Spoon: Annotation-Driven Program Transformation — the AOP Case

Renaud Pawlak
INRIA Futurs, LIFL, France
renaud.pawlak@inria.fr

ABSTRACT
This paper presents Spoon and its AOP extension. Spoon is a pure Java 5 framework for implementing source-level and annotation-driven program transformations. It aims to be a powerful tool to build and integrate middleware. Spoon allows for the definition of program processors and annotation processors that use Compile-Time reflection, which is achieved with an extension of Sun's APT. In particular, Spoon provides an AOP extension under the form of a set of annotation processors. With Spoon, it is possible to do comprehensive and efficient AOP in pure Java, without relying on any specific language or IDE support.

Categories and Subject Descriptors
D.3.4 [**Processors**]: Code Generation, Preprocessors.

Keywords
Program Transformation, Annotations, Compile-Time Reflection, AOP.

1. INTRODUCTION
Program transformation techniques are a powerful way to increase software productivity. They are widely used in middleware, to generate the software infrastructures and seamlessly integrate technical concerns into the end-user components. Moreover, they have been increasingly used for business code, in order to automatically integrate the latter to middleware layers and to achieve better separation of concerns within the application design. In particular, program transformation techniques can deal with crosscutting concerns by applying systematic changes throughout numerous program elements.

An issue of program transformation is the parameterization of the transformations. In other words, how does the programmer specify *where* and *when* the transformations occur? Typically, the parameterization can be done by the programs themselves (leading to complex configuration code and maintainability issues), or by external configuration files (leading to complex infrastructure and IDE support). Other attempts for managing program transformations consist of defining new languages. In Java, AOP solutions such as AspectJ [4], JAC [5], JBoss-AOP [1], and Spring-AOP [3] can be seen as transformation tools that include configuration primitives under the form of pointcut languages. However, theses solutions necessitate non-standard tool support (for instance the AJDT plugin for Eclipse) and require the learning of new languages. AOP frameworks (JAC, JBoss-AOP, Spring-AOP) rely on more standard tools than the AOP languages (AspectJ). However, since they are pure Java solutions, frameworks are less efficient and can hardly benefit from good IDE and language support when it comes to matters such as navigation, contextual help, and debugging.

The new version of Java (Java 1.5 Tiger, or Java 5 for short) introduces a mechanism called *annotation*, or metadata, which allows the programmers to parameterize their programs with standard and natively supported Java constructs. Java 5 annotations open the door for a flexible and standard way for transformation parameterization, which can be fully integrated and supported in the programs, as well as in the standard IDEs such as Eclipse.

In this paper, we present Spoon and its AOP extension, Spoon-AOP. Spoon is a program transformation framework that fully takes advantage of the new features provided by Java 5, especially the annotation feature. In section 2, we give an overview of the Java 5 annotation mechanism. In section 3, we explain the Spoon implementation and architecture, and show how the program transformations can be easily defined and parameterized in pure Java. In section 4, we then apply Spoon to implement Spoon-AOP. We then evaluate our implementation and briefly show some concrete applications in the last two sections.

2. THE JAVA 5 ANNOTATION FEATURE
Here, we give an overview of the Java 5 annotation feature and its associated tool: APT (Annotation Processing Tool) [6].

2.1 Java 5 Annotations
In Java 5, programmers can define a new annotation type similarly to a new class or interface. The goal of an annotation type is to provide the definition of some metadata that will be attached to some program elements (such as classes, fields, and methods). For instance, an annotation can be defined to limit the number of elements in a stack so that no sub-classing or parameterization code is needed to ensure that the used stack is bounded. To do so, we can define the following annotation type:

```
public @interface Bound {
    int max() default 10;
}
```


AOMD '05, November 28- December 2, 2005 Grenoble , France

Once this annotation type is defined, the programmer can instantiate it by annotating the stack class:

```
@Bound(max=5)
public class Stack<T> {
  List<T> elements=new Vector<T>();
  public void push(T element) {
    elements.add(0,element);
  }
  public T pop() {
    T element=elements.get(0);
    elements.remove(0);
    return element;
} }
```

In this program, an annotation of the `Bound` type is instantiated and attached to the `Stack` class. The annotation values are initialized with the values given during the annotation's construction (here, the only value is `max`). Note that only constants of primitive types, strings, enum, classes and single dimension arrays of those are allowed in an annotation.

2.2 APT

Coming with the JSDK, Java 5 provides the Annotation Processing Tool (APT) [10]. APT is an extension of the Java compiler (*javac*) that implements some hooks towards programmatically defined annotation processors. An annotation processor is created by a factory, and is in charge of dealing with some annotations at compile-time. As such, it declares the annotation set it consumes and instantiates the processors that will process the annotations.

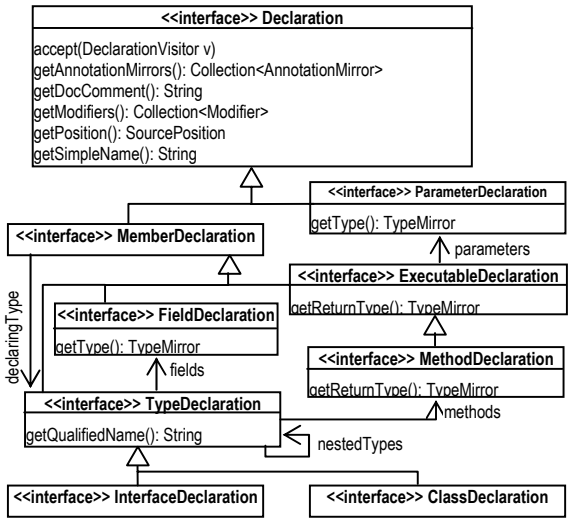

Figure 1. APT meta-model.

In APT, processors access to the compilation environment, which contains an annotation-aware meta-model defined in the `com.sun.mirror.declaration` package. This meta-model can be visited by implementing the `com.sun.mirror.util.DeclarationVisitor` interface (visitor pattern). Typically, a processor will generate a set of new files out of the processed program. These new files will then be recursively processed by APT, so that the processing repeats itself until all the annotations are consumed by the processors.

Figure 2 shows an (incomplete) excerpt of the APT declaration API: This can be seen as an abstraction of the Java AST or, also, as a meta-model for Java 5. Note that this meta-model also fully supports *generics* (formal parameters), *varargs*, and *enums*, which have not been shown in the figure. Hence, it is a useful API for introspecting Java 5 programs.

APT only allows for the reading access of the programs, so that it can be useful to generate new elements out of the programs (such as documentation, and proxies to existing class) but it is harder to modify existing programs. Besides, it is impossible to access the bodies of the methods with this API.

3. SPOON

In this section, we present our Spoon framework: an annotation-driven program transformation tool implemented on the top of APT.

3.1 The Compile-Time Reflection API

The goal of Spoon is to allow the programmer to specify program transformations, which are parameterized with Java 5 annotations. Beyond a program transformation tool, Spoon must be seen as an open compiler built on the top of APT and using compile-time (CT) reflection [4]. CT reflection is a branch of meta-programming that allows the programmers to modify the semantics of the language by implementing meta-programs, which are able to modify the programs' structure and behavior. To do so, Spoon provides the user with a meta-model that is an extension of the `com.sun.mirror.declaration` API. This meta-model is defined in the `spoon.reflect` package and allows both for reading and writing. Each interface of this package is a CT program element (`CtElement`), which extends the corresponding declaration of the mirror API in order to provide the modifiers to the model elements, as shown in figure 2 (`Declaration` and `MemberDeclaration` cases).

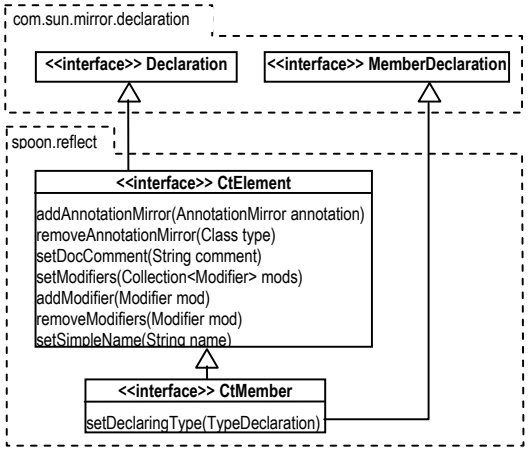

Figure 2. Spoon meta-model.

Note that Spoon allows for accessing and modifying the code of the executable elements (methods and constructors), a feature not provided by APT.

3.2 The Program Processing API

The Spoon framework contains a `spoon.processing` package that defines the API for code processing in general. The main interface is the `Processor` interface:

```
public interface Processor {
  CtEnvironment getEnvironment();
  void setEnvironment(
    CtEnvironment environment);
  void process(CtElement element)
    throws ProcessingException;
}
```

Spoon users can implement this interface to define a new code processor. At compile-time, Spoon will visit (visitor pattern) the program model and will up-call the `process` method of all the registered processors. The processors can then perform any program checks or transformation by using the currently visited element (passed as a parameter) and the compile-time environment, which is set by Spoon, and that allows the processor to access the currently compiled program through the CT reflection API described in section 3.1.

Since the philosophy of Spoon is to trigger transformations based on program annotations, a more specific kind of processor is provided:

```
public interface AnnotationProcessor
       extends Processor {
  void process(Annotation annotation,
               CtElement element)
    throws ProcessingException;
  Collection<Class<? extends Annotation>>
    getProcessedAnnotationTypes();
  Collection<Class<? extends Annotation>>
    getConsumedAnnotationTypes();
}
```

An *annotation processor* processes annotated element and takes the currently processed annotation as an additional parameter. It also must define the set of the processed annotations so that Spoon knows when to trigger the processor. When annotation processing leads to annotation consuming, the annotations should be removed from the model. To do so automatically, you can define the consumed annotations by implementing the `getConsumedAnnotationTypes` method. For program simplification, Spoon provides default abstract implementations of these interfaces.

Once the program has been processed, Spoon automatically generates and compiles the processed code. The code generator is a simple pretty printer implemented as a visitor of the `spoon.reflect` metamodel.

3.3 Example

In this section, we implement a simple processor for the `Bound` annotation defined in section 2.1. This processor transforms the annotated stack classes so that the `push` method implements a test to check that the `elements` collection's size of the stack is bounded by the `max` value. For the sake of the example, this processor is specific to our stack but it would be possible to generalize it by adding parameters to the `Bound` annotation. In our simple case, the processor is defined as follows:

```
public class BoundProcessor
    extends AbstractAnnotationProcessor {
  public BoundProcessor () {
    addConsumedAnnotationType(Bound.class);
  }
  public void process(Annotation a,
    CtElement e) throws ProcessingException {
    CtMethod m = ((CtClass) e).getMethod(
      "push", new ParameterType("T"));
    m.setBodyExpression(
      "if (elements.size()>=" +
      ((Bound) a).max() +
      ") { throw new RuntimeException\
      (\"overflow\"); }" +
      m.getBodyExpression());
  }
}
```

This code is a typical CT metaprogram. We first get the CT representation of the `push` method. Then, we set its body so that the new body is formed of the test expression followed by the old body expression. When our processor is applied, Spoon automatically removes the `Bound` annotation from the processed program, since `Bound` is defined as a consumed annotation. Thus, the generated code for our stack is:

```
public class Stack<T> {
  public void push(T element) {
    if (elements.size()>=5) {
      throw new RuntimeException("overflow");
    }
    elements.add(0,element);
  }
  public T pop() {
    T element=elements.get(0);
    elements.remove(0);
    return element;
} }
```

CT reflection is a very flexible technique. Any transformation can be applied to the program by using the `spoon.reflect` API. For instance, new interfaces, methods, or fields can be added or removed, and method bodies can be manipulated as string. Through the Spoon factory, new classes or interface can be generated from scratch. In middleware, this can be particularly useful for generating new delegators or proxies that shall be implemented differently depending on some architectural information.

3.4 Configuration

In order to be effective, Spoon needs to be configured so that it knows what code needs to be processed and what are the processors to be used. The most straightforward way of doing this is to use the Ant task provided by Spoon. For our stack example, a typical configuration would be:

```
<target name="stack.example">
  <mkdir dir="${build}" />
  <taskdef name="spoon"
           classname="spoon.SpoonTask"> ...
  </taskdef>
  <spoon
      sourceRootDir="examples/stack"
      spoonedDir="${spooned}"
      destinationDir="${build}" ...>
    <processor type="BoundInstrumentor"/>
  </spoon>
</target>
```

In the configuration, `sourceRootDir` is the directory where to find the source code, `spoonedDir` is the intermediate directory where the processed and generated code is stored, `destinationDir` is the target directory for the compiled classes, and `processor` defines a processor to be used for processing the source code. Note that any number of processors can be declared here and that Spoon will apply them in the order they have been declared.

4. THE AOP CASE

Spoon can be used to implement any kind of program transformation. Amongst those, AOP transformations deserve to be focused on since they provide some basic mechanisms for better separation of concerns, which is an important property when developing or using middleware. In this section, we discuss how to implement AOP with Spoon and we evaluate our solution compared to other AOP solutions.

4.1 Spoon-AOP

Spoon-AOP defines a set of annotations and program transformations that can be used for AOP. Our goal here is not to support all the concepts provided by complex AOP languages or frameworks, but to demonstrate that, thanks to Spoon, we are able to implement efficient AOP in pure Java 5. Moreover, Spoon-AOP differs from classical AOP approaches since it is entirely annotation driven.

As a consequence of being annotation driven, AOP with Spoon-AOP consists of annotating the base program to define where and how the aspects weave to the base program. Spoon-AOP defines three main annotations:

- `Advised({list_of_advice_classes})`: this annotation is applicable to methods or constructors to indicate that they are advised by some advice defined in some external classes. Note that we only support execution advising (although calling advising would be possible to implement).
- `Mixined({list_of_mixin_classes})`: this annotation is applicable to classes to indicate that some mixin classes are applied. All the methods and fields defined by the mixin classes will be introduced into the target class definition. If one of the mixin classes is an interface, then the target class will automatically implement this interface.
- `CFlow ({list_of_cflow_types})`: this annotation is applicable to methods or constructors to indicate that they are the starting points of the cflow types given as parameters.

Other types of annotations are also defined by Spoon-AOP. For example, the `AdviceClass` annotation is applicable to classes to define a class containing advice methods. On each advice method, an `Advice` annotation of the form `Advice(type=AdviceType, context=AdviceContextType)` must be defined, where `AdviceType` is an *enum* (`BEFORE`, `AFTER`, `AROUND`), and `AdviceContextType` is also an enum that defines the context that can be accessed by the advice method. For instance, you can use the `TARGET_ONLY` context if you do not need to access any parameters of the advised method execution and `FULL` if you want to access its parameters.

4.2 Example

Let us now assume that we want to implement with AOP the same bounded stack we implemented in section 3.4. For this we need to define a before advice:

```
@AdviceClass
public class BoundAdvice {
  @Advice(
    type = AdviceType.BEFORE,
    context = AdviceContextType.TARGET_ONLY)
  public void checkBound(Object target) {
    if(((Stack)target).elements.size() >= 5) {
      throw new RuntimeException("overflow");
} } }
```

The advice is then applied using annotations:

```
public class Stack<T> {
  List<T> elements=new Vector<T>();
  @Advised(BoundAdvice.class)
  public void push(T element) {
    elements.add(0,element);
  }
  public T pop() {
    T element=elements.get(0);
    elements.remove(0);
    return element;
} }
```

Note that this program will only compile if the `elements` field is accessible from the `BoundAdvice` class. This is not always the case. When the field is private for instance, a good way to access it is to use a mixin. Here, we can create a new interface implemented with a mixin:

```
public interface Access {int getSize();}
public class AccessMixin implements Access {
  List elements; // not reintroduced in target
  public int getSize() {
    return elements.size();
} }
```

Then, you need to apply the mixin to the stack class:

```
@Mixin({Access.class,AccessMixin.class})
public class Stack<T> { ...
```

Finally, you can change the implementation of the advice to access the size with the `Access` interface:

```
... if(((Access)target).getSize()>= 5) { ...
```

The advantage of this solution compared to the processor of section 3.4 is obvious: the added code is not manipulated as strings anymore, but as a regular Java program. As a consequence, it is easier and safer to implement.

4.3 Pointcuts

One could object that Spoon-AOP breaks the obliviousness principle. Indeed, base programs need to be annotated and have some (restricted) knowledge of the aspects. When the obliviousness principle really needs to be preserved, one can define a pointcut, which is a Spoon processor that annotates the program in order to trigger the AOP processors in a seamless way. For instance, to make our stack oblivious to the aspect, we can define the following processor:

```
public class StackPointcut
  extends AbstractProcessor {
 public void process(CtElement element)
    throws ProcessingException {
  if((element instanceof CtClass) &&
    element.getSimpleName().equals("Stack")) {
   CtFactory f=getEnvironment().getFactory();
   f.annotate(element,Mixined.class,"value",
              Access.class,AccessMixin.class);
   CtMethod method=((CtClass)element)
    .getMethod("push",new ParameterType("T"));
   f.annotate(method,Advised.class,"value",
              BoundAdvice.class);
} } }
```

This pointcut programmatically creates the `Mixined` annotation on the `Stack` class and the `Advised` annotation on the `push` method. Note that this processor must be declared before the `Mixined` and `Advised` processors in the configuration.

4.4 Evaluation

The AOP processors provided by Spoon perform the same kind of transformations as AspectJ [6] ones; the runtime performances are therefore similar. For instance, in our example:

```
public class Stack<T> implements Access {
  List<T> elements=new Vector<T>();
  public void _org_push(T element) {
    elements.add(0,element);
  }
  public void push(T element) {
    BoundAdvice.__INSTANCE__.checkBound(this);
    _org_push(element);
  }
  public int getSize() {
    return elements.getSize();
  }
  ...
```

The main difference with AspectJ is that the weaving and its tuning are more explicit. For instance, if you want to access the parameters of a method execution, in an advice method, you need to use the context type `FULL`. Note that the parameters are then accessed as they are declared in the advised method prototype: there is thus no performance loss and type safety can be ensured by the compiler. To access the parameters as an array of objects (which is more generic but slower and not type safe), you can use the context type `GENERIC`. In a similar way, you can tune a mixin to be directly introduced into the target class (default policy), or to be kept as a standalone object to which the target class delegates. The former strategy is more efficient, since it saves one delegation invocation. The latter one (the one adopted by AspectJ) is less efficient but makes the generated code easier to debug.

Compared to other frameworks (such as JAC [7], Spring-AOP [4], and JBoss AOP [1]), Spoon is by far more efficient since it avoids reflection or proxying when unneeded. It is also more naturally integrated within existing development environments.

Since we use standard Java 5 annotations to weave and tune the aspects, the AO programs are more straightforward to maintain than with other AO frameworks or languages. For instance, you can fetch all the methods of your project advised by a given advice type, simply by using a standard reference searching tool (`ctrl+alt+G` in Eclipse). You can also navigate directly from an advised method to the corresponding advice class (`F3` in Eclipse). The same principles apply to mixins. When the annotations are programmatically defined through a pointcut processor such as the one presented in section 4.3, Spoon can be configured to generate an intermediate annotated program so that the same principles apply too.

Last but not least, Spoon can fall back on CT reflection [2] when AOP reaches its limits. Reflective processors are indeed more suited for specific optimizations, parameterized aspect code for generic aspects, or for generating new classes.

5. APPLICATIONS

We have used Spoon and Spoon-AOP to develop specific frameworks for middleware component integration. These frameworks are annotation-driven, so that they are easy-to-use for the end-user. In particular, we have implemented an annotation-driven EJB-like environment (based on a Bean model called Spoon-Bean). With Spoon-Bean, most of the EJB-specific features (such as Home or Remote interfaces) can be automatically implemented and configured by using annotations. In addition, some deployment information can be added through annotations.

We have also developed Pepper, a persistence framework that uses Spoon-AOP and that can be used consistently with the Spoon-Bean model. Pepper uses annotations to define the O/R mapping and uses Spoon-AOP to seamlessly read and write the data from and to the database trough JDBC. The following code show an excerpt of a typical Spoon bean, which is configured to be persistent.

```
@Mixined(PersistentObjectMixin.class)
@TableName("T_COMPANY")
public class Company {
    @ColumnName("NAME")
    @ColumnConstraints({"NOT NULL"})
    @Getter("getName")
    String name;

    @OppositeRole("company")
    List<Employee> employees
      =new Vector<Employee>();

    @Advised({RoleAdvice.class,
              PersistentObjectAdvice.class})
    public void addEmployee(Employee employee) {
        employees.add(employee);
    }

    @Advised(PersistentObjectAdvice.class)
    public List<Employee> getEmployees() {
        return employees;
    }

    @Advised(PersistentObjectAdvice.class)
    @GetterFor("name")
    public String getName() {
        return name;
    }

    @Advised(PersistentObjectAdvice.class)
    public void setName(String name) {
        this.name = name;
    }
}
```

Like in regular beans, our framework tries to deduce as much information as possible using naming conventions and program introspection. For instance, it finds the getters and the setters through naming conventions. In addition, it also uses a default O/R mapping policy, which most of the time avoids unnecessary configuration. In this context, annotations are used to tune the bean model and the O/R mapping.

For instance, the `Getter` annotation explicitly defines a field's getter. Also, the bean model allows for the automatic handling of associations through the use of the `OppositeRole` annotation.

For Pepper O/R mapping, the tuning can be done by using, for instance, `TableName` (that defines the name of the table where the class' instances are stored), `ColumnName` (that defines the name of the column that corresponds to the annotated field), or `ColumnConstraints` (that defines a DBMS-specific constraint for the column that corresponds to the annotated field).

All the configuration information is then used in the advice and mixin classes, which implement all the logic that needs to be separated from the bean business model, such as association management (`RoleAdvice`) and the persistence concern (`PersistentObjectAdvice` and `PersistentObjectMixin`).

Of course, the goal of this paper is not to describe a full-fledged framework for persistence, but it is to show that it is possible to integrate middleware components and APIs by using Spoon. This integration is done in pure Java 5 and shows useful information to the final programmer, so that the configuration is more straightforward and more flexible than with classical frameworks. In particular, we think that AOP annotation-driven development is well suited for incremental development, where the application is built through several refinement steps that require frequent code-level reengineering.

6. RELATED WORKS

Annotations have been pointed out to be a powerful language construct for separating concerns [2], especially within the context of AOP [7][8]. The existing proposals consist of defining annotation-driven pointcut (like it is possible in AspectJ latest releases). However, these approaches suggest that the programmer should define semantic annotations, which requires an intermediary definition step that is not always straightforward at early development stages. Spoon-AOP provides core-level AOP annotations (`Advised`, `Mixined`, `CFlow`) that have several advantages over semantic annotations. Firstly, they are more straightforward to use, since their effect is well-known and locally defined. Secondly, they take as parameters the Java types that define the associated aspect behaviors; it implies that they are always well-typed and that there is direct IDE support to manage the dependencies between the program and the aspects. This is not always the case when using semantic annotations which introduce an indirection level that can be sometimes tedious to understand.

Spoon widely uses CT reflection [4]. Several tools and open compilers based on this principle have been defined in C++ [3], Java [11], or other languages. Spoon innovation compared to early approaches is that it provides annotation-driven CT reflection. It takes advantage of the annotation feature to allow the Java language to be extended without having to introduce any particular syntactic sugar or ad-hoc mechanisms (source code comments, for instance), like it is the case in languages that do not support annotations in a native way.

7. CONCLUSION

Through this paper, we have presented Spoon, a program transformation tool that takes advantage of Java 5 annotations to define and parameterize user defined transformations. The kernel uses a CT reflection engine that we built on the top of APT. We have shown how to do AOP with Spoon and compared our approach with other AOP tools. We demonstrated that Spoon-AOP is flexible, efficient, intuitive, and well integrated into classical development environments without any requirements for dedicated plugins.

Because Spoon allows for annotation-driven program transformation and generation by using AOP and CT reflection, it is a tool suited for integrating middleware components. In the context of Spoon, the integration can be done by defining annotations that drive the deployment of the application on middleware layers. As shown in section 5, our first experimentations in this direction are promising.

8. REFERENCES

[1] Burke B. et al. JBoss AOP. http://www.jboss.org/products/aop

[2] Cachopo J. Separation of Concerns through Semantic Annotations. *In Companion of OOPSLA 2002.*

[3] Chiba S. A Metaobject Protocol for C++. *In proceedings of OOPLSA 1995*.

[4] Chiba S. A Study of Compile-time Metaobject Protocol. *PhD Dissertation, 1996*.

[5] Johnson R. J2EE development frameworks. *Computer, volume: 38, issue: 1, Jan. 2005.*

[6] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., G. Griswold W. An Overview of AspectJ, *In proceedings of ECOOP 2001*.

[7] Kiczales G., Mezini M. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. *In proceedings of ECOOP 2005*.

[8] Laddad R. AOP and Metadata: a Perfect Match. *AOP@Work, IBM, 2005*.

[9] Pawlak R., Seinturier L., Duchien L., Florin G. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. *In proceedings of Reflection 2001*.

[10] SUN. Annotation Processing Tool. http://java.sun.com/j2se/1.5.0/docs/guide/apt/

[11] Tatsubori M., Chiba S., Itano K., Kilijian M.-O. OpenJava: A Class-Based Macro System for Java. *In proceedings of OORaSE 1999*.