Bridging Java and AspectJ through Explicit Join Points

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

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

ABSTRACT

Through AspectJ, aspect-oriented programming (AOP) is becoming of increasing interest and availability to Java programmers as it matures as a methodology for improved software modularity via the separation of cross-cutting concerns. AOP proponents often advocate a development strategy where Java programmers write the main application, ignoring cross-cutting concerns, and then AspectJ programmers, domain experts in their specific concerns, weave in the logic for these more specialized cross-cutting concerns. However, several authors have recently debated the merits of this strategy by empirically showing certain drawbacks. The proposed solutions paint a different development strategy where base code and aspect *programmers* are aware of each other (to varying degrees) and interactions between cross-cutting concerns are planned for early on.

Herein we explore new possibilities in the language design space that open up when the base *code* is aware of cross-cutting aspects. Using our insights from this exploration we concretize these new possibilities by extending AspectJ with concise yet powerful constructs, while maintaining full backwards compatibility. These new constructs allow 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. We illustrate the value of our extensions through an example based on transactions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Procedures, functions, and subroutines*

General Terms

Design, Languages.

PPPJ 2007, September 5–7, 2007, Lisboa, Portugal.

Keywords

Aspect-oriented programming, join point, modularity.

1. INTRODUCTION

The principles of *aspect-oriented programming* (AOP) [21] 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 [20], 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 [11]."

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 [8], 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 [1, 7, 10, 17, 22, 23, 27, 28].

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 [28], reduce aspect modularity [10], and cause safety and feasibility problems [23].

To address these challenges authors have proposed ways in which to plan for, restrict, infer, or document aspects' interaction with base code [1, 17, 22, 28]. With the exception of [22] 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 models on which they are built, but rather seek to adjust, guide, or constrain systems such that modularity is improved.

In this paper we explore new territory starting from a new premise, considering the improved modularity and safety that are possible when aspects are part of the software design process and when aspects and base code can *explicitly* communicate. 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.

This paper presents the full EJP 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.

2. 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.

3. 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.

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 summarizes our work. The Appendix details our extensions to the AspectJ and Java grammars.

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. [11] These two concepts are so fundamental that some authors accept the assertion that AOP *is* quantification and obliviousness.

AspectJ [20] 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.

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 [6, 10, 13, 12, 28, 30]. One empirical case study [10] 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.

The issues involved in obliviously using aspects to add transactional semantics to base code were studied in [23]. 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 issues, as there is no way for aspects to enforce constraints upon the base code being advised.

In [17] 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 3.5. 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 [28] 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 crosscutting programming interfaces (XPIs), extending the ideas of [17]. 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 [16] 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 [17] for the same reasons.

Aldrich takes a different approach by proposing *open modules* [1]. Aspects are only allowed to advise the join points explicitly exported by a module. If a module carefully de-

fines 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 [28], is often the case.

A similar approach to improving modularity is taken in [22], 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. DESIGN PRINCIPLES

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.

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

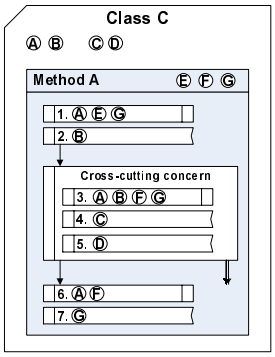

Figure 1: Inlined approach

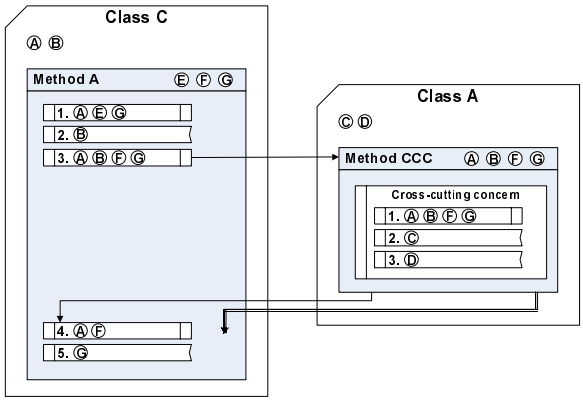

Figure 2: Refactoring approach

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.

3.3 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

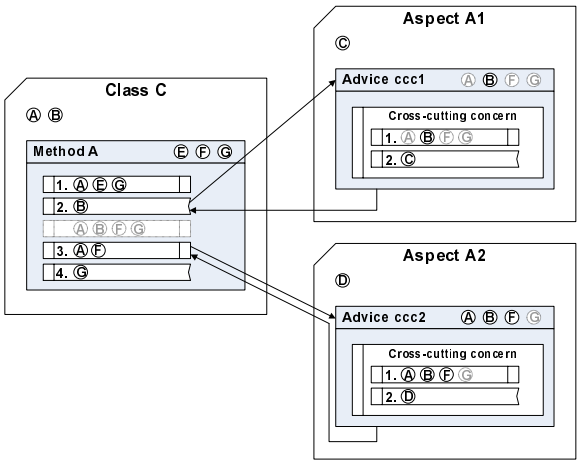

Figure 3: AspectJ approach

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 [27].

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.

```
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

3.4 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 [29], 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 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 [5]. Other examples include interaction with legacy systems that are incompatible with the transactionalizing mechanisms, physical or network device (I/O), etc. [18].

Using AspectJ to implement transactions and related mechanisms is an area of active interest in research and industry [4, 14, 24, 31]. 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, etc.

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 [24] (based on JDBC and JDA) and in [23] (in the context of the OPTIMA transaction framework).

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`). Barring failures and potential state–point separation issues, the AspectJ approach works well — the business logic is free from

```
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

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.

3.5 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 [9] 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. First, 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 [10, 28].

2. Second, 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. Finally, 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 [26] 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 extendability study [19].

4. EXPLICIT JOIN POINTS

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

4.1 Simple EJPs

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 (OO) instead of aspects weaving logic into base code (AO).

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.

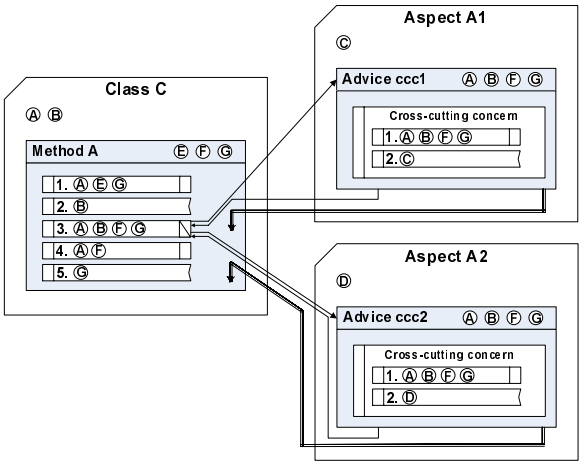

Figure 4: AspectJ with EJPs approach

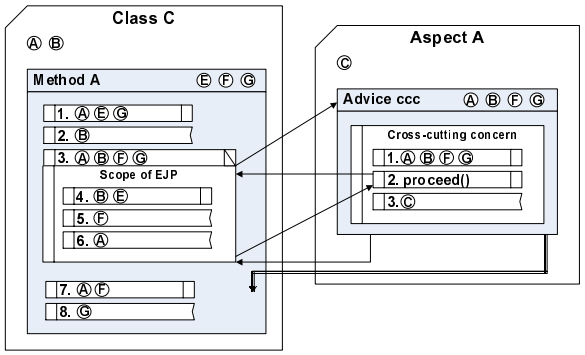

Figure 5: Advising arbitrary code blocks using scoped EJPs

Listing 3 overviews syntax for the fully fledged EJPs and previews the advanced EJP features that we introduce in the sections below, while Appendix A precisely defines our syntactic extensions. Refer to Listing 4 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.[1] 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 [25], 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 Beyond simple EJPs

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 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 5. Refer to Listings 3 and 4 and Appendix A 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. First, it has been shown empirically that refactoring methods to expose new join points decreases object cohesion and artificially increases system complexity [10] (as measured by the well-founded metrics of that study).

2. Second, 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

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

```
aspect A { ...
  // Syntax for EJP declarations:
  [<JavaModifiers>] [scoped] joinpoint
    <ReturnType> <Name> "(" <Formals> ")"
      [pointcutargs "(" <DefPCArg> ["," <DefPCArg>]* ")"]
      [handles ...]
      [throws ...]
      ["=" <DefaultValueExpr>] ";"
  //where <DefPCArg> is <ArgName> "(" <Formals> ")"
}
class C {
  void m() {
    // Syntax for using EJPs in base code:
    <AspectName> "." <EJPname> "(" <Args> ")"
      [pointcutargs <PCArg> ["," <PCArg>]*] ] ";"
    // Syntax for using scoped EJPs in base code:
    <AspectName> "." <EJPname> "(" <Args> ")"
      [pointcutargs <PCArg> ["," <PCArg>]* ]
    "{"
      // block of code to be advised
    "}" ";"
    //<PCArg> is <ArgName>"("<Formals>")" ":" <Pointcut>
  }
}
```

Listing 3: Overview of explicit join point syntax

```
aspect A {
  // Examples of ejp and ejpScope pointcut modifiers:
  pointcut p1(...): call(ejp(<EJP name>)) && args(...);
  pointcut p2(): cflow(call(ejpScope(<scoped EJP name>)));
}
```

Listing 4: Matching EJPs in pointcuts

of these methods. Each concern added to the system makes refactoring increasingly difficult [10], hindering methodologies such as Extreme Programming [3].

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 multi-stage advice would need to be referenced and understood.

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.[2] In essence, it is a promise to the base code that aspects 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.

```
1 interface CompensationRecord {
2   void compensate();
3   void log(...);
4 }
5 abstract aspect TranConcern {
6   scoped joinpoint void enterTrans(int isolationLevel)
7           pointcutargs ignoreableCompensations()
8           throws TranException;
9   joinpoint int defaultIsolationLevel() = 0;
10  joinpoint void addCompensation(CompensationRecord r);
11 }
```

Listing 5: EJP interface defining the transaction cross-cutting concern

```
1 class Agent {
2   CardProcessor cc = ...;
3   void createTrip(Person p, Flight f, Hotel h)
4       throws TranException {
5     int level = TranConcern.defaultIsolationLevel();
6     TranConcern.enterTrans(level) {
7       f.reserveSeat(p);
8       h.reserveRoom(p);
9       cc.debit(p.getCC(), ...);
10      //The TranConcern.addCompensation EJP would
11      //be referenced here because of the cc.debit call
12    };
13  }
14 }
```

Listing 6: Pseudo-code of business logic with EJPs

4.4 Transactions with EJPs example

With our new language features we are now better equipped to effectively implement the transaction cross-cutting concern. Listings 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 3.4. First of all, the `enterTrans` EJP declaration on line 8 of Listing 5 specifies that references to that EJP must handle that checked exception (see Listing 6, line 4) and that aspects advising that EJP are allowed to throw a checked exception of type `TranException`. Whereas in the AspectJ implementation the `TransactionImpl` aspect 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.

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

```
1 aspect TransactionImpl {
2   pointcut tranScopes(int isolationLevel):
3       call(ejpScope(TranConcern.enterTrans))
4     &&  args(isolationLevel);
5   //A transaction begins and ends at the outermost
6   //instance in the control flow of the enterTrans EJP
7   void around(int isolationLevel) throws TranException:
8       tranScopes(isolationLevel)
9     && !cflowbelow(tranScopes(..))
10  {
11    TransContext t = ...;
12    t.beginTrans();
13    proceed();
14    t.commitTrans();
15  }
16  after() throwing throws TranException: tranScopes(..){
17    TransContext t = (...).getActiveContext();
18    if (t != null) t.abortTrans();
19  }
20  //Some aspect could advise the addCompensation EJP
21 }
```

Listing 7: Pseudo-code of aspect implementing transaction concern with EJPs

An error handling aspect would soften the EJP's checked exception using a robust pointcut based on the EJP name:

```
declare soft(): TranException:
    call(ejpScope(TranConcern.enterTrans))
```

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.

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 5, line 9). 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 6, line 5).

Additionally, the implementation is robust to changes in how transactions are implemented. If one desires to try a different aspect in the place of `TransactionImpl` then the only step is 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 7, lines 2, 7, and 8). 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 [23] 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

```
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

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 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 EJP FEATURES

In this section two additional features of EJPs are introduced: *pointcut arguments* and *policy enforcement*.

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 `<EJP name>.<arg name>`.

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 Listing 3). 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 mod-

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

Listing 9: Base code adding to a pointcut argument

ifiers `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 5 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 6. 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. We add the `enterTrans.ignoreableCompensations` pointcut argument (see Listing 5), 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 5:

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

5.2 Policy enforcement

Another advanced feature complementing EJPs is our contribution of *policy enforcement* mechanisms. In AspectJ, compilation errors can be generated when join points match certain pointcut expressions. We extend this mechanism 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:

```
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
```

This 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. CONCLUSIONS

We have presented and precisely defined explicit join points with their advanced features, which enhance the power of Java and AspectJ and increase modularity and safety. The value of EJPs were demonstrated by implementing the transactions cross-cutting concern with both AspectJ and EJPs and comparing and contrasting the two methods in detail. Through EJPs we advocate a cooperative AOP approach where base code and aspects actively cooperate. An extension of the AspectBench compiler [2] that implements EJPs is freely available under terms of the LGPL[3]. We anticipate future research exhibiting new design patterns made possible by the enhanced power of AspectJ with EJPs.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

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

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

[3] K. Beck. Embracing change with Extreme Programming. In *Computer*, volume 32, pages 70–77. IEEE Computer Society Press, 1999.

[4] 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.

[5] M. Butler, T. Hoare, and C. Ferreira. *Communicating Sequential Processes*, volume 3525/2005 of *Lecture Notes in Computer Science*, chapter A Trace Semantics for Long-Running Transactions, pages 133–150. Springer, 2005.

[6] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena. Composing design patterns: a scalability study of aspect-oriented programming. In *AOSD'06*, pages 109–121, 2006.

[7] C. Clifton and G. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT'03*, 2003.

[8] A. Colyer. Towards widespread adoption of AOSD. In *AOSD'03*, 2003.

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

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

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

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

[3] http://www.cs.purdue.edu/homes/kjhoffma/

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

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

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

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

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

[18] T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3), 2005.

[19] K. Hoffman and P. Eugster. Aspects and exception handling: The case of explicit join points. Technical Report ejp-200703-1, Purdue University, 2007.

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

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

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

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

[24] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*, chapter 11, pages 356–390. Manning, 2003.

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

[26] H. Rajan and K. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE'05*, pages 59–68, 2005.

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

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

[29] P. L. Tarr and S. M. Sutton Jr. Programming heterogeneous transactions for software development environments. In *ICSE'93*, pages 358–369, 1993.

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

[31] C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *AOSD'03*, pages 130–139, 2003.

APPENDIX

A. SYNTAX EXTENSIONS

Herein we formally define our extensions to the AspectJ and Java grammars. Our grammar extensions are LALR(1), and their descriptions follow the conventions used in the Java language specification [15].

Syntax added to AspectJ:

ExplicitJoinPointDeclaration:
ExplicitJointPointHeader *FormalPointCutParams*$_{opt}$ *Handles*$_{opt}$ *Throws*$_{opt}$ *ExplicitJoinPointInitializer*$_{opt}$

ExplicitJointPointHeader:
Modifiers$_{opt}$ **scoped**$_{opt}$ *ExplicitJointPointDeclarator*

ExplicitJointPointDeclarator:
joinpoint *ResultType* *Identifier* **(***FormalParamList***)**

FormalPointCutParameters:
pointcutargs *FormalPointCutParameterList*

FormalPointCutParameterList:
FormalPointCutParameter
FormalPointCutParameter **,** *FormalPointCutParameterList*

FormalPointCutParameter:
Identifier **(***FormalParameterList*$_{opt}$**)**

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 *Granularity* **:** *PointcutExpression* **:** *PointcutExpression* **:** *StringConstant*

Granularity:
by package
by class
by method
by ejp

Syntax added to Java:

ExplicitJoinPointExpression:
ExplicitJoinPoint *PointcutArguments*$_{opt}$ *Block*$_{opt}$**;**

ExplicitJoinPoint:
AspectName **.** *ExplicitJointPointName* **(***ArgumentList*$_{opt}$**)**

PointcutArguments:
pointcutargs *PointcutArgumentList*

PointcutArgumentList:
PointcutArgument
PointcutArgument **,** *PointcutArgumentList*

PointcutArgument:
PointcutName **(***FormalParamList*$_{opt}$**)** **:** *PointcutExpression*