Design and Implementation of An Aspect Instantiation Mechanism

Kouhei Sakurai[1], Hidehiko Masuhara[2]
Naoyasu Ubayashi[3], Saeko Matuura[1], and Seiichi Komiya[1]

[1] Shibaura Institute of Technology,
{sakurai@komiya.ise,matsuura@se,skomiya@sic}.shibaura-it.ac.jp
[2] University of Tokyo,
masuhara@acm.org
[3] Kyushu Institute of Technology,
ubayashi@acm.org

Abstract. This paper describes the design and implementation of *association aspects*, which are a linguistic mechanism for the AspectJ language that concisely associates aspect instances to object groups by extending the per-object aspects in AspectJ. This mechanism allows an aspect instance to be associated to a group of objects, and by providing a new pointcut primitive to specify aspect instances as execution context of advice. With association aspects, we can straightforwardly implement crosscutting concerns that have stateful behavior related to a particular group of objects. The new pointcut primitive can more flexibly specify aspect instances when compared against previous implicit mechanisms. We implemented a compiler for association aspects by modifying the AspectJ compiler (ajc), which reduces the size of data structures for keeping associations. Our benchmark tests confirmed that the overheads of association aspects are reasonably small when compared against functionally equivalent aspects in pure AspectJ that manually manage associations. The expressiveness of association aspects is demonstrated through development of an integrated development environment (IDE) with and without association aspects.

1 Introduction

In aspect-oriented programming (AOP), an aspect is the unit of modular definitions of crosscutting concerns. Aspects may be provided as a different module system from existing ones (e.g., in AspectJ[1]), or may be defined by using an existing module system (e.g., in Hyper/J[2]). In both cases, an aspect serves as the encapsulation of state and behavior, which are represented by instance variables and advice declarations, respectively, in AspectJ-like languages.

AspectJ-like languages run an advice body *in the context of* an aspect instance, in a similar sense that object-oriented languages run a method body in the context of an object. A problem is how to determine an aspect instance as the context of an advice execution, since aspect instances are not usually obvious

during the program execution. AspectJ, for example, offers a few mechanisms[4] to this problem:

- *singleton* aspects create only one aspect instance for each aspect declaration. This type of aspects are useful to implement concerns that have system-wide behaviors.
- *per-object* aspects *associate* a unique aspect instance for each object. When an operation in terms of an object triggers an advice execution, the system automatically looks up the aspect instance associated to the object, and uses the instance as the execution context. This type of aspects are useful to implement concerns that have a unique state for each object.

Those mechanisms are useful to certain kinds of crosscutting concerns, but Sullivan et al. pointed out that they do not straightforwardly support *behavioral relationships*, which are the concerns that integrate the behaviors of collections of objects by extending or modifying their respective behaviors[3]. With above mechanisms, such behavioral relationships are usually implemented by creating a singleton aspect with a table for associating the states unique to object groups. The resulted implementations have to have not only the code for the core behavior but also the code for managing association in a single aspect definition.

Subsequently, Rajan and Sullivan proposed instance-level advising by aspect instances as a solution, as demonstrated in their AOP language Eos[4]. In Eos, the programmer dynamically creates an aspect instance to represent behavioral relationships. Each aspect can be associated to the objects in its representing relation. When a method is called during program execution, the advice body is executed in the context of each aspect instance that is associated to the target of the call. As a result, the mechanism can cleanly implement such behavioral relationships. However, the mechanism can still be improved with respect to the following problems: (1) it is not flexible in the selection of aspect instances as it always selects with respect to the target object, and (2) it requires additional language constructs in order to distinguish associated objects of the compatible types.

This paper proposes an alternative mechanism called *association aspects*, which also allows us to associate an aspect instance to a group of objects. The mechanism addresses the above-mentioned problems by providing a new pointcut primitive that can more flexibly select aspect instances upon advice execution, and can distinguish associated objects without introducing other language constructs. The mechanism is implemented by modifying an AspectJ compiler(`ajc`[5]). Our benchmark tests showed that the association aspects can be implemented with acceptable amounts of overheads in comparison to the singleton or per-object aspects that manually manage tables.

The rest of the paper is organized as follows. Section 2 presents an example of behavioral relationships. Section 3 explains the design of association aspects, our proposed mechanism. Section 4 describes how association aspects are compiled

[4] There are also mechanisms based on the control flow, but they are not directly relevant to the topic of the paper.

into native Java programs. Section 5 gives the result of our benchmark tests to compare the efficiency of association aspects with respect to the programs in pure AspectJ. Section 6 shows an application program written with association aspects for comparing expressiveness against pure AspectJ. Section 7 compares association aspects to similar approaches. Section 8 concludes the paper.

2 Motivating Example

This section presents an example system to motivate the need for association aspects. Section 2.1 presents a system integration that becomes a crosscutting concern in object-oriented programming, and starts with prerequisites and requirements of the example system. Section 2.2 presents an object-oriented implementation of the system integration using a design pattern. Section 2.3 then shows that AspectJ implements the concern in an awkward manner. Section 2.4 analyzes the conditions when such problems happen.

The problem presented in this section was first pointed out by Sullivan, Gu and Cai[3].

2.1 System Integration

Integration of independently developed systems often raises crosscutting concerns; it often requires modifications on many descriptions of participating systems[3, 6, 7]. For example, assume that one builds an integrated development environment (IDE) by integrating a text editor and a compiler[6, 7]. Without AOP, descriptions for the integration concern has to appear in several places in both sub-systems; e.g., a "save" method not only writes to a file, but also needs to invoke the compiler. We will revisit this example in Section 6.

For the concreteness, we consider integration of `Bit` objects, which was originally introduced in by Sullivan et al.[3]. A `Bit` object has a boolean instance variable and methods for setting, clearing, and getting the value of the variable:

```
class Bit {
    boolean value = false;
    void set()    { value = true; }
    void clear()  { value = false; }
    boolean get() { return value; }
}
```

The integration concern is to synchronize the states of particular `Bit` pairs, which is represented by relations. A relation consists of a type (either equality or trigger) and a pair of `Bit` objects. The relations are created dynamically during program execution.

Figure 1 shows three `Bit` objects (illustrated as ovals) connected by two equality relations (illustrated as diamonds). An equality relation propagates `set` and `get` calls on the left-hand side to the right-hand side and vice versa. Therefore, when `set` is called on b_2, the top equality relation calls `set` on b_1, which in

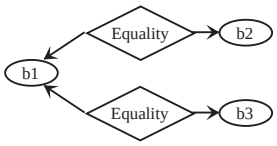

Fig. 1. Integration of `Bit`s

turn makes the bottom equality relation to call `set` on b_3. Note that the relations must not cause an infinite loop; i.e., the call on b_1 by the top equality relation should not be propagated back to b_2.

We require the following properties for implementing the equality (and other) relations for comprehensiveness, maintainability, and extensibility.

non-intrusiveness: The implementation does not require modification of the definition of `Bit`.

variability: Not only equality, but also other kinds of relations are supported simultaneously. For example, a trigger relation, which merely propagates calls on the left-hand side to right-hand side should also be used.

simplicity: When the programmer uses relations, he/she needs not consider the implementation details of the relations.

2.2 A Solution in Java with Observer Pattern

Figure 2 shows an implementation of the `Bit` integration system in Java with the Observer Pattern. The Observer Pattern is one of the GoF design patterns[8] that is to define a dependency relationship between one to many objects. When the monitored object changes its state, it notifies the depending objects.

In order to implement the `Bit` integration system, we let `Bit` objects play the `Subject` role and define `Equality` objects to represent describing equality relationships, and let `Equality` play `Observer` role so that they can propagate operations on `Bit` objects.

An `Equality` relation establishes an association by calling the `attach` method on two `Bit` objects. Propagation of the set and clear operations is achieved by inserting a call to the `change` method at the end of those methods. When the `change` is called, it calls the `update` method of the `Equality` object associated to the `Bit` object. The `update` method determines the opponent `Bit` object of the relation, determines whether it should call `set` or `clear` method by sensing the state of the changed object, and then calls the method on the opponent. The instance variable `busy` of `Equality` is an algorithmic state for avoiding cyclic calls of `update` on the same object.

With respect to the required properties presented in the last section, the implementation is intrusive because we have to modify `Bit` class. The modifications are not only at the end of each propagated operation, but also in the inheritance

```
interface Observer {
  void update(Subject s);
}

class Subject {
  List observers =
    new LinkedList();
  void attach(Observer o) {
    observers.add(o);
  }
  void detach(Observer o) {
    observers.remove(o);
  }
  void change() {
    for (Iterator iter
       = observers.iterator();
       iter.hasNext();) {
     Observer o
      = (Observer) iter.next();
     o.update(this);
  } }
}
```

```
class Equality implements Observer {
  Bit l, r;
  boolean busy;
  Equality(Bit l, Bit r) {
    this.l = l; this.r = r;
    l.attach(this); r.attach(this);
  }
  public void update(Subject s) {
    Bit b = (s == l) ? r : l;
    if (!busy) {      //to avoid
      busy = true;    //infinite loop
      if (((Bit) s).get())
        b.set(); else b.clear();
      busy = false;
  }}
}

class Bit extends Subject {
  boolean value = false;
  void set()   {value=true;  change();}
  void clear() {value=false; change();}
  boolean get() {return value;}
}
```

Fig. 2. Bit Integration System with Observer Pattern

hierarchy. This suggests that we can not apply the implementation to a class in the middle of an inheritance hierarchy. The implementation is also less variable. When we introduce a different kind of relation that is to propagate a different set of operations, the pattern forces every relation to receive notifications of all kinds of state changes, even if the change is not relevant to a specific relation.

2.3 A Solution in AspectJ

It is possible to define aspects in AspectJ that implement the above relations.

Figure 3 shows a possible definition of the equality relation in AspectJ[5]. In order to represent the state of each relation, the aspect defines an inner-class called `Relation`, which has references to the related `Bit` objects and a `busy` flag. The aspect adds a list of `Relation`s to each `Bit` object, so that the advice can find `Relation`s from a `Bit` object.

Two advice declarations capture `set` and `clear` calls, respectively, to any `Bit` object. The bodies of advice obtain a `relations` list from a target object. For each `Relation` in the list, it checks the flag and invokes the same method when the advice is not recursively executed for the same `Relation`.

[5] The definition is written by the authors who follow the outline originally presented by Sullivan, et al.

```
aspect Equality {
  static class Relation {
    Bit left, right;
    boolean busy = false;
    Bit getOpp(Bit b) {
      return b==left? right:left;
  } }
  private List Bit.relations
           = new LinkedList();

  static void associate(
           Bit left, Bit right) {
    Relation r = new Relation();
    r.left = left;
    r.right = right;
    left.relations.add(r);
    right.relations.add(r);
  }

  after(Bit b): call(void Bit.set())
                && target(b) {
    for (Iterator iter
         = left.relations.iterator();
         iter.hasNext(); ) {
      Relation r
         = (Relation) iter.next();
      if (!r.busy) {    //to avoid
        r.busy = true;  //infinite loop
        r.getOpp(b).set();
        r.busy = false;
    } }
  }
  //advice for the clear
  //method goes here
  //...
}
```

Fig. 3. An Implementation of Equality Relation in AspectJ

The static method `associate` creates a relation. When the method is called with two `Bit` objects, it creates a `Relation` object and registers it into each of the `relations` lists in the given `Bit` objects. The integrated system of `Bit`s specified in Figure 1 can be constructed by executing the following code fragment:

```
Bit b1 = new Bit(), b2 = new Bit(), b3 = new Bit();
Equality.associate(b1,b2); //connect b1 and b2
Equality.associate(b1,b3); //connect b1 and b3
```

2.4 Problems of AspectJ Solution

The AspectJ solution is better than the pure Java solution, but it still has problems.

Here, we analyze the AspectJ implementation with respect to the required properties in Section 2.1:

non-intrusiveness: `Equality` aspect is not intrusive as its pointcut and advice captures calls to `Bit` objects without modifying the class declaration of `Bit`.

variability: AspectJ allows the programmer to define relations other than `Equality` without major interference. However, such a relation cannot share the implementation with `Equality` as those different relations manage the relations among objects in different ways.

simplicity: The solution is not simple enough as it has to declare a separate inner-class for representing relations, and each advice body has to have an iteration to find all the relevant relations. The latter point would be significant when there are more advice declarations for more complicated relationships. At design level, an equality relation is an entity that encapsulates the state

(related objects and a busy flag) and the behavior (detection and propagation of method calls). It would be straightforward if a relation is modeled by an instance at the programming level. However, the AspectJ solution models the relation as an aspect declaration (for the behavior) and an instance of an inner-class (for the state).

To summarize, aspect instantiation mechanisms in AspectJ are not sufficient to straightforwardly implement concerns that affect a group of objects and have stateful behavior. As it is a natural idea to encapsulate the state and behavior in an aspect instance and a mechanism that enables to create aspect instances on a per-object-group basis is useful.

In other words, the *singleton* aspects in AspectJ are not suitable because they can create no more than one instance. As a result, the implementation would have to allocate the states in different objects, and manage a table to keep those objects.

The *per-object* aspects in AspectJ, namely `pertarget` and `perthis` aspects, are not suitable either. This is because only one per-object aspect instance is allowed to exist for each object. In order to represent relations between objects, more than one aspect instances exists for one object.

Although one may think standard protocols or APIs for managing relations could solve problems of simplicity and variability, they actually help little for achieving both. If we designed the protocols or APIs variable enough to support various usages of relations, the resulted aspects would be no longer simple as the protocols and APIs would require a number of descriptions such as iterators, unsafe type casting, subclassing, and so forth.

We do not believe that this problem is unique to large-scale system integrations. Rather, similar problems could be observed in smaller-scale systems. For example, in the AspectJ implementation of the GoF design patterns[8] by Hannemann and Kiczales[9], 6 out of 23 patterns manage the relations and their states by using tables.

3 Association Aspects

3.1 Overview

We propose an extension to the AspectJ's aspect instantiation mechanism, called *association aspects*, that allows the programmer to associate an aspect instance to a tuple of objects. Association aspects are designed to straightforwardly model crosscutting concerns like behavioral relations, which coordinate behavior among a particular group of objects.

Two basic functions support the association aspects: (1) a function to associate an aspect instance to tuples of objects, and (2) a function to select aspect instances based on the association at advice execution.

Figure 4 shows the `Bit` integration example rewritten with the association aspects. The `perobjects` modifier on the first line declares that its instance is to be associated to a pair of `Bit` objects. The following statements builds the integrated `Bits` in Figure 1:

```
aspect Equality perobjects(Bit, Bit) {
  Bit left, right;
  Equality(Bit l, Bit r) {
    associate(l, r);       //establishes association
    left = l; right = r;
  }
  after(Bit l) :
    call(void Bit.set()) && target(l) && associated(l,*){
    propagateSet(right);   //when left is called, call set on right
  }
  after(Bit r) :
    call(void Bit.set()) && target(r) && associated(*,r){
    propagateSet(left);    //when right is called, call set on left
  }
  boolean busy = false;    //indicates if the relation is active
  void propagateSet(Bit opp) {
    if (!busy) {           //call set on opp
      busy = true;         //unless it already has propagated
      opp.set();
      busy = false;
  } }
  // advice decls. for clear method go here
}
```

Fig. 4. Equality Relation with Association Aspects

```
Bit b1 = new Bit(), b2 = new Bit(), b3 = new Bit();
Equality a1 = new Equality(b1,b2);
Equality a2 = new Equality(b1,b3);
```

The `new` expressions create `Equality` aspect instances. The constructor of `Equality` associates the created instance to the given `Bit` objects.

The `associated` pointcuts in the advice declarations specify what aspect instances shall be used as the execution context of the advice bodies. The combination of pointcuts `target(l) && associated(l,*)` selects aspect instances that are associated to the current target object. The selected aspect instances serve as execution context of advice; i.e., the body of advice runs with accesses to the instance variables of the selected aspect instances.

For example, when a program evaluates `b2.set()`, aspect instance `a1` is selected by the second advice, and executes the advice body. The advice checks `busy` flag in `a1`, and calls `set` on `left`, which is bound to `b1` in `a1`.

We hereafter refer to the process that selects aspect instances and runs advice body in the context of selected instances as *advice dispatching to aspect instances*.

3.2 Properties of Association Aspects

Association aspects satisfy the three properties that are presented in Section 2.1.

non-intrusiveness: `Equality` in Figure 4 is not intrusive as well as the one in AspectJ.

variability: By combining associated pointcuts with AspectJ's abstraction mechanism such as abstract aspect and pointcut overriding, association aspects are variable. As we will see in Section 3.4, `associated` pointcuts are powerful enough to describe both symmetric and asymmetric relations. For the `Bit` integration, it therefore is possible to define an aspect that has an abstract pointcut, and sub-aspects with concrete pointcuts to specify either symmetric or asymmetric relation.

simplicity: `Equality` in Figure 4 is simple than the one in Figure 3 because association aspects hide the implementation details of relations, which are explicit in Figure 3 (e.g., the field `Bit.relations` and the use of iterator in the after advice). Moreover, composition of associated pointcuts with free variables, which will be explained in Section 3.4, avoids the duplication of advice declarations.

The following subsections explains the association and advice dispatching mechanisms in greater detail.

3.3 Creating and Associating Aspect Instances

Association aspects are declared with `perobjects` modifiers. They are defined by the following syntax:

$$\texttt{aspect } A \texttt{ perobjects(}T\texttt{,...) \{ } \mathit{mdecl} \ldots \texttt{ \}}$$

where A is the name of the aspect, T is the type of objects to be associated, and *mdecl* is the member declaration including constructor, method, variable, advice, etc.

An association aspect can be instantiated by executing a `new` A`(...)` expression in a similar manner to object instantiation. Creation of a new aspect instance also invokes a constructor for initialization. A newly created aspect instance is not associated to any objects.

The `perobjects(`T_1`,`T_2`,...,`T_n`)` modifier automatically defines an `associate` method in A. It takes n objects of type $T_1, \ldots, T_n$, and associates the aspect instance to the given objects $o_1, \ldots, o_n$. The modifier also defines a `void` A`.delete()` method, which revokes association.

In contrast to per-object aspects in AspectJ, creation and association of association aspects are explicit. This is due to the typical usage of association aspects, in which they represent explicit artifacts such as the `Equality` relations in the `Bit` integration example. When association aspects are required for objects in certain join points, it is possible to make those operations non-intrusive by defining advice as we will see in Section 3.5.

3.4 Dispatching to Aspect Instances

Semantically, dispatching advice to aspect instances is realized by trying to execute the same advice in the context of *all* aspect instances, and only the instances

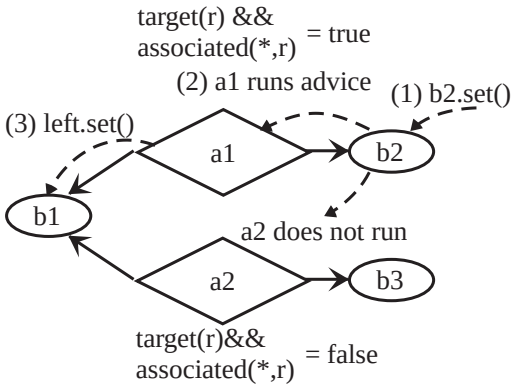

Fig. 5. Advice Dispatching to Associated Aspects

that satisfy the pointcut *actually* run the body. In order to select associated aspect instances, we provide the `associated` pointcut primitive.

Figure 5 illustrates the semantics in terms of the example presented at the beginning of the section. The evaluation of `b2.set()` creates a call join point (1). We here focus on the execution of the second advice declaration. Each aspect instance tests the pointcut. Since the pointcut is satisfied only when an aspect instance is associated to `b2` as the second parameter, `a1` is the only aspect instance to run the advice (2). The advice body propagates the call by accessing the `left` instance variable stored in the execution context, `a1` (3).

Aspect instances are ordered in undetermined order to test-and-execute an advice declaration. For around advice, the following four steps are executed. First, an aspect instance is randomly selected from all aspect instances. Second, the selected aspect instance test-and-executes the advice declaration. Third, when the aspect instance executes a `proceed` form, or the aspect instance does not match the advice declaration, a next aspect instance is selected and repeats from the second step. When the aspect instance does not execute a `proceed` form, it continues the execution without running the join point. Fourth, when there are no more aspect instances at the first step or the last part of the third step, it continues the join point.

An `associated` pointcut determines how an aspect instance is associated to objects. In an aspect declared with `perobjects(`T_1`,...,`T_n`)`, the pointcut is written as `associated(`v_1`,...,`v_n`)` where v_i is either

- a variable that is bound by another pointcut (e.g. by `target(`v_i`)`),
- an asterisk (`*`) as a wild card, or a free variable.

An additional restriction is that an `associated` pointcut has at least one bound variable in its parameter.

The pointcut `associated(`v_1`,...,`v_n`)` is evaluated to true for an aspect instance that is associated to $\langle o_1, \ldots, o_n \rangle$, if, for any $1 \leq i \leq n$, v_i is either an asterisk or a free variable, or a variable bound to o_i. The asterisks and free variables allow more than one aspect instances to match the same join point.

Note that the pointcut distinguishes parameter positions. This is useful to define directed relations that capture different events on the different sides of the relations.

Binding to Associated Objects The `associated` pointcut can bind variables to associated objects when free variables are written instead of wild cards. For example, the following declaration, which is slightly modified from the first advice declaration in Figure 4, has a free variable `r` instead of the wild card:

```
after(Bit l, Bit r) : call(void Bit.set())
  && target(l) && associated(l,r) {
  propagateSet(r);
}
```

The modified advice has the same behavior as the original one except that it binds `r` to each associated object at the second parameter position when it executes the body.

The binding feature can give shorter definitions to symmetric association aspects, which equally treat their associated objects. For example, the following single advice declaration can be substituted for the first two advice declarations in Figure 4:

```
after(Bit b,Bit o): call(void Bit.set()) && target(b)
  && (associated(b,o) || associated(o,b)) {
  propagateSet(o);
}
```

This is because the combination of `associated` pointcuts by an disjunctive operator identify aspect instances that are associated to the target object regardless parameter position, and then the binding feature binds `o` to the associated object that is not the target.

3.5 Static Advice

Association aspects can declare *static* advice, which provides similar semantics to the advice declarations in singleton aspects. When an advice declaration has a `static` modifier, pointcut matching and execution is performed exactly once regardless the number of existing aspect instances. Obviously, a static advice declaration may not use an `associated` pointcut. The execution context of static advice is the aspect-class; the advice body can only access to static (or class) variables.

The static advice declarations are typically useful for bootstrapping. In order to create a new aspect instance by using the advice mechanism, a static advice declaration should be used because there are no aspect instances at the beginning. For example, the advice in the following code creates an `Equality` instance when `callSomeMethod()` happens:

```
aspect Equality perobjects(Bit,Bit) {
 ...
 static void showAll(Bit b) { }    // empty body
 after(Bit b) :
    call(void Equality.showAll(Bit)) && args(Bit b)
    && (associated(b,*) || associated(*,b)) {
  System.out.println(this);  //this is bound to
} }                          //associated instance
```

Fig. 6. An Idiom to Enumerate Aspect Instances

```
aspect Equality perobjects(Bit, Bit) {
  static after(Bit l, Bit r) : callSomeMethod() && args(l,r) {
    new Equality(l,r); //creates an aspect instance
  }
  ...
}
```

3.6 Idioms to Find Aspect Instances

It is sometimes necessary to check if there is any aspect instance associated to a particular tuple of objects, or to do something on all aspect instances associated to a particular object (e.g., deleting all aspect instances associated to an object). Those operations can be realized by means of advice declarations with `associated` pointcuts. We therefore do not provide specific primitives for such purposes.

An example is to prevent creating no more than one `Equality` aspect instance for the same pair of objects. The next advice does the job:

```
aspect Equality perobjects(Bit,Bit) {
  ...
  Equality around(Bit l, Bit r) :
      call(Equality.new(Bit,Bit)) && args(l,r)
      && (associated(l,r) || associated(r,l)) {
    return this;
} }
```

When a program executes `new Equality(`b`,`b'`)` and there is an aspect instance a associated to $\langle b, b' \rangle$ or $\langle b', b \rangle$, the above advice returns a instead of creating new one. When there is no such an aspect instance, a new `Equality` instance will be created because the advice does not run at all.

Enumerating all aspect instances associated to a particular object can be realized by an empty static method with an advice declaration. For example, execution of `Equality.showAll(b)` in Figure 6 displays all aspect instances that are associated to `b`.

4 Implementation

The mechanisms for association aspects are implemented[6] by modifying the AspectJ compiler (`ajc`) version 1.2.0. Similar to the original compiler, it takes class and aspect declarations as inputs, and generates Java bytecode as compiled code. We first review how the original AspectJ compiler generates compiled code. We then show how the extended compiler generates code for association aspects. For readability, we present compiled code at the Java source-code level.

4.1 Compilation of Regular AspectJ Programs

AspectJ compiler translates an aspect declaration into a class, and an advice body into a method of the class, respectively[10]. Advice is executed by the inserted method calls into locations where the pointcut of the advice statically matches. Dynamic conditions in the pointcut (e.g., `cflow` and `if`) are translated into conditional statements inserted at the beginning of translated advice. The second author et al. gave a semantic model of the translation by using partial evaluation of an interpreter[11].

Consider the following (non-association) aspect definition which counts invocations of a method on a per-target-object basis:

```
aspect Counter pertarget(callSet()) {
    pointcut callSet() : call(void Bit.set());
    int count = 0;
    after() returning() : callSet() {
        count++;
}   }
```

Compilation of `Counter` aspect with `Bit` class yields the code shown in Figure 7[7]. A statement `b.set();` where `b` is of type `Bit` is translated into the following statements:

```
b._bind();          //create&associate if not yet
b.set();
b._aspect._abody0();//advice dispatching
```

The `Counter` aspect is translated into a class. The variable `count` becomes an instance variable, and the after advice becomes a method.

The `Bit` class has an instance variable `_aspect`, which keeps an aspect instance (i.e., a `Counter` object) associated to the `Bit` object. The `_bind` method creates an associated `Counter` instance for a `Bit` object if it is not yet created.

[6] The implementation is available at `http://www.komiya.ise.shibaura-it.ac.jp/~sakurai/`.

[7] Note that the code is drastically simplified from what the actual compiler generates. For readability, we inlined method calls and renamed compiler-generated methods and fields, and removed unimportant access modifiers.

```
class Bit {           //translated
  Counter _aspect;    //associated aspect instance
  boolean value;      //original instance variable
  public synchronized void _bind() {
    if (_aspect == null) _aspect = new Counter();
  }
  //definitions of set, clear and get methods
  //...
}

class Counter {                //translated
  int count = 0;               //instance variable
  public final void _abody0() {//body of the after
    count++;                   //advice
} }
```

Fig. 7. Compiled Code by AspectJ

The translated call to `set` method is surrounded by a call to `_bind` and a call to run the advice body. The latter call is realized by invoking an instance method of `Counter` class. As a result, the body of the advice is executed in the context of an associated aspect instance.

4.2 Overview of Compilation of Association Aspects

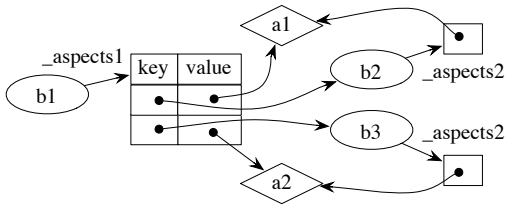

Fig. 8. Implementation of Association with Maps

Compilation of Bit Integration Example Association aspects are compiled into Java classes in a similar manner to other aspects, except for association and advice dispatching. We first show how the `Bit` class and the `Equality` aspect in Figure 4 are compiled.

The translated `Bit` class [8] has a field `_aspects1` to keep a map from `Bit` to `Equality`, and a field `_aspects2` to keep a list of Equality. The fields are

[8] We use Java 1.5 notation for collection types. `Map<`T_1`,` T_2`>` denotes the type of map objects from T_1 to T_2. `List<`T`>` denotes the type of lists of T. The syntax `for(`T v `:` e`)` s is a shorthand for looping s for each v of type T in iterator e.

of different types because of optimizations reasons, which will be explained in Section 4.3.

```
class Bit {// translated
  Map<Bit, Equality> _aspects1 = new HashMap();
  List<Equality>     _aspects2 = new ArrayList();
  ...
}
```

Those two collections are used for processing pointcuts `associated(b,*)` and `associated(*,b)`, respectively. They preserve the following invariants: when an aspect instance a associated to $\langle b_1, b_2 \rangle$, b_1.`_aspects1.get(`b_2`)` $= a$ and b_2.`_aspects2.contains(`a`)` $=$ `true`.

Note that those fields are not symmetric even though the `Equality` aspect definition treats the first and second `Bit` objects equally. This is because our compiler minimizes the collection types to reduce memory overheads. The detailed compilation strategy is described in the next section.

Figure 8 shows how the implementation represents the associations of the integrated `Bit`s in Figure 1.

Advice dispatching is translated into a loop over all key-value pairs in a map or into a loop over a list. A statement `b.set();` is translated into the following code for dispatching the two advice declarations:

```
b.set();                              //original call
for(Bit v: b._aspects1.keys()) {      //for the first
   Equality a=b._aspects1.get(v);     //after-advice
   a._abody0(b);
}
for(Equality a: b._aspects2) {        //for the second
   a._abody1(b);                      //after-advice
}
```

The two for-loops correspond to the two advice declarations. Since the first advice has the `associated(l,*)` pointcut where `l` is the target of the call, it processes all the aspect instances `a` in the `_aspects1` map of the target object, and runs the body of the advice by invoking the instance method of `a`. The code for the second advice corresponds to the `associated(*,r)` pointcut where `r` is the target of the call and processes all the aspect instances `a` in the `_aspects2` list of the target object, and runs the body of the advice by invoking the instance method of `a`.

When all parameters to the `associated` pointcut are bound, advice dispatching is translated into simple look-up in the map. For example, the parameters to the `associated` pointcut in the following advice are both bound by `args`:

```
after(Bit l, Bit r) : call(Equality.new(Bit,Bit))
    && args(l,r) && associated(l,r) {
  System.out.println("duplicated!");
}
```

Then the translation of an expression `new Equality(b1,b2)` yields the next statements subsequent to the original expression:

```
Equality a = b1._aspects1.get(b2); //for the third
if (a != null) a._abody2(b1,b2);   //after advice
```

4.3 Compilation Process

The general compilation process is slightly more complicated because we allow aspects to be associated with arbitrary numbers of (i.e., even more than two) objects, and to use wild cards at any parameter positions in associated pointcuts. Note that free variables are regarded as wild cards.

The compilation takes place in the following steps:

1. For each aspect declaration with a `perobjects` modifier, it enumerates a set of *parameter combinations* that serve as keys for dispatching advice execution.
2. It computes a set of *sequences of parameter indices*. Each sequence represents a type of data structure that records associations for specific `associated` pointcuts.
3. Based on the set of sequences, it installs fields into the associated types for recording associations, and generates methods for registering associations.
4. Finally, for each join point shadow that matches an `associated` pointcut, it inserts a code fragment for dispatching advice.

Below, we assume aspect A is declared with `perobjects`$(T_1, T_2, \cdots, T_n)$ and advice declarations with `associated` pointcuts. We write the ith occurrence of `associated` pointcut as $p_i =$ `associated`$(v_{i1}, v_{i2}, \ldots, v_{in})$ where v_{ij} is either a bound variable or a wild card. Free variables are regarded as wild cards.

We define a *parameter combination* of an `associated` pointcut as a set of indices of bound variables in the pointcut. The parameter combination of p_i is written as τ_i. When $p_i =$ `associated(v1, v2, *)`, $\tau_i = \{1, 2\}$.

For each `associated` pointcut, the compiler uses a *sequence of indices* σ_i to determine the type of the data structure for recording associations, and to generate a code fragment to dispatch advice execution. We write $|\sigma_i|$ as the length of the sequence, and $\sigma_i(j)$ as the jth index in σ_i for $1 \leq j \leq |\sigma_i|$. The sequence σ_i contains all indices in τ_i at the first $|\tau_i|$ positions; i.e., $\forall k \in \tau_i, \exists j$ such that $j \leq |\tau_i|$ and $\sigma_i(j) = k$.

Given a sequence σ_i, we use a map of type $T_{\sigma_i(1)} \rightarrow T_{\sigma_i(2)} \rightarrow \cdots \rightarrow T_{\sigma_i(|\sigma_i|)} \rightarrow A$ for recording associations. When objects $o_1, o_2, \ldots, o_{|\sigma_i|}$ of type $T_{\sigma_i(1)}, T_{\sigma_i(2)}, \ldots, T_{\sigma_i(|\sigma_i|)}$ are given, the dispatching procedure is to apply o_1, $o_2, \ldots, o_{|\sigma_i|}$ to the map, in order to obtain a reference to the associated aspect instance.

Usually, there are several possibilities to choose a set of σ_is for the given aspect. We will discuss this issue after presenting how associations are managed based on τ_is.

$$U_{ij} = \begin{cases} \texttt{Map<}T_{\sigma_i(j+1)}, U_{ij+1}\texttt{>} & j < |\sigma_i| \\ A & j = |\sigma_i| = n \\ \texttt{List<}A\texttt{>} & j = |\sigma_i| < n \end{cases}$$

```
void associate(T1 v1,T2 v2,...,Tn vn) {
   install1
   install2
   ⋮
}
```

$$install_i = \begin{cases} U_{i1}\texttt{m}_{i1} \texttt{ = v}_{\sigma_i(1)}\texttt{._aspects}_i\texttt{;} \\ U_{i2}\texttt{m}_{i2} \texttt{ = getOrCreate}_{i2}\texttt{(m}_{i1}\texttt{,v}_{\sigma_i(2)}\texttt{);} \\ U_{i3}\texttt{m}_{i3} \texttt{ = getOrCreate}_{i3}\texttt{(m}_{i2}\texttt{,v}_{\sigma_i(3)}\texttt{);} \\ \ldots \\ U_{i|\sigma_i|-1}\texttt{m}_{i|\sigma_i|-1} \texttt{ = getOrCreate}_{i|\sigma_i|-1}\texttt{(m}_{i|\sigma_i|-2}\texttt{,v}_{\sigma_i(|\sigma_i|-1)}\texttt{);} \\ addaspect; \end{cases}$$

$$addaspect = \begin{cases} \texttt{m}_{i|\sigma_i|-1}\texttt{.put(v}_{\sigma_i(n)}\texttt{, this)} & |\sigma_i| = n \\ U_{i|\sigma_i|}\ \texttt{m}_{i|\sigma_i|}\texttt{= getOrCreate}_{i|\sigma_i|}\texttt{(m}_{i|\sigma_i|-1}\texttt{,v}_{\sigma_i(|\sigma_i|)}\texttt{);} & |\sigma_i| < n \\ \qquad\qquad \texttt{m}_{i|\sigma_i|}\texttt{.add(this)} & \\ \texttt{v}_{\sigma_i(1)}\texttt{._aspects}_i\texttt{= this} & |\sigma_i| = 1 \end{cases}$$

Fig. 9. Rules for Generating `associate` Method

Managing Associations In order to maintain an association between objects and an aspect instance, the compiler actually installs fields into type declarations of associated objects, and generates an `associated` method in the aspect declaration by following the rules in Figure 9.

Given a sequence of indices σ_i for pointcut p_i, the compiler first installs a field $\texttt{\_aspects}_i$ of type U_{i1}(defined in Figure 9) into class $T_{\sigma_i(1)}$.

The associate method, shown in Figure 9, consists of the statements $install_i$, which installs an aspect instance into a sequence of maps for each σ_i.
Therefore, the compiler adds the statements of Figure 9 for each σ_i in the associate method.

$\texttt{getOrCreate}_{ij}\texttt{(}m\texttt{, }v\texttt{)}$ in the $install_i$ statements Figure 9 returns a value of type U_{ij} for key v in `Map` m if it is registered. If not, it creates an empty map of type U_{ij}, registers it in m with key v, and returns the created object.

The last line of the $install_i$ registers the aspect instance depending on the length of σ_i.

For example, assume we have an aspect definition shown in Figure 10 and the compiler uses sequences of indices $\sigma_1 = \langle 2, 3\rangle$ and $\sigma_2 = \langle 1, 2, 3\rangle$ for the first and second `associated` pointcuts, respectively. It inserts fields declarations `_aspects1` of type `Map<T3,List<A>>` into type `T2` and `_aspects2` of type

```
aspect A perobjects(T1, T2, T3) {
  before(T2 v2, T3 v3): call(* *.*(..))
      && args(v2, v3) && associated(*, v2, v3) { ... }
  before(T1 v1, T2 v2, T3 v3): call(* *.*(..))
      && args(v1, v2, v3) && associated(v1, v2, v3) { ... }
}
```

Fig. 10. An Example Aspect Definition

```
void associate(T1 v1, T2 v2, T3 v3) {
   Map<T3, List<A>> m1_1 = v2._aspects1;
   List<A>          m1_2 = getOrCreate1_2(m1_1, v3);
   m1_2.add(this);

   Map<T2, Map<T3, A>> m2_1 = v1._aspects2;
   Map<T3, A>          m2_2 = getOrCreate2_2(m2_1, v2);
   m2_2.put(v3, this);
}
```

Fig. 11. `associate` Method Generated for Figure 10

`Map<T2, Map<T3,A>>` into type `T1`. It then generates the `associate` method in Figure 11 into `A`.

Dispatching Advice Execution The compiler realizes advice dispatching by inserting a call to a method that dispatches advice execution at each join point shadow that statically matches the pointcut. The dispatching method receives l parameters from the context (i.e., the join point)[9], finds all aspect instances associated to those parameters, and calls method `_abody` on each aspect instance. The `_abody` is the method translated from the advice body, which first checks conditions due to dynamic pointcuts (e.g., `if` and type-tests), followed by the body of the advice.

For brevity, we here explain the cases for before and after advice declarations. The case for around advice is explained in the Appendix.

The rules for generating the dispatching method, which is shown in Figure 12, depend on the parameter sequence σ_i. Due to the sharing of parameter sequences among pointcuts (which will be explained later), σ_i can be longer than the number of parameters available at the join point. The dispatching method therefore consists of two parts: the first half looks up the maps by using the parameters, and the latter half iterates over all the elements in the maps.

For example, with the aspect declaration in Figure 13, the rules in Figure 12 generate the `_dispatch1` method in Figure 14.

[9] Actually, `thisJoinPoint` and other arguments used in pointcuts other than `associated` should be passed to the advice body.

```
static void _dispatch( T_{σ_i(1)} v_1, T_{σ_i(2)} v_2,...,T_{σ_i(l)} v_l) {
  if(!⟨parameterless dynamic conditions⟩) return;
  U_{i1} m_{i1}=v_1._aspects_i; if(m_{i1}==null)return;
  U_{i2} m_{i2}=m_{i1}.get(v_2); if(m_{i2}==null)return;
  ...
  U_{il-1} m_{il-1}=m_{il-2}.get(v_{l-1}); if(m_{il-1}==null)return;
  U_{il} m_{il}=m_{il-1}.get(v_l); if(m_{il}==null)return;
  for (U_{il+1} m_{il+1} : m_{il}.values()) {
    for (U_{il+2} m_{il+2} : m_{il+1}.values()) {
      ...
        for (U_{i|σ_i|} m_{i|σ_i|} : m_{i|σ_i|-1}.values()) {
          invokebody
        }
      ...
} } }
```

$$invokebody = \begin{cases} \texttt{for(}A\ \texttt{a :m}_{i|\sigma_i|}\texttt{) a._abody(v}_1\texttt{,..., v}_l\texttt{);} & |\sigma_i| < n \\ \texttt{m}_{i|\sigma_i|}\texttt{._abody(v}_1\texttt{,..., v}_l\texttt{);} & |\sigma_i| = n \end{cases}$$

Fig. 12. Rules for Generating Dispatching Method for Pointcut p_i

```
aspect A perobjects(T1, T2, T3, T4) {
 before(T1 v1, T2 v2): call(void m1(T1, T2))
    && args(v1,v2) && associated(v1,v2,*, *) {
   System.err.println("m1 with " + v1 + "," + v2);
 }
}
```

Fig. 13. An Example Aspect Definition

Sharing Maps The compiler minimizes the number of map types that record associations by sharing maps among different `associated` pointcuts. This avoids inefficiency of using redundant data structures when `associated` pointcuts use different parameters for dispatching advice.

Normally, an advice declaration can reuse a map object if its parameter sequence appears in the head of a parameter sequence of another advice declaration. Take the example in Figure 10 again. When the compiler uses the sequences $\sigma_1 = \langle 2, 3\rangle$ and $\sigma_2 = \langle 1, 2, 3\rangle$ for the two pointcuts, we have to have two maps. When $\sigma_1 = \langle 2, 3\rangle$ and $\sigma_2 = \langle 2, 3, 1\rangle$, a field in `T1` of type `Map<T3, Map<T1,A>>` is sufficient for dispatching those two advice declarations.

In order to share maps among pointcuts, the compiler computes a set of sequences S that cover all parameter combinations P in an aspect declaration by applying the algorithm in Figure 15.

```
static void _dispatch1(T1 v1, T2 v2) {
 Map<T2, Map<T3, List<A>>> m1 = v1._aspects1; if(m1==null) return;
 Map<T3, List<A>> m2 = m1.get(v2);            if(m2==null) return;
 for (List<A> m3: m2.values()) {
   for (A a : m3)
     a._abody(v1, v2);
 }
}
```

Fig. 14. Dispatching Method Generated for Figure 13

$S \leftarrow \{\}$
while $P \neq \{\}$
$\tau_{min} \leftarrow \min_{|\tau|}\{\tau | \tau \in P\}$
$P \leftarrow P \backslash \{\tau_{min}\}$
$\sigma_{max} \leftarrow \max_{|\sigma|}\{\sigma | \sigma \in S \cup \{\langle\rangle\}, \tau_{min} \text{ contains } \sigma\}$
$S \leftarrow S \backslash \{\sigma_{max}\} \cup \{append(\sigma_{max}, \tau_{min})\}$
end

Fig. 15. Algorithm to Compute a Set of Parameter Sequence S for Sharing Maps

In the algorithm, τ contains σ if $\forall k \in \{\sigma(1), \ldots, \sigma(|\sigma|)\}.\ k \in \tau$, and $append(\sigma, \tau) = \sigma'$ is a shortest sequence that has the same first $|\sigma|$ elements to σ, and has all the elements in τ; i.e., $\forall j \in \{1, \ldots, |\sigma|\}.\sigma(j) = \sigma'(j)$ and $\forall k \in \tau.\ \exists j \in \{1, \ldots, |\tau|\}.\ \sigma'(j) = k$.

When $P = \{\{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$, one of the solutions of the algorithm is $S = \{\langle 3,1\rangle, \langle 2,3\rangle, \langle 1,2,3\rangle\}$.

After computing the set of sequences S, σ_i for τ_i is selected as the shortest sequence in S whose first $|\tau_i|$ elements have all elements in τ_i.

5 Performance Evaluation

We carried out micro-benchmark tests for comparing run-time efficiency between (1) programs with association aspects, (2) programs with singleton aspects that manually manage associated states, and (3) programs with per-object aspects in AspectJ.

All benchmark tests were executed by Java HotSpot Client VM version 1.4.2, running on a PowerPC G4 1.25GHz MacOS X 10.4 machine with 512MB memory. Each execution time was measured by averaging the execution time, which is obtained through `currentTimeMillis`, of a loop that runs more than one second.

5.1 Performance of Basic Operations

We measured the costs of the basic operations, namely object creation, aspect instantiation and association, and method invocation with before advice execution. They are measured by executing programs with aspect declarations associated to n objects. The programs perform each of the following operations:

1. (**OBJ:**) create objects that can be associated to aspect instances
2. (**ASSOC:**) create an aspect instance and associate it to the n objects, and
3. (**BEFORE:**) invoke the empty method on an object.

in the aspect declarations, there are advice declarations that use the `associated` pointcut with 1 to n bound variables:

```
aspect Test perobjects(C,...,C) {
   int x1, x2, x3, x4, x5;
   Test(C o1,...,C o_n) {
      associate(o1,...,o_n);
   }
   before(C o1, ...,C o_n): callEmptyMethod()
      && args(o1,...,o_n) && associated(p_1,...,p_n) {
      x1++; x2++; x3++; x4++; x5++;
} }
```

where p_i is either `o`$_i$ or `*`.

We compare the following three aspect implementations:

AA: that uses association aspects (shown above),
SNG: that uses singleton aspects in AspectJ with inner-class objects stored in collections for associated states (see below), and
PO: that uses per-object aspect in AspectJ (namely `pertarget`). This is used only for $n = 1$.

SNG uses the same collection structures to those in the AA. For example, the SNG aspect declaration for $n = 2$ with one bound variable looks as in Figure 16.

Table 1 shows the execution times of those basic three operations for different n and different variations of `associated` pointcuts. The column p shows the parameters of the `associated` pointcuts. Since our current implementation uses the same set of map structures, OBJ denotes the time for generating one object. OBJ and ASSOC give the same figures for the same n. The rightmost column shows the relative execution times of AA with respect to SNG. We omit the cases for `(*,*,o3)` and `(*,o2,o3)` because they are identical to the cases for `(*,o2,*)` and `(o1,*,o3)`, respectively.

As we can see, AA poses at most 19% overheads compared to the manual implementation, SNG, except for the aspects associated to one object (i.e., P =`(o1)`). [10] Those numbers are reasonable as the compiled code for AA basically does the same operations to what SNG does, yet in much more concise descriptions.

[10] The relative overheads are increased from our previous measurement, which was 14%[12]. We presume that the additional overheads are introduced by the guard

```
aspect Test {
 static class Relation {
   int x1, x2, x3, x4, x5;
   C o1; C o2;
 }
 HashMap C.relations;
 static void associate(C o1, C o2) {
   Relation r = new Relation();
   r.o1 = o1; r.o2 = o2;
   HashMap m1 = o1.relations;
   if (m1 == null) {
     m1 = new HashMap();
     o1.relations = m1;
   }
   m1.put(o2, r);
 }
 before(C o1): callEmptyMethod() && args(o1, *) {
   if (o1.relations == null) return;
   for (Iterator i = o1.relations.values().iterator();
     i.hasNext(); ) {
     Relation r = (Relation) i.next();
     r.x1++; r.x2++; r.x3++; r.x4++; r.x5++;
} } }
```

Fig. 16. Declaration of an SNG Aspect

5.2 Performance of Bit Integration

We also compared the performance by running the `Bit` integration example in AA and SNG implementations (as shown in Figure 3 and 4, respectively). The benchmark programs first create 100 `Bit` objects, which are randomly connected via n equality and trigger relations, and then invoke `set` or `clear` methods on randomly selected objects for 1000 times.

The overall execution times are shown in Table 2. As seen in the rightmost column on the table, the relative execution times of AA with respect to SNG range 1.0 to 1.2, depending on the density of the relations. We conjecture the differences in the implementation details caused those differences. Especially, we presume that the major overheads come from the guard code in the AA implementation that allows safe addition and deletion of associations during advice dispatching.

---

code in the implementation of the association aspects that guarantees safe addition and deletion of associations during advice dispatching. Since the guard code adds constant overhead to each method invocation, we predict the ratio AA/SNG will not change significantly for the cases $n > 3$ though we have not measured.

	n	P	AA	SNG	PO	AA/SNG
OBJ	1		0.068	0.068	0.068	0.994
	2		0.133	0.140		0.946
	3		0.264	0.267		0.988
ASSOC	1		0.135	0.113	0.163	1.194
	2		1.762	1.719		1.025
	3		5.454	5.404		1.009
BEFORE	1	`(o1)`	0.050	0.032	0.072	1.566
	2	`(o1, *)`	0.382	0.379		1.009
		`( *,o2)`	0.326	0.322		1.012
		`(o1,o2)`	0.139	0.117		1.191
	3	`(o1, *, *)`	0.743	0.721		1.031
		`( *,o2, *)`	0.683	0.667		1.023
		`(o1,o2, *)`	0.476	0.464		1.025
		`(o1, *,o3)`	0.416	0.404		1.030
		`(o1,o2,o3)`	0.229	0.201		1.135

Table 1. Execution Times (in μsec.) of Basic Operations

n	AA	SNG	$\frac{\text{AA}}{\text{SNG}}$	n	AA	SNG	$\frac{\text{AA}}{\text{SNG}}$
10	0.345	0.330	1.046	50	3.450	3.183	1.084
20	0.525	0.504	1.041	60	21.081	18.612	1.133
30	0.804	0.742	1.084	70	62.731	57.124	1.098
40	1.338	1.197	1.118	80	347.120	287.047	1.209

Table 2. Execution Times (in msec.) of Bit Integration with n Relations

6 Expressiveness Evaluation

In this section, we illustrate a practical application program built with association aspects. We then compare the implementation of the application against the same application implemented differently, namely with Java and with pure AspectJ. The comparison illustrates the advantages of association aspects.

6.1 An Application: Integrated Development Environment

We developed a simple integrated development environment(IDE) by integrating existing application programs. Note that we used major open source software as the applications to be integrated, rather than toy programs. Figure 6.1 shows a screenshot of our developed IDE, consisting of:

- jEdit[11] text editor on the left window.
- Apache-ant building system[12] with our own simple GUI called `AntManager`, whose role is to list source files in a project description file `build.xml` and

[11] http://www.jedit.org
[12] http://ant.apache.org/

to launch the ant process with the project file, on the bottom right window, and

– our own IDE front-end that starts the `AntManager` after letting the user choose a `build.xml` file, and coordinates between the `AntManager` and jEdit.

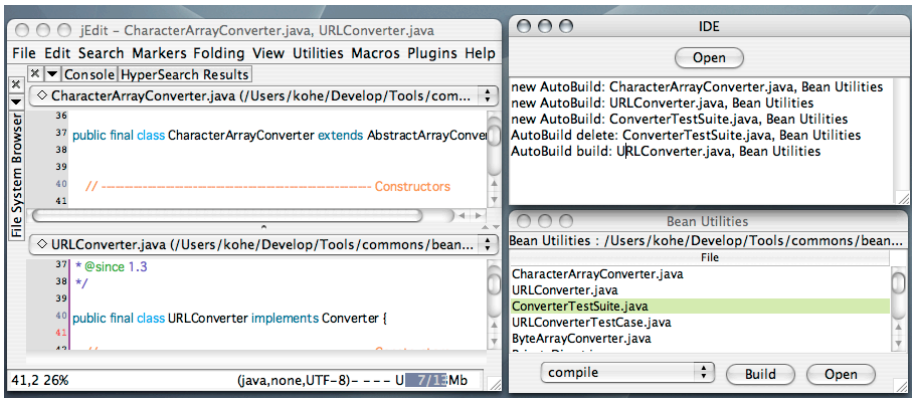

Fig. 17. A Screenshot of a Tiny IDE System built with Association Aspects

The IDE front-end uses association aspects called `AutoBuild` in order to build a project after saving a file in the jEdit text editor, and in order to save jEdit buffers before building a project.

The IDE instantiates an `AutoBuild` aspect when a user selects a source file from `AntManager` or opens a file with jEdit. The instantiated aspects are associated to a `Buffer` object in jEdit and the `AntManager` object itself. The `Buffer` object contains a copy of the text in the opened file. It has a method `save` to write the modified text to the file. The `AntManager` object is instantiated on a per definition file of a project (`build.xml`) basis. Its method `build` calls ant with the build file.

When an aspect `AutoBuild` observes a call of method `save` to an associated `Buffer` object, it invokes the method `build` of the associated `AntManager` object. Moreover, when method `build` is called, the aspect `AutoBuild` invokes `save` method to an associated `Buffer` object so that ant can build the project with latest files.

6.2 Implementation of Integrated Development Environment

The code of the aspect `AutoBuild` from the design which was described in the previous section is represented by Figure 18.

We implemented the same IDE in Java and in AspectJ without using association aspects. By comparing those implementations against the implementation in AspectJ with association aspects, we observed the following problems:

```
aspect AutoBuild perobjects(Buffer, AntManager) {
  private int busy;

  after(Buffer buffer, AntManager project):
    execution(public boolean Buffer.save(..))
    && target(buffer) && associated(buffer, project) {
    if (busy > 0) return;
    busy++;
    project.build();
    busy--;
  }

  before(final Buffer buffer, AntManager project):
   execution(public void AntManager.build())
   && associated(buffer, project)
   && target(project) && if(buffer.isDirty()) {
   if (busy > 0) return;
   busy++;
   SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      if (buffer.save(jEdit.getLastView(), null, false)) {
        buffer.setDirty(false);
        jEdit.getLastView().getEditPane()
            .getBufferSwitcher().updateBufferList();  }
      busy--;
   } });
  }
  //...advice delete AutoBuild when file is closed,
  //   and so on, goes here
}
```

Fig. 18. The Code of AutoBuild with Association Aspects

The Pure Java Implementation In the pure Java Implementation, the code for integration crosscuts the underlying applications. We had to define a generic `Listener` interface and to modify the `Buffer` and `AntManager` classes to implement the interface, and to insert code fragments to notify the `Listener` into several methods of those classes.

AspectJ Implementation without Association Aspects Figure 19 shows an implementation of `AutoBuild` in AspectJ without using association aspects. It basically follows an implementation technique discussed in Section 2.3. The implementation adds the fields that store references to `Relation` objects into `Buffer` and `AntManager` by means of inter-type declarations.

The implementation even tries to modularize the code for managing associations by declaring an abstract aspect `Association` and by letting `AutoBuild` inherit from `Association`. The reusability of this approach is however limited

Table 3. The Result of Comparing Code Size

File name	AALOC	File name	AJLOC	AALOC/AJLOC
AutoBuild.aj	50	AutoBuildAj.aj	83	0.60
		Association.aj	77	
AntFile.java	85		85	
AntManager.java	128		128	
IDE.java	132		132	
Total	395		505	0.78

as we have to define `Relation` inner class in `AutoBuild` and to explicitly write loops over `Relation` objects in each advice body.

6.3 Comparison of Code Size

Table 3 compares implementations in AspectJ with and without association aspects in terms of code size. In the table, the AALOC column shows the lines of code of the implementation with association aspects on a per-file basis. The AJLOC column shows the numbers without association aspects, in which `AutoBuild` is defined by two aspects: `AutoBuildAj.aj` and `Association.aj`.

The two implementations share the Java classes `AntFile.java`, `AntManager.java` and `IDE.java` that define GUI for ant and IDE.

As we can see, AALOC of `AutoBuild.aj` is 50 and AJLOC of `AutoBuildAj.aj` is 83. In other words, the implementation with association aspects has merely 60 percent code size when compared against the implementation without association aspects.

The difference in the code size can be observed as the additional lines in the implementation without association aspects. The comments in Figure 19 classifies the additional lines into the next three groups:

+REL: code for `Relation` class declaration,
+LOOP: code for loops to access all relations, and
+GET: code for retrieving states in `Relation` objects.

Conversely, the advantages of association aspects are to provide language constructs for those operations.

The advantages of association aspects would become more significant when we develop more practical IDEs. This is because such an IDE would have more integrated operations not only between `Buffer` and `AntManager`, but also among multiple projects (e.g., build depending projects before building a project), between a text editor and a source file versioning system, between a text editor and a compiler for handling error messages, and so on. Implementing those additional features by using association aspects would be good for assessing extensibility and adaptability of aspects. We would like to explore this in future work.

```
public aspect AutoBuildAj extends Association {
  private static class AutoBuildRelation extends Relation { //+REL
   private int busy;
   ... //following getter and setter of busy definitions    //+REL
  }                                                         //+REL
  protected Class getRelationClass() {                      //+REL
   return AutoBuildRelation.class;                          //+REL
  }
  after(Buffer buffer):
   execution(public boolean Buffer.save(..))
   && target(buffer)  {
   for (Iterator iter = (Iterator) associated(buffer, ANY); //+LOOP
      iter.hasNext();)  {  // writing explicit loop by hand //+LOOP
    AutoBuildRelation r = (AutoBuildRelation) iter.next();  //+LOOP
    AntManager project = (AntManager) r.getRight();         //+GET
    int busy = r.getBusy();                                 //+GET
    if (busy > 0) return;
    r.setBusy(busy+1);
    project.build();
    r.setBusy(busy-1);
 } }
  ... //following other advice definitions
}
```

Fig. 19. The Code of AutoBuild by the Original AspectJ

7 Discussion

7.1 Comparison with Eos

As the work on the association aspects is based on the work on Eos[4], we here discuss the difference in detail.

The most notable difference is that Eos implicitly uses the current target object when selecting aspect instances at advice execution. In contrast, association aspects can use arbitrary objects that are explicitly specified by pointcuts. The mechanism in Eos is less flexible for the following situations: (1) when aspect instances should be selected by using a non-target object; e.g., when advising a call to a class method, and (2) when aspect instances should be selected by using more than one object; e.g., when a security concern is to prevent method calls from object A to B, it can be realized by an aspect instance associated to A and B. When a call from A to B happens, all the aspect instances associated to B run an advice body in Eos, even though the caller object A could be used for selecting aspect instances.

Both association aspects and Eos can distinguish roles of associated objects. Eos, however, distinguishes by introducing additional role constructs around advice declarations, which might make it difficult to reuse aspects. For example, even though `Trigger` and `Equality` aspects in Section 2.1 only differ in what

objects should be used at advice dispatching, the declarations in Eos have different program structures as the former has to enclose advice declarations in a role construct. Since association aspects distinguish roles of objects by the parameter positions in the `associated` pointcuts, the declarations of those aspects can only differ in the pointcuts. Our approach, in which advice dispatching is governed by pointcuts, would fit the other language features in AspectJ, as it usually reuses aspects through the abstraction mechanisms of pointcuts (i.e., the named pointcuts and the abstract pointcuts).

Both Eos and association aspects should be careful about the performance penalty for the objects with no associated aspect instance. For the `Bit` integration example, a `set` call to a `Bit` object that has no associated `Equality` instances should not have significant overhead. On this regard, there are two possible dimensions to the overhead.

The first is the number of aspect instances. A naive implementation (which is called the first work-around[4]) would significantly degrade its performance to look up a system-wide table of aspect instances. Both Eos and association aspects avoid this problem by having a list of associated aspects in each object.

The second is the number of advice declarations that statically match to the call. Association aspects would linearly degrade the performance as each advice declaration adds a getting a field and null checking into the method call expression. Eos avoids this problem by having a list of thunks for each method call expression. However, the approach in Eos requires more memory and more operations for associating/unassociating aspect instances.

Those differences in implementation would result in the difference of performance characteristics. However, we would need more programs written with association aspects in order to carry out quantitative comparison. This is because the difference of performance depends on the number of advice declarations at a join point shadow, and the number of join point shadows that are advised by different set of aspect instances.

7.2 Other related work

Prior to the proposal of AOP, there have been studies on language mechanisms that support evolution of collaborative behavior for object-oriented languages namely the contracts by Helm and Holland[13], and the context relations by Seiter et al.[14, 15]. Those languages rely on different mechanisms from the pointcut and advice. Although there are many commonalities between those language mechanisms and association aspects, there are also differences when compared closer. For example, a context relation can be associated to an arbitrary number of objects while an association aspect can be associated to at most one object for each parameter position. Conversely, an association aspect can be associated to an object pair in the same type, which does not seem to be possible in the context relations.

There are AOP languages that have similar mechanisms to association aspects, namely CarsarJ proposed by Mezini and Ostermann [16–18] , EpsilonJ proposed by Tamai, et al.[19] and ObjectTeams proposed by Herrmann, et al.[20,

21]. Those language models can support integration concerns by using role objects and collaboration contexts.

In CaesarJ, a `cclass` object corresponds to an aspect instance, which can be instantiated with several `cclass` instances that wrap objects as role member. However, CaesarJ has no mechanism to associate a `cclass` instance to a group of objects and to find associated instances. Supporting integration concerns would need manual management of wrapper instances. A more recent version of CaesarJ supports variable management implementations by a mixin-like reuse mechanism.

EpsilonJ and ObjectTeams have a construct to define a context that encloses several role definitions. A context (or a team in ObjectTeams) can be instantiated explicitly and can bind a role to an arbitrary object. EpsilonJ realizes one to many relations by introducing a mechanism that broadcasts calls to all role objects of a specified type in a context, ObjectTeams can generalize the mechanism by using an abstract team.

JAsCo[22] is an aspect-oriented language for component-based software development. JAsCo introduces new language constructs such as a hook and a connector. Although they have different granularity from the module system of the association aspects (or AspectJ), we think that association aspects are useful to connect(integrate) existing components as well as JAsCo.

Ostermann et al. proposed an expressive pointcut language ALPHA based on logical queries over dynamic properties of a program execution[23]. Unlike other extensible AOP languages that can query over static structures of a program, ALPHA is so powerful that it can define pointcuts that examine a past state in a program execution with individual object references. As a result, it is possible to write a pointcut like "when called `set` or `clear` to a target `bit1`, there was a call `associate(bit1,bit2)` in the past, then bind the second parameter to `bit2`". However, it is not clear whether such a pointcut can be as efficiently implemented as association aspects.

Colman and Han developed the ROAD framework[24] using association aspects for defining a coordination system that manages an organizational system. In the ROAD framework, objects are modeled as the roles in a specific organization. Association aspects act as the stateful contract between role objects. When an association aspect picks up a message between the role objects, the advice of the aspect coordinates the role objects

The current version of association aspects is implemented by modifying the `ajc` compiler[10]. There is another AspectJ compiler named `abc` developed by de Moor, et al. [25]. `abc` is an extensible compiler for implementing new language constructs such as association aspects. We expect that association aspects for the `abc` compiler can be achieved by applying the same compilation strategy described in Section 4.

8 Conclusion

We presented association aspects as an extension to in AspectJ. They are based on the notion of instance-level aspects in Eos[4], and extended with the pointcut-based advice dispatching mechanisms that enable flexible yet concise descriptions of aspects whose instances are associated to more than one object. As a result, the association aspects can give straightforward representations of crosscutting concerns that have stateful behavior with respect to a particular group of objects.

We developed a compiler for association aspects by modifying the AspectJ compiler (`ajc`). The compiler employs an optimization strategy that reduces the number of data structures. The benchmark tests exhibited that the slowdown factors of the programs using association aspects with respect to the regular AspectJ programs are 1.0 to 1.2.

As an application of association aspects, we developed an tiny IDE by integrating existing applications in a non-intrusive way. Although this is merely one particular example, we observed that the use of association aspects reduced the code size of the core integration aspect in the IDE to approximately 60% from the one defined without association aspects. Our future plan is to quantitatively evaluate association aspects by using software metrics other than code size. In particular, evaluation criteria used for comparing GoF Design Pattern implementations in Java and AspectJ[26] would be useful.

Bridging between design level concepts and association aspects at implementation level is also left for future work. Association aspects would be suitable vehicle to implement many design level concepts such as relation objects in UML, roles in the collection designs and composites of concepts in CoCompose[27]. It would be useful to investigate methodologies to design these concepts by assuming association aspects, and to derive proper implementations from those concepts.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments that helped us to clarify discussion and to fix English problems in an early drafts of the paper.

We also thank Kevin Sullivan and Hridesh Rajan for the detailed information on Eos, Alan Colman for the feedback from his experience in using association aspects, and the members of the TM-seminar and the Kumini project at University of Tokyo and the members of Komiya's Laboratory at Shibaura Institute of Technology for valuable comments and suggestions.

References

1. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, Springer-Verlag GmbH (2001) 327–353

2. Ossher, H.: Multi-Dimensional Separation of Concerns: The Hyperspace Approach. In: Proceedings of Software Architectures and Component Technology. (2000)
3. Sullivan, K., Gu, L., Cai, Y.: Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ. In: Proceedings of the 1st international conference on Aspect-Oriented Software Development, ACM Press (2002) 19–26
4. Rajan, H., Sullivan, K.: Eos: Instance-Level Aspects for Integrated System Design. In: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2003) 297–306
5. The AspectJ project at Eclipse.org: http://www.eclipse.org/aspectj/.
6. Sullivan, K.: Mediators: Easing the Design and Evolution of Integrated Systems. PhD thesis, Department of Computer Science, Unversity of Washington (1994) publichsed as TR UW-CSE-94-08-01.
7. Sullivan, K.J., Notkin, D.: Reconciling Environment Integration and Software Evolution. ACM Trans. Softw. Eng. Methodol. **1** (1992) 229–268
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1995)
9. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2002) 161–173
10. Hilsdale, E., Hugunin, J.: Advice Weaving in AspectJ. In: Proceedings of the 3rd international conference on Aspect-Oriented Software Development, ACM Press (2004) 26–35
11. Masuhara, H., Kiczales, G., Dutchyn, C.: A Compilation and Optimization Model for Aspect-Oriented Programs. In: Proceedings of the 12th International Conference Compiler Construction 2003, Springer-Verlag GmbH (2003) 46–60
12. Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., Komiya, S.: Association Aspects. In: Proceedings of the 3rd international conference on Aspect-Oriented Software Development, ACM Press (2004) 16–25
13. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In: OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (1990) 169–180
14. Seiter, L.M., Palsberg, J., Lieberherr, K.J.: Evolution of Object Behavior using Context Relations. In Garlan, D., ed.: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, San Francisco, ACM Press (SIGSOFT) (1996) 46–57
15. Seiter, L.M., Palsberg, J., Lieberherr, K.J.: Evolution of Object Behavior using Context Relations. IEEE Transactions on Software Engineering **24** (1998) 79–92
16. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: AOSD '03: Proceedings of the 2nd international conference on Aspect-Oriented Software Development, New York, NY, USA, ACM Press (2003) 90–99
17. Mezini, M., Ostermann, K.: Integrating Independent Components with On-Demand Remodularization. In: OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, ACM Press (2002) 52–67

18. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of Software Engineering, New York, NY, USA, ACM Press (2004) 127–136
19. Tamai, T., Ubayashi, N., Ichiyama, R.: An Adaptive Object Model with Dynamic Role Binding. In: ICSE '05: Proceedings of the 27th international conference on Software engineering, New York, NY, USA, ACM Press (2005) 166–175
20. Veit, M., Herrmann, S.: Model-View-Controller and Object Teams: A Perfect Match of Paradigms. In: AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2003) 140–149
21. Herrmann, S., Hundt, C., Mehner, K., Wloka, J.: Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation. In: DAW '05: Dynamic Aspects Workshop (held in conjunction with AOSD 2005). (2005) 93–101
22. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In: AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2003) 21–29
23. Ostermann, K., Mezini, M., Bockisch, C.: Expressive Pointcuts for Increased Modularity. In: ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming. (2005)
24. Colman, A., Han, J.: Coordination Systems in Role-based Adaptive Software. In: Proceedings of the Seventh International Conference on Coordination Models and Languages (COORDINATION '05) (Lecture Notes in Computer Science, Vol. 3454), Springer (2005) 63–78
25. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: An extensible AspectJ compiler. In: AOSD '05: Proceedings of the 4th international conference on Aspect-Oriented Software Development, New York, NY, USA, ACM Press (2005) 87–98
26. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: Modularizing Design Patterns with Aspects: A Quantitative Study. In: AOSD '05: Proceedings of the 4th international conference on Aspect-Oriented Software Development, New York, NY, USA, ACM Press (2005) 3–14
27. Wagelaar, D., Jonckers, V.: A Concept-Based Approach to Software Design. In: Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA 2003). (2003)

Appendix

Generating Code of Dispatching with Around Advice

The compilation rules of around advice are slightly different from those of before and after advice due to the `proceed` mechanism in around advice. Figure 20 shows the skeletons of the methods and an auxiliary class, namely the `_dispatch` and `_abody` methods and `_Closure` class.

When an around advice is to run, instead of directly running the advice body, the compiled code first creates a `_Closure` object with a list of associated aspects that match the pointcut.

n	AROUND	BEFORE	AROUND-BEFORE
0	0.786	0.029	0.757
1	2.016	0.327	1.689
25	11.986	3.578	8.408
50	24.103	6.948	17.155
75	39.369	10.268	29.101
100	56.223	13.642	42.581

Table 4. Execution Times (in μsec.) of Around Advice

The `_Closure` object serves as a continuation of advice body. When called, it runs the advice body in the context of the next aspect instance, or performs the original operations of the join point.

Assume the declaration of aspect `A` in Figure 21. As shown in Figure 22, the Compiler generates the `_dispatch1` and `_abody1` methods in to class `A` and an auxiliary class `_Closure`.

The compiler replaces every call to `m` with a called to `_dispatch1`; which in turn runs the body of advice in the context of an aspect instance or runs method `m` when no more matching aspect instances are found. Note that the former case creates a new `_Closure` object for handling `proceed` in the advice body. This is needed to cope with AspectJ's language design that allows around advice declarations to call `proceed` more than once.

Performance of Around Advice The implementation of around advice in association aspects has some overheads when compared against before advice. The overheads include 1) collecting n aspect instances, and 2) dispatching n closures with `proceed`. Table 4 illustrates differences of execution times between around and before advice with n aspect instances. Those figures are insensitive to the number of associated objects and the number of bound parameters in `associated` pointcuts.

From the figures on the table, we can approximate the overhead of around advice execution by the following formula:

$$AROUND(n) = 0.375n + 0.757 + BEFORE(n)$$

where $AROUND(n)$ and $BEFORE(n)$ are execution times of around and before advice with n instances, respectively.

This suggests that the around advice has overheads of approximately 0.757 microseconds for each join point and 0.375 microseconds for running an advice body in the context of an aspect instance.

```
static T_jp _dispatch( T_σi(1) v1, ...,T_σi(l) vl) {
  if(!⟨parameterless dynamic conditions⟩) return _jp(v1,...,vl);
  U_i1 m_i1=v1._aspects_i; if(m_i1==null)return _jp(v1,...,vl);
  U_i2 m_i2=m_i1.get(v2); if(m_i2==null)return _jp(v1,...,vl);
  ...
  U_il m_il=m_il−1.get(vl);
  if(m_il==null)return _jp(v1,...,vl);
  List as = new ArrayList();
  for (U_il+1 m_il+1 : m_il.values()) {
    ...
      for (U_i|σi| m_i|σi| : m_i|σi|−1.values()) {
        collecting
      }
    ...
  }
  return new _Closure(as, 0).run(v1 ,···,vl);
}


T_jp _abody(T_σi(1) v1, ..., T_σi(l) vl, _Closure c) {
   if (!⟨dynamic conditions⟩) return _jp(v1,...,vl);
   //statements in the advice body...
   //proceed are translated to c.run(...)
}

class _Closure {
   List as; int i;
   _Closure(List as, int i) { this.as =as; this.i =i; }
   T_jp run(T_σi(1) v1,...,T_σi(l) vl) {
      if(i < as.size()) {
         return ((A)as.get(i))._abody(v1, ..., vl, new _Closure(as, i+1));
      } else { return _jp(v1,...,vl); }
} }
```

$$collecting = \begin{cases} \texttt{as.addAll(m}_{i|\sigma_i|}\texttt{);} & |\sigma_i| < n \\ \texttt{as.add(m}_{i|\sigma_i|}\texttt{);} & |\sigma_i| = n \end{cases}$$

Fig. 20. Code for Around Advice Dispatching and Body

```
aspect A perobjects(T1, T2, T3, T4) {
 int around(T1 v1, T2 v2): call(int m(T1, T2))
    && args(v1, v2) && associated(v1,v2,*,*) {
   return proceed(v1, v2);
} }
```

Fig. 21. An Example Aspect with an Around Advice Declaration

```
static int _dispatch1(T1 v1, T2 v2) {
 Map<T2, Map<T3, List<A>>> m1 = v1.aspects1;
                                    if(m1==null) return m(v1,v2);
 Map<T3, List<A>> m2 = m1.get(v2);  if(m2==null) return m(v1,v2);
 List as = new ArrayList();
 _Closure c = new _Closure(as, 0); //create a closure
 for (List<A> m3: m2.values())      //collect all matching
    as.addAll(m3);                  //   aspect instances
 return c.run(v1,v2);               //run the first advice
}

static int _abody1(T1 v1, T2 v2, _Closure c) {
  return c.run(v1, v2);             //the body of advice
}

class _Closure {
  List as; int i;
  _Closure(List as, int i) { this.as = as; this.i = i; }
  int run(T1 v1, T2 v2) {
    if (i < as.size()) {
      A a = as.get(i);                  //run advice body
      return a._abody1(v1,v2, new _Closure(as, i+1));
    } else {
      return m(v1,v2);
    }
} }
```

Fig. 22. Generated Methods and Class for Around Advice