TWO-STEP WEAVING WITH REFLECTION USING ASPECTJ

Pierre-Charles David, Thomas Ledoux & Noury M. N. Bouraqadi-Saâdani [*]
{pcdavid, ledoux}@emn.fr, bouraqadi@ensm-douai.fr
Ecole des Mines de Nantes - Dpt. Informatique - 44307 Nantes Cedex3 - France

A. Introduction

While Aspect-oriented programming (AOP) [Kic+97] enables modularization of crosscutting concerns, Reflection [Smi82] is commonly recognized as a valuable instrument for separation of concerns [Coi99]. In this short paper, we explore the relationships between reflection and AOP. This study is carried out in the context of a project about dynamic adaptability of distributed systems [RNTL00]. We propose a combination of the two approaches to obtain the best of both worlds. The proposed prototype shows that AspectJ [Kic+01] and run-time reflection in Java can be combined together to support run-time weaving.

B. AOP and Reflection

In this section, we explore the relationships between reflection and AOP, and we propose to discuss AOP terminology in the context of reflection.

1. Separation of Concerns

AOP allows the separation of multiple concerns (aspects) in the software development process. Reflective programming also allows *separation of concerns* [HL95] because it provides a clear separation between the program code (the base-level) and its description and/or control (the *meta-level*). In a reflective OO language, base-level objects are controlled by objects at the meta-level called *meta-objects*. This relationship is materialized by a link between base and meta-objects, named *meta-link*. Meta-objects control of base-level objects follows a precise protocol named a Meta-Object Protocol (MOP) [KdB91].
Finally, meta-objects, because they can control the way non-functional concerns (such as persistence, synchronization, tracing, etc) affects base-level objects, can be used to represent aspects. Figure 1 shows two kinds of meta-objects concerning a distribution aspect.

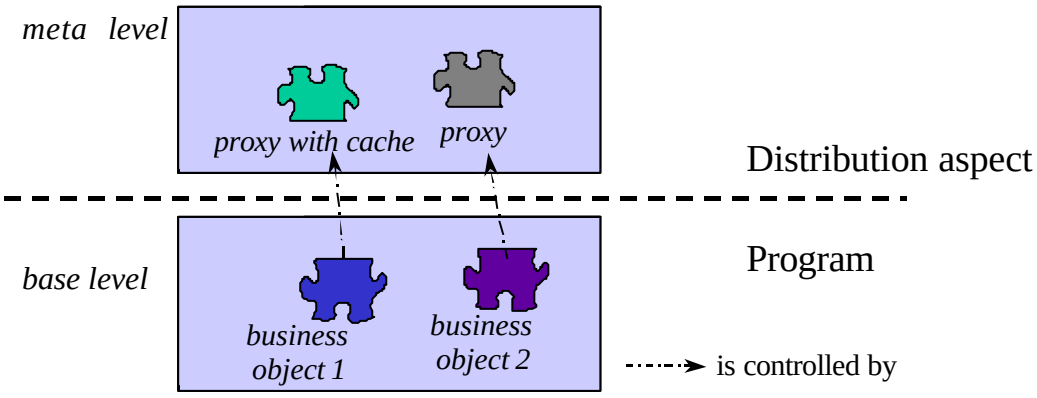

Figure 1-Meta-objects and aspects

[*] *Author's current address: Ecole des Mines de Douai - Dpt. GIP - 59508 Douai Cedex- France*

2. Weaving in Reflection

Separation is a good idea only if we can recompose the separated concerns together! The AOP paradigm proposes the *weaving* process to compose the different concerns. This consists in a coordination process between aspect code and non-aspect (basic program) code at the appropriate *join points*, i.e. key points in the dynamic execution of the program. How to weave? When to weave? are recurrent questions for AOP systems designers.

2.1 *How to Weave?*

In a reflective system, base objects and meta-objects are defined separately and then connected through the meta-link. Functionalities provided by each base object are then executed using the mechanisms defined in the meta-objects it is linked to.

Thus, the answer to "How to weave?" in the context of reflection can be formulated in the following way: jump from the base-level – via the meta-link – to the meta-level and execute the meta-object code (which implements the aspects). This jump is done at some key places in the base-level code with that we named *hooks*: fragments of code implementing an indirection to the meta-level.

Figure 2 illustrates this idea. An object `hello` receives the message `sayhello`. This message is trapped by a hook and redirected to `hello`'s meta-object which executes the aspect code (in this case, generating a log message) before invoking the base-level method.

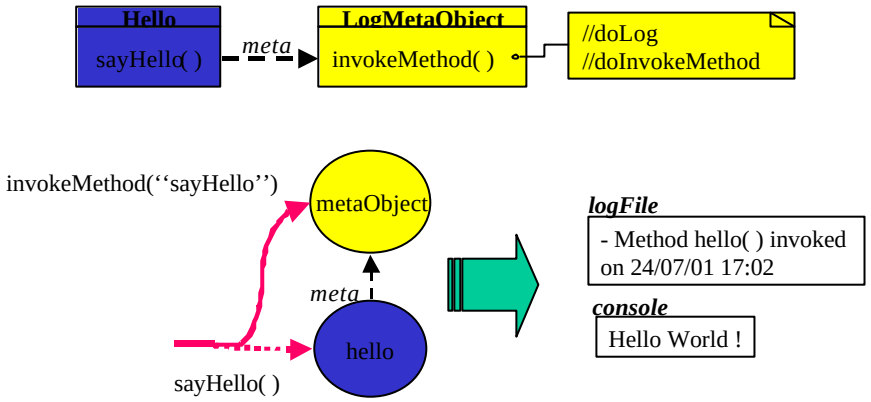

Figure 2-How to weave with meta-objects

2.2 *When to Weave?*

In [CV01], the authors propose a classification of MOPs that uses as a criterion the time period in the life of a program when meta-objects are actually in use. The three periods of time they consider are *compile-time*, *load-time* and *run-time*. Since meta-objects can represent aspects, we can conclude that it should be possible to develop AOP systems – with reflection – dealing with compile-time, load-time or run-time weaving.

Since our goal is to support run-time adaptability, we focus on run-time MOPs that allow postponing weaving to run-time. In the new classification proposed by [CV01], the *proxy-based run-time MOP* approach seems to be the most interesting one to build run-time MOPs in Java since it guarantees the VM portability. Proxy-based run-time MOPs introduce hooks into the program in order to reify run-time events such as method invocation or access to fields. The hooks can be introduced either at compile-time, for example by using a compile-time MOP, at load-time by modifying the bytecode for a class or even at runtime by using objects that implement the proxy pattern [GoF95].

2.3 *Conclusion*

By studying AOP terminology in the context of reflection, we can notice that the join point model proposed in [Kic+01] shares common concepts with the proxy-based run-time MOPs. The join point model is used to weave the non-aspect (basic program) code with the aspect code at the appropriate join points, while proxy-based MOP introduce *hooks* in the program code in order to shift to the meta-level and execute the meta-object code.

C. **Our Solution: two-step weaving with Reflection using AspectJ**

As stated earlier, this study takes place in the context of a research about the dynamic adaptability of applications to their execution environment in distributed systems [RNTL00]. We are thus looking for the best solution to build adaptive applications.
We are interested in the join point model and the fine-grained poincuts declaration proposed by AspectJ, which allows to clearly compose aspects dealing with non-functional concerns. However, its compile-time approach to weaving does not suit our goals, as it prevents run-time adaptation.
The analogy between joint points declaration and hooks introduction gave us the idea to connect AspectJ to our proxy-based run-time MOP, named RAM (Reflection for Adaptable Mobility), developed during a former project [BSLS01]. Indeed, AspectJ proposes a rich set of join points where to introduce hooks reifying run-time events. This section presents this solution[1].

1. Principle

We propose a weaving between aspect code and non-aspect (basic program) code in two steps. The first one takes place in AspectJ at compile-time, the second one in RAM at run-time.

1.1 *1st step : in AspectJ*

At compile-time, we weave in a base-level class code – at the join points declared by user – a *generic aspect* introducing an indirection to our Java MOP (cf. C.2). We obtain a *loose weaving* where the generic aspect will be specialized at run-time via RAM. The proposed join points in AspectJ are used to place the hooks in the base-level class code. Instances of the resulting class can then be connected through a meta-link to some meta-objects complying with RAM's MOP.

1.2 *2nd step : in RAM*

At run-time, we can choose to use some meta-objects within the RAM library in order to introduce some particular aspect. Then, it is possible to dynamically attach and detach meta-objects to specific instances of the transformed class. We can adapt this attachment when needed, thus allowing to re-weave aspects at run-time. Because RAM includes a support for meta-objects composition, we can even attach multiple aspects to an instance thus achieving the composition of many aspects.

1.3 *Illustration*

We can now illustrate the software development process in such a context. Let `Account.java` be a standard Java source file. Transforming it into a reflective class involves introducing hooks at the appropriate join points, which have to be declared in a configuration file (cf. C.2). This file and the `.java` file are the input files of our prototype based on AspectJ. The output file is `Account.class`, a java byte code file which contains the appropriate hooks to jump to the meta-level in RAM (cf. left side of Figure 3). Then, we can create instances of class

[1] In this experiment, we replace the former program transformation of RAM to introduce hooks and we keep the RAM kernel and the meta-objects library.

`Account` in any standard `Main.java` and dynamically attach meta-objects implementing particular aspects to these instances. This allows run-time weaving (cf. right side of Figure 3).

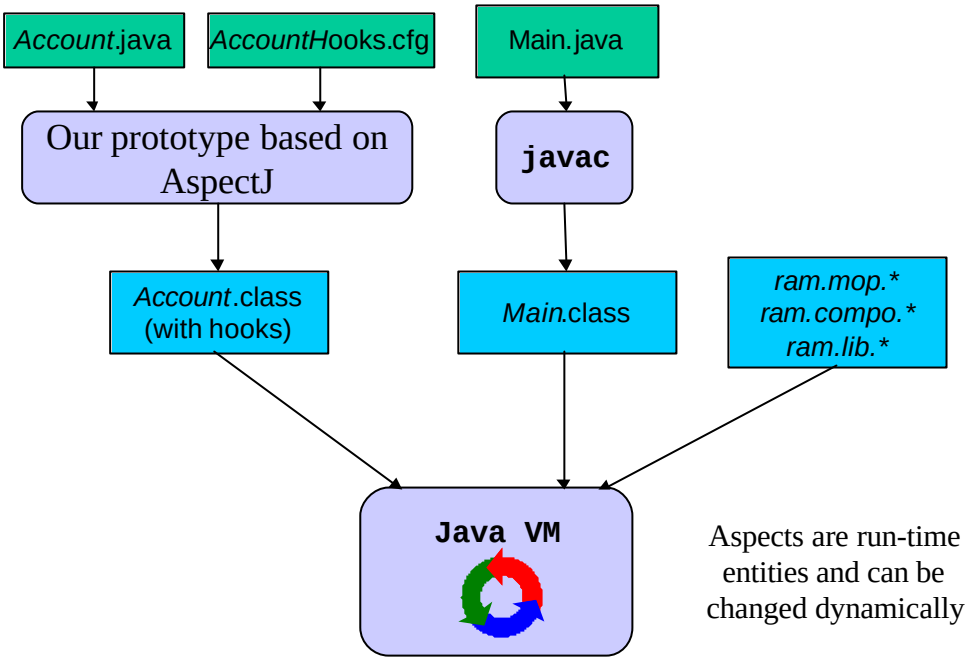

Figure 3-Software development process

2. Implementation Details

2.1 *Hooks declaration*

During the building of our prototype, our objective was to use AspectJ as a black box. Since users would have to declare the appropriate join points for hooks introduction, we had to define a simple syntax which can be mapped in AspectJ poincuts declaration. These declarations are put in a configuration file taken as input by our prototype.

Using the following example, the user can declare that calls to the `withdraw(double)` and `deposit(double)` methods in the class `Account` in the package `demo` should be handled at the meta-level:

```
[demo.Account]
invoke: * Account.withdraw(double)
invoke: * Account.deposit(double)
```

A configuration file contains the name of the class to transform and a list of hooks declarations, each being of the form `<kind>: <location>.` The kinds of hooks available are : `create` (object creation), `gets` and `sets` (fields access and modification), `invoke` (method invocation), and `serialize` (object serialization). They correspond to different kinds of join points and activate the MOP in the appropriate ways. The right-hand side of a hook declaration specifies the exact location of the hook using standard AspectJ notation for join point designators; this join point designator must be compatible with the kind of hook to introduce (i.e. it is not possible to introduce a method invocation hook on a join point designating a field access).

2.2 *Use of AspectJ*

Given a configuration file like the one described above, our tool generates one AspectJ source file for each class to be made reflective. Each such file defines a aspect where:

- Join points corresponds to the ones declared in the configuration file.
- Associated advices are generic hooks of the right kind to jump to the meta-level.

There is one generic advice per kind of hook (`create`, `invoke`…). Each of these advices have the same structure: first get contextual information (name and parameters of a method invocation for example), and then call the MOP with the correct parameters.

The contextual information is available from AspectJ through a variable named `thisJoinPoint`. This variable, similar to Java's `this` keyword, is a reification of the join-point which triggered the execution of the current advice. AspectJ provides a set of classes representing all possible kinds of join-points, from which it is possible to get the information necessary to call our MOP. The following example shows a simplified version of the generic hook used for method invocation, to illustrate this mechanism (others generic hooks have a very similar structure):

```
around() returns Object: invoke() {
  ReceptionJoinPoint jp = (ReceptionJoinPoint) thisJoinPoint;
  ReflectiveObject receiver = (ReflectiveObject) jp.getExecutingObject();
  MetaObject mo = receiver.getMetaObject();
  Method meth = findOriginalMethod(receiver.getClass(), jp.getSignature());
  return mo.invokeMethod(receiver, meth, jp.getParameters());
}
```

The informations necessary to call RAM (base-level object receiving the message, name and parameters of the method invocation) are all made available by AspectJ's reflective API. The `findOriginalMethod` method is used to retrieve the original, unhooked method to make it available to the meta-object, allowing its invocation from the meta-level without creating an infinite recursion. The `invoke()`pointcut in the above code could be replaced by whichever join points were declared by the user.

The actual implementation is actually a little more sophisticated: because these advices can be made fully generic, they are defined only once, in an abstract aspect called `ReflectiveAspect`. This aspect contains the definition of all these advices, but the join-points activating them (`invoke()` in the example) are made abstract. The *sub-aspect* generated by our tool thus contains almost only the concrete definitions for these join-points, and inherits all the advice code from its super-aspect, `ReflectiveAspect`. Figure 4 illustrates the concrete architecture of our implementation.

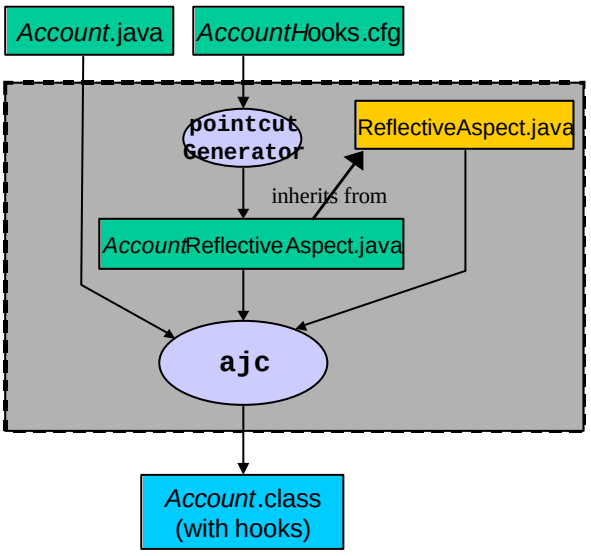

Figure 4

D. **Open Issues**

Because we rely on AspectJ for the hooks insertion, we inherit some of its strengths and weaknesses. First, the source code of the base-level code (but only the base-level code) is needed to insert hooks, and we need to know at compile time where to introduce hooks. This was not an issue in our experiment, but it could become one in other contexts, like reusable components development: how to choose the relevant join points in the program code? Although aspect weaving is made more dynamic, hooks are still introduced at compile-time! Other problems are the run-time overhead compared to AspectJ (but it is the price for more flexibility) and potential security issues which we did not explore.

E. **Conclusion and Future Work**

This experiment casts some light on the relationships between AOP and reflective programming. More specifically, it shows that aspects and meta-objects on the one hand, and join-points and hooks on the other hand have lots of similarities which would need to be explored further.

We were able to leverage AspectJ's very rich and fine-grained point cut model in the context of our run-time MOP to get the best of both worlds! We believe that the more general technique of generic advices introduction is a good way to reuse some of the work done on AspectJ (the point cut model) to explore new areas in AOP, and in particular dynamic weaving.

F. **Bibliography**

[BSLS01] Bouraqadi-Saâdani M.N., Ledoux T. and Südholt M. – A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based Implementation. Technical Report Ecole des Mines de Nantes, 2001.

[Coi99] Cointe P. Editor. – *Proceedings of Reflection'99*. LNCS 1616, Springer-Verlag, July 1999.

[CV01] Denis Caromel and Julien Vayssiere – *Reflections on MOPs, Components, and Java Security.* In proceedings of ECOOP'01. LNCS 2072, Springer-Verlag, June 2001.

[GoF95] Gamma E. and al. – *Design Patterns*. Addison Wesley Publisher, 1995.

[HL95] W. Hürsch and C. Videira Lopes – *Separation of concerns*. Technical Report NU-CCS-95-03, Northeastern University, 1995.

[KdB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow – *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), USA, 1991.

[Kic+97] Kiczales G. and al. – *Aspect-Oriented Programming*. In proceedings of ECOOP'97. LNCS 1241, Springer-Verlag, June 1997.

[Kic+01] Kiczales G. and al. – *An Overview of AspectJ.* In proceedings of ECOOP'01. LNCS 2072, Springer-Verlag, June 2001.

[RNTL00] RNTL 2000 – Projet ARCAD (Architecture Répartie extensible pour Composants ADaptables). Ecole des Mines de Nantes, France Telecom R&D, INRIA Rhône-Alpes, INRIA Sophia-Antipolis et laboratoire I3S. *In french*

[Smi82] Smith, B.C. – *Reflection and Semantics in a Procedural Languag*e. PhD: M.I.T. 1982.