Pluggable Aspect Instantiation Models*

David H. Lorenz and Victor Trakhtenberg

Dept. of Mathematics and Computer Science,
Open University of Israel, Raanana 43107 Israel
Email: lorenz@openu.ac.il, victortr75@gmail.com

Abstract. An aspect encapsulates not only crosscutting behavior, but also crosscutting state. When aspects are stateful, there is a need to specify and control their instantiation. Unfortunately, aspect instantiation is a hard-wired feature in ASPECTJ. This feature cannot be customized by the application programmer. Specifically, there are six pre-defined instantiation models to choose from, each designated by a keyword: **issingleton**, **perthis**, **pertarget**, **percflow**, **percflowbelow**, **pertypewithin**. In this work, we introduce a new language mechanism and keyword '**perscope**' that lets third-parties define custom aspect instantiation models. This new keyword replaces the six existing keywords in ASPECTJ, and reduces the need for introducing future ones.

1 Introduction

A *stateful aspect* [4] is a unit of modular definition of a crosscutting concern that encapsulates not only crosscutting behavior, but also *crosscutting state*. An *aspect instantiation model (AIM)* is a policy that defines the manner in which stateful aspects are to be instantiated and managed [13]. This work is concerned with third-party customization of AIMs.

The need for custom AIMs has historically influenced the evolution of the ASPECTJ language [6]. Initially, ASPECTJ assumed only singleton aspects. A *singleton aspect* is instantiated once, i.e., there is only a single aspect instance for each aspect declaration. Singleton aspects are useful for implementing concerns that have system-wide behaviors.

Gradually, ASPECTJ was extended to include finer-grained AIMs. A *per-object aspect* (**perthis** and **pertarget**) associates a unique aspect instance with each advised object. A *per-flow aspect* (**percflow** and **percflowbelow**) associates a unique aspect instance with every flow of control that is matched by a specified pointcut argument. These per-clauses were added in early versions of ASPECTJ. A later version of ASPECTJ 5 introduced a new *per-type* clause (**pertypewithin**) for associating a unique aspect instance with each type matched by a specified type pattern.

This evolution begs the question: what other sorts of per-clause may be needed, and how can the authority and responsibility for defining such AIMs be decoupled from the evolution of the aspect language?

* This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 926/08.

1.1 Background

In ASPECTJ, aspect instantiation is a hard-wired feature of the aspect language. This feature cannot be customized or extended. ASPECTJ does give the application programmer some control over the instantiation of aspects, but only through a limited set of pre-defined AIMs. For example, one cannot associate a separate aspect instance for every execution thread (i.e., a **`perthread`** aspect). Consequently, the aspect state has to be managed manually in, e.g., a JVM ThreadLocal control. Another example is dealing with user state. In J2EE applications it is often useful to define a **`persession`** aspect. Unfortunately, ASPECTJ does not support this (yet). The developer must therefore manage the aspect state manually using J2EE controls, namely an HTTP session and HTTP filters.

When the pre-defined AIMs do not meet the need, the programmer may either manage the aspect state in the application code, or introduce a new AIM into ASPECTJ by modifying the compiler code (e.g., ajc). The first option results in code tangling, increasing code complexity, diminishing maintainability and robustness. The second option is a tedious task even for an experienced compiler programmer. The code fragments implementing the built-in AIMs in ajc are scattered, tangled, and strongly coupled with the implementation of other features of the language. This also requires a change to the ASPECTJ language to let the application code refer to the new AIM (i.e., introduce new syntax). Both changes to the language and changes to the compiler need to be redone whenever a new version of ASPECTJ or `ajc` is released.

1.2 Contribution

This work contributes a framework for enabling the definition and use of third-party AIMs in ASPECTJ. All the built-in AIMs in ASPECTJ can be defined in this framework. New AIMs not currently part of ASPECTJ can also be defined. From the application programmer perspective, the framework provides a way to define and implement AIMs that are required in the application design but currently not available in ASPECTJ. From the compiler programmer perspective, extending the language with new AIMs, whether implemented in-house or supplied by third-party vendors, can be accoplished in a modular way without modifying the ASPECTJ compiler code.

The framework provides the flexibility of realizing an AIM as either a pluggable static mechanism that is invoked during compilation, or as a pluggable dynamic mechanism that is invoked at runtime. This has the advantage of supporting the implementation of built-in AIMs with static mechanisms that produce woven code identical to the one generated by the existing ASPECTJ compiler, while permitting new AIMs to be quickly implemented and tested as dynamic mechanisms.

We extended the ASPECTJ language with a new language feature for supporting custom AIMs. Syntactically, the feature subsumes all of the six ASPECTJ built-in aspect instantiation keywords into a single new keyword: **`perscope`**. Semantically, this feature allows the programmer to use custom third-party AIMs. We implemented the framework by refactoring the `ajc` compiler.

2 Approach

In ASPECTJ, aspects are instantiated implicitly according to the AIM specified in the aspect declaration. The application programmer expresses declaratively which AIM is required. The ASPECTJ compiler weaves the specified AIM into the aspect class, producing efficient woven code. The main drawback of ASPECTJ is in its fixed set of available AIMs and the inability to define new ones.

This work enables the definition and use of third-party AIMs in ASPECTJ. We present a pluggable framework for AIMs. The AIM framework exhibits the following desired properties:

Extensible New AIMs can be introduced, and pre-defined AIMs can be refined. Generic AIMs can be authored by third-party providers independent of the ASPECTJ language evolution, or implemented and customized by ASPECTJ developers.

Declarative Use of AIMs can be specified in a declarative manner similar to the way AIMs are specified in ASPECTJ today, allowing their implementation to vary independently of their use.

Expressive Realistically complex AIMs can be defined. For the very least, it is possible to implement third-party AIMs that provide the same semantics as the six AIMs available in ASPECTJ today.

Efficient Implementation of custom AIMs can be optimized to produce as efficient woven bytecode as their hardwired counterparts implementation in ASPECTJ. In particular, it is possible to implement the six standard AIMs of ASPECTJ and generate the same bytecode as would ASPECTJ, thus providing not only identical functionality but also the same runtime performance.

Flexible The framework supports both compile-time and runtime AIM mechanisms. An AIM can be initially implemented via a mechanism that is invoked at runtime, possibly surrendering some performance for simplicity. The implementation can later be replaced with a more efficient mechanism that is invoked during compilation, without altering code that already uses the AIM.

The support for both compile-time and runtime mechanisms promotes not only adoption but also experimentation with new AIMs. A runtime AIM mechanism is often simpler to implement than a compile-time mechanism, because programming is done directly in terms of the runtime elements. The programmer that is unfamiliar with the internals of bytecode manipulation may implement the runtime version first and migrate to a compile time implementation when performance becomes crucial.

2.1 Language Extension

To support pluggable AIMs we introduce a new **`perscope`** keyword to ASPECTJ. This single keyword subsumes the existing six ASPECTJ keywords (and hopefully reduces the need to introduce future AIM keywords). The changes to the ASPECTJ grammar are listed in Figure 1.

`perscope` is a parameterized keyword. The first argument is mandatory. It is a name of the class that implements the AIM mechanism. The additional arguments are

```
<aspect_decl> ::= [<modifiers>]
                  "aspect" <identifier>
                  [<super>] [<interfaces>]
                  [<perclause>]
                  <aspect_body>
  <perclause> ::= "perscope" "(" <id_list> ")"
    <id_list> ::= <identifier>
                | <id_list> "," <identifier>
```

Fig. 1. Changes to the ASPECTJ Grammar

```
public aspect IndividualClientCaching
  extends SimpleCaching //an abstract aspect defined elsewhere
  perscope(Perthis, cachedOperation(Object)) {
}
```

Listing 1.1. Example of client specific cache

optional and depend on the specific AIM definition. For example, in case of **perthis** a second argument is needed to represent the pointcut. In case of **pertypewithin** the second argument represents a type. When the **perscope** clause is omitted, the default **issingleton** AIM is assumed, as is the case in ASPECTJ.

Listing 1.1 illustrates the use of the **perscope** keyword for a client specific cache. In the listing, `SimpleCaching` is an abstract aspect defined elsewhere, and *`Perthis`* is the name of a class that implements the **perthis** AIM. The implementation of *`Perthis`* can be either as a static or as a dynamic mechanism. For this purpose two new interfaces are introduced: *`StaticPerscope`* and *`DynamicPerscope`*. Only a class that implements one of these interfaces is permissible as the first argument to the **perscope** keyword. With these interface in place, there is no need to modify the compiler code intrusively whenever a new AIM is needed. The compiler programmer can implement a new AIM by implementing either *`StaticPerscope`* or *`DynamicPerscope`* and ship the AIM with a new version of the compiler or provide it as a library.

2.2 Pluggable Framework

The **perscope** keyword together with the *`StaticPerscope`* and *`DynamicPerscope`* interfaces provide a pluggable and flexible mechanism for defining and implementing AIMs in ASPECTJ. With these tools in hand, language designers are able to design new AIMs. Programmers are able to implement custom AIMs. Libraries may offer generic AIMs. The AIMs can be invoked during compilation, dealing with bytecode manipulations, producing efficient code. The AIMs can be implemented as runtime plug-ins and later optimized to run at compile-time, as needed. Whatever the implementation choice may be, the end user is able to use the AIMs in a declarative way.

In order to implement a custom AIM, the implementer specifies what should happen when an aspect instance is handled (during compilation or at runtime). This is done as

```
public interface StaticPerscope {
  void instantiateMethodObjects(ReferenceType typeX, ClassScope scope,
       SourceTypeBinding binding);
  void generateMethodAttributes(ClassFile classFile, AspectDeclaration
       aspectDeclaration, final ReferenceType typeX);
  void generateMethodBodies(AspectDeclaration aspectDeclaration, ReferenceType
       typeX);
  ParametersParser getParametersParser();
}
```

Listing 1.2. StaticPerscope interface

```
public interface DynamicPerscope {
  void bindAspect(Object aspekt,Object object);
  Object aspectOf(Class<?> aspectType, Object object);
  boolean hasAspect(Class<?> aspectType, Object object);
}
```

Listing 1.3. DynamicPerscope Interface

```
public class Perthread implements DynamicPerscope {
  private static ThreadLocalAspect theAspect = new ThreadLocalAspect();
  public Object aspectOf(Class<?> aspectType, Object object) {
    return theAspect.get();
  }
  public void bindAspect(Object aspekt, Object object) {
    if (theAspect.get() == null) {
      theAspect.set(aspekt);
    }
  }
  public boolean hasAspect(Class<?> aspectType, Object object) {
    return theAspect.get() != null;
  }
  private static class ThreadLocalAspect extends ThreadLocal<Object> {
  }
}
```

Listing 1.4. Perthread implementation

```
public aspect PerThreadCaching
  extends SimpleCaching
  perscope(Perthread, cachedOperation(Object)){
}
```

Listing 1.5. Perthread use

part of the process of implementing the `StaticPerscope` interface (Listing 1.2) or implementing the `DynamicPerscope` interface (Listing 1.3).

The modifications to the ajc code include calls to the `StaticPerscope` interface instead of calling the hard-wired weaving code when the mechanism specified in the **perscope** clause implements `StaticPerscope`. The implementation of this interface deals with bytecode manipulations mostly and produces efficient code. This interface can be implemented by third-party infrastructure programmers that wish to provide novel generic AIMs. Alternatively, the AIM mechanism may be realized by implementing the `DynamicPerscope` interface to specify how the aspect instance should

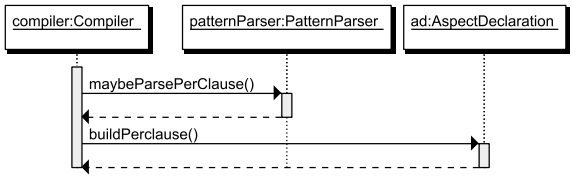

Fig. 2. Regular aspects compilation

be bound and later retrieved. A `DynamicPerscope` implementation is invoked at runtime. Thus, it does not need to deal with bytecode manipulations but rather may be expressed in plain JAVA code.

It is also possible for the application programmer to customize the compiler directly by implementing an AIM that is not available in the current compiler version. A third-party vendor can develop an independent set of AIMs and provide them as a library too. For example, if a **`perthread`** AIM is needed for caching, and if that AIM is not available in ASPECTJ, then it can still be defined and used via a `Perthread` class (Listing 1.4) that implements the `DynamicPerscope` interface (Listing 1.3), and then used with **`perscope`** (Listing 1.5).

3 Implementation

The **`perscope`** framework was implemented by refactoring the ASPECTJ ajc compiler (version 1.6.5). The compiler reads class and aspect declarations as input, and generates Java bytecode as output. We first briefly review how ajc generates code for regular aspects. We then explain how our implementation generates code for **`perscope`** aspects.

The ASPECTJ compiler translates an aspect declaration into a class, and places the advice body into a method of that class. To ensure that the advice gets executed, method calls are inserted into locations where the pointcut of the advice statically matches [9]. First, the `Compiler` (an ajc internal class) checks whether any of the six per-clause keywords are specified, by invoking `maybeParsePerClause` on the `PatternParser` (Figure 2). When a per clause is detected, the parser runs special code for handling that keyword by invoking `buildPerclause` in the class `AspectDeclaration`.

For this purpose, there are a number of *if-then-else* blocks that are spread all over in the compiler code (`AspectDeclaration`). These blocks control the generation of synthetic methods. Synthetic methods are methods not derived directly from the user specified code. Aspect instantiation synthetic methods for the **`perthis`** aspect are `hasAspect`, `aspectOf` and `ajc$perObjectBind` (Figure 3). There are also methods that are responsible for the creation of the needed ajc objects that are used for synthetic methods generation. Finally, there is the code that actually generates the bodies of these synthetic methods.

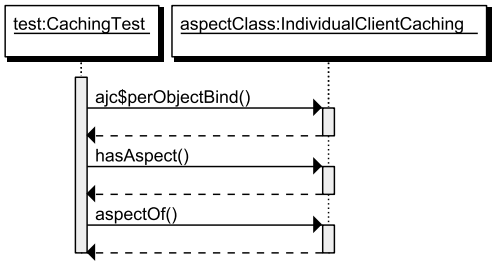

Fig. 3. Runtime of regular aspects

```
public interface ParametersParser {
  PerClause parseParameters();
}
```

Listing 1.6. ParametersParser interface

3.1 The StaticPerscope Interface

Since AIM related code is scattered all over the `AspectDeclaration` class, it is difficult to override and extend it. The purpose of the *`StaticPerscope`* (Listing 1.2) interface is first and foremost to provide the abstraction necessary for customiaing AIM related code.

`perscope` aspects are compiled into Java classes in a similar manner as regular aspects. The compiler code was modified in a way that it delegates to the appropriate `StaticPerscope` interface method. The compilation flow of **`perscope`** aspects is depicted as a sequence diagram in Figure 4. In phase 1, the ajc `Compiler` invokes the `maybeParsePerClause` method of `PatternParser` (also ajc internal class). The modified behavior of the `PatterParser` is to create a new instance of the user specified implementation of `StaticPerscope` interface, fetch the `ParametersParser` implementation from it in order to parse the **`perscope`** parameters. This parsing process produces the concrete instance of `PerClause`. It is then used to store the created *`StaticPerscope`* instance. In phase 2, the `Compiler` invokes the `buildPerclause` method of the `AspectDeclaration`, which holds the reference to the `PerClause` instance that was created in phase 1. The *`StaticPerscope`* instance is fetched from it and the synthetic methods generation is delegated to it.

The six built-in ASPECTJ strategies are provided as classes that implement the *`StaticPerscope`* interface in a way that executes the original `AspectDeclaration` code. The original code was moved to the implementing classes without change. It thus produces the same bytecode for the built-in strategies. *`StaticPerscope`* is open for other implementations as well. These custom implementations allow implementing custom AIMs. The actual `StaticPerscope` implementation is realized at aspect parsing time according to the aspect declaration.

A **`perscope`** strategy may have as many parameters as needed. In order to implement this ability the `ParametersParser` interface should be implemented (List-

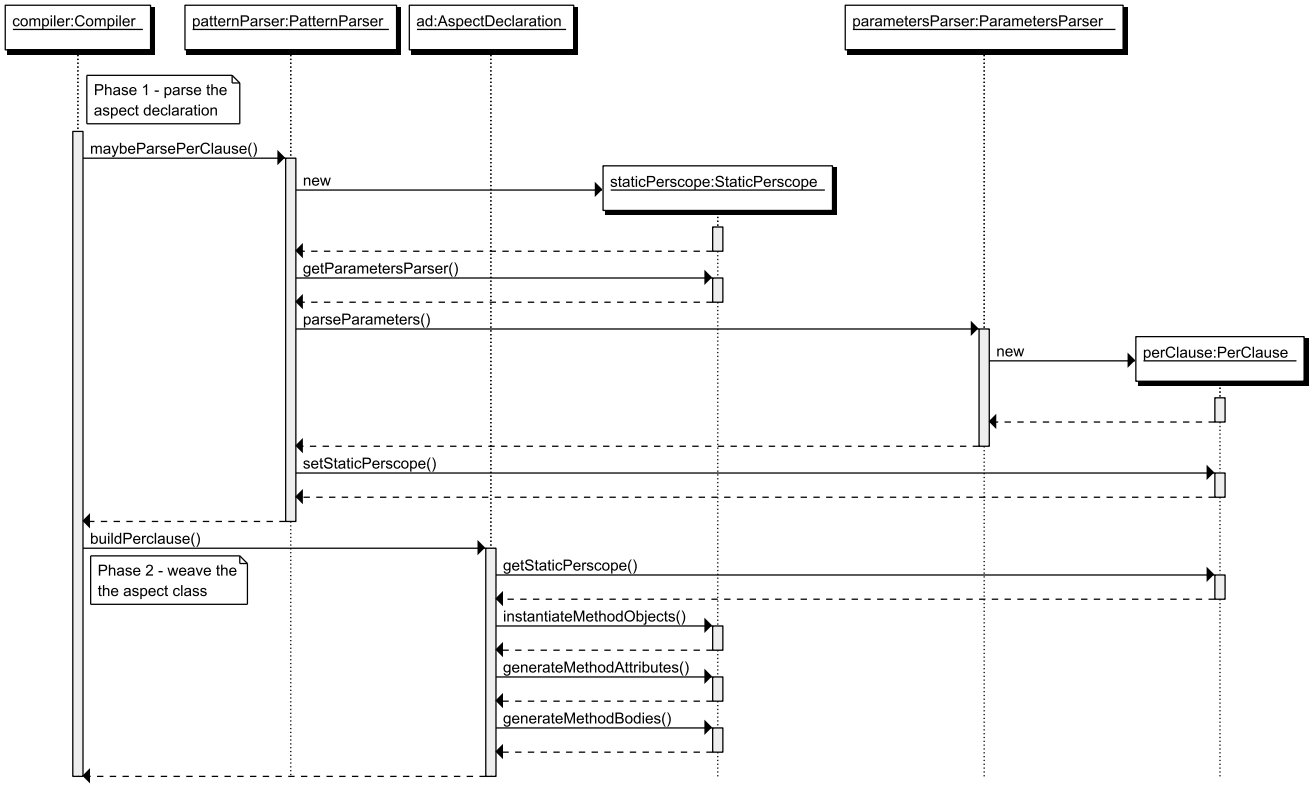

Fig. 4. Compilation of perscope aspects

ing 1.6). There are two built-in implementations. One `PointcutParametersParser` that parses the name of the pointcut and the `TypeParametersParser` that parses the name of the Java type. The interface method `parseParameters` returns the `PerClause` instance, which is the ajc built-in type that is used later on during the compilation process.

3.2 The DynamicPerscope Interface

The *`StaticPerscope`* implementation used at compile-time, deals with bytecode manipulations. In order to make the AIM implementation easier, another built-in `PerscopeDelegator` implementation is provided. This implementation weaves into the aspect code special hooks that delegate at runtime to the user specified *`DynamicPerscope`* implementation.

The *`DynamicPerscope`* implementation is used at runtime for aspect instantiation. The `bindAspect` method is responsible for the registration of an aspect. It gets as its parameters the aspect instance and the optional object instance that is related to this aspect instance. This is needed in AIMs like **`perthis`** and **`pertarget`**. Other AIMs

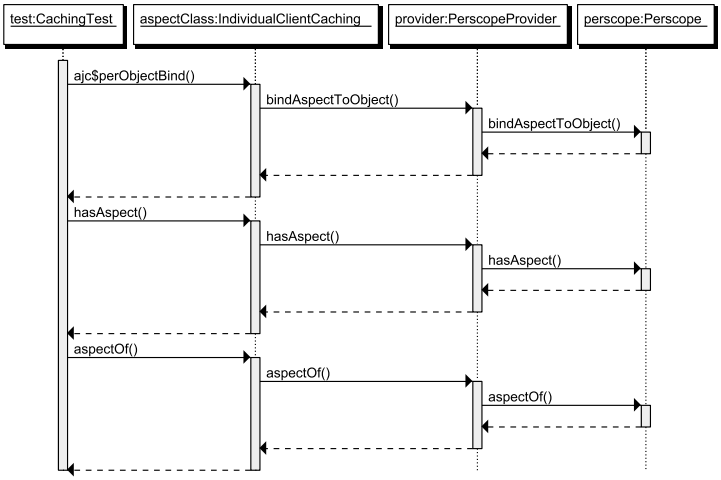

Fig. 5. Runtime behavior of perscope aspects

may ignore the `object` parameter, e.g., **`perthread`**. The parameters are the type of the needed aspect and the concrete object that the aspect is bound to. It is worth noting that for **`perthis`** a **`this`** object is essential and for **`pertarget`** a **`target`** object is essential, but they are never both needed. Therefore, a single argument is sufficient. The `object` parameter is optional here as well. The `hasAspect` method is similar to `aspectOf`, but it queries whether the aspect instance exists. The `object` parameter in the interface methods can be used to implement strategies that depend on the state of an object, such as the value of a certain field of the object.

The compiler uses the utility class `PerscopeProvider` (Listing 1.8) for delegating the invocation of the `hasAspect`, `aspectOf`, and all binding methods to the programmer's specified class. Invocations of `aspectOf`, `hasAspect`, and binding methods are delegated to the `PerscopeProvider` class. For example, Listing 1.7 presents the decompiled bytecode generated for the `PerThreadCaching` class (Listing 1.5). Note that the name of the specified user defined *`Perthread`* class name is provided as an argument to `PerscopeProvider` (lines 8, 12 and 16 in Listing 1.7). Also, the class object of the aspect class itself (`class1`) and the actual object in use (`obj`) are arguments in these calls.

3.3 Summary

The implementer of a **`perscope`** AIM has two options. It is possible to implement a compile-time custom AIM by implementing the *`StaticPerscope`* interface or implement a runtime custom AIM by implementing the *`DynamicPerscope`* interface. In the first case, the implementation is usually more complex as it involves bytecode manipulation, but it is also more efficient. The *`StaticPerscope`* interface is intended for the

```
1  public class PerThreadCaching extends SimpleCaching {
2    public static PerThreadCaching aspectOf(Object obj) {
3      // load aspect class
4      Class class1 = Class.forName("PerThreadCaching");
5      // delegate to PerscopeProvider aspectOf()
6      // with the name of the user specified DynamicPerscope
7      // implementation class name
8      return (PerThreadCaching)PerscopeProvider.aspectOf("Perthread", class1, obj);
9    }
10   public static boolean hasAspect(Object obj) {
11     Class class1 = Class.forName("PerThreadCaching");
12     return PerscopeProvider.hasAspect("Perthread", class1, obj);
13   }
14   public static void perObjectBind(Object obj) {
15     PerThreadCaching threadCaching = new PerThreadCaching();
16     PerscopeProvider.bindAspect("Perthread", threadCaching, obj);
17   }
18 }
```

Listing 1.7. PerThreadCaching generated class with perscope

more advanced programmer. The *DynamicPerscope* interface is intended for regular infrastructure programmers.

These implementation trade-offs are transparent to the end user of the AIM. As soon as the strategy is implemented (either by implementing *DynamicPerscope* or *StaticPerscope*) all the end user needs to do is to provide the class name as the **perscope** keyword argument.

4 Evaluation

In this section we discuss and summarize various experiments that we ran to validate our approach. First, we verified that with **perscope** aspects we can implement all of ASPECTJ built-in strategies (Section 4.1). Second, we implemented additional AIMs that do not exist in ASPECTJ (Section 4.2). We conclude that **perscope** enables third-party providers to easily add new strategies in a modular and extensible way, to be invoked during compilation or at runtime. The implementation and test code can be download at: `http://aop.cslab.openu.ac.il/research/perscope/`.

4.1 Implementing ASPECTJ Built-in Strategies

Using the special *StaticPerscope* interface, we have implemented AIMs corresponding to all six existing ASPECTJ keywords [8]. For the aspect examples we tested, our *StaticPerscope* implementation generates exactly the same bytecode as the original ASPECTJ implementation. As explained in Section 3.1 this is a result of the refactoring, which merely moved aspect instantiation related code from hard-wired *if-then-else* blocks to the appropriate *StaticPerscope* implementations. As depicted in Figure 6, the classes that implement the six existing ASPECTJ keywords are: *Perthis*, *Pertarget*, *Percflow*, *Percflowbelow*, *Pertypewithin* and `Persingleton`.

```
public class PerscopeProvider {
  private static DynamicPerscope aspectInstantiation;
  private PerscopeProvider() {}
  public static DynamicPerscope getStaticPerscope(String implName) {
    if(aspectInstantiation) == null) {
      try {
         Class<?> clazz = Class.forName(implName);
         Object object = clazz.newInstance();
       aspectInstantiation = (DynamicPerscope) object;
      }catch (Exception e) {
       throw new RuntimeException("Failed to instantiate the aspects instantiation
            implementation");
      }
    }
    return aspectInstantiation;
  }
  public static void bindAspect(String implName, Object aspekt,Object object) {
    getStaticPerscope(implName).bindAspect(aspekt, object);
  }
  public static Object aspectOf(String implName, Class<?> aspectType, Object object
       ) {
    return getStaticPerscope(implName).aspectOf(aspectType, object);
  }
  public static boolean hasAspect(String implName, Class<?> aspectType, Object
       object) {
    return getStaticPerscope(implName).hasAspect(aspectType, object);
  }
}
```

Listing 1.8. PerscopeProvider class

To test the **perscope** static implementation we implemented and ran a caching example (found in [3]) using **perscope**. Both ajc and our **perscope** implementation produced the same bytecode and therefore exhibit the same behavior and performance.

We also ran the benchmark implementation of the Law of Demeter [7], which is part of the AspectBench *abc* compiler distribution [2]. The Law of Demeter example uses both **pertarget** and **percflow** intensively. We compared the implementation that uses ASPECTJ 1.6.5 with an implementation that uses **perscope** aspects and observed the same functionality and performance in both.

4.2 Implementing Non ASPECTJ Strategies

The **perscope** aspects allow the application programmer to implement AIMs that do not exist in ASPECTJ.

Strategies found in other AOP languages ASPECTS [5] is an aspect language for SMALLTALK (SQUEAK), which supports certain cflow advice activations that are not available in ASPECTJ. The *Class First* and *Class all-but-first* cflow advice trigger activation on an object-recursion *first* and *other than first* method invocation, respectively. Similarly, *Instance First* and *Instance all-but-first* cflow advice will trigger activation on a method-recursion *first* and *other than first* method invocation, respectively. With **perscope**, it is possible for the application programmer to implement similar AIMs without modifying the compiler code.

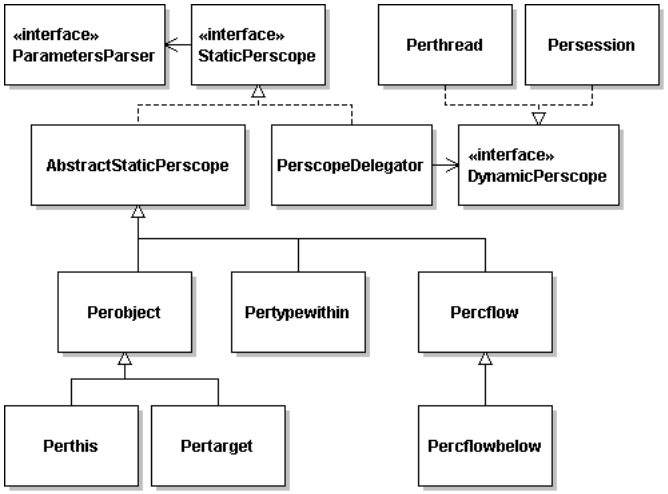

Fig. 6. AIM class diagram

With **perscope** it is possible to implement the **perthread** AIM, i.e., specifying that there should be a dedicated aspect instance per execution thread, as provided in the JAsCo [14] language. If we want to have a **perthread** caching, we would declare the aspect as it appears in Listing 1.5. The *Perthread* is the class that implements the **perthread** AIM (Listing 1.4). The *Perthread* class implements the *DynamicPerscope* interface (Figure 6).

Novel strategies not found in other languages It is also possible to define and use a **persession** AIM. This AIM creates a dedicated aspect instance per specific user session. The Persession class also implements the *DynamicPerscope* interface (Figure 6).

4.3 Code Complexity

With **perscope** aspects the complexity of AIM implementation is reduced significantly compared to what would have been required to modify ajc directly. The approach is modular and extensible. We refactored the AspectJ compiler code that is responsible for generating aspect instantiation methods, which resides in the AspectDeclaration class (Listing 1.9). Instead of dealing directly with the aspect instantiation method generation via *if-then-else* blocks, the refactored code delegates the task to the proper user defined class, which implements the *StaticPerscope* interface (Listing 1.10). If the user specified as a **perscope** parameter a class that implements the *DynamicPerscope* a special built-in *StaticPerscope* implementation will be used to weave the code in a way that a real aspect instance binding or retrieval will be postponed until runtime.

There are three places in the AspectDeclaration class with such *if-then-else-else* statements. All of them were replaced by a two-line API call to the **perscope** interface. The total size of the AspectDeclaration class in AspectJ version 1.6.5

```
if (perClause.getKind() == PerClause.SINGLETON) {
     generatePerSingletonAspectOfMethod(classFile);
     generatePerSingletonHasAspectMethod(classFile);
     generatePerSingletonAjcClinitMethod(classFile);
   } else if (perClause.getKind() == PerClause.PERCFLOW) {
     generatePerCflowAspectOfMethod(classFile);
     generatePerCflowHasAspectMethod(classFile);
     generatePerCflowPushMethod(classFile);
     generatePerCflowAjcClinitMethod(classFile);
   } else if (perClause.getKind() == PerClause.PEROBJECT) {
     TypeBinding interfaceType = generatePerObjectInterface(classFile);
     generatePerObjectAspectOfMethod(classFile, interfaceType);
     generatePerObjectHasAspectMethod(classFile, interfaceType);
     generatePerObjectBindMethod(classFile, interfaceType);
   } else if (perClause.getKind() == PerClause.PERTYPEWITHIN) {
     // PTWIMPL Generate the methods required *in the aspect*
     generatePerTypeWithinAspectOfMethod(classFile); // public static <aspecttype>
          aspectOf(java.lang.Class)
     generatePerTypeWithinGetInstanceMethod(classFile); // private static <
          aspecttype> ajc$getInstance(Class c) throws
     // Exception
     generatePerTypeWithinHasAspectMethod(classFile);
generatePerTypeWithinCreateAspectInstanceMethod(classFile); // generate public
     static X ajc$createAspectInstance(Class
     // forClass) {
     generatePerTypeWithinGetWithinTypeNameMethod(classFile);
```

Listing 1.9. AspectDeclaration implementation in ajc

```
StaticPerscope staticPerscope = perClause.getStaticPerscope();
StaticPerscope.generateSyntheticMethods(classFile, this, typeX);
```

Listing 1.10. AspectDeclaration with perscope

is 1102 lines of code. The refactored **perscope** implementation code size was reduced to 648 lines of code. The ASPECTJ compiler programmer can extend the language with new AIMs easily, just by implementing one of the interfaces, without changing the complex `AspectDeclaration` class.

4.4 Threats to Validity

The **perscope** feature was successfully implemented just for the ajc compiler. It was not tested with other ASPECTJ language implementations, such as abc. However, since the feature is language-specific (rather than implementation-specific), it should be equally possible to implement **perscope** in other implementations.

The ASPECTJ six built-in strategies were implemented using **perscope** aspects and this implementation was tested on the caching example and on the Law of Demeter benchmark example. While two examples are not proof of correctness, the fact that the bytecode generated for these examples was identical to the bytecode generated by ajc version 1.6.5 on the same examples is a strong indication that the refactoring and *StaticPerscope* implementations for the six strategies were behavior preserving.

Various AIMs were demonstrated in this paper and all of them were implemented using **perscope** strategies. There may be other strategies that cannot be implemented

with **perscope** (without compiler modifications). However, while **perscope** may not be perfect it is nonetheless an improvement over the current ASPECTJ implementation, which does not support implementing customized strategy at all.

5 Related Work

The closest related work to ours are aspect factories in the JASCO language [14]. There are five built-in aspect factories: **perobject**, **permethod**, **perall**, **perclass** and **perthread**. It is also possible to implement custom AIM by implementing an `IASpectFactory` interface. The As with **perscope** the usage of aspect factories is declarative. One of the predefined keyword may be specified in the connector code or the **per** keyword may be used if a custom AIM is needed. The JASCO `IAspectFactory` implementation is used at runtime for the aspect instantiation management. With **perscope** the user has a choice: implement the *`DynamicPerscope`* interface to get a behavior similar to the JASCO `IAspectFactory` or implement the *`StaticPerscope`* interface and specify how the AIM should be implemented at compile-time gaining more efficient code in most cases.

Tanter et al. [15, 16] discuss the need for dynamic deployment of aspects. They propose *deployment strategies* for parametrized dynamic aspect deployment. To a certain extent, AIMs could be viewed as a special simplified case of deployment strategies. However, dynamic deployment of aspects is not supported in ASPECTJ. In our work, we focus on decoupling the AIMs that are supported, rather than extending ASPECTJ to support dynamic deployment of aspects.

There are certain similarities between **perscope** and CAESAR [10] wrappers. The ASPECTJ built-in AIMs can be implemented with wrapper instances in CAESAR. Wrapper instances roughly correspond to aspect instances in ASPECTJ, but they can be manually instantiated and associated with objects. In addition, the wrapper recycling mechanism helps to retrieve associated wrappers from objects. Other AIMs can be implemented as well. Both do not need new keyword to express semantics for new AIMs, making the language simpler. Both can express semantics that cannot be easily expressed in ASPECTJ. There are also clear differences between the **perscope** approach and CAESAR. In CAESAR the aspects are instantiated explicitly, while with **perscope**, aspects are instantiated implicitly according to the aspect declaration, just as in ASPECTJ.

Association aspects [12] is a mechanism for associating an aspect instance with a group of objects. This mechanism introduces the **new** keyword, which allows to instantiate an aspect. The aspect instantiation is explicit. In comparison, a dedicated aspect instance per pair of objects or per group of objects can also be achieved with **perscope**. A class that manages the logic of such aspect instantiation can be implemented and used as an argument to the **perscope** keyword.

EOS [11] is an AOP langauge that has support for instance level aspect weaving. Instance level aspect weaving is the ability to differentiate between two instances of a class and to weave them differently if needed. It allows the developer to differentiate between instances. Their work characterizes the AOP languages in respect to pointcut

language richness and instance level aspect weaving support. This differs from AIMs, which is the focus of **perscope** aspects.

Skotiniotis et al. [13] claim that aspect instantiation is an essential feature in AOSD language. They show that the programs complexity increases in situations where multiple instances of aspects are present. They contrast ASPECTJ with the approach of aspect instantiation taken by ASPECTS [5]. ASPECTS permit the instantiation of an aspect (via **new**) and the activation of an aspect (via install). Obtaining a reference to an aspect instance in ASPECTS is a matter of a simple assignment at instantiation time, and deployment of the aspect becomes as simple as a call to its install method.

6 Conclusion

Various AOP languages take different approaches regarding aspect instantiation. Some of them grant the programmer explicit control over when and where aspects are instantiated. In these languages, the aspect instantiation mechanism is extensible.

ASPECTJ hides the instantiation details from the programmer and permits the programmer to specify the instantiation model only in a declarative way. In ASPECTJ there are six built-in AIMs. Their usage is simple; the programmer just needs to declare the aspect and specify the AIM by using the appropriate keyword. However, the aspect instantiation mechanism in ASPECTJ is not extensible. It is not possible to specify strategies other than those built-in to the language.

This work contributes a framework for enabling the definition and use of third-party AIMs in ASPECTJ. To support pluggable AIMs we introduce a new **perscope** keyword to ASPECTJ. **perscope** is both declarative and extensible. It lets one define and implement various AIMs that are not part of the ASPECTJ language. These AIMs may be provided by third-parties. Once a new AIM is implemented, it can be immediately used by programmers in a declarative way, in the spirit of ASPECTJ.

References

1. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, March 17-21 2003. ACM.
2. P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 87–98, Chicago, Illinois, USA, March 14-18 2005. ACM.
3. A. Colyer. Implementing caching with AspectJ (blog). The Aspect Blog, `http://www.aspectprogrammer.org/blogs/adrian/2004/06/`, June 2004.
4. T. Cottenier, A. van den Berg, and T. Elrad. Stateful aspects: the case for aspect-oriented modeling. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 7–14, New York, NY, USA, 2007. ACM.
5. R. Hirschfeld. Aspect-oriented programming with AspectS. In M. Akşit and M. Mezini, editors, *Proceedings of the 3rd International Conference Net.ObjectDays, NODe 2002*, pages 219–235, Erfurt, Germany, October 7-10 2002.

6. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. Springer Verlag.
7. K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the Law of Demeter with AspectJ. In AOSD'03 [1], pages 40–49.
8. D. H. Lorenz and V. Trakhtenberg. Perscope aspects: Decoupling aspect instantiation interface and implementation (poster). In *IBM Programming Languages and Development Environments Seminar*, Haifa, Israel, April 14 2010. IBM Research - Haifa.
9. H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction*, number 2622 in Lecture Notes in Computer Science, pages 46–60, 2003.
10. M. Mezini and K. Ostermann. Conquering aspects with caesar. In AOSD'03 [1], pages 90–99.
11. H. Rajan and K. Sullivan. 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*, pages 297–306, New York, NY, USA, 2003. ACM Press.
12. K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 16–25, Lancaster, UK, March 17-21 2004. ACM.
13. T. Skotiniotis, K. Lieberherr, and D. H. Lorenz. Aspect instances and their interactions. In *Proceedings of the AOSD'03 Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT'03)*, Boston, Massachusetts, March 18 2003. ACM.
14. D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In AOSD'03 [1], pages 21–29.
15. E. Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 168–179, Brussels, Belgium, 2008. ACM.
16. E. Tanter, J. Fabry, R. Douence, J. Noyé, and M. Südholt. Expressive scoping of distributed aspects. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 27–38, Charlottesville, Virginia, USA, 2009. ACM.