Enhancing Base-code Protection in Aspect-Oriented Programs

Mohamed ElBendary
University of Wisconsin-Milwaukee
Milwaukee, WI 53211
mbendary@cs.uwm.edu

John Boyland
University of Wisconsin-Milwaukee
Milwaukee, WI 53211
boyland@cs.uwm.edu

ABSTRACT

Aspect-oriented programming (AOP) promises to localize concerns that inherently crosscut the primary structural decomposition of a software system. Localization of concerns is critical to parallel development, maintainability, modular reasoning, and program understanding. However, AOP as it stands today causes problems in exactly these areas, defeating its purpose and impeding its adoption. First, the need to open up systems' modules for aspects' interaction competes with the need to protect those modules against possible fault injection by aspects. Second, since aspects are written in terms of base code interfaces, base system components must be stable before aspect components can be developed. This dependency hinders parallel development. This work proposes a language-based solution that allows base code classes to regulate aspect invasiveness, and provides loose coupling of aspects and base code.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Aspect-Oriented Programming; D.2 [**Software Engineering**]: Compilers

General Terms

Algorithms, Design

1. INTRODUCTION

Aspect-oriented software development (AOSD) is supposed to apply over a system's entire lifetime, positively impacting software measures such as cost, quality, and time-to-market [8]. AOP promises to localize cross-cutting concerns by providing language-based mechanisms for explicitly representing their structure and/or behavior. AOP does provide a cleaner separation of concerns. However, AOP negatively impacts modularity by crossing module boundaries in a completely unregulated fashion [10].

This work is an attempt to resolve two points of contention that are impeding the adoption of AOP. The first point is the competition between the need to open up systems' modules for AOP and the need to protect those modules against possible fault injection by AOP. The second point is the need to have base system components stabilized before aspect components can be developed, which reduces opportunities for parallel development.

We believe that pure obliviousness (currently, the dominant approach to AOP, as in AspectJ) is problematic for the following reasons. First, while clients (including aspects) may be insensitive to changes in implementation details of the components they use, they are tightly coupled to their interfaces. For example, an aspect that references method `m` in class `C` breaks if method `m` is now called `n`, this problem is referred to as the *fragile pointcut problem*. Second, pure obliviousness offers no information on the base side regarding what elements of an interface are being advised or which aspects are involved. Aspects can infuse the component's internals through introductions and advice mechanisms making it impossible to reason about a base component by examining it in isolation. Third, pure obliviousness renders the base code component entirely helpless in the face of *harmful aspects*. A harmful aspect is an aspect that violates a base code policy as it extends (advises) base components, for example, by replacing the body to be executed at a joinpoint with something entirely different.

Our philosophy is that since the base code and the aspect code participate in making up a module's interface, they should explicitly cooperate to preserve the module's boundary. We see a module boundary extending beyond traditional class or aspect module boundaries with base code classes being responsible for establishing their module boundaries within the system, using advising constraints to limit aspects' invasiveness.

We believe that pure obliviousness can be sacrificed to maintain ease of reasoning, ease of maintenance, separation of concerns, and code locality. This work focuses on separation of concerns, code quality, and ease of maintenance as primary "concerns".

2. INTERFACE IMAGE (I2) APPROACH

An Interface Image (I2) is a level of indirection through which all advising requests are carried out. It provides a mechanism by which a class exposes a set of joinpoints through aliasing base code interface elements. The image incorporates advising constraints per exposed joinpoint. Aspects are developed against the aliases defined in the interface images of base code classes. Aspects are not allowed to advise classes directly. This indirection limits the scope of

FOAL'09, March 2, 2009, Charlottesville, Virginia, USA.

image_declaration ::= `image {`
`[opento: {` *TypeAccess** `};]`
`[`*alias definition**`]`
`}`
alias_definition ::=
method-header `=` *method-header* `{` *constraints* `}`
| `* = * {` *constraints* `}`

method-header ::=
`[`*modifiers*`]` *RT method-name*`(`*P*`)` *throws_list*

modifiers: *Java-style member method modifiers*
RT ::= *TypeAccess*
method-name: *Java-style method identifier*
P: *Java-style method parameter list*
throws_list ::= *TypeAccess**
TypeAccess ::=
Java-style type access (qualified and simple type names)

constraints ::= `kind: { `*Advice_Kind** `};`
| `(origin=`*Origin*`, boundary=`*Boundary*`);`
| `exceptions: { `*Exception_Type** `};`

Advice_Kind ::= `before` | `after` | `after_returning`
| `after_throwing` | `around`
Origin ::= `internal` | `external`
Boundary ::= `method` | `class` | `package`
Exception_Type ::= *TypeAccess*

Figure 1: Interface image syntax.

dependency of aspects on base code to that of images only. I2 lends itself to a feature-obliviousness design, as the next section will show. Our design requires cooperation from the base code developers so it is not language-level oblivious. I2 is not designer-oblivious either since it assumes designers are aware of aspects realizing functionality.

In this design, an I2 provides the following benefits:

1. The base code is now an active participant in the advising process since it is up to each class to expose joinpoints on which it permits advice. For each exposed joinpoint, advising constraints can be attached to disallow unwanted aspect advising.

2. Response to changes in the interface of a class is limited to updates in the class' interface image. Aspects are not involved. Parallel development can benefit from this loose coupling.

3. The I2 serves as a specification of advisable interface elements for base code and aspect developers alike.

This work studies the interface image approach in the context of classes only. We leave augmenting interfaces and aspects with interface images for future work.

Interface images are defined using the `image` construct. An `image` can only appear within the scope of a class definition. Figure 1 shows the syntax and Fig. 2 shows an example instance. An empty image `image{}` exposes the class to unrestricted (AspectJ-style) advising.

```
class Point extends Shape {
  protected int x, y;
  public void moveby(int dx, int dy){
        x += dx; y += dy;
  }

  image {
    opento: {CheckScene};
    public void moveby(int dx, int dy) =
    public void translate(int dx, int dy){
     kind: {after};
     (origin=external, boundary=class);
     exceptions: {SceneInvariantViolation};
    }
  }
}
```

Figure 2: Example `image` for a class `Point`

2.1 The `opento` clause

The `opento` clause allows a class to provide a list of aspects allowed to perform introductions on it. If an image does not declare an `opento` list, then the enclosing class will accept introductions from any aspect. An empty `opento` list prohibits any aspects from performing introductions on the declaring class.

2.2 Aliases

An alias definition has a signature on the left-hand side of an equal sign followed by an alias signature and an attached scope for declaring advising constraints. The aliases are used to name aspect joinpoints—only aliased methods can be advised.

A class can only alias methods that it explicitly declares. Both instance and static methods are aliasable. The wildcard form `* = *` permits all declared methods in the class to be advised under a single set of advising constraints.

An image can also declare multiple aliases for the same joinpoint to further allow constraint refinement per joinpoint. If two aspects implementing two different concerns each with a different set of, possibly conflicting, advising constraints at the same joinpoint, accommodating both is easily done by defining different aliases on the same joinpoint and having each aspect use a different alias definition. Providing different hooks (aliases) with different advising constraints essentially allows joinpoints to "fan-out" different channels for aspects to communicate with the base code.

2.3 Advising Constraints

The `kind` clause lists the advice kinds allowed at this joinpoint. For example (Fig. 2), the clause `kind: {after}`, would only allow `after` advice at this joinpoint for this alias. If an aspect declares a pointcut that matches this alias, and declares an advice of a kind other than this kind (e.g. `around`), this advice application will be disabled. This is useful for enforcing the design intent that the translation cannot be skipped.

An empty `kind` clause turns off *any* advising on this joinpoint through this alias. If a `kind` clause is omitted, all advice kinds are allowed.

The `(origin, boundary)` pair, if it exists, specifies whether advising is permitted for calls originating inside (`internal`) or outside (`external`) module boundary or both. A module boundary is either `method`, `class`, or `package`. If the `(origin, boundary)` pair is omitted, all calls may be advised.

The `exceptions` clause, if it exists, lists all exception types that if thrown by this joinpoint cannot be "softened" by an aspect with a matching pointcut of a `declare soft()` statement. Omitting the `exceptions` clause allows all exceptions to be softened. An empty exceptions list prevents softening of any exception through this alias.

2.4 Summary

The interface image technique has the following benefits to the base code designer: The base code can limit advice to join points that are semantically relevant to the outside, and can limit which aspects are permitted to perform introductions. The base code can determine which exceptions can be safely softened. No extra aspect is required to check advising constraints.

On the other hand, the aspect designer can still make use of the full power of aspect orientation and is now only dependent on the alias names, not the actual method names. This avoids the overloading of method signatures with two different meanings, from the client's perspective they are service access points while from the aspect's perspective they are joinpoints.

3. EXAMPLE

This banking authorization example is adapted from Laddad [9]. Ladded developed it to showcase modularity of an AspectJ solution over a conventional Java solution. The example is an authorization service in a banking system. The base code (not shown) for this example consists of methods for performing simple operations on bank accounts: debit, credit and transfer.

Laddad used an abstract aspect, shown in Fig. 3 to implement an authorization protocol. The abstract pointcut `authOperations()` acts as a hook for concrete derived aspects to quantify which operations in the system they want to apply the authorization protocol to. A derived concrete aspect, `BankingAuthAspect`, that fully implements the authorization concern is shown in Fig. 3.

The first `before()` advice in Fig. 3 performs authentication if the subject accessing the system has not been authenticated yet. The `around()` advice wraps authorization around the banking operations' calls. The example uses banking methods' names as the permission names. We rely on the `JoinPoint.StaticPart` parameter to access method names at the joinpoints in the body of `getPermission()`.

The implementation given uses strictly two kinds of advice, `before` and `around`. So it is safer to allow exactly those kinds of advice and explicitly disable all others, possibly allowing more as new concerns are added. It is important to note that allowing `around` does not automatically include `before` and `after` kinds, even though their effects may be possible.

The solution provided by Laddad [9] does not soften the checked exception, `InsufficientBalanceException`, used by the application to prevent invalid withdrawals and/or transfers. The solution provides an aspect implemented specifically for preserving this exception. We believe that,

```
public abstract aspect AbstractAuthAspect{
  public abstract pointcut authOperations();

  before() : authOperations() {
   // authentication logic
  }

  public abstract Permission getPermission(
   JoinPoint.StaticPart joinPointStaticPart);

  Object around()
    : authOperations() &&
      !cflowbelow(authOperations()) {
    // Perform authorized operation
  }

  before() : authOperations(){
    // Authorization logic
  }

}  // Abstract aspect ends here
```

Figure 3: AbstractBankingAuthAspect [Laddad].

```
public aspect BankingAuthAspect
        extends AbstractAuthAspect{
  public pointcut authOperations()
    : execution(public * banking.Account.*(..))
   || execution(
        public *
    banking.InterAccountTransferSystem.*(..));

  public Permission getPermission(
    joinPoint.StaticPart joinPointStaticPart){
      return new BankingPermission(
        joinPointStaticPart.getSignature().
                getName());
    }
}
```

Figure 4: Concrete aspect BankingAuthAspect [Laddad].

```
  image {
    opento: {};
    * = * {
      kind: { before, around };
      exceptions: { InsufficientBalanceException };
      (origin=internal, boundary=package);
    }
  }
```

Figure 5: Banking example interface image declaration.

for an exception that is part of the contract of a core class method, the decision of whether it is softened or not, is for the core class to make. Clients on the base code may want to "know" about this situation and handle it in their own specific ways. The `exceptions` clause also saves substantial coding, since it replaces an entire aspect that had to be developed in the classical AspectJ solution.

Given the nature of this application and its operations, calls to `credit` and `debit` originating outside `banking` should be considered dubious. Currently, AspectJ guards against this using pointcut matching. However, a single misplaced wild card, could jeopardize the integrity of the application and the base code is simply helpless. Instead, base code classes could easily add the `(origin, boundary)` pair in Fig. 5, improving the robustness of advising and preventing the potentially erroneous behavior of *unintended* wild-cards.

The image in Fig. 5 is the finished product, this is all the code we need in order to alleviate the potential problems outlined above. In this example, the aspect side does not perform introductions of any sort. However, it does not make sense to keep the base classes open for intertype declarations, even though no errors will be caused in this case. This is because it is safer to progressively open classes to specific aspects as needed than leaving them open for all aspects and deal with possible maintenance problems later.

4. IMPLEMENTATION

I2 is implemented as an extension to AspectJ within the *aspectbench compiler* (abc) [2]. Our implementation uses the `JastAdd` [4] front-end of abc.

4.1 Image Semantic Checking

An image has to pass an error checking phase before it can be translated, including that the enclosing class does not have multiple image declarations, and that aliases are for methods declared in the enclosing class. We also check for duplicate constraint declarations.

If an aspect wishes to make introductions, the effected classes must have a `image` declaration and permit the introduction (see `opento`). Then for each joinpoint that the advice potentially applies to, we check the advice for violations of `kind` and `exceptions` clauses.

4.2 Image Translation

The image construct is translated internally into a privileged static nested aspect that performs method introductions into the class declaring the image. Being privileged allows the translated aspect to refer to private members of enclosing classes, which may be aliased.

One introduced wrapper method that called the original method is generated for each alias definition. An `around` advice is used to intercept calls to the original method and direct them to the wrapper. This advice in controlled by the `(origin, boundary)` pair constraint for the alias. Table 1 shows the translation to AspectJ conditions.

Additionally, we add `&& !within(imageAspect)` to each generated pointcut to prevent internal aspects from advising themselves, where "`imageAspect`" is some identifier used only internally that identifies the generated aspect.

We use AspectJ's precedence declaration to ensure that the generated aspects are applied before the concern-specific aspects:

Table 1: Translation of *(origin, boundary)* pairs to pointcuts.

(origin,boundary)	Pointcut Translation
(internal, method)	within(*signature(m)*)
(internal, class)	within(*enclosing type*)
(internal, package)	within(*enclosing package*)
(external, method)	!within(*signature(m)*)
(external, class)	!within(*enclosing type*)
(external, package)	!within(*enclosing package*)

```
class Point extends Shape {
  protected int x, y;
  public void moveby(int dx, int dy){
        x += dx; y += dy;
  }

  privileged static imageAspect {
   public void Point.translate(int dx, int dy){
     moveby(dx, dy);
   }
   void around(Point p):
    target(p) && !within(imageAspect) &&
    !within(Point) &&
    call(
      public void Point.moveby(int dx,int dy)){
        p.translate(dx, dy);
   }
  }
}
```

Figure 6: Class Point after translation

```
public aspect _internalOrderingAspect {
  declare precedence: *..*imageAspect*, *;
}
```

Any concern-specific precedence declarations are rewritten to add the pattern `*..*imageAspect*` to the front of the precedence list of each one of them.

Class `Point` shown in Fig. 2 translates internally to the one shown in Fig. 6. It shows the call to the aliased method `moveby()` wrapped inside introduced method `translate()`. It also shows the `around` advice and the pointcuts generated from the `(origin, boundary)` constraint. The `target(p)` pointcut exposes `p` for use in the advice body.

5. EVALUATION

This section presents evaluation of the I2 approach. This quantitative evaluation uses the AspectJ Development Tools' (AJDT) cross-cutting map generator to measure coupling. In the context of this study, coupling means the existence of a "cross-cutting relationship" between two components. A cross-cutting relationship in the source code arises from intertype declarations and advice declarations.

This study uses two examples from the AspectJ programming guide published by the Eclipse foundation [6]. These are, the `Observer` and the `Telecom Simulation`. The third is the `ants simulation` program used to demonstrate Open

Table 2: Coupling data reported by AJDT.

Example	AspectJ	AJI2	Coupling Change(%)
Observer	38	48	+26.3%
Telecom	30	24	-20.0%
Ants	670	626	-6.6%

Modules [1]. For each example, two implementations are analyzed, a classical AspectJ implementation and an AspectJ with I2-style implementation.

In order to allow AJDT to process I2 sources, we simulate the effects of I2 syntax by modifying the implementations of the examples as follows. We use a field introduction of an aspect instance into the class that uses `opento`, to account for classes referring back to aspects using the `opento` clause. We use one introduction declaration within each aspect referenced in the `opento` list. We also change method names in pointcuts to use alias names instead of the original names of their signatures in classes. For every alias that we use, we write a method in base code classes selected by the pointcut using the alias. This additional method bears the alias signature and wraps the call to the original aliased method. This arrangement simulates limiting the scope of aspects' references to images as opposed to the entire program as in AspectJ. The table above summarizes the "cross-cutting relationships" data reported by AJDT for each example.

In weighing these percentages a few details need to be taken into consideration. In `ants simulation`, the 6.6% corresponds to 44 less cross-cutting relationships in the program, which in turn corresponds to 27 spots in the source code where independent evolution is possible. We can only expect the number of spots to grow for larger programs with a similar feature mix. The 27 spots represent a 9% reduction in coupled spots in the code, a substantial percentage in a large program.

6. RELATED WORK

In Open Modules (OM) [1], a module exposes pointcuts that can be advised by external aspects as part of the module's interface. Thus only "external origin" calls are advisable. Also unlike I2, Open Modules do not allow advice to crosscut modules unless the modules are related through inheritance.

Cross-cutting Pointcut Interfaces (XPI) [5] separate a traditional aspect into three aspects: one to specify pointcuts and advising constraints; another aspect with advice implementations; and a third aspect to check advising constraints. Roughly, I2 can be seen as a way to automatically create and check the advising constraints of XPI.

Using Explicit Join Points (EJPs [7]), the base code adds syntactic hooks that look like static method calls at points where advising is required. This is more powerful than AspectJ or I2 because these hooks can occur in arbitrary blocks of code, at the cost of requiring the base code designer to put forward more effort.

Modular Aspects with Ownership (MAO [3]) enables modular reasoning using ownership to constrain heap effects. Advice can be declared as having no or only limited control effects, or no or limited data (heap) effects. Unrestricted aspects must be "accepted" by the base code to which they apply, in a similar mechanism to I2's "opento" declaration, which only applies to aspects with introductions. I2 preserves more feature obliviousness while MAO enables stronger modular reasoning.

Ptolemy [11] solves the fragile pointcut problem by introducing named event (joinpoint in AspectJ terminology) types that are declared independently from the modules (aspects) that announce/handle them. This is arguably superior to the pattern-matching used by AspectJ (and I2) to identify joinpoints. Ptolemy does not appear to support advising constraints.

7. CONCLUSION

This work addresses two modularity problems in AOSD: (1) aspect brittleness and sensitivity to base code changes; (2) the inability for base code to control advising. We provide a language-level solution to both problems in the form of a new construct added to classes that exports a view of the advisable class interface for aspects. Advising constraints can be attached to joinpoints that restrict what advice may apply. A prototype implementation as an AspectJ extension along with evaluation studies show it is possible to realize a design that loosely couples the evolution of the base code interfaces from the aspect-oriented code advising base components. We hope that these ideas can encourage greater adoption of AOP by the software engineering community.

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. In *AOSD 2005*, pages 87–98, 2005.

[3] C. Clifton, G. T. Leavens, and J. Noble. Mao: Ownership and effects for more effective reasoning about aspects. In *ECOOP '07*, pages 451–475, 2007.

[4] T. Ekman and G. Hedin. The jastadd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[5] W. G. Griswold, K. Sullivan, Y. Song, Y. Cai, M. Shonle, N. Tewari, and R. Hridesh. Modular software design with crosscutting interfaces. *IEEE Softw.*, pages 51–60, 2006.

[6] A. P. Guide. The AspectJ programming guide. http://www.eclipse.org/aspectj/doc/released/progguide/.

[7] K. Hoffman and P. Eugster. Bridging java and AspectJ through explicit join points. Technical Report ejp-200705-1, Purdue University, 2007.

[8] I. Jacobson. A case for aspects. *Software development Magazine*, October 2003.

[9] R. Laddad. *AspectJ IN ACTION, Practical Aspect-Oriented Programming.* Manning Publications Co., 2003. ISBN 1-930110-93-6.

[10] G. T. Leavens and C. Clifton. Multiple concerns in aspect-oriented language design: A language engineering approach to balancing benefits, with examples. Technical Report TR 07-01a, Iowa State University, 2007.

[11] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08*, pages 155–179, 2008.