point. For example, an aspect cannot safely introduce exceptions into the base code and require during compilation that these exceptions be handled. The aspect could throw a soft exception and specify that some other aspect catch these soft exceptions, but there are no compile time constraints available to ensure that this actually happens. Advice can use a `throws` clause to indicate the advice can throw a checked exception, but it is not allowed to throw any checked exception that cannot already be thrown by the join point it is advising (so new checked exception types cannot actually be introduced by aspects). For cross-cutting concerns with semantics requiring proper error handling this lack of static enforcement is undesirable and is a safety hazard, and we address this issue with our language extensions as demonstrated later.

3.5 Transactions example

To ground our presentation of EJPs in practice, we introduce a running example showing pseudo-code to implement the transactions cross-cutting concern. Since different flavors of transactions exist [47], and extending programming languages with specific support is expensive in many senses, aspects seem like a promising solution.

Listing 1 shows pseudo-code for a method representing business logic we would like to transactionalize. The `Person`, `Flight`, and `Hotel` objects represent transactionalizable objects (or send commands to a database in a manner compatible with the transactionalizing mechanisms). The `reserveSeat` and `reserveRoom` method calls represent business logic within the transaction

```
class Agent {
  CardProcessor cc = ...;
  void createTrip(Person p, Flight f, Hotel h) {
    ...
    f.reserveSeat(p);
    h.reserveRoom(p);
    cc.debit(p.getCC(), ...);
  }
}
```

Listing 1. Pseudo-code of business logic in example

that may cause exceptional control flow (e.g., the flight or hotel is booked out). The `debit` operation on the `CardProcessor` object represents an action that cannot be rolled back automatically, requiring explicit compensation [7]. Other examples include interaction with legacy systems that are incompatible with the transactionalizing mechanisms, physical or network device (I/O), etc. [27]. [2]

Using AspectJ to implement transactions and related mechanisms is an area of active interest in research and industry [6,20,38,49]. In this discussion we focus on the fundamental difficulties in using AspectJ to implement transactions rather than focusing on the details of any one system. The actual implementation could employ optimistic or pessimistic concurrency control, be based on databases, objects, or enterprise APIs, be centralized or distributed, but the issues discussed herein are applicable to any such system.

The general strategy in applying transactions using AspectJ is to use `around` advice to wrap transactionalizing logic around method calls that should be executed with transactional semantics or that affect the state within the transaction. The aspect is responsible for creating or referencing a transaction context object to manage the state for the transaction (typically one context is required per top level transaction). Examples of how to do this with AspectJ can be found in [38] and in [36].

The pseudo-code showing this strategy (abstracted away from the details of any particular system) is depicted in Listing 2. Not shown are concrete aspects that would specify pointcut expressions indicating where transactions should begin (e.g., to match the call to `Agent.createTrip`). Not considering exceptional conditions and potential state–point separation issues, the AspectJ approach works well — the business logic is free from the implementation details of transactions and different aspects could be written to implement transactions utilizing different mechanisms, allowing weave-time adaptation of the business logic according to need and context.

[2] Recently compensation is becoming more widespread by being part of the specification for transactions in Web Services [31].

```
abstract aspect TransactionImpl {
  abstract pointcut startTranIsoLevel0();
  ...
  abstract pointcut startTranIsoLevel3();
  void around() : startTranIsoLevel0() {
    TransContext t = (...).getContext(0);
    t.beginTrans();
    proceed();
    t.commitTrans();
  }
  ...
  void around() : startTranIsoLevel3() {
    TransContext t = (...).getContext(3);
    ...
  }
  after() throwing : startTranIsoLevel0()
      || ... || startTranIsoLevel3()
  {
    TransContext t = (...).getActiveContext();
    t.abortTrans();
  }
  declare soft: TranException: within(TransactionImpl)
}
// Now declare aspects that derive from TransactionImpl
```

Listing 2. Pseudo-code of transactionalizing mechanisms with AspectJ

3.6 Challenges

However, the possibility of failure introduces important problems. If a transaction fails then the error must be handled – a correct program cannot ignore the failure and continue with a rolled back state of the system, especially if the rollback is incomplete [13] and/or compensation is required for certain actions. Since aspects cannot require new checked exception types to be handled (exceptions must be softened instead) a safety hazard is introduced.

Additionally, the modularity and flexibility of the transaction cross-cutting concern is restricted in the following ways:

(1) While the actual implementation is within an abstract aspect, concrete aspects must be used to override the abstract pointcuts and describe every join point in the program where a transaction should be opened. When the concrete aspects with the pointcuts are specified they have to explicitly indicate the base aspect they are deriving from, tying the pointcut definitions to the actual implementation technique that should be used. If one wants to try a different implementation aspect, all of the concrete aspects have to be changed so they inherit from the new aspect.
    Also, because these concrete aspects are separate from and yet tightly coupled to the base code, as the base code changes these pointcuts have to be maintained. Each cross-cutting concern adds a potentially large amount of new pointcuts to be maintained, limiting the scale of the technique and reducing overall modularity. In these ways the subclasses of

the abstract aspect are tightly coupled both to the abstract aspect itself (the transaction implementation technique) and to the base code being advised (through their pointcut descriptors). These limitations on aspect modularity have been observed in empirical case studies [14,46].

(2) Because the concrete aspects cannot specify the isolation level to use through a parameter in the pointcuts they specify (as pointcut arguments can only be populated via state from base code) a separate abstract pointcut must be used for each possible isolation level. Alternatively a separate concrete aspect could be used along with a field within the aspect to indicate the isolation level, but either way the number of pointcuts or concrete aspects required grows multiplicatively as the number of parameters affecting the advice increases. In this example the total number of possibilities was only 4, but in other practical scenarios the total number of distinct parameter values required could become unmanageable.

(3) AspectJ does not allow aspects to advise abstract pointcuts in other aspects. This means that if one wanted to implement another concern (e.g., a concern monitoring the timing *and* source location of transactions) then the aspect implementing the new concern could not reuse the pointcut definitions from the concrete subclasses of the `TransactionImpl` abstract aspect – all of the pointcuts would have to be specified again (observe that it could not simply advise method calls to the `TransContext` class because the concern also wants to record the source locations in the base code where the transactions are starting or ending). This greatly increases the difficulty of adding new concerns that need to advise similar join points to existing concerns and is in contrast to the stated goals of AOP. Classpects [41] have been proposed to address this challenge by removing any distinction between class and aspect, thus removing anonymity for advice.

These arguments are further substantiated in an empirical case study involving EJPs and exception handling [30].

4 Explicit join points

In this section we present *explicit join points* (EJPs), illustrating how they compare and contrast to the techniques presented in the previous section. The running example of implementing transactions via aspects is continued, showing one example of how EJPs are effective in practice.

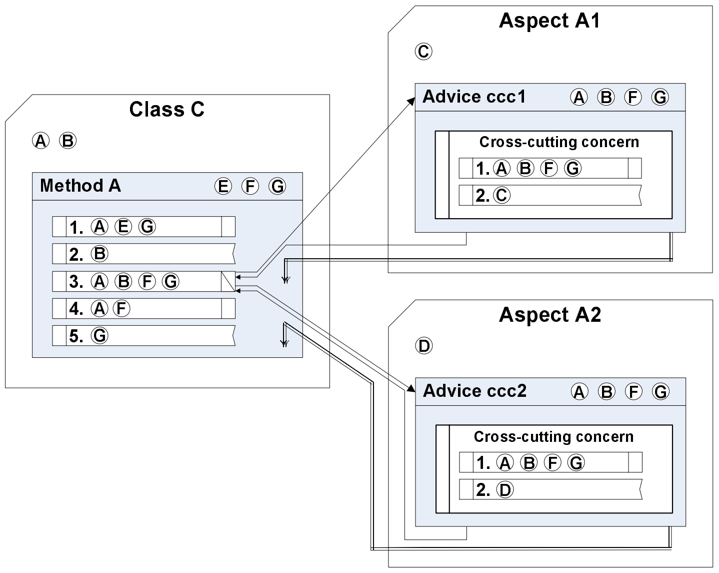


Fig. 4. AspectJ with explict join points approach: the base code makes allowance for aspects to advise it

4.1 Simple explicit join points

By removing from our language design the assumption that all base code will be oblivious to aspects, we are now free to introduce explicit communication mechanisms. One simple communication mechanism would be for base code to invoke methods within aspects. However, while this is actually possible in AspectJ (as all aspects are also classes), this wouldn't be aspect-oriented because base code would be invoking logic in aspects (object-oriented) instead of aspects weaving logic into base code (aspect-oriented).

Instead, we define the *simplest* EJP declaration to have the same form as a method declaration within an interface, having an identifier, return value, formals, and a throws list (but no method body). We do allow specification of a default return value to be used if no aspect advises the EJP. The base code then references these EJP declarations using syntax similar to static method invocation. A new pointcut modifier, `ejp`, is defined (used within the `call` pointcut) to pattern match EJPs within base code.

Figure 8 overviews syntax for the fully fledged EJPs and previews the advanced EJP features that we introduce in the sections below, while Figure 5 defines our syntactic extensions to AspectJ, and Figure 6 defines those made to Java, both in syntax following the conventions used in the Java language specification [22]. Refer to Figure 9 for examples of pointcuts that match against EJPs.

Figure 4 shows the structure, data flow, and control flow in a program designed using the EJP pattern. Because we have reintroduced explicitness into the base code, aspect(s) are able to access all information required by the cross-cutting concern. The `throws` list of the EJP signature ensures during compilation that the base code handles any exceptions that could be raised (or alternatively that aspects handle these exceptions and soften them using `declare soft`).

The EJP serves as an explicit representation of an abstract contract between the base code and aspects, modeling the information required by the cross-cutting concern and also any constraints to be enforced upon base code wishing to be advised. In contrast to method invocation, however, the target class is not predefined, and the base code is not coupled to any specific implementation. [3] At a minimum the EJP designer should informally specify any high-level pre- and post- conditions and the semantics of the EJP's formals. Additionally, the EJP designer could attach some abstract promise of functionality to EJP declarations. In accordance with the *information hiding* design principle [39], the contract that the EJP represents should be as abstract as possible so that concern implementations hide their implementation details, thus facilitating future enhancement (via changing, composing, or recomposing aspects) without requiring the base code to be modified.

4.2 Scoped explicit join points

The EJPs described in the above form represent a change in design methodology more than a new or novel language construct – the simplest EJPs can be used today without extending AspectJ by modeling EJPs as static methods. However, the increased power and real benefits of EJPs come as we add new language features that bring aspect-awareness to base code. As we allow base code to use aspect-oriented concepts directly we approach a paradigm shift towards *cooperative aspect-oriented programming*, where the distinction between base code and aspects is blurred, and the scenario where aspects are advising other aspects in complex ways becomes much more common. Indeed, one of the issues limiting the scalability of AspectJ with respect to the number of cross-cutting concerns is that it is difficult for aspects of one cross-cutting concern to advise aspects implementing another cross-cutting concern, due to the anonymous nature of advice and the unilateral advising model in AspectJ.

The need for aspects to robustly advise other aspects is not limited to theory, but becomes more necessary as system complexity increases. For example, a fault analysis engine would arguably need to advise code *within* exception handling aspects, not just advise the join points that the exception handling aspects advise. This is similar to arguing that following a strict layering approach (where a layer can only communicate with its immediately neighboring

[3] Using the *command* design pattern [17] would reduce coupling vs. method invocation, but would not be as effective as aspects.

ExplicitJoinPointDeclaration *:*

  ExplicitJointPointHeader [***FormalPointCutParameters***] [***Handles***]

    ↪ [*Throws*] [***ExplicitJoinPointInitializer***]

ExplicitJointPointHeader *:*

  [*Modifiers*] [***scoped***] ***ExplicitJointPointDeclarator***

ExplicitJointPointDeclarator *:*

  joinpoint *TypeArguments ResultType Identifier* (*FormalParamList*)

FormalPointCutParameters *:*

  pointcutargs ***FormalPointCutParameterList***

FormalPointCutParameterList *:*

  FormalPointCutParameter

  FormalPointCutParameter , ***FormalPointCutParameterList***

FormalPointCutParameter *:*

  Identifier ([*FormalParameterList*])

Handles *:*

  handles *ClassTypeList*

ExplicitJoinPointInitializer *:*

  = *Expression*

MethodConstructorPattern:

  ...

  thisclass

  thismethod

  thisblock

  ExplicitJoinPointPattern

  ExplicitJoinPointScopePattern

ExplicitJoinPointPattern *:*

  ejp (*MethodPattern*)

  ! ***ejp*** (*MethodPattern*)

ExplicitJoinPointScopePattern *:*

  ejpscope (*MethodPattern*)

  ! ***ejpscope*** (*MethodPattern*)

DeclareDeclaration:

  ...

  declare error ***nomatch*** ***NoMatchGranularity*** : *PointcutExpression* :

    ↪ *PointcutExpression* : *StringConstant*

NoMatchGranularity *:*

  by package

  by class

  by method

  by ejp

Fig. 5. Syntax added to AspectJ; conventions are that of the Java Language Specification [23]: production names are italicized, new productions are bolded, new keywords are italicized and bolded, optional expressions are bracketed

ExplicitJoinPointExpression:
    ExplicitJoinPoint [***PointcutArguments***] [*Block*];
ExplicitJoinPoint:
    AspectName . [*NonWildcardTypeArguments*] *Identifier* [*Arguments*]
PointcutArguments:
    `pointcutargs` ***PointcutArgumentList***
PointcutArgumentList:
    PointcutArgument
    PointcutArgument, PointcutArgumentList
PointcutArgument:
    PointcutName ([*FormalParamList*]) : *PointcutExpression*

Fig. 6. Syntax added to Java

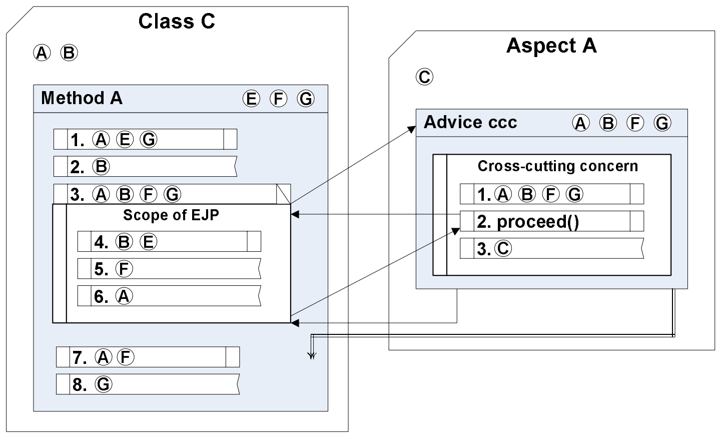


Fig. 7. Advising arbitrary code blocks using scoped explicit join points

layers) is not always practical or desirable. We seek to empower aspect-oriented programming to better address these cases.

The first new language feature we add is a `scoped` modifier for EJP declarations, which when specified allows base code to attach a block of code to an EJP reference using syntax similar to anonymous class instantiation. We add the `ejpscope` pointcut modifier, which matches executions of blocks of code attached to EJPs. This allows arbitrary *blocks* of code to be advised, as depicted in Figure 7. Refer to Figures 8 and 9 for related syntax.

It is worth noting how scoped EJPs are advantageous over a well-known technique in the AspectJ community – refactoring blocks of code into new methods in order to advise that block of code. The refactoring approach has two significant drawbacks:

(1) It has been shown empirically that refactoring methods to expose new join points decreases object cohesion and artificially increases system complexity [14] (as measured by the well-founded metrics of that study).

(2) The negative effects multiply as additional cross-cutting concerns advise base code. The base code becomes increasingly splintered as differing concerns require the advising of different blocks of code and more and more state is passed as arguments through these "intermediate methods." Additionally, if these refactored blocks of code change multiple local variable values existing in an outer scope the lack of pass by reference parameters further complicates the signatures of these methods. Each concern added to the system makes refactoring increasingly difficult [14], hindering methodologies such as Extreme Programming [4].

These effects are in direct contrast to one of the foremost goals of AOP – to make it easy to adapt, evolve, and refactor (via the separation of cross-cutting concerns).

In contrast, scoped EJPs do not require the introduction of new methods, preserving cohesion and avoiding the muddling of the primary decomposition of the program. With scoped EJPs, the `cflow` construct is more powerful and less fragile, as now explicit meaning can be associated with a single scoped EJP as opposed to having to attach meaning to a list of method names using a pointcut. Scoped EJPs also scale better in the presence of multiple cross-cutting concerns – because the blocks of code being advised remain inlined in their original methods and can access and modify local variables in all visible outer scopes, method signatures can remain unchanged and concerns can be added or removed simply by wrapping or unwrapping a block of code within a scope. Each concern is orthogonal to the other in relation to its effect on the base code, as code can freely reach across multiple scopes without additional effort. This also has the side effect of making the base code more readable and comprehensible, as all significant cross-cutting semantics are visible via the EJPs. Without EJPs in order to comprehend these semantics complex pointcuts and multistage advice would need to be referenced and understood. While these arguments are qualitative, these benefits have also been shown empirically in related work [30].

4.3 Handles list

Now that aspects can advise arbitrary blocks of code we can introduce new and interesting ways in which aspects and base code can interact. One such way is facilitated by the addition of a new `handles` list to an EJP declaration. Listing checked exception types in the `handles` list means that for each type in the list *some* aspect (at least one) will advise that point joint and handle that exception type.[4] In essence, it is a promise to the base code that aspects

[4] This can be enforced during compilation when base code and aspects are woven all at once.

```
aspect ExampleAspect {
  // Syntax for EJP declarations:
  [JavaModifiers] [scoped] joinpoint
    ["<" GenericTypeParameters ">"]
    ReturnType EJPName "(" Formals ")"
      [pointcutargs "(" DefPCArg ["," DefPCArg]* ")"]
      [handles ...]
      [throws ...]
      ["=" DefaultValueExpr] ";"
  // Syntax of DefPCArg:
  ArgName "(" Formals ")"
}
class ExampleClass {
  void exampleMethod() {
    // Syntax for using EJPs in base code:
    AspectName"."[TypeArgs]EJPName "("Args")" [pointcutargs PCArg [","PCArg]*]]";"
    // Syntax for using scoped EJPs in base code:
    AspectName"."[TypeArgs]EJPName "("Args")" [pointcutargs PCArg [","PCArg]*]
    "{"
      // block of code to be advised
    "}" ";"
    // Syntax of TypeArgs:
    "<" GenericTypeArg ["," GenericTypeArg]* ">"
    // Syntax of PCArg:
    ArgName "(" Formals ")" ":" Pointcut
  }
}
```

Fig. 8. Overview of explicit join point syntax: keywords are bolded, 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; `ExampleAspect`, `ExampleClass`, and `exampleMethod` demonstrate the required context for the specified syntax, but are not part of the syntax specification

```
aspect ExampleAspect {
  // Examples of ejp and ejpscope pointcut modifiers:
  pointcut p1(...): call(ejp(EJPName)) && args(...);
  pointcut p2(): cflow(call(ejpscope(ScopedEJPName)));
}
```

Fig. 9. Examples of how to match explicit join points in pointcuts

will handle certain exceptions (the base code need not be concerned with how and where the exceptions are handled – only that they are). In the base code referencing a scoped EJP with a `handles` list has the same effect on checked exceptions as if that block of code were associated with catch blocks for the checked exceptions types in the `handles` list.

4.4 Generics and explicit join points

One of the significant enhancements in AspectJ 5 over previous versions is that it integrates Java 5 generics into AOP related constructs. Combining generics with EJPs in a similar fashion complements the cooperative AOP approach in that it allows aspects to become more type-agnostic while allowing base code

```
// Declaration of EJPs promising to provide common exception handling patterns.
aspect CommonExceptionHandlers {
  // When an exception of type T is caught, prints and then rethrows the exception.
  public scoped joinpoint <T extends Exception>
      void printAndRethrow(String msg, Class<T> handleType) throws T;

  // When an exception of type T is caught, sets
  // the value of the first entry in outValue to value.
  public scoped joinpoint <T extends Throwable>
      void onExceptionSetValue(Class<T> handleType, Object[] outValue, Object value)
        handles T;
}
```

Listing 3. One possible benefit from combining generics with EJPs; common exception handling patterns are modeled with EJPs while preserving exception safety

```
interface CompensationRecord {
  void compensate();
  void log(...);
}

abstract aspect TranConcern {
  scoped joinpoint void enterTrans(int isolationLevel)
          pointcutargs ignoreableCompensations()
          throws TranException;
  joinpoint int defaultIsolationLevel() = 0;
  joinpoint void addCompensation(CompensationRecord r);
}
```

Listing 4. Explicit join point interface defining the transaction cross-cutting concern

```
class Agent {
  CardProcessor cc = ...;
  void createTrip(Person p, Flight f, Hotel h) throws TranException {
    int level = TranConcern.defaultIsolationLevel();
    TranConcern.enterTrans(level) {
      f.reserveSeat(p);
      h.reserveRoom(p);
      cc.debit(p.getCC(), ...);
      // The TranConcern.addCompensation EJP would
      // be referenced here because of the cc.debit call
    };
  }
}
```

Listing 5. Pseudo-code of business logic with EJPs

to specify concrete types for increased type safety. In combination with scoped EJPs and the `handles` clause, EJPs can be defined that represent generic exception handling *patterns*, such as "catch, log, and rethrow" or "catch and throw a different exception." Listing 3 presents example code for a few such patterns. Additional details on these patterns and their benefits can be found in [30].

```
aspect TransactionImpl {
  pointcut tranScopes(int isolationLevel):
       call(ejpscope(TranConcern.enterTrans)) && args(isolationLevel);

  // A transaction begins and ends at the outermost
  // instance in the control flow of the enterTrans EJP
  void around(int isolationLevel) throws TranException:
        tranScopes(isolationLevel) && !cflowbelow(tranScopes(..))
  {
    TransContext t = ...;
    t.beginTrans();
    proceed();
    t.commitTrans();
  }

  after() throwing throws TranException: tranScopes(..) {
    TransContext t = (...).getActiveContext();
    if (t != null) t.abortTrans();
  }

  // Some aspect could advise the addCompensation EJP
}
```

Listing 6. Pseudo-code of aspect implementing transaction concern with EJPs

4.5 Transactions with explicit join points example

With our new language features we are now better equipped to effectively implement the transaction cross-cutting concern. Listings 4, 5, 6, and 7 show the implementation using EJPs, in contrast to the AspectJ implementation shown in Listings 1 and 2. Note that any compensation related constructs are irrelevant in this section.

Observe that the EJP implementation does not suffer from the downfalls of the AspectJ method as discussed in Section 3.5. First of all, the `enterTrans` EJP declaration on lines 7–9 of Listing 4 specifies that references to that EJP must handle that checked exception (e.g., see Listing 5, line 4) and that aspects advising that EJP are allowed to throw a checked exception of type `TranException`. Whereas in the AspectJ implementation the aspect `TransactionImpl` had to soften `TranException` (and had no guarantee this exception would be handled), advice can now declare the `TranException` type in their throws list without a compiler error without requiring softening. Instead of adding the `TranException` type to the throws list in the base code an aspect could be used to implement the exception handling logic for that exception. Listing 7 presents such an aspect that uses a robust pointbut based on the EJP name. Note that in the EJP case without an exception handling aspect (or another mechanism to handle the `TranException` checked exception) a compiler error is raised, whereas with the AspectJ method the original exception being thrown is softened, so no compiler error is raised if the exception is not handled in some way.

```
aspect HandleTransactionExceptions {
  pointcut tranScopes():
      call(ejpscope(TranConcern.enterTrans));

  declare soft(): TranException : tranScopes();
  after() throwing(TranException): tranScopes() {
    ...
  }
}
```

Listing 7. Pseudo-code of an error handling aspect that employs a robust pointcut matching an explicit join point

Also observe that the implementation aspects are concrete and no additional pointcuts or aspects need to be defined now or in the future – the pointcuts are robust, in contrast to the pointcuts required by the AspectJ implementation.

Usually base code is not concerned with the isolation level, so we defined the `defaultIsolationLevel` EJP (Listing 4, line 10). This allows aspects to customize the isolation level in the general case and then for the base code to directly override it as desired (see Listing 5, line 5).

Additionally, the implementation is robust to changes in how transactions are implemented. To use a different aspect instead of `TransactionImpl`, the only step would be to define the new aspect and the one pointcut that refers to the EJP – no pointcuts need to be changed or duplicated and no changes need to be made to aspect inheritance, in contrast to the AspectJ technique.

Another important benefit of the EJP approach is that the advice can be customized on a per join-point basis (through the formals of the EJP), as shown by the `isolationLevel` parameter in this example (Listing 6, lines 2, 8, and 9). Whereas the AspectJ technique required four nearly identical advice to model the four different possible isolation levels the EJP technique could model this with just one. (While the advice could be refactored somewhat to eliminate redundancy, this is hindered by the need to ultimately call `proceed`, although `proceed` can actually be captured within a closure.) Although the parameters are specified in the base code in the example (and as discussed in [36] it is often desirable to do so) they could have just as easily been specified by aspects, as discussed next.

To preserve obliviousness in the base code Listing 8 demonstrates how aspects could be used to "reference" EJPs at appropriate places within the base code. However, we warn that this technique only shifts the pointcut coupling seen in the AspectJ technique to a different set of aspects, and advocate instead that in most cases base code should directly reference EJPs. However, this technique would help AspectJ code begin using EJPs with minimal change while allowing new code to reap the full benefits of EJPs. Without the notion of a scoped

```
abstract aspect ReferenceTransEJP {
  abstract pointcut startTranIsoLevel0();
  ...
  abstract pointcut startTranIsoLevel3();
  void around() : startTranIsoLevel0() {
    TranConcern.enterTrans(0) {
      return proceed();
    };
  }
  ...
  declare soft: TranException: within(ReferenceTransEJP)
}
```

Listing 8. Pseudo-code of using aspects to indirectly reference EJPs in base code

EJP, this new level of indirection that facilitates pointcut robustness would not be possible.

Finally, the use of EJPs (whether they are referenced directly in base code or indirectly through aspects) facilitate adding new concerns that advise the same join points being advised by existing aspects. To return to our running example, an aspect implementing the monitoring of timing and source location of transactions would only require one pointcut (to match the `enterTrans` EJP), in contrast to the AspectJ method that requires the respecification of all join points in the base code where transactions begin and end.

5 Advanced Explicit Join Point Features

In this section two additional features of EJPs are introduced: *pointcut arguments* and *policy enforcement*. These features facilitate the cooperative aspect-oriented programming methodology and provide mechanisms to check for errors of omission in new and existing code.

5.1 Pointcut arguments

The explicit presence of AOP features in base code allow us to introduce a new construct for cooperative AOP – *pointcut arguments*. When declaring an EJP optionally one can also specify a set of pointcut arguments to be attached to the EJP. Each pointcut argument attached to an EJP declaration can be thought of as declaring an *abstract* pointcut whose full name is `EJPName.ArgName`. These pointcut arguments have the same semantics as regular pointcuts can be used exactly the same way as other pointcuts can; however, they are not defined at the point where they are declared.

Instead of deriving aspects to specify the concrete definitions for these abstract pointcuts, base code is given the opportunity to specify these pointcuts *incrementally* in the following way: References to EJPs in base code optionally use the `pointcutargs` keyword (see Figure 8). In this context the list of pointcuts following the `pointcutargs` keyword add to the definition of the named pointcut arguments. The pointcut arguments in each EJP declaration are fully defined by the disjunction of all of their corresponding pointcut patterns within the base code that references them.

Pointcut arguments allow the base code to use the power of pointcuts to tell aspects additional places they should (or should not) advise. This is advantageous in scenarios where the base code has a better understanding than aspects of which join points should be advised. If pointcut fragility cannot be avoided then at least the definition of the fragile pointcut can be moved as close as possible to the code upon which it is tightly coupled. We define new pointcut modifiers `thisclass`, `thismethod`, and `thisblock`, applicable only within pointcut patterns within base code (referring to the containing class, method, or EJP block), to mitigate pointcut fragility by avoiding references to explicit typenames.

To illustrate pointcut arguments we continue our transactions example. Listing 4 shows how explicit compensation might be modeled in the EJP interface. The base code would reference the EJP to record actions requiring manual compensation upon rollback, as denoted in Listing 5. Note that an aspect could be used to reference the EJP obliviously if desired but might face state–point separation challenges.

Sometimes code may need to ignore compensation because it is compensated as part of a larger transaction. In Listing 4 we add the pointcut argument `enterTrans.ignoreableCompensations`, which describes compensations that should be ignored. The pointcut modeled by this pointcut argument is incrementally defined as base code references the `enterTrans` EJP. Through pointcut arguments, base code specifies additional join points where compensation should be ignored.

Listing 9 gives an example where all compensation is ignored within transactions started by the EJP block in the `testTrans` method. The following is an example pointcut that uses the pointcut argument defined in Listing 4:

```
pointcut newRecords(): call(ejp(*.addCompensation))
                       && !enterTrans.ignoreableCompensations();
```

In this particular case, the `ignoreableCompensations` pointcut argument of the `enterTrans` EJP represents all those join points where new compensation actions (references to the `addCompensation` EJP) should be ignored. The `newRecords` pointcut is therefore matching all EJP references that are adding

```
class Testing {
  void testTrans() {
    TranConcern.enterTrans(...)              // EJP reference
      pointcutargs ignorableCompensations():  // Add to the pointcutarg definition.
        (call(ejp(*.addCompensation)) &&
         cflow(call(thisblock))
    {
      // compensation is now ignored in cflow of this block
    };
  }
}
```

Listing 9. Base code adding to a pointcut argument

new compensation actions, except for those locations where compensation should be ignored as specified by the pointcut argument.

Pointcut arguments are one of the distinguishing features of EJPs that combine the quantification benefits of AOP with the explicit style of OO programming. When applied judiciously, EJPs with pointcut arguments improve modularity by allowing the base code to shape how aspects are woven in to the base code, especially when the aspect needs to interact with the base code in precise and intricate ways. EJPs are distinguished from other techniques such as feature-oriented programming [40] and mixin layers [43] because they provide the rich and characteristic quantification model of AOP within the base code through pointcut arguments. In this way, pointcut arguments facilitate the cooperative AOP paradigm.

5.2 Policy enforcement

The use of EJPs to implement a cross-cutting concern shifts the potential for errors of omission from the aspects to the base code. An approach using AspectJ without EJPs relies on the correctness of the patterns within aspects' pointcuts to be correctly written and complete initially, and then as the base code evolves these patterns must be updated and checked for correctness and completeness. Whereas static analysis tools such as [45] help developers ensure correctness, they are less effective at ensuring completeness as new code is written (since a static analysis tool does not know whether a new piece of code must match certain pointcuts in other aspects for correct program behavior). Consequently, some programmer must review the new code and any related aspects to ensure all relevant aspects apply where needed.

Likewise, with the EJP approach, programmers must also review new code to ensure that the relevant aspects are applied where needed (in this case by explicitly referencing the appropriate EJPs). However, once the new code has been written and the application tested for correctness, the resulting application is much more resilient as the software evolves. Whereas without EJPs there is the potential for pointcuts to inadvertently start or stop matching evolving code, with the EJP approach a programmer must explicitly delete or insert a reference to an EJP, making it less likely for an error to be inadvertently introduced.

In both approaches as new code is added there is the potential for aspects not to properly be applied to the new code (either because of incomplete pointcut patterns or missing EJP references in the base code), even though these aspects should be applied in order to implement cross-cutting concerns in the new code. While AspectJ has the `declare error` construct to catch errors of *commission* (when certain things happen but they shouldn't) at compile time, there is no corresponding mechanism to catch errors of *omission*, such as the type of errors described above. To help mitigate this issue, we introduce another advanced feature complementing EJPs, *policy enforcement* mechanisms.

We extend the `declare error` construct in AspectJ via the `nomatch` keyword so that errors are generated when a pointcut expression does *not* match any join points within a given lexical scope (package, class, method, or EJP block). Consider the following example that ensures certain EJP blocks contains certain EJP references:

```
declare error nomatch by ejp:          // Check each EJP block that...
  call(* CardProcessor.*):             // contains these join points
  call(ejp(*.addCompensation(..))):    // (what to require)
  "missing compensation!";             // (error message)
```

The above example illustrates how policy enforcement can be used to ensure that any scoped EJP containing method calls on a `CardProcessor` object should also contain a reference to the `addCompensation` EJP. In this way policies can be written that ensure that base case does not "forget" to reference EJPs that are part of a larger protocol.

6 Implementation

We implemented EJPs in AspectJ by extending the AspectBench research compiler (`abc`) [1,3]. 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 [28] and distribution under the GNU Lesser General Public License (LGPL).

Due to `abc`'s effective design for extensibility, we were able to implement EJPs via subclassing and new classes. Support for non-scoped EJPs was straight-

```
class Agent {
  CardProcessor cc = ...;

  void createTrip(Person p, Flight f, Hotel h) throws TranException, ... {
    int level = TranConcern.defaultIsolationLevel();
    TranConcern.enterTrans(level);

    class ejpScopedInner1 {
      Person p;
      Flight f;
      Hotel h;
      /* field not created for cc, because it was not a local or formal in block */
      final void JPTRANS$npxw33$8$25(int ejpvar1) {
        f.reserveSeat(p);
        f.reserveRoom(p);
        cc.debit(p.getCC(), ...);
      }
    }

    final ejpScopedInner1 local_ejpScopedInner1 = new ejpScopedInner1();
    local_ejpScopedInner1.p = p;
    local_ejpScopedInner1.f = f;
    local_ejpScopedInner1.h = h;
    local_ejpScopedInner1.JPTRANS$npxw33$8$25(level);
    p = local_ejpScopedInner1.p;
    f = local_ejpScopedInner1.f;
    h = local_ejpScopedInner1.h;
  }
}
```

Listing 10. Source representation of code after the EJP compiler transforms scoped EJP references in the `createTrip` method of Listing 5 (actual transformation is performed on the AST, not source code)

forward via abstract syntax tree rewriting and by extending the type system, while support for scoped EJPs proved to be more interesting. We needed all pointcut designators, including cflow, to efficiently advise scoped EJPs without creating restrictions on the type of code placed within a scoped EJP or restricting the degree of concurrency of the executing code. The use of `cflow` with scoped EJPs is especially interesting because it allows the programmer very fine granularity in specifying join points to advise, without cluttering the class interface unnecessarily.

Our approach lifts the code within each scoped EJP reference and places it into a new anonymous class, instantiated at the point of the EJP reference. To avoid modifying the underlying Java compiler to make our language extensions as unobtrusive as possible, and because Java only allows inner classes to reference final local variables, references to local variables or formals declared outside the scope of the EJP are converted into fields in the inner class. Code is automatically generated to instantiate a new instance of the anonymous class, populate its fields, call the method with the lifted code, and copy changed field values back to local variables. Listing 1 and Listing 10 illustrate how our technique transforms the `createTrip` method.

Note that because a unique anonymous class is created for each scoped EJP reference and because each execution of the scoped EJP reference results in the temporary instantiation of its associated anonymous class, our technique does not add any synchronization issues to the underlying code. Additionally, there is no code redundancy – the code inside the scoped EJP is moved into the anonymous class, but no other copies of that code are needed. Semantically, this implementation strategy is the same as if the block of code inside the scoped EJP was refactored into a new method; however, since this transformation is performed by the compiler during the compile process, this extra complexity is hidden from the developer, and the code within the scoped EJP can access formals and local variables as expected.

Our technique accomodates corner cases, placing no restriction on the type of code that can be contained within a scoped EJP. Around advice matching scoped EJPs that call `proceed` several times (or not at all) operate efficiently and correctly, as the around advice is only applied to the actual inner class method call, not to the surrounding support code. The AST rewriting was broken into several subpasses so that scoped EJPs nested within the same method are compiled, advised, and execute correctly.

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 Java Version Machines [21] (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 [37] could facilitate additional compile-time optimization.

The method pattern matcher was extended to recognize patterns that have the `ejp` and `ejpscope` modifier. The pattern matcher for the `ejpscope` modifier checks to see if the pattern matches the EJP associated with the scoped EJP instance instead of matching against the method of the inner class containing the lifted code.

Our non-invasive modification of `abc` allows the EJP compiler extension to be distributed as an extension `JAR` to the standard `abc` distribution.

7 Extensibility Study

One of the central themes of aspect-oriented programming is Filman and Friedman's oft-quoted statement: [15] "Just program like always, and we'll be able

Attributes	Metrics	Definition
Size	Lines of Code	Number of uniformly formatted lines of code, excluding whitespace and comments.
	Pointcut Size	Number of terms (primitive pointcuts or pointcut references) within all pointcuts.
	Number of Operations	Number of declared methods and advice.
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.
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

to add the aspects later." Intuitively one might expect EJPs to affect the ability to extend functionality in this oblivious way due to the additional explicit links between base and aspect code. To verify this, we performed an experiment involving the AspectJ and EJP versions of Java Pet Store [30,32] where the exception handling concern was already aspectized with AspectJ in [14] and for EJPs in [30]. We extended both versions of Java Pet Store to handle an *additional* cross-cutting concern that was not planned for nor anticipated when the exception handling concern was originally aspectized. Herein we present the new concern that was chosen and our techniques in implementing the new concern completely obliviously. To clarify the discussion, we refer to the aspects containing the exception handling logic as the handler base code. We then compare the two implementations, using metric measurements as material for discussion.

7.1 Study setting

Our target application is Java Pet Store [32], which is an e-commerce reference application written to demonstrate best practices for enterprise systems with respect both to Java technologies and to software engineering principles. The project consists of more than 340 classes and interfaces across 17,800 LOC.

We leverage and build on established techniques for empirically evaluating aspect-oriented software. Aspect-oriented specific metrics, building on well-known OO empirical metrics [10], have been proposed in [9,42,50]. These metrics have recently been used in several empirical case studies focused on aspect-oriented software [8,14,30,18,19,24,48].

Table 1 summarizes the metrics we employed for this study. The size and Coupling Between Modules metrics are adaptations of the well-known CK metrics [10] modified to consider aspect-oriented language features [42]. The Coupling on Intercepted Modules [9] metric measures how tightly coupled an aspect's pointcuts are with respect to the code it is advising. We additionally define the Pointcut Size metric to evaluate the effort required to specify how to weave the aspects into the base code and also to measure the potential for pointcut fragility. We also employ the separation of concern metrics in [42]. We primarily used the AOPMETRICS tool [44] to automatically compute metric values for the source code.

7.2 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 [16], 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, which is useful to understand when faults are finally handled. 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. The interface modeling the requirements of our new cross-cutting concern is presented in Listing 11. 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 or EJPs when we implemented our additional concern.

7.3 Implementation techniques

The new concern was implemented in a purely oblivious fashion for both the AspectJ and EJP versions by creating new aspects to advise the handler base code. We analyzed the Java Pet Store application for common exception handling patterns and how these patterns could be mapped to the requirements

```
/** Interface would be implemented by some fault analysis engine to receive data.
 *  Used by the exception monitoring concern to provide fault data. */
public interface FaultAnalyzerDataSink {
  // Tells the fault analyzer about a message printed by an exception handler.
  public void recordExceptionMessage(String msg, boolean willBeRethrown);

  // Tells the fault analyzer about a thrown exception and intended response.
  public void recordException(Throwable e, boolean willBeRethrown);
}
```

Listing 11. Interface modeling the data that our new concern must provide to some fault analysis engine

```
/** Abstract aspect that feeds exceptions and related data to a fault analyzer. */
public abstract aspect ExceptionLoggingAspect {
  // Retrieves reference to fault analyzer object
  static FaultAnalyzerDataSink getFaultAnalyzer() {...}

  // Pointcuts that should be defined by concrete aspects to capture the actual
  // occurrences of these exception handling patterns in the application.

  // Pointcuts capturing exception handlers
  abstract pointcut handlerWillNotBeRethrown(Throwable e);
  abstract pointcut handlerWillBeRethrown(Throwable e);

  // Pointcuts capturing messages printed within exception handlers
  abstract pointcut msgExcWillNotBeRethrown(String msg);
  abstract pointcut msgExcWillBeRethrown(String msg);

  // Advice implementing the logic to provide data to
  // the fault analyzer for each of the known patterns.

  after(Throwable e): handlerWillNotBeRethrown(e) {
    getFaultAnalyzer().recordException(e, false);
  }

  after(Throwable e): handlerWillBeRethrown(e) {
    getFaultAnalyzer().recordException(e, true);
  }

  Object around(String msg): msgExcWillNotBeRethrown(msg) {
    getFaultAnalyzer().recordExceptionMessage(msg, false);
    // We do not call proceed because the message is provided
    // to the fault analyzer instead of being printed.
    return null;
  }

  Object around(String msg): msgExcWillBeRethrown(msg) {
    getFaultAnalyzer().recordExceptionMessage(msg, true);
    return null;
  }
}
```

Listing 12. Abstract aspect common to both versions that provides information to the fault analyzer for the most common exception handling patterns

```
public aspect AdminWebHandler {
  /* Note that the fault analysis aspect will need to capture the original
   * exception type (ServiceLocatorException). Since it can't capture the sle
   * variable in a pointcut, it will have to match on the call to getMessage. */
  after() throwing (ServiceLocatorException sle) throws AdminBDException :
      adminRequestBDHandler() || updateOrdersHandler() {
    sle.printStackTrace();
    throw new AdminBDException(sle.getMessage());
  }

  /* Note that the fault analysis aspect will need to distinguish between
   * this advice and the advice above (because this advice does not rethrow
   * the exception). Because advice are anonymous, the distinction is made
   * by type or how exceptions are used, which may break as this evolves. */
  String around(ApplRequestProcessor arp) : processRequestHandler() && target(arp){
    try {
      return proceed(arp);
    } catch (ParserConfigurationException pe) {
      return arp.replyHeader + "..." + pe.getMessage() + "...";
    }
  }
}
```

Listing 13. An example aspect from the exception handling concern in the AspectJ version highlighting some of the challenges in capturing the join points needed to implement the new exception monitoring concern

```
// Aspect implementing the new exception monitoring concern for the AspectJ version
public privileged aspect AJExceptionLoggingAspect extends ExceptionLoggingAspect {
  /* For advice that rethrow exceptions, we need to capture the original exception
   * type by matching join points within the advice that use the original exception
   * (because advice are anonymous in AspectJ and can't be explicitly named).
   * For example: 4 patterns of how Throwable objects are used within a handler. */
  pointcut callExcGetMessage(Throwable e): call(* *.getMessage(..)) && target(e);
  pointcut callExcToString(Throwable e): call(* *.toString(..)) && target(e);
  pointcut callExcGetArg1(Throwable e): call(Throwable+.new(..)) && args(e);
  pointcut callExcGetArg2(Throwable e): call(Throwable+.new(..)) && args(..,e);

  /* Special case for AdminWebHandler--not all of the aspect's handlers will rethrow
   * the exception, so we have to exclude them based on the type of the Exception.*/
  pointcut adminWebHandlerCase():
    target(ParserConfigurationException) || target(SAXException) || target(...);

  /* Capture originally thrown exception for advice that rethrow exceptions.
   *  Note that callExcGetMessage handles the special case for AdminWebHandler. */
  pointcut throwObjectFromTarget(Throwable e):
        (within(AdminWebHandler) && callExcGetMessage(e) && !adminWebHandlerCase())
    ||  (within(CatalogClientHandler) && callExcGetMessage(e))
    ||  (...) // 11 other similar explicit references to other modules
    ||  (within(WafViewTemplateHandler) && callExcToString(e));

  pointcut throwObjectFromArg1(Throwable e):
        (within(EJBExceptionGenericAspect) && callExcGetArg1(e))
    ||  (...); // 10 other similar explicit references to other modules

  pointcut throwObjectFromArg2(Throwable e):
        (...); // 3 explicit references to other modules
  ...
  // In total 16 concrete pointcuts couple this aspect to 45 other modules.
}
```

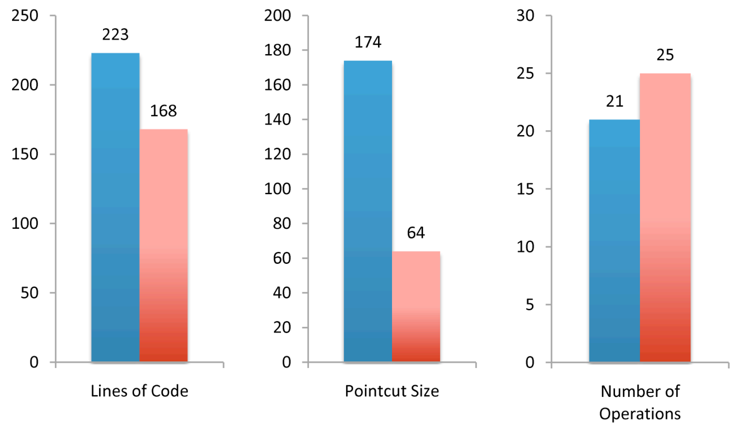
Listing 14. Pointcuts capturing a few of the join points for the exception monitoring concern for the AspectJ version; most pointcuts in this aspect match based on exception type within a specific class, as opposed to naming specific methods

```
public privileged aspect EJPExceptionLoggingAspect extends ExceptionLoggingAspect {
  ...

  /** Matches parameterized EJPs that will NOT rethrow the caught exception */
  pointcut excWNBR(Class excType):
       (execution(ejpScope(ExceptionUtil.ejpIgnoreExc)) && args(excType))
    || (execution(ejpScope(ExceptionUtil.ejpGetExcMessage)) && args(excType,..))
    || (execution(ejpScope(ExceptionUtil.ejpPrintExc)) && args(excType,..))
    || (execution(ejpScope(ExceptionUtil.ejpPrintExcAndExit)) && args(excType,..))
    || (execution(ejpScope(ExceptionUtil.ejpOnExcSetValue)) && args(excType,..))
    || (execution(ejpScope(AdminClientHandler.ejpShowPanel)) && args(excType,..))
    || (execution(ejpScope(SignonWebHandler.ejpRedirectResp)) && args(excType,..))
    || (execution(ejpScope(*.ejpTraceException)) && args(excType,..));

  /** Matches parameterized EJPs that WILL rethrow */
  pointcut excWR(Class excType):
       (execution(ejpScope(ExceptionUtil.ejpPrintAndRethrow)) && args(*,excType,..))
    || (execution(ejpScope(ExceptionUtil.ejpRethrowDiff)) && args(*,*,*,excType,..))
    || (execution(ejpScope(ExceptionUtil.ejpConvertExcToItsCause)) && args(excType));

  /* 10 other EJPs and 2 other methods are explicitly named in other pointcuts.
   * In total 21 EJPs and 2 methods are referenced explicitly in pointcuts.
   * These 23 references couple this aspect to 9 other modules. */
}
```

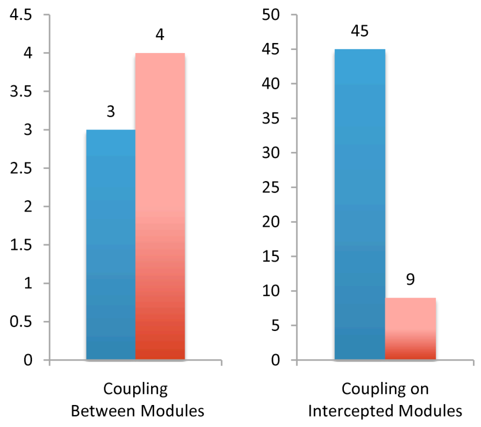
Listing 15. Pointcuts capturing most of the join points for the exception monitoring concern for the explicit join point version

of the `FaultAnalyzerDataSink` interface shown in Listing 11. We then created an abstract aspect, shown in Listing 12, that provides data to the fault analyzer for the most common patterns. Note that two pointcuts were required to model a caught exception (both the `handlerWillNotBeRethrown` and `handlerWillBeRethrown` pointcuts) because the concrete aspect cannot easily parameterize the advice in the base aspects, as discussed further in challenge number 2 of Section 3.6. For the AspectJ and EJP versions we then derived separate concrete aspects to specify the pointcuts capturing these patterns, as well as for any additional advice for exception handlers that did not fit one of the patterns handled by the abstract aspect. Listing 14 and Listing 15 present a part of these concrete aspects for the AspectJ and EJP versions respectively, showing the kind of pointcuts required to implement the new concern.

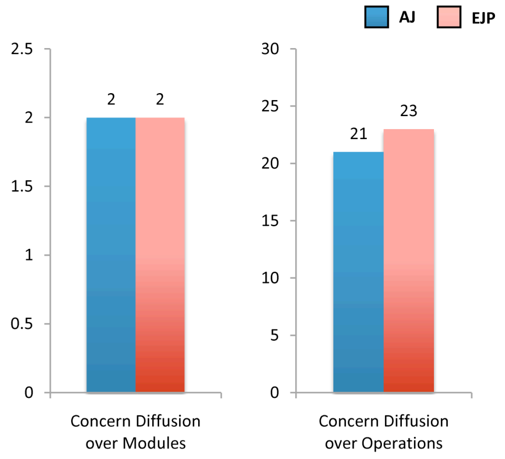
The new concerns requirement for contextual information about faults required the new aspects to classify handler blocks as to whether or not a handled exception would be rethrown. However, this classification was difficult because it is not possible to lexically pick out handler blocks within certain 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



(a) Size



(b) Coupling



(c) Separation of Concerns

Fig. 10. Metrics results for an aspect implementing a new concern in Java Pet Store (lower is better)

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 exception handling scoped EJP reference that was exited (either normally by reaching the end of the scope or via a thrown exception, on a per-thread basis). This 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.

7.4 Results

Figure 10 show the results of the size, coupling, and separation of concerns metrics. Metrics that are not shown in the figure have 0 values in all cases.

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.

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. This prevented us from using more general pointcuts that did not explicitly name specific handler aspects. 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, requiring review of the related pointcuts in the new aspect. In contrast, the EJP version pointcuts were structured around EJP names and would be robust against all but significant changes in their targeted EJP.

Moreover, the new aspect in the AspectJ version is fragile to changes in the handler base code. For example, if a handler aspect that contained only catch blocks that did not rethrow an exception was extended with an advice with a catch block that did rethrow an exception, the pointcuts in the new aspect would have to be changed in order to correctly reclassify catch blocks in the changed handler aspect. An example of this can be seen in Listing 13 and Listing 14. Assume in an older version of Java Pet Store the `AdminWebHandler` aspect only contained a handler for the `ServiceLocatorException` exception (the first handler). If this was the case, then all handlers in that aspect would rethrow the exception and so the `adminWebHandlerCase` pointcut would not yet be needed. If the other handlers were then added to the `AdminWebHandler` as the application evolved but the `adminWebHandlerCase` pointcut was not added, then the exception monitoring aspect would incorrectly mark all of the exceptions thrown from the `AdminWebHandler` aspect as being rethrown.

In contrast, in the EJP version both base code and handler base code were structured by the use of EJPs, allowing the new aspect to advise classes of exception handling patterns (e.g., catch and print, catch and rethrow, etc.) based solely on the name of the EJP. Continuing the example above, in the EJP version the `AdminWebHandler` aspect does not exist, and instead the base code directly references the relevant EJPs. These factors imply that the new aspect for the EJP version is resilient to changes in the handler base code, as is confirmed by the significantly lower coupling and pointcut size metric results.

The EJP version required 2 more operations to implement the concern, resulting in the Number of Operations metric to be slightly higher for the EJP version. However, the methods in the EJP version tended to be simpler and easier to comprehend (qualitatively).

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.

Coupling Between Modules was nearly identical in both cases. The one additional coupling in the EJP version was from the use of a generic EJP helper class. For both the AspectJ and EJP versions the Lack of Cohesion in Operations was 0, due to the tightly focused nature of the new objects and aspects in both cases.

For both versions the aspects implementing the new concern are completely oblivious and have nearly identical Separation of Concerns metric values. Concern Diffusion over Modules was 2 in both cases, indicating that logic for the exception monitoring concern was implemented entirely within 2 modules. The EJP version required 2 more operations to actually implement the concern, resulting in the Concern Diffusion over Operations metric to be slightly higher for the EJP version. The Concern Diffusion over Lines of Code metric was 0 for both versions. This indicates that the code implementing the new cross-cutting concern was completely isolated from the base code in both versions, verifying that both versions had purely oblivious implementations.

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, base code is structured in semantic ways and more contextual information is accessible to oblivious pointcuts, making oblivious programming in the future more powerful and robust.

8 Conclusions

We have presented and precisely defined explicit join points, which enhance the power of Java and AspectJ and increase modularity and safety. The value of EJPs were explored by implementing the transactions cross-cutting concern with both AspectJ and EJPs and comparing and contrasting the two methods in detail. We empirically evaluated how EJPs affect future oblivious extensibility, and the data indicates for our study of Java Pet Store that extensibility is enhanced when cross-cutting concerns are appropriately implemented using EJPs. Through EJPs we advocate a *cooperative aspect-oriented programming* (Co-AOP) approach where base code and aspects actively cooperate, which will thereby benefit software quality when applied judiciously. Versus the AspectJ version the EJP version had 25% fewer lines of code, 63% less pointcut size, and 80% less coupling on intercepted modules, while having identical or nearly equal values for all other quality metrics. An extension of the AspectBench compiler [3] that implements EJPs is freely available. [28] We anticipate future research exhibiting new design patterns made possible by the enhanced power of AspectJ with EJPs that further empowers cooperative aspect-oriented programming.

9 Acknowledgements

The authors express thanks to Jan Vitek and the anonymous reviewers for insightful comments.

References

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

[2] J. Aldrich, Open Modules: Modular reasoning about advice, in: ECOOP'05, 2005.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, abc: An extensible AspectJ compiler, Transactions on Aspect-Oriented Software Development 1 (2005) 293–334.

[4] K. Beck, Embracing change with Extreme Programming, in: Computer, vol. 32, IEEE Computer Society Press, 1999.
URL `http://dx.doi.org/10.1109/2.796139`

[5] J. Bonér, AspectWerkz–dynamic AOP for Java, in: AOSD'04, 2004.

[6] L. Bussard, Towards a pragmatic composition model of CORBA services based on AspectJ, in: Workshop on Aspects and Dimensions of Concerns of ECOOP'02, 2000.

[7] M. Butler, T. Hoare, C. Ferreira, Communicating Sequential Processes, vol. 3525/2005 of Lecture Notes in Computer Science, chap. A Trace Semantics for Long-Running Transactions, Springer, 2005, pp. 133–150.

[8] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, C. Lucena, Composing design patterns: a scalability study of aspect-oriented programming, in: AOSD'06, 2006.

[9] M. Ceccato, P. Tonella, Measuring the effects of software aspectization, in: 1st Workshop on Aspect Reverse Engineering, Delft, The Netherlands, 2004.

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

[11] C. Clifton, G. Leavens, Obliviousness, modular reasoning, and the behavioral subtyping analogy, in: SPLAT'03, 2003.

[12] A. Colyer, Towards widespread adoption of AOSD, in: AOSD'03, 2003.

[13] C. Fetzer, K. Hogstedt, P. Felber, Automatic detection and masking of nonatomic exception handling, IEEE Transactions on Software Engineering 30 (8) (2004) 547–560.

[14] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhao, A. Garcia, C. M. F. Rubira, Exceptions and aspects: the devil is in the details, in: FSE'06, 2006.

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

[16] M. Fuad, D. Deb, M. Oudshoorn, Adding self-healing capabilities into legacy object oriented application, in: ICAS'06, 2006.

[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[18] A. Garcia, C. Sant'Anna, C. Chavez, V. T. da Silva, C. de Lucena, A. von Staa, Software Engineering for Multi-Agent Systems II, chap. Separation of Concerns in Multi-agent Systems: An Empirical Study, Springer, 2004, pp. 49–72.

[19] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa, Modularizing design patterns with aspects: a quantitative study, in: AOSD'05, 2005.

[20] S. Ghosh, R. B. France, D. M. Simmonds, A. Bare, B. Kamalakar, R. P. Shankar, G. Tandon, P. Vile, S. Yin, A middleware transparent approach to developing distributed applications, Software: Practice and Experience 35 (12) (2005) 1131–1154.

[21] S. Goldman, D. Detlefs, S. Dever, K. Russell, New compiler optimizations in the Java HotSpot virtual machine, in: JavaOne 2006, 2006.

[22] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, 3rd ed., Addison-Wesley, 2005.
URL `http://java.sun.com/docs/books/jls/`

[23] J. Gosling, B. Joy, G. Steele, G. Bracha, Java Language Specification (3rd Edition), Addison-Wesley Professional, 2005.

[24] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, A. Rashid, On the impact of aspectual decompositions on design stability: An empirical study, in: ECOOP'07, 2007.

[25] W. Griswold, K. Sullivan, W. Song, M. Shonle, N. Tewari, Y. Cai, H. Rajan, Modular software design with crosscutting interfaces, IEEE Software 23 (1) (2006) 51–60.

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

[27] T. Harris, Exceptions and side-effects in atomic blocks, Science of Computer Programming 58 (3).

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

[29] K. Hoffman, P. Eugster, Bridging Java and AspectJ through explicit join points, in: PPPJ'07, 2007.

[30] K. Hoffman, P. Eugster, Towards reusable components with aspects: An empirical study on modularity and obliviousness, in: ICSE'08, 2008.

[31] IBM, BEA Systems, Microsoft, Arjuna, Hitachi, IONA, Web Services Transactions Specifications (2005).

[32] Java Pet Store, `https://blueprints.dev.java.net/petstore/`.

[33] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: ECOOP'01, 2001.

[34] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: ECOOP'97, 1997.

[35] G. Kiczales, M. Mezini, Aspect-oriented programming and modular reasoning, in: ICSE'05, 2005.

[36] J. Kienzle, R. Guerraoui, AOP: Does it make sense? the case of concurrency and failures, in: ECOOP'02, 2002.

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

[38] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, chap. 11, Manning, 2003, pp. 356–390.

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

[40] C. Prehofer, Feature-Oriented Programming: A Fresh Look at Objects, in: ECOOP'97, 1997.

[41] H. Rajan, K. Sullivan, Classpects: unifying aspect- and object-oriented language design, in: ICSE'05, 2005.

[42] C. SantAnna, A. Garcia, C. Chavez, C. Lucena, A. von Staa, On the reuse and maintenance of Aspect-Oriented software: An assessment framework, in: 17th Brazilian Symposium on Software Engineering, 2003.

[43] Y. Smaragdakis, D. Batory, Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs, ACM Transactions on Software Engineering and Methodology 11 (2) (2002) 215–255.

[44] M. Stochmialek, aopmetrics, `http://aopmetrics.tigris.org/`.

[45] M. Stoerzer, J. Graf, Using pointcut delta analysis to support evolution of aspect-oriented software, in: ICSM'05, 2005.

[46] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, H. Rajan, Information hiding interfaces for aspect-oriented design, in: FSE'05, 2005.

[47] P. L. Tarr, S. M. Sutton Jr., Programming heterogeneous transactions for software development environments, in: ICSE'93, 1993.

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

[49] C. Zhang, H.-A. Jacobsen, Quantifying aspects in middleware platforms, in: AOSD'03, 2003.

[50] J. Zhao, Measuring coupling in Aspect-Oriented systems, in: METRICS'04, 2004.