Prototyping and Composing Aspect Languages
using an Aspect Interpreter Framework

Wilke Havinga, Lodewijk Bergmans, and Mehmet Aksit

University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
{w.havinga,l.m.j.bergmans,m.aksit}@ewi.utwente.nl

Abstract. Domain specific aspect languages (DSALs) are becoming more popular because they can be designed to represent recurring concerns in a way that is optimized for a specific domain. However, the design and implementation of even a limited domain-specific aspect language can be a tedious job. To address this, we propose a framework that offers a fast way to prototype implementations of domain specific aspect languages. A particular goal of the framework is to be general enough to support a wide range of aspect language concepts, such that existing language concepts can be easily used, and new language concepts can be quickly created.

We briefly introduce the framework and its underlying model, as well as the workflow used when implementing DSALs. Subsequently, we show mappings of several domain specific aspect languages to demonstrate the framework. Since in our approach the DSALs are mapped to a common model, the framework provides an integrating platform allowing us to compose programs that use aspects written in multiple DSALs. The framework also provides explicit mechanisms to specify composition of advices written in multiple DSALs.

1 Introduction

The benefits of using domain specific aspect languages (DSALs) are widely recognized [8, 16, 24]. In fact, the idea of expressing each crosscutting concern using a dedicated domain-specific language was at the very heart of the first proposals called “AOP” [12].

However, designing and implementing DSALs can be a tedious job. For example, each aspect language has to define under which circumstances an aspect should influence the program, and implement mechanisms to facilitate this (e.g. using bytecode weaving).

In addition, most applications will need to express concerns from different problem domains, making it desirable to write programs using multiple DSALs. That way, each DSAL could be used to effectively address the concerns within its specific domain.

It is not trivial to compose aspects expressed in several DSALs however, as each language typically constructs its own model of the program; unless a lot of care is taken, the effects of one aspect may not be reflected in the models constructed by other DSALs. In addition, aspects written in several DSALs may interact with each other, possibly in undesirable ways (depending on the situation).

Our paper contributes the following to address these problems:
We propose an aspect interpreter framework that can be used to prototype domain-specific aspect languages. As our framework supports a wide range of aspect language concepts, it can be used to prototype diverse DSALs in a reasonable amount of time, as we will show in section 3.

Using our approach, aspects written in several (domain-specific) languages are mapped to a common model. As a result, we can compose applications that are written using multiple DSALs, as we will show in section 4.1.

The framework provides explicit mechanisms to specify composition of advices, even if advices are written in several DSALs. This is discussed in section 4.3.

In this paper, we show implementations of only three DSALs. However, our work is based on a thorough study of aspect oriented languages [19], as well as the modeling of their possible implementation mechanisms using an interpreter, as presented in [7].

In the next section, we briefly introduce the framework itself. Section 3 presents more details about the framework by showing the implementations of several DSALs using our framework. Section 4 discusses the composition of aspects written in multiple DSALs, including specifications to resolve the interactions between aspects. Section 5 discusses several design and implementation considerations related to our framework. We discuss related work in section 6, and conclude the paper in section 7.

2 JAMI - an aspect interpreter framework

One of the defining features of AOP languages is the support for "implicit invocation" of application behavior. That is, behavior can be invoked without an explicit reference (such as a method call statement) being visible in the (source) code at the point of invocation. Implicit invocation is a key feature of the interface between the base program and the aspect program. The framework to model aspect language mechanisms we present in this paper is strongly based on implicit invocation as the connection between the base program and the aspects (advices).

In this section, we briefly discuss the concepts used in the aspect language domain, based on a reference model proposed in [19]. We then propose a framework that provides common implementations of these concepts, while supporting variations on these concepts found in different aspect languages. The general design and architecture of the framework was first proposed in [7] and [11]. Finally, we briefly outline the workflow used to prototype DSALs using this framework.

2.1 Common aspect language concepts

Aspect languages must first of all support the concept of pointcuts. Pointcuts define the circumstances under which an aspect influences a program – for example, at certain locations (such as entering a particular method) or under particular runtime conditions (e.g., only when a variable $x$ equals 5). Pointcuts can be seen as predicates or conditions over the execution state of a program. The execution state may, in a broad sense, include information about the call stack, objects, or even the execution trace and structure of the program. As pointcuts may match at several places or moments during the execution of a program, the concept of joinpoints is used to model references to the relevant
execution state (e.g., which method is being intercepted) whenever a pointcut matches. Pointcuts can be bound to *advices*, which may add to or replace parts of the original program behavior and/or its runtime state. Advices can use the joinpoint information to adapt their behavior based on the current runtime situation. Finally, *bindings* specify how pointcuts and advices are connected and grouped into modules (usually called aspects). In addition, bindings are also used to bind “aspect state” – data stored by aspects, such that it can be shared between advices (i.e., similar to sharing state between methods by using instance variables).

2.2 Framework implementation

Aspect languages adopt varying implementations of the concepts listed above. We provide a framework that implements the behavior of these high-level concepts, and allows for their refinement to facilitate specific language implementations.

![Fig. 1. The Java Aspect Metamodel Interpreter - an overview](image)

Figure 1 shows a global overview of our framework, called the Java Aspect Metamodel Interpreter (JAMI) [1]. Basically, this framework enforces the high-level structure and control flow of aspects, while providing implementations of common concepts at an abstraction level that is appropriate when prototyping DSALs – as we intend to demonstrate in section 3. By enforcing a fixed high-level control flow, our framework provides a common platform that enables composition of aspects written in multiple DSALs, as we will show in section 4. To provide the flexibility required to model features of particular languages, each concept can be either instantiated in a dedicated configuration of framework elements, or refined (extended) when necessary. JAMI provides many of the common implementations found in different aspect languages.

**Control flow** We briefly discuss the high-level control flow within JAMI. In principle, the base program (a normal Java application) runs as it would without the interpreter. However, the *interception mechanism* (see figure 1) intercepts the control flow at any point that is of potential interest to the aspect interpreter. Our current implementation uses a regular AspectJ aspect to intercept *all* method calls and field assignments. Apart from intercepting method calls, the mechanism keeps track of *context* information that may be of interest to the framework. Currently, it keeps track of the call stack, senders, targets, and method signatures of all calls on the stack, as well as field assignments.
Upon interception of the control flow, the mechanism creates a joinpoint object representing the current joinpoint. A refinement class exists for each different joinpoint type, such as MethodCallJoinpoint, MethodReturnJoinPoint or AssignmentJoinpoint. Each of these joinpoint objects keeps a reference to the relevant context information - e.g., the method that was executing upon interception, etc.

Subsequently, each pointcut registered with the aspect interpreter is evaluated against the current joinpoint (see figure 1). As indicated before, pointcuts are basically conditions that either match or do not match a particular joinpoint. Thus, the main pointcut class consists of only an evaluate method, which returns true or false based on whether it matches the current joinpoint. Refinement classes are provided for many common pointcut conditions; new ones can be created if necessary to implement DSAL-specific pointcut types. For example, we provide pointcuts that match based on the type of joinpoint, method signature, or target object type. In addition, there are pointcut classes that can combine other pointcuts using regular logic expressions (and, or, not). Many concrete examples of implemented pointcut conditions will be shown in section 3.

One or more pointcuts can be associated to one or more advices using bindings (see figure 1). For each matching pointcut, the interpreter looks up the corresponding advice through these bindings. Advice can be expressed in terms of elementary advice “building blocks” provided by JAMI, which allows the expression of many common types of advice without creating custom implementations for each advice. In addition, advice can be expressed using normal base code, when necessary. Advices may also want to share state among each other, or among different executions of the same advice. Therefore, we also provide bindings to aspect state; this will be discussed in more detail in section 3.1.

When several pointcuts match at the same joinpoint, the order of advice execution has to be resolved. This issue is discussed in detail in section 4.

Further details about the implementation, we refer to the JAMI manual and API documentation available on the JAMI website [1].

2.3 Prototyping DSALs: general workflow

We briefly describe the steps involved in prototyping a DSAL using JAMI.

First of all, we define a grammar that can conveniently express the domain concepts of a particular DSAL, as well as the relations between those concepts. Next, a parser is needed - using a parser generator is typically the most convenient way to implement this (we use Antlr, but any Java-based parser generator could be used). Subsequently, we convert the abstract syntax trees (ASTs) obtained from the parser to an object-based version, such that the domain concepts are semantically represented by objects. This conversion can be implemented using handwritten code, or by using generated “tree walkers”. The final step is to convert the object-based AST representation of domain concepts to JAMI elements. We currently implement this conversion using handwritten code.

Once an aspect written in a DSAL is (automatically) converted to JAMI elements in the way described above, the JAMI interpreter framework can run the aspect as part of a normal Java application. Typically, we write an explicit instruction to load and deploy
the aspect at application startup. Once the aspect is deployed, JAMI ensures that the
aspect behavior is called at the appropriate times, as described in the previous section.
In the following section, we introduce several examples to demonstrate the frame-
work in detail. We focus on expressing each example using JAMI elements, showing
the object structure of the JAMI representations of each aspect. The full examples,
including parsers and code that executes the mapping steps as described above, can be
downloaded from the JAMI website.

3 Features of JAMI, demonstrated by example

In this section, we show 3 aspect languages optimized for a specific task, implemented
using the Java Aspect Metamodel Interpreter. We first introduce a running example that
we will use to demonstrate each language.

![Diagram of a simple word processor application]

Figure 2 shows the UML class-diagram of a simple word processor application.
Within this application, class Document defines some methods to modify a document
(addLine() and setContent()), a method to obtain the document content (getContent), as
well as a method that counts the current number of words in the document (wordCount).
In the following subsections, we extend this example using aspects written in several
domain-specific aspect languages. These extensions will allow us to: (1) create an au-
tosave mechanism using a modularized version of the decorator pattern, (2) synchronize
access to documents, such that multiple threads can access its content at the same time
(for example, to run a background spellchecker), and (3) cache the results of expensive
method calls, as long as variables on which the method depends are unchanged.

3.1 A domain-specific language for the decorator pattern

Suppose we want to add autosave behavior to our word processing application. We can
implement this using the decorator pattern by defining a class AutoSaveDocument.
This class implements the same methods as class Document, but adds the behavior to
save any changes made to the document (e.g., to a file), before forwarding method calls
to the original document object - see figure 3.

Listing 1 shows how we could use this decorator class:

```java
1 public class WordProcessor {
2     Document doc;
3     AutoSaveDocument autoSaveDoc;
```

In fact, the object diagrams in this paper are based on observing (using the Eclipse debug-
ger) the actual runtime JAMI ASTs, which where automatically converted from the DSAL
notations.
There are two issues with this code. (1) When writing this in plain Java, we can still make calls to the object that is being decorated (also called the *decoratee*) - see line 10. This is almost certainly unintended, as the behavior of the decorator is not invoked this way. (2) Part of the code dealing with the decorator pattern is visible in the client (class *WordProcessor* in this example) – it is not fully modularized. We experiment with simple domain-specific extensions to Java to solve these issues.

**Enforcing the decorator pattern** We start with the issue of enforcing the decorator pattern. Once a *decorator* is associated with a *decoratee* (listing 1, line 7), all subsequent calls should be made to the *decorator*. We define a small domain specific aspect language (DSAL) to enforce this - by automatically forwarding calls to a *decoratee* object to the *decorator*. In the first version of our language, a program in this DSAL defines which classes may act in the *decorator* and *decoratee* roles, respectively, see listing 2:

```java
    decorate: Document -> AutoSaveDocument
```

We take the specification in listing 2 to mean the following: objects of type *Document* may be decorated by objects of type *AutoSaveDocument*. However, we do not want to simply decorate every object of type *Document*. Doing so would defeat the purpose of the decorator design pattern, which is used to decide *dynamically* which objects should be decorated. Therefore, our first implementation will automatically infer the *decorator-decoratee* relationship between objects from the occurrence of constructor calls such as shown in listing 1, line 7. That is, an association is established upon calling a constructor of a class that is indicated to be a *decorator* in our DSAL specification (listing 2), of which the first argument is of the corresponding specified *decoratee* class.

**Mapping to JAMI** To think of the above language in terms of an aspect language definition, we consider the aspect language concepts as shown in figure 1. The main
The task of programs written in this language is to intercept calls to *decoratee* objects, and forward them to the associated *decorator* object. In aspect terminology, the interception specification can be seen as a *pointcut*, whereas the forwarding part is an *advice* specification.

For the above pointcut/advice definition to make sense, the aspect program needs to know which objects are *associated* in the roles of *decoratee* and *decorator*, i.e. we need to establish and store this association as part of the aspect.

Therefore, to create the association, we intercept (using another pointcut) calls to the constructor of the type acting as *decorator*. The connected *advice* is to create an association between the object being created (the *decorator* object), and the first argument of the constructor call (which we assume to be the *decoratee* object, as discussed above). This association is stored as “aspect state”, such that it can be shared between advices.

We now explain the mapping to JAMI in detail, by showing object diagrams that represent the aspect program given in listing 2. The object structures shown here consist largely of elements (classes) predefined by the JAMI framework. All these elements are described in the JAMI API documentation and reference manual, which is available online [1].

![Diagram](image)

**Fig. 4.** Bindings between the parts of a decorator aspect

Figure 4 shows an aspect definition expressed using JAMI elements. The figure contains two subviews specifying “selector-advice-bindings”, constructs connecting a particular pointcut to a particular advice. These subviews refer to figures 5 and 6, which we will discuss shortly. We model the grouping of several pointcut/advice combinations into a single aspect module to facilitate the sharing of state (data) between related advices. An “aspect” module can define its own variables, which can have different kinds of instantiation policies. For example, a “singleton” policy means that there is one instance of the variable for the entire program, a “per object” policy means there is one instance of the aspect variable for each target object (where the current target object depends on the join point context), etc. In JAMI, each variable can have its own instantiation policy, i.e. even variables within the same aspect module can have different instantiation policies.

Instantiation is usually implicit: new instances are automatically created when needed (using the default constructor of the specified variable type), i.e. on first use in a particular context. However, explicit instantiations are possible as well, as we will show in this example. Figure 4 shows two “aspect variables”, *decorator* and *decoratee*, which have a ‘per association’ instantiation policy. This policy means that when the value of
one variable is known, the value of the other one can be retrieved through it. In our
implementation, this is done through the hashtable shared by the two instantiation policy
objects. The associations themselves have to be explicitly instantiated, as we will show
in the “AssociationCreation” subview (figure 5). The implementation of “association
variables” is similar to the concept of “Association aspects” as proposed in [21].

Fig. 5. Association by intercepting constructor calls

Figure 5 shows how associations between \textit{decoratee} and \textit{decorator} objects are cre-
ated. The top element, a SelectorAdviceBinding connects a pointcut definition (on the
left) to an advice definition (on the right). The pointcut definition (called “join point se-
lector” in JAMI) in this case consists of several “primitive” selection criteria, which can
be combined using the standard logic operators (i.e., \textit{and}, \textit{or}, \textit{not}). The pointcut in this
figure selects (a) constructor calls, (b) for which the type of the created object is “Au-
toSaveDocument”, and (c) the constructor being called has 1 argument of type “Docu-
ment”. This pointcut is connected to a custom advice class – extending the framework
– which explicitly creates the association between \textit{decorator} (the constructed object)
and \textit{decoratee} (the value of the first argument). The class “AssociateDecoratorAdvice”
(consisting of ca. 20 lines of code) is the only extension to JAMI needed to implement
our decorator aspect language.

Fig. 6. Forwarding calls from decoratee to decorator

To finish the example, figure 6 shows the definition of the call forwarding part of the
aspect. The pointcut in this figure (on the left) selects method calls for which the target
object occurs as a \textit{decoratee} value in the association table, \textit{except} those for which the
caller object is a \textit{decorator}. Omitting this exception would make it impossible to reach
the \textit{decoratee} object at all, and would in addition lead to an infinite loop on the first
call to the `decorator` object (as the `decorator` will at some point call the `decoratee`, see figure 3).

The advice attached to this pointcut is composed of 4 predefined JAMI elements. The first, `BindObjectToVariableAction`, binds the value of the current target value to the variable `decoratee`. Next, the instruction `SetTargetObjectFromVariableAction` modifies the target object of the call to the value of the `decorator` variable, which can now be looked up through the corresponding aspect variable. Finally, we instruct the interpreter to execute the method call on the current target (in this case, set by the previous instruction). As we did not modify the signature of the called method, effectively a method with the same signature (as referred to by the current joinpoint context) is called, except on a different object (i.e. the `decorator` object instead of the `decoratee`). Finally, the instruction `SkipOriginalCallAction` instructs the interpreter not to execute the original call.

The sample program defining a JAMI-based “aspect-AST” (abstract syntax tree) as presented above can be downloaded from the JAMI website; when we initialize the aspect interpreter using this aspect, together with the “word processor” base program presented earlier, we can now write code calling the `decoratee` (as in listing 1, line 10), and still get the correct behavior. The call to the `decoratee` object (Document) is automatically forwarded to the `decorator` object (AutoSaveDocument).

Modularizing the decorator pattern  The simple aspect language defined above does not fully modularize the decorator pattern: associations are created in base code using explicit constructor calls to a `decorator`. If we want to fully separate the decorator from the base code, we have to find a way to specify when a `decorator` has to be created, and to which `decoratee` object it should apply. This is non-trivial, as the decorator pattern is (usually) to be applied selectively, i.e. not simply to all objects of a particular class.

In this section, we look at one way to specify decorator associations from an aspect. There are many possible ways to specify this, each having their own language design trade-offs. A benefit of using JAMI is that we can quickly prototype several proposals, so that we can experiment with the resulting language using real programs.

In the proposal we suggest, a programmer can specify which particular instance variable (indicated by name) should be decorated, and by which decorator class. We could specify this as shown in listing 3, line 1. The decorator is created and associated whenever a new value is assigned to the `decoratee` instance variable – variable “doc”, in this case. In listing 3, this happens on line 7. The forwarding behavior stays the same as before, i.e. on line 9, the call will be forwarded to the auto-saving decorator.

```java
1 decorate: Document WordProcessor.doc -> AutoSaveDocument;
2
3 public class WordProcessor {
4    Document doc;
5
6    public void testAutoSave() {
7       doc = new Document();
8    doc.addLine("AutoSaved!");
9   }
10 }
```

Listing 3. Decorator example: modularized version
To implement this, we only have to replace the “association creation” part as it was shown in figure 5. Instead, we create the structure as shown in figure 7. The pointcut combines 3 criteria using the logical AND operator: the join point must be of the type field assignment, must be contained by class WordProcessor, and have the name/identifier doc. The advice is a custom advice class (extending the JAMI model) that associates the value assigned to the field (i.e. the decoratee) to a newly created and initialized decorator object.

As shown above, we could implement our proposal by writing a minimal amount of (new) source code: a method constructing the (partial) aspect AST such as displayed in figure 7, as well as the custom advice class ConstructAndAssociateDecoratorAdvice. Everything else is already handled by the (existing) framework and interpreter. It took us 4 days to implement the entire language, enabling both the enforcement as well as modularization of the decorator pattern.

### 3.2 Using the D/COOL domain-specific aspect language for synchronization

To show that JAMI can be used to conveniently accommodate more complex domain-specific languages as well, we implement a representative subset of the coordination aspect language “COOL”, which is part of the D language framework. The language is documented extensively in the dissertation describing this framework [16].

Suppose we want to add a spellchecker to our word processor, which runs concurrently with the user interface by using a separate thread. To ensure correct behavior when multiple threads may access the document concurrently, we use a synchronization specification written in COOL, as shown in listing 4. By using COOL, we do not have to put any synchronization-related code in the Java source code itself.

```
coordinator Document {
  selfex addLine, setContent;
  mutex [addLine, setContent];
  mutex [addLine, getContent];
  mutex [addLine, wordCount];
  mutex [setContent, getContent];
  mutex [setContent, wordCount];
}
```

**Listing 4.** Using COOL to synchronize reader/writer access

Listing 4 specifies that we want to coordinate instances of class Document. Line 2 specifies that the methods addLine and setContent are self-exclusive; i.e. only 1 thread
at a time may be running those methods. Line 3 specifies that these methods are mutually exclusive in addition; i.e. only one thread may be active in either addLine or setCon-
tent at a given time. Note that self-exclusion does not imply mutual exclusion: without
mutual exclusion, it could still occur that one thread is running addLine, while another is running setContent. Vice versa, mutual exclusion does not imply self-exclusion either:
although only one of the methods in a mutual exclusion specification may be running at the same time, multiple threads may be executing that one method.

Lines 5-8 also specify pairs of methods not allowed to run at the same time - addLine and setContents are writer methods, and should not run at the same time as reader
methods getContent or wordCount.

By default, COOL synchronizes method access per object, i.e. in the above example, several threads can still run method addLine at the same time, as long as they do so within different object contexts. COOL also allows to specify a per class modifier, which makes the synchronization “global” for the specified class.

Mapping to JAMI We now describe a mapping of the subset of COOL described above to JAMI. First, for each method involved in a synchronization (i.e. selfex/mutex)
specification, we calculate the set of methods that may not be entered while another
thread is active within that method. For method addLine, this “exclusion set” contains
addLine itself (because of the selfex specification on line 2), as well as methods set-
Contents, getContent and wordCount (because of the mutex specifications on line 3, 5
and 6). How these exclusion sets are determined exactly is documented in [16]; we do
not repeat the details here.

Fig. 8. Expressing COOL coordinators using JAMI concepts

A coordinator is modeled (see figure 8) as an aspect that defines one AspectVariable
named coordinator. This variable has a “per object” or “per class” instantiation policy,
depending on the specified granularity of the coordinator. Thus, the variable is shared
between advices belonging to this coordinator, and can be used to regulate the synchro-
nization. The instantiation policies in JAMI automatically give us the desired granu-
larlarity as proscribed by the synchronization specification. Two selector-advice-bindings
are defined for each method involved in a synchronization specification; one will be
executed upon entering the method, one upon leaving.

Figure 9 shows the object diagram for the selector-advice-binding executed upon en-
tering method addLine. It matches only join points of type MethodCall, of which the
target object is of type \textit{Document}, and of which the signature of the called method is \texttt{addLine}. Before the call is executed, we execute the advice \texttt{EnterSyncedContextAction}, an advice class specific to this language.

We show the source code of this advice in listing 5. First, the advice retrieves (line 2-4) the \texttt{coordinator} aspect variable instance belonging to this specific context (i.e. object or class, depending on the instantiation policy). This \texttt{coordinator} object ensures that the synchronization “bookkeeping” itself is properly synchronized. While the advice holds a lock on this object (line 6-21), it can safely inspect the \texttt{MethodState} objects for this coordinator. For each method (involved in synchronization), such a \texttt{MethodState} object tracks which threads are currently running that method. While other threads are active in any method in the exclusion set of the currently invoked method (line 11,12), the advice waits (releasing the lock on the \texttt{coordinator} while waiting) until this is no longer the case (line 14-16). When the loop is left, it means the method is free to run - after the advice registers the current thread with the corresponding \texttt{MethodState} object (line 20) and releases the lock on the coordinator object.

```java
public boolean evaluate(InterpreterContext metaContext) {
    CoordinatorImplementation coord =
        (CoordinatorImplementation) metaContext.getAspect().
            getDataSource(metaContext, "coordinator");

    synchronized(coord) {
        boolean shouldWait = false;
        do {
            shouldWait = false;
            // Wait while any other thread is active in any method in our exclusionset
            for (String excludedMethod : exclusionSet)
            shouldWait |= coord.getMethodState(excludedMethod).isActiveInOtherThread();
            if (shouldWait) {
                try { coord.wait(); } 
                catch (InterruptedException e) { }
            }
        } while (shouldWait);
        // This method is now allowed to run, register it
        coord.getMethodState(myMethodName).enteringMethod();
        return true;
    }
```

\textbf{Listing 5.} Advice executed when entering a synchronized method
Similarly, another pointcut is created to intercept join points that occur upon leaving any of the methods involved in the synchronization specification. The object diagram is analogous to figure 9, except the pointcut now matches only join points of type `Method-Return`, and executes an advice of type `LeaveSyncedContextAction`. We show the source of this advice in listing 6. The advice waits until it obtains a lock on the coordinator object within the given context (object or class, as in the previous advice), allowing it to update the synchronization “bookkeeping”. Once the lock is obtained, it deregisters the current thread from the `MethodState` object for this method (line 6). It then notifies all waiting threads (if any), such that they can re-evaluate their waiting conditions (line 8).

```java
1 public boolean evaluate(InterpreterContext metaContext) {
2 CoordinatorImplementation coord = ...; // as in previous listing
3 synchronized (coord) {
4 // deregister this thread from running this method
5 coord.getMethodState(myMethodName).leavingMethod();
6 // Notify all threads (not just one), as potentially several may be allowed to continue
7 coord.notifyAll();
8 }
9 return true;
10 }
Listing 6. Advice executed when leaving a synchronized method
```

This concludes our implementation of (a subset of) COOL. The above essentially describes the same implementation mechanisms as used in [16], except using an interpreter-based implementation instead of source-code weaving. We believe that this exercise demonstrates the usefulness of JAMI in several ways. First, we successfully mapped an existing, complex language proposal to JAMI. In addition, it took minimal effort to build a functional prototype, which can be used on real base programs. It took 4 days to write the prototype, and it consists of only 500 lines of code. The advice code as shown in listings 5 and 6 comprises the majority of the actual implementation mechanism; in addition we created a parser for the subset of COOL used in this example (ca. 100 lines of code), an object-based representation of this AST (ca. 100 lines of code), as well as code to map such object-based COOL ASTs to “aspect-AST” structures such as shown in figure 8 and 9 (ca. 200 lines of code). Thus, JAMI proves useful as a “testbed” to prototype DSALs.

### 3.3 An experimental DSAL to implement caching

As a final example, we implement an experimental language that introduces a modular way to specify caching of method return values (also called memoization).

Methods (or functions) to which memoization is applied, traditionally have to conform to the following conditions: (1) the method depends on its (input) parameters only; (2) given the same input parameter values, it should return the same result every time; (3) the method should have no side effects. Our implementation maintains the last two requirements. However, the first requirement is often violated in object-oriented programming, as results of a method call are often influenced by instance variables (within the same object) or specific method calls (on the same object). Therefore, our implementation extends the notion of memoization as defined above, by allowing cached
results to be invalidated when the value of particular fields changes, or when particular methods are called.

In our example application from figure 2, the method `wordCount` is a good candidate for memoization, as repeatedly calculating the number of words – even when the document has not changed – can become quite time consuming on large documents. The method has no side effects, but depends on the value of instance variable `content`. This variable is written by method `setContent`, as it contains the statement “this.content = newContent;”. The method `addLine`, containing the statement “content.add(line);” does not overwrite the instance variable itself; it does however modify its contained object structure. Therefore, calls to method `addLine` should also invalidate the return value of `wordCount`.

We specify the above using a domain specific aspect language as shown in listing 7.

```
cache Document object {
  memoize wordCount,
  invalidated by assigning content or calling addLine(java.lang.String);
}
```

Listing 7. Example specification of a memoization aspect

This specification means the following: apply a caching aspect on each `Document` object (line 1). This caching aspect will memoize the return value of method `wordCount` (line 2). The cache will be invalidated when a new value is assigned to instance variable `content` within the corresponding `Document` object (line 3), or when the method `addLine(..)` is called on the `Document` object (line 4).

```
Aspect
getInstance(Context)
cache_wordCount :
AspectVariable
: PerObjectInstantiationPolicy
instantiationPolicy
before(wordCount)
(subview)
selectorAdvBinding

after(wordCount)
(subview)
selectorAdvBinding

invalidate wordCount
by assigning content
(subview)
selectorAdvBinding

invalidate wordCount
by calling addLine
(subview)
```

Fig. 10. Mapping a caching aspect to JAMI concepts

### Mapping to JAMI

We now show how to map the specification shown in listing 7 to JAMI. As figure 10 shows, we create an aspect variable of type `Cache` for each `memoize` declaration. Its instantiation policy can again be specified as per object or per class - in the example above, we want to cache the return value of method `wordCount` for each object of type `Document`. The class `Cache` models a simple wrapper object that can store and retrieve an object, as well as clear its currently stored value.

For each memoized method, we need a pointcut that intercepts calls to that method, coupled to an advice that returns the cached value (if one is stored). Another pointcut intercepts `returns` from the memoized method, coupled to an advice that stores the return value in the cache. Finally, a pointcut is needed for each cache validation specification,
coupled with an advice that invalidates the cache. In this example there are two such pointcuts, corresponding to the invalidation specifications in line 3 and 4 of listing 7).

As shown in figure 11, we intercept calls to the method of which the results should be cached. The advice that is executed is shown in listing 8. First, the advice retrieves the aspect variable corresponding to this memoize declaration (line 2+3). If the cache currently contains a value (which means it must have been set after a previous call), we instruct the interpreter not to execute the original call after it finishes executing this advice (line 7), and instead to set the return value to the value found in the cache (line 8).

```java
public boolean evaluate(InterpreterContext metaContext) {
    Cache cache = (Cache)metaContext.getAspect();
    .getDataFieldValue(metaContext, cacheVarName);
    if (cache.hasValue()) {
        // Use cached value!
        metaContext.setExecuteOriginalCall(false);
        metaContext.setReturnValue(cache.getValue());
    }
    return true;
}
```

Listing 8. Advice: retrieving a cached value

After the method returns, the advice in listing 9 is called, which stores the return value of the method.

```java
public boolean evaluate(InterpreterContext metaContext) {
    Cache cache = (Cache)metaContext.getAspect();
    .getDataFieldValue(metaContext, cacheVarName);
    cache.setValue(metaContext.getReturnValue());
    return true;
}
```

Listing 9. Advice: storing a cached value

First, the advice retrieves the cache variable (line 3+4). Next, it stores the return value of the called method, which can be obtained through the interpreter context (line 6). Note that we do not take method parameters into account in this implementation (fortunately, the method wordCount() does not have any). This is done to avoid cluttering the example; adding this behavior would be straightforward.
To finalize our example, we show one of the pointcut-advice-bindings used to invalidate the cache. Figure 12 shows a pointcut that will match field assignments, but only to the field named `content`, and when the assignment takes place within an object of type `Document`. The advice is to call the method `clearValue` on the aspect variable `cache_wordCount`. The cache can also be invalidated by particular method calls; this binding reuses the same advice, but has a pointcut selecting the specified method, in a way equivalent to many of the examples shown above, e.g. in figure 11.

It took us 3 days to implement this language, including a parser for specifications as shown in 7 and an automated mapping of the parsed structure to the JAMI object diagrams as shown in this section.

4 Composition of multiple DSALs

As each DSAL is designed to address concerns within a particular problem domain, we would often want to combine the use of several such languages within a single application\(^2\). Implementing this is not straightforward however, as partial programs expressed in several languages have to be composed into a single combined, working application. Even if this is technically feasible (which is not necessarily the case), running the combined application may reveal unexpected and/or undesired results.

In this section, we discuss how several aspects written in different DSALs (all implemented using JAMI) can be composed and used within the same application. We discuss several difficulties that may occur in this case, and explain how JAMI can help to address these issues.

4.1 DSAL composition in JAMI

In general, the composition of multiple aspect languages is far from trivial. As an example, consider the common implementation of aspect languages as transformation of the

\(^2\) Note that the entire discussion about the composition of DSALs technically also holds for the composition of general purpose aspect languages, or a mixture of these. However, we believe composition of DSALs is much more realistic to expect, hence we focus on this.
source code or byte code representation of the base program (where each of these aspect language implementations may, or may not, share a common infrastructure). This would require the sequential execution of aspect language implementations over the incrementally transformed base code. Typically, such a byte code transformation is not commutative, meaning that the behavior of the resulting program could vary, according to the normally undefined execution order of the aspect language implementations.

In section 3, we have shown how aspects written in several DSALs are mapped to JAMI elements. Such aspects, expressed in terms of JAMI elements, or refinements of JAMI elements, can be deployed within a single application—even though they originate from different aspect languages. This is enabled by the common runtime platform provided by JAMI.

This platform defines common abstractions and a common data structure for the representation of aspects (e.g. in terms of pointcut expressions, advice-selector bindings, ordering constraints, etc.). Further, the framework imposes a unified high-level control flow for the execution of aspects, as shown schematically in figure 1. At the same time, while adopting these predefined abstractions and high-level control flow, for each language there is a large freedom to define in varying ways how e.g. pointcuts can be defined and matched.

Thus, using JAMI, it is possible to execute aspects written in different DSALs within a single application. This does not require any tailoring or design decisions that are specific to the other DSALs that are combined. However, this does not guarantee that the resulting application will show the “correct” or “desired” behavior. As is the case with aspects written in a single language, interactions or interference may also occur between aspects written in different DSALs.

This phenomenon has also been observed before: e.g. in [17], two categories of aspect interactions are distinguished:

- **co-advising**: the composition of advice of multiple aspect languages at a shared join point.
- **foreign advising**: this corresponds to the notion of ”aspects on aspects”, where advice from one aspect language may apply to a join point associated with the execution of advice in another aspect language.

In the remainder of section 4, we first discuss the issue of co-advising, followed by an explanation of the advice composition mechanism of JAMI in section 4.3, and finally a discussion of foreign advising. These problems are all illustrated by combining the aspects shown in section 3 within the same application.

### 4.2 Co-advising

When multiple pointcuts match at the same join point, the order in which advices bound to these pointcuts are executed may lead to different behavior [20, 9], if there are dependencies between the aspects. Reversely, in the absence of any ordering specification at shared join points, the application behavior may be non-predictable and undesirable.

---

3 [17] defines these terms using a description based on weaving semantics, we reformulated these in terms of aspect execution.
The above is also true if the shared join points originate from programs written in different aspect languages. For individual languages, many mechanisms exist to deal with this.

However, when pointcuts originate from different languages, there are two additional issues:

- We need improved or additional mechanisms to compose advices from different aspect languages. The reason is that we (want to) assume DSALs to be developed independently, so that aspects written in a particular DSAL are likely (and preferably) unaware of those written in another DSAL. JAMI supports a uniform constraint model (first proposed in [20]) that facilitates ordering constraints within as well as between languages. We demonstrate this below.
- There is a distinction between language-level and program-level composition [17]. In particular for DSALs, composition constraints may be specific to a combination of DSALs, and should apply to all aspects written in those DSALs (i.e. language-level constraints). However, it may –in addition– be possible that some constraints are program-specific (i.e. program level).

**Example: composing the synchronization and caching aspects** When we deploy the aspects for synchronization (shown in listing 4) and caching (listing 7) within our original application (see figure 2), we observe that several shared join points occur, as most calls to methods within class Document are advised by both aspects. Therefore, we need to determine in what order these advices should be executed.

As an example, we consider the join point that occurs when returning from method `wordCount`.

![Fig. 13. Concurrent execution; correct advice ordering](image)

![Fig. 14. Incorrect advice ordering](image)

At this join point, a caching advice will store the value that was returned by the method. The synchronization advice leaves the critical section that was entered before the method was executed, as shown in listing 6. In this case, the caching advice –at the end of a method– should be executed before the synchronization advice. This is illustrated in figure 13, whereas figure 14 illustrates a specific scenario of two threads where –in both cases– the synchronization advice precedes the caching advice. In the latter case, a different thread executing a `writer` method may invalidate the cache as soon as the critical section is left, while subsequently the caching aspect stores an (already invalidated!) value in the cache. In that case, the next call to `wordCount` would return a cached value that is incorrect.

To generalize the example, we observe that any caching advice should occur within the critical sections as imposed by the synchronization advice. Specifically, for ad-
vices executed at a shared `MethodCallJoinpoint`, the synchronization advice should have precedence, while at a shared `MethodReturnJoinpoint`, the caching advice should have precedence. This is an example of a language-level composition constraint.

**Example: composing further with the decorator aspect** As a test case, we illustrate the composition of three DSALs in JAMI, including a join point where advices from all three languages must be applied. In addition to the synchronization and caching aspects discussed above, we add the decorator aspect as shown in listing 2 to our application. This means that for objects of type `Document` that are decorated, all calls to methods within `Document` will be forwarded instead to `AutoSaveDocument`. Thus, shared join points may occur where all three aspects (each originating from a different language) want to execute an advice. In this case, the desired behavior is more complex than simply ordering the advices.

When a call is redirected by a forwarding advice (as defined by a decorator aspect), the original call does no longer lead to the execution of a method –as specified by the `SkipOriginalCallAction` in figure 6. Therefore, after the execution of the advice of the decorator aspect (in this case), there is no method execution join point active. Effectively, this means that no other advices should be executed at this join point. This implies that any advice from the decorator aspect should be executed before advices specified by both other languages. After all, it would be illogical to continue and cache or synchronize the execution of a method that will not be executed at all.

![Fig. 15. Control flow combining aspects from 3 DSALs](image)

In figure 15, we illustrate that the intended behavior is obtained by ordering the advices per language as described above. When method `Document.wordCount` is called, all three aspects match this join point. The decorator advice will be executed first, and forwards the call to `AutoSaveDoc.wordCount`, as a result canceling the original call. In addition, it ensures that no other advices are executed at this join point. Subsequently, the implementation of `AutoSaveDoc.wordCount` calls `Document.addLine` again. The decorator aspect does not match this (new!) join point, as internal calls from decorator to decoratee should not be intercepted (as defined by the pointcut expression in figure 6). The other two aspects both match this join point however, and are executed in an order such that caching takes place within the critical section of the synchronization advice.
This example illustrates that with JAMI, composition of more than two aspect languages is supported, even in the presence of delicate interdependencies; JAMI supports the expression of the necessary composition constraints such that the intended effect of each aspect is preserved.

4.3 The advice composition mechanism of JAMI

JAMI offers two complementary advice composition mechanisms. First, it implements a generic ordering constraint mechanism as proposed in [19, 20]. At shared join points, constraints may limit which advices are currently applicable. Such constraints may be conditional, and may for example depend on which advices where already executed (at the same join point). Even so, the application of constraints may still leave several advices eligible for execution. Second, JAMI therefore supports a “scheduling” interface to determine the further selection of advice execution. Different strategies can be implemented to disambiguate the selection of advice. Our default implementation picks an arbitrary element from the set of applicable bindings, and in addition prints a warning that the program is potentially ambiguous. In addition, the scheduler can decide to cancel further advice executions at a given join point, if requested to do so by particular advice actions.

Constraints are specified over selector-advice-bindings, as these are the primary elements over which we want to express ordering criteria—as opposed to ordering specified per advice, pointcut, or aspect. The reason is that advices (and pointcuts) can be reused in several bindings; the desired ordering may be different per binding. In addition, selector-advice-bindings are the most “low-level” construct within JAMI to which an ordering can be applied. Ordering between aspects can be expressed in terms of (several) constraints between selector-advice-bindings. These are examples of program-level constraints. Language-level constraints are also expressed in terms of (several) constraints between individual selector-advice-bindings. The framework could include ‘convenience methods’ to allow to directly express constraints between all bindings within particular aspects, or between all bindings of all aspects written in a particular language. We believe that the constraint mechanism adopted by the framework can be used as the basis of any such higher-level ordering mechanism.

Constraints are decoupled from the “aspect modules” (as shown in e.g. figure 4), and are instead kept as a separate set of entities within the aspect evaluation framework. This enables the specification of constraints between selector-advice-bindings that are part of several aspects, or that even originate from several aspect languages. It would even be possible to define a “constraint language” that can express constraints over aspects from several different DSALs, although this requires that DSALs make it possible to identify (e.g. by name) the entities to which an ordering may need to be applied.

As discussed above, for our example we want to specify language-level composition based on the originating language of each selector-advice-binding. We do not need any program-level constraints, in this case. Therefore, we simply create constraints between all selector-advice-bindings, such that caching advices occur within the critical section

---

4 This corresponds to the run-time detection and resolution of aspect interactions in [23].
created by synchronization aspects (if the advices apply at the same join point), and decorator advices get even higher priority.

Finally, we extend the “forwarding” advice of the decorator aspect (as shown in figure 6) with an advice action that instructs the scheduler to cancel any other advices at the current join point. We argue that this action should be part of the decorator language, as its “forwarding” advice effectively negates the occurrence of the join point at which it is executed. Therefore, no other advices should be executed at that join point.

A functional implementation (in JAMI) that composes aspects written in all three DSALs discussed in this paper — including the constraints as discussed in this section, is downloadable as part of the example discussed throughout this paper [1].

4.4 Foreign advising

Another way to compose aspects, which we did not discuss so far, is the application of aspects on aspects. In particular, the application of advice written in one DSAL on another advice written in a different DSAL, is also called “foreign advising” [17].

Within JAMI, advices expressed in any DSAL are eventually executed in terms of the same kind of instructions used by the base program (e.g. using objects and method calls). Such advice instructions can again be advised, like any normal base language construct. A weaver-based approach must make sure to weave aspects (written in several DSALs) in a particular order, to ensure that the effects of one DSAL can be advised by another —or to ensure that they are not advised by another DSAL. As no weaving takes place within JAMI, the execution of advice is simply a runtime occurrence that can be intercepted like any other join point, if needed.

We test the application of “foreign advice” within JAMI by applying a decorator aspect to the caching aspect discussed in listing 7. Our example decorator class logs the actions of the caching aspect, which can be useful when e.g. debugging the caching aspect. We use the following decorator specification:

```
1 decorate: MemoizeRetrieveAction -> MemoizeLogDecorator
```

This specification intercepts all executions of the memoization advice, and redirects them instead to an instance of class MemoizeLogDecorator, which logs the activity of the caching aspect and then forwards the call back to the original advice method. Thus, it applies an advice to the caching advice.

As a final note, we remark that the advising of advice can quite easily lead to infinite interception loops. Therefore, by default, we exclude all framework classes (and extensions thereof, such as class MemoizeRetrieveAction) from interception by the framework. However, in cases where interception is desired, an annotation can be used to indicate that a particular class should be included for interception. On the other hand, method executions within the base language that are caused by advice executions are by default intercepted.

5 Discussion

In this section, we discuss some of the important design decisions and implementation characteristics of JAMI.
5.1 Efficiency vs. flexibility

JAMI is designed to provide maximum flexibility while designing or testing new aspect languages or language features. As a result, in cases where we had to choose between flexibility and efficiency, we chose the former. For example, the interception of all joinpoints within an application is not very attractive from a performance point of view. However, using this mechanism makes it much simpler to experiment with pointcuts that depend on complex combinations of runtime state (including historic information). While designing a language, the developer can ignore such details as e.g. deciding which part of a pointcut can be evaluated statically, and which remaining dynamic checks have to be woven at such statically determined “shadow join points”. In addition, when composing several DSALs, the effects of one DSAL may influence the static evaluation of pointcuts in other DSALs. Using JAMI, such issues do not occur, as the entire pointcut evaluation takes place at runtime.

However, when creating an efficient language implementation is a primary design goal rather than the prototyping of (new) language features, the use of other frameworks such as abc [2] may be more suitable.

A research framework that aims specifically at modeling efficient implementation techniques of AO mechanisms is ALIA [4]. We are currently exploring the possibility to (manually) map JAMI models to ALIA models. In this way, one could use the JAMI framework to experiment with the language design, and subsequently use the ALIA framework to implement the language to be as efficient as possible.

5.2 Modeling different types of advice

JAMI does not make a distinction between different types of advice, such as around, before or after advice. Join points before and after method calls are simply considered different (kinds of) join points altogether (i.e. MethodCall and MethodReturn join points). The two defining features of around advice (as it is known in e.g. AspectJ) are the ability (1) to share state between the before and after advice parts around a method call (using local variables within the advice definition), and (2) to decide whether or not to execute the code at the location of the current join point (determined by whether or not the advice “executes” a proceed construct). Both can be accomplished using JAMI as well; the first by using aspect variables with an appropriate instantiation policy, the second by using meta-advice actions that instruct the interpreter whether or not to execute the code at the current join point. For an example using both features, see section 3.3. We believe that our advice model allows more freedom over the specification of e.g. advice ordering and the semantics of particular advice actions, while it is also able to accommodate existing advice models, as indicated above.

6 Related work

In [18], Masuhara and Kiczales propose the Aspect Sand Box, an interpreter framework to model aspect mechanisms. Using this framework, the effects of aspects are defined in terms of weaving semantics. The weaving process is modeled by extending
or modifying the interpreter of a base language that models a single-inheritance OO language (which can be seen as a core subset of Java). In comparison, JAMI defines a common runtime environment for aspects, which allows us to express explicit ordering constraints between advices, and enables the deployment of multiple aspect languages within a single application. As discussed in section 4, it would be harder to define a single weaver that models the composition of multiple languages. Essentially, JAMI can be seen as an implementation of a single, parameterized (by using different predefined framework classes) aspect composition process as indicated in the future work section of [18]. In this sense, JAMI can be seen as an elaboration on the ideas proposed in the Aspect Sand Box project, thus enabling the implementation of additional features such as mentioned above.

The AspectBench Compiler (abc) [2] is a workbench that – like JAMI – facilitates experimentation with new (aspect) language features. Unlike JAMI however, it focuses mainly on extensions to AspectJ, and strives to provide an industrial-strength compiler architecture that facilitates efficient implementations of extensions to the AspectJ language. In contrast, while designing JAMI we specifically tried to avoid design decisions that would limit the flexibility of our framework, as discussed in section 5.1. In addition, abc is not designed to handle composition between multiple languages.

In [3], Bagge and Kalleberg propose to implement DSALs by creating a library that implements a program transformation system (cf. weaver), in addition to a notation that “configures” the behavior of this library. The paper does not discuss the composition of multiple DSALs, or the ordering of advices at shared joinpoints.

In [5], Bräuer and Lochmann describe how to integrate multiple DSLs based on a common semantic metamodel, using an MDA-based transformation approach. Like JAMI, this provides a model to integrate modularized specifications written in several DSLs. However, the paper aims to propose a common semantic model for the composition of multiple DSLs in general – as opposed to expressing crosscutting functionality (i.e. aspects) in particular, as is the focus of JAMI. Likely as a result of this, the paper does not discuss the interference issues that may result from composing multiple crosscutting specifications.

The work from Kojarski and Lorenz [17, 14, 13] is strongly related to ours; in particular, they also investigate the issue around the composition of multiple aspect languages.

In [14], seven interaction patterns among features of composed aspect languages are described. Some of these, such as emergent advice ordering, are also discussed in this paper. However, because (1) JAMI introduces its own set of abstract features, such as selector-advice bindings, and (2) in our interpreter-based approach, individual aspect languages are not translated into base language terminology, hence, there is never accidental interaction, not all interaction patterns are applicable. However, the proposed analysis approach could also be applied in the context of our work.

In [17, 13], the AWESOME framework is described; instead of an interpreter-based approach, this adopts a weaver-based approach, that also addresses foreign advising, and language-level, but currently – according to [17] – not program level co-advising (which we presented in section 4.3).

The Reflex AOP kernel [24, 23] is also closely related work; it is a reflection-based kernel for AOP languages, with a specific focus on the composition of aspect programs.
To this extent, it provides an (extensible) set of composition operators, which can be used when translating an aspect specification to a representation in terms of the kernel-level abstractions. Although there are many similarities with JAMI, a key difference of the current implementation is that it is weaving-based, rather than interpreter-based. Mostly due to this, the support for foreign advising is limited (as e.g. exemplified in [14]).

The XAspects project [22] implements a system to map DSALs to AspectJ source code. The approach addresses the need to compose aspects written in multiple DSAL, but does not provide explicit mechanisms to deal with interactions between aspects, other than suggesting the use of the AspectJ declare precedence construct. Compared to this, JAMI offers more elaborate ways to specify the composition of aspects.

In [6], Brichau et.al. propose the definition and composition of DSALs (“Aspect-Specific Languages”) using Logic Metaprogramming. Although their approach is not based on a typical OO framework, it does allow the reuse and refinement of aspect languages. It is based (in [6]) on static source code weaving (by method-level wrapping). The composition of aspect languages (program level composition is not supported) is achieved by explicit composition of languages into new, combined languages. In our opinion, this is less flexible, as it requires explicit composition for each configuration of aspect DSALs that occur in an application, and the late addition of a new aspect language in a system may not be possible without restructuring the composition hierarchy.

7 Conclusion

In this paper, we introduced an aspect interpreter framework aimed at the prototyping of (domain-specific) aspect languages. We demonstrated the framework by implementing three domain-specific aspect languages, and also briefly discussed the workflow used to implement these. Using our framework, it took only 3-4 days (per language) to create functional prototypes of these diverse DSALs. Aspects written in these DSALs can be composed with regular Java programs at runtime, in an interpreted style.

We have used JAMI in a programming language course to teach the common aspect language concepts and various implementations thereof. As part of this course, students successfully developed small DSALs within limited allotted time. This supports our claim that JAMI can be used to prototype DSALs while requiring relatively little effort, even including the learning curve of the framework itself.

We contribute the effectiveness of JAMI as a framework for prototyping DSALs in large part to its flexibility and expressiveness. For example, as aspects are completely dynamically evaluated, it is easy to experiment with pointcuts that express complex selection criteria over the runtime state. In addition, our support for “aspect state” using variables that each may have different instantiation policies provides a flexible way to implement aspect language features, while requiring relatively little effort.

We have shown that our framework supports applications composed of aspects written in several DSALs. In addition, we have discussed interactions that may occur when combining multiple DSALs, and demonstrated mechanisms implemented as part of JAMI to specify aspect composition – also of aspects written in different languages.

The complete framework as well as the examples shown in this paper can be downloaded from the JAMI website [1].
8 Acknowledgments

This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe).

References

10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts, 1994.