Towards
Unified Aspect-Oriented Programming

Noury Bouraqadi Abdelhak Seriai Gabriel Leblanc

{*bouraqadi, seriai*}*@ensm-douai.fr*
École des Mines de Douai - Dépt. G.I.P.
941, rue Charles Bourseul - B.P. 10838
59508 Douai Cedex - France

Abstract

Aspect-Oriented Programming (AOP) is a paradigm that aims at improving software modularization. Indeed, aspects are yet another dimension for structuring applications. The notion of aspect refers to any crosscutting property. Such crosscutting can be either dynamic or static. Dynamic crosscutting refers to applications execution flow. While, static crosscutting refers to applications structure. Although many AOP approaches does enable these two kinds of crosscutting, this support is not always satisfactory. Aspects code is complex and often requires different constructs for expressing static and dynamic crosscutting. We present in this paper the foundation for an AOP platform that unifies the description of both kinds of crosscuttings. This solution relies on reflection and mixin-based inheritance.

Key words: aspect-oriented programming, static crosscutting, dynamic crosscutting, reflection, mixin-based inheritance

1 Introduction

Aspect-Oriented Programming (AOP) [15,10,11] is among key post-object paradigms that appeared during the last decade. This programming approach supports separation of concerns. Building an application using the AOP approach leads to defining on the one hand one *application core*, and on the other hand an arbitrary number of *aspects*. Application core is usually a set of classes. Aspects are concerns that cross-cut application core. Aspects are not only separated from application core, they are also isolated one from the other. Hence, AOP promotes modularization. Development responsibilities of aspects and application core can be dispatched among members of a project

team. Once all modules (aspects and application core) are ready, the full application can be "integrated" through the process of *weaving*.

The notion of aspect refers to any property crosscutting a software. Such crosscutting can be either *dynamic* or *static* [16].

Dynamic crosscutting: A crosscutting is said to be dynamic if it affects applications behavior, *i.e.* execution flow. The implementation of dynamic crosscuttings relies on the concepts of *pointcuts* (set of points within the execution flow) and *advices* (blocks of code to evaluate at some points of the execution flow).

Static crosscutting: A crosscutting is said to be static if it affects applications structure. The implementation of static crosscuttings relies on *introductions* of new building blocks (*e.g.* classes, methods, instance variables) and restructuring their relationships (*e.g.* inheritance).

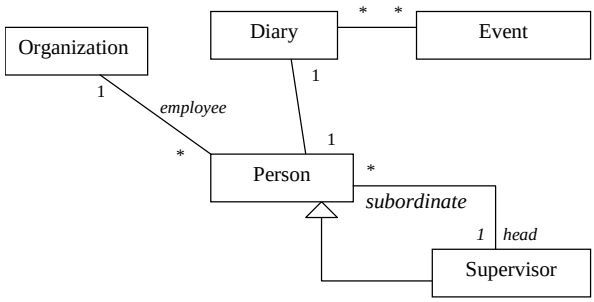

Fig. 1. Distributed Diary System Core

We illustrate these two of aspects using an example of a distributed diary system. Application core for this system is a set of classes describing employees, organization, diaries and events (see figure 1). This system can have different aspects. We present in the following two of them: log which illustrate dynamic crosscutting and absence management which illustrate static crosscutting.

The log aspect displays on a console traces describing system's activity. So, logs can be produced on the addition a new employee to the organization or events addition/removal to/from diaries. Log is a typical aspect with dynamic crosscutting. Trace production is triggered by the application execution. If no execution happens, the log aspect computations (*i.e.* logging) isn't performed. This is often the case of "infrastructure oriented" aspects.

The absence management aspect deals with employees vacations. Each person has a certain amount of available vacation days and can request vacations. Vacation requests has to be validated by the requester's boss before a new event is added to the requester's diary. Storing available vacation days requires a new instance variable to be inserted into the Person class. Handling requests and updating available vacation days count requires new methods to be inserted in classes Person and Supervisor. Absence management is a typical

aspect with static crosscutting. It extends existing classes with new instance variables and methods. This is often the case of "business oriented" aspects.

Among existing AOP approaches, some does support only one kind of crosscutting [7,2,8,9,19]. Others [16,13,21] do support both static and dynamic crosscuttings. But, these platforms lead quickly to complex code, even for simple aspects. And, often aspects definitions are non uniform: different constructs are used for expressing code introductions and advices. This is particularly true for AspectJ the most popular AOP language, as we show in section 2.1.

In this paper, we setup the foundations for a platform that unifies the programming of crosscuttings should they be static or dynamic. We use *reflection* [23,18] and *mixin-based inheritance* [6] to extend an object-oriented platform in order to supports AOP. No new language construct is needed, and only a minimal set of concepts is introduced and applied uniformly to express both static and dynamic crosscutting. We believe that this platform provides simplicity and uniformity, without scarifying expressiveness. Implementations of simple aspects remain simple, and those of complex aspects are still possible.

Reflection is the ability of a system to reason on and to act upon itself. In the context of programming languages, reflection provides developers with two programming levels: a *base-level* and a *meta-level*. The base-level is where applications building blocks (*i.e.* structure) are defined. The meta-level is where applications semantics (*i.e.* behavior) is defined. Programming within both base- and meta-levels is uniform since it relies on the same constructs in a reflective language.

Having access to applications structure and behavior is not enough to define aspects. Applications need to be decomposed so that each aspect definition is isolated and separated from the others. We use mixin-based inheritance to achieve this separation. Mixin-based inheritance is an alternative to multiple-inheritance which avoids automatic linearization issues. A mixin can be viewed as a subclass parametrizable with its superclass. In the proposed solution, each aspect is defined as a set of mixins. This description applies uniformly to express both static and dynamic crosscuttings.

With our model, each class of the application core is linked to a meta-level class. The process of weaving inserts mixins into class hierarchies. Mixins related to static crosscuttings are inserted into the hierarchy of base-level classes. While, mixins related to dynamic crosscuttings are inserted into the hierarchy of meta-level classes.

The remainder of this paper is organized as follows. Section 2 motivates the need for a platform supporting the definition of both functional and non-functional aspects. This motivation is illustrated using a distributed diary example that will be used throughout the paper. Then, foundations of unified

AOP based on reflection and mixin-based inheritance is described in section 3. Last, after discussion related work in section 5, section 6 ends the article ends with concluding remarks and some perspectives.

2 Motivation

In this section, we motivate the need of a unified AOP based on AspectJ, the AOP mainstream platform. This motivation is illustrated using the distributed diary example exposed in the introduction. We show some limitations of AspectJ when it comes to building reusable aspects with static and dynamic crosscutings, namely the log aspect and the absence management aspect.

2.1 Some AspectJ Limitations

2.1.1 A First Absence Management Aspect

```
01: public aspect SimpleAbsenceManagement {
02:  private int Person.vacationDaysCount;

03:  public int Person.getVacationDaysCount(){
04:    return this.vacationDaysCount;}

05:  public void Person.setVacationDaysCount(int newVacationDaysCount){
06:    this.vacationDaysCount = newVacationDaysCount;}

07:  public String Person.toString(){
08:    return "\nAvailable Vacation Days = " + this.getVacationDaysCount();}

09:  public int Person.defaultVacationDaysCount(){
10:    return 30;}

11:  pointcut constructorExec(Person aPerson):
12:    execution(Person.new(String, String)) && target(aPerson);

13:  after(Person aPerson): constructorExec(aPerson){
14:    aPerson.setVacationDaysCount(aPerson.defaultVacationDaysCount());}

                                    ...
```

Fig. 2. A simple implementation of the absence management aspect in AspectJ

Figure 2 gives a first version [1] of the absence management aspect in AspectJ.

[1] We provide here only part of the actual code of the aspect.

This aspect is not reusable since it directly refers to application core class Person. Indeed, a new field named vacationDaysCount is introduced in class Person for counting available vacation days. This vacation days counter is used for vacation requests (not shown on figure 2) and also for the string describing person returned by the toString() method.

In order to initialize the counter, the only possibility is to use an advice (lines 13 and 14) that acts after the execution of the constructor of class Person (lines 11 and 12). Figure 3 provides a simple program using the classes Person and Supervisor after weaving the SimpleAbsenceManagement aspect.

Evaluated Code

```
Supervisor chief = new Supervisor("Bart", "Simpson");
Person joe = new Person("Joe", "Dalton");
joe.setBoss(chief);
System.out.println("---println(joe)---\n" + joe);
System.out.println("---println(chief)---\n" + chief);
```

Console Display

```
---println(joe)----
Available Vacation Days = 30
---println(chief)---
Available Vacation Days = 30
Boss of 1 person(s)
```

Fig. 3. A code evaluated and its result after weaving the SimpleAbsenceManagement aspect

In this example, we can see different limitations of AspectJ. First, a simple extension of an existing code can lead to somewhat complex code. This is the case with the initialization of the vacationDaysCount field. Such initialization which simply requires extending an existing constructor is actually performed using a rather "unnatural" code based on a pointcut and an advice.

Another problem can arise on evolution. Suppose we add a toString() method into the Person class definition. In this case, AspectJ fails weaving the SimpleAbsenceManagement aspect and reports a conflict. The only solution to this conflict is to replace the defintion of toString() method provided by the aspect with pointcut and advice constructs (see figure 4). Note that the within(Person) condition in the pointcut description ensures that the advice is performed only once: for the toString() method defined within the Person class. Otherwise, the advice would be performed twice for instances of the Supervisor class, since this latter does redefine the toString() method and does and super send.

A similar problem arise when two aspects introduce methods with the same signature in the same class. In this case, weaving fails and one needs to rewrite at least one of the two aspects and replace method introduction with statements based on the pointcut and advice constructs.

```
1: pointcut toStringExec(Person aPerson):
2:  execution(String Person.toString()) && target(aPerson) && within(Person);

3: String around(Person aPerson): toStringExec(aPerson){
4:  String initialString = proceed(aPerson);
5:  return initialString +
6:    "\nAvailable Vacation Days = " + aPerson.getVacationDaysCount();}
```

Fig. 4. Replacement of the toString() method definition with a pointcut and an advice in the SimpleAbsenceManagement aspect

Note that AspectJ weaver does handle the case of homonymous fields. Fields scopes are restricted to aspects where they are defined. For example, let A1 and A2 two aspects that introduce within the same class fields of the same name. Methods introduced in A1 will access the field introduced in A1. And methods introduced in A2 will access the field introduced in A2. The same solution applies if an aspect introduces a field with a name already used in core application code. Although this solution is convenient for most cases, sometimes one may want to merge such fields in order to share data.

Last, we can note that using AspectJ one can easily end up "hardwiring" aspect definitions to a particular application. This is the case of the SimpleAbsenceManagement which explicitly refer to the Person class. In the following, we'll see that disciplined programming can avoid this pitfall and enable aspect reuse. However, we'll face other limitations.

2.1.2 A Reusbale Absence Management Aspect

Figure 5 provides the definition of a reusable absence management aspect [2] in AspectJ. Actually, there are two aspects. The first one AbsenceManagement (lines 1 to 15) is reusable cause not bound to any application code. The second aspect AbsenceManagementImpl (lines 16 and 17) extends the former with links to the core application code.

The AbsenceManagement introduces an new "marker" interface named AbsenceRequestor (line 2). All classes which implement this interface will be extended with members introduced in lines 3 to 10. The actual class which is extended this way is Person which is referenced in AbsenceManagementImpl (line 17). The Person class is linked to the marker interface AbsenceRequestor using the "declare parents" statement.

The use of a marker interface has a consequence on the pointcut declaration which enables the initialization of the vacationDaysCount field (lines 11 to 13).

[2] We provide here only part of the actual code of the aspect.

```
01: public abstract aspect AbsenceManagement {

02:  public interface AbsenceRequestor {}

03:  private int AbsenceRequestor.vacationDaysCount;

04:  public int AbsenceRequestor.getVacationDaysCount(){
05:    return this.vacationDaysCount;}

06:  public void AbsenceRequestor.setVacationDaysCount(int newCount){
07:    this.vacationDaysCount = newCount;}

08:  public int AbsenceRequestor.defaultVacationDaysCount(){return 30;}

09:  public String AbsenceRequestor.toString(){
10:    return "Available Vacation Days = " + this.getVacationDaysCount();}

11:  pointcut constructorExec(AbsenceRequestor requestor):
12:    execution(AbsenceRequestor+.new(..)) && target(requestor) &&
13:    !cflowbelow(execution(AbsenceRequestor+.new(..)));

14:  after(AbsenceRequestor requestor): constructorExec(requestor){
15:    requestor.setVacationDaysCount(requestor.defaultVacationDaysCount());}

                                    ...

16: public aspect AbsenceManagementImpl extends AbsenceManagement{
17:   declare parents : Person implements AbsenceRequestor;
                                    ...
```

Fig. 5. A reusable implementation of the absence management aspect in AspectJ

Because interfaces does not hold constructors, the execution statement should refer to constructors of classes implementing the interface. This is what the "+" refers to in the expression execution(AbsenceRequestor+.new(..)). However, this definition covers not only classes directly implementing the interface (Person in our example), but also their subclasses (Supervisor in our example). In order to avoid executing the advice twice, we need to complexify a bit more the pointcut declaration. This is what is stated by line 13. Note however, that we don't get exactly the behavior provided in figure 2 (page 4). Indeed, with the reusable definition of the absence management aspect (introduced in this section), the initialization for instances of class Supervisor (subclass of Person) is done after all constructors (defined in classes Supervisor and Person) are executed. While in the non-reusable definition of the aspect (introduced in section 2.1.1 page 4), the advice is done right after the execution of the constructor of class Person.

Another problem with the reusable definition of the AbsenceManagement as-

pect provided on figure 5 is caused by the introduction of new methods such as toString() (lines 9 and 10). This extension performs well if the Person class does not implement a method with the same signature. However, if Person does implement a such method, than the extension is simply ignored without warning. Similarly two aspects introducing in a same class two methods with the same signature, the weaver does actually silently introduce only one methods without warning. Even if warnings were available, aspect integrators would have to change the aspects definitions and hence loose part of the benefice of reuse.

```
1: pointcut toStringExec(AbsenceRequestor requestor):
2:  execution(String AbsenceRequestor.toString()) &&
3:  target(requestor) &&
4:  !cflowbelow(execution(String AbsenceRequestor.toString()));

5: String around(AbsenceRequestor requestor): toStringExec(requestor){
6:  String initialAnswer = proceed(requestor);
7:    return initialAnswer + "\nAvailable Vacation Days = "
8:      + requestor.getVacationDaysCount();}

//Default method
9: public String AbsenceRequestor.toString(){return "";}
```

Fig. 6. Replacement of the toString() method definition with a pointcut and an advice in the AbsenceManagement aspect

To avoid such problems, one should replace every method with pointcuts and advices. Figure 6 provides such rewriting for the toString() method. The pointcut declaration captures the execution of method toString() by instances of classes implementing the AbsenceRequestor interface. The !cflowbelow(...) part of the declaration avoids performing the advice twice when there are super.toString() sends. However, the resulting semantics is a bit different from the one obtained with the non-reusable version of the aspect (Figure 2 page 4). As shown by figure 7, the string corresponding to available vacation days is appended at the end of supervisors descriptions. While in the non-reusable aspect definition (Figure 3 page 5) available vacation days string is inserted before the string providing the number of subordinates.

Yet another problem with AspectJ is that the code provided by figure 6 (page 8) need to insert a *default* implementation of the toString() method (line 9). This definition is useful for cases where the core application classes does not provide such a method. When such method is available, the default implementation is simply ignored. While the use of a default method implementation allows reusing the aspect in multiple applications providing or not the introduced method, it causes a non resolvable conflict when two aspects provide two default implementations of the same method. Indeed, the default method implementation is just a programming style and the weaver is not aware of

Evaluated Code

```
Supervisor chief = new Supervisor("Bart", "Simpson");
Person joe = new Person("Joe", "Dalton");
joe.setBoss(chief);
System.out.println("---println(joe)---\n" + joe);
System.out.println("---println(chief)---\n" + chief);
```

Console Display

```
---println(joe)----
Available Vacation Days = 30
---println(chief)---
Boss of 1 person(s)
Available Vacation Days = 30
```

Fig. 7. A code evaluated and its result in the context of the reusable absence management aspect

it. So, weaving fails, and application integrators has to change one aspect and remove the corresponding default method implementation.

2.1.3 Summary of AspectJ Limitations

To sum up, AspectJ has several limitations regarding a uniform description of reusable aspects.

- AspectJ does not encourage reuse. Building reusable aspects mainly relies on developers discipline.
- AspectJ introduces extra complexity for developers. They are offered two different sets of constructs for implementing cross-cutting code: inter-type declarations for static cross-cutting, and pointcuts and advices for dynamic cross-cutting.
- Reusable definitions of simple aspects is complex and unatural, since it requires having for each introduced method an inter-type declaration with the default implementation of the method, a pointcut declaration capturing a single execution of the method and an advice performing the desired processing.
- It is difficult if not impossible to always get the desired semantics when building reusable aspects.
- It is not possible to build fully reusable aspect. Application integrator may always face conflicts requiring modifying aspects code.

2.2 Problem Statement

Starting from AspectJ limitations, we list here issues that should be addressed by an AOP platform allowing to build reusable aspects with both static and

dynamic crosscutting. Such platform should be easy to learn and use, especially when it comes to maintain existing aspects. As proved by languages such as Self [24] and Smalltalk [12], uniformity and simplicity can go along with the language expressive power. We believe that this philosophy should be adopted in AOP platforms. Issues to be addressed are the following:

Reusable Aspects: An AOP platform should encourage building reusable aspects, by encouraging decoupling aspect's code from other applications parts.

Uniform Description of Crosscuttings: Having a small set of constructs uniformly used to express both static and dynamic crosscuttings would ease the learning and the understanding of aspects.

Uniform Conflict Management: Conflicts can occur between two static crosscuttings or two dynamic ones alike. Developers should be provided the same tools to handle both of them.

Crosscuttings Interactions: Dynamic crosscuttings should be able to alter the whole application code including static crosscuttings.

3 Foundations for Unified AOP

Our proposal to support unified AOP relies on *reflection* [23,18] and *mixin-based inheritance* [6]. Starting from plain Smalltalk, we introduce a minimal set of concepts and apply them uniformly to express both static and dynamic crosscutting. We believe that with this platform provides simplicity and uniformity, without scarifying expressiveness.

In this section, we first briefly remind reflection and mixin-based inheritance. We then provide a description of aspects in a platform supporting unified AOP. Last we describe the process of weaving aspects into application core.

3.1 Background: Reflection and Mixin-Based Inheritance

3.1.1 Reflection

Refection is the ability of a system to reason and to act upon itself. In the context of object-oriented languages, reflection gives access to languages semantics. A reflective OO language provides programmers with two programming levels: *base-level* and *meta-level*. The base-level includes all application objects (e.g. diary, person, supervisor, ...). The meta-level includes so-called *meta-objects* which are objects describing the reflective language's constructs (e.g. classes) and how programs are evaluated (e.g. message dispatch).

We use in the reminder of this paper the Meta-Object Protocol (MOP) of MetaclassTalk [3] [5,4], a reflective extension of Smalltalk. MetaclassTalk MOP allows controlling objects creation and memory allocation, instance variable reads and writes, message sends and receptions, and method lookup and evaluation.

3.1.2 Mixins

The concept of mixins has been introduced as an alternative to both single and multiple inheritance. It provides more code sharing than allowed with single inheritance, while avoiding issues arising with multiple inheritance and its automatic linearization. A mixin can be viewed as an abstract subclass parameterized with its superclass. This parameterization allow using a same mixin in different class hierarchies.

The mixin model we use in the reminder of this paper is inspired by the one introduced in CLOS [14]. A class can have many superclasses (mixins or plain classes). But, we go further than CLOS where mixin-based inheritance is just a programming style. We constrain the model to allow only multiple inheritance of mixins [3,4]. A subclass can inherit from an arbitrary number of mixins, but can have only one non-mixin superclass. Linearization chain of a subclass starts with mixins in the order provided in the subclass definition. The non-mixin superclass appears after mixins. So, methods are looked up first in mixins and then in the non-mixin superclass.

3.2 Structure of Unified Aspects

Our proposal relies on using mixins to build unified aspects. A unified aspect is a compound of: a Set of mixins, a pre-weaving script, and a post-weaving script. Mixins provide descriptions of crosscutting code. While, pre-weaving and post-weaving scripts are sequences of Smalltalk statements describing initialization operations to be performed before and after weaving crosscutting code into application core.

A crosscutting be it static or dynamic is implemented using a set of mixins. Mixins describing static crosscuttings are aimed to be inserted (on weave-time) into base-level class hierarchies. While, mixins describing dynamic corsscuttings are aimed to be inserted (on weave-time) into meta-level class hierarchies.

Besides aspects, developers have also to provide application core. That is a set of classes organized using composition and inheritance relationships. These

[3] http://csl.ensm-douai.fr/MetaclassTalk

classes define basic structure and behavior of application objects. They do not hold any instance variable or method related to any aspect.

3.3 Weaving Unified Aspects

As mentioned before, weaving relies on reflection, and mixin-based inheritance. For every class A in application core, the weaver builds a meta-object class AMeta. Each instance of A is controlled by an instance of AMeta. So, instances of AMeta provide the semantics of behavior (message sends and receptions) and structure (instance variables reads and writes) of instances of A. An application is obtained once mixins provided by various aspects are linked through inheritance to application core classes and corresponding meta-object classes. It worth noting that the meta-object class hierarchy is parallel to the class one. So, given B a subclass of A, BMeta the class of meta-objects of instances of B is built by the weaver as a subclass of AMeta

Once the weaver creates meta-object classes, it repeats the four following steps for every aspect.

(1) Provide a map stating which aspect's mixins to link to which core application classes. This step is necessary because the aspects' definitions does not refer to application core classes.
(2) Evaluate the current aspect's pre-weaving script.
(3) Link the current aspect's mixins to application core classes.
(4) Evaluate the current aspect's post-weaving script.

Because of the use of mixin-based inheritance, the cross-cutting code remains isolated from application core (though it is linked). Relying on Smalltalk dynamicity our solution supports not only dynamic weaving, but also dynamic unweaving. To complete this support, the structure of unified aspects includes also pre-weaving and post-weaving scripts. The pre-weaving script is evaluated before unlinking classes and mixins. The post-weaving script is evaluated after unlinking classes and mixins.

Joint point are expressed using the map and the MOP. That is, joint point cover class definitions, message sends and receptions, and instance variables reads and writes.

3.4 Aspects Interactions and Conflicts

Aspects does not only alter the structure and behavior of the application core, they also may affects each other execution. Consider an aspect A1 that makes

a core application class C inherit from some mixin M1. Suppose also that we weave into this application another aspect A2 that adds some other mixin M2 to the superclass list of class CMetaObject, the class of meta-objects of instances of class C. Therefore, the semantics of code introduced by the A1 aspect using the M1 mixin is altered by the A2 aspect which introduces the M2 mixin.

Conflicts may arise when two aspects link mixins with homonymous methods or instance variables to a same class. Mixin-based inheritance provides us with a first solution to this open issue. Indeed, developers can order mixins linked to each class. Methods introduced in mixins appearing first in a class definition override homonymous methods defined in other mixins. This solution currently implemented in our prototype is rather coarse grain and does not address the case of homonymous instance variables. Van Limberghen and Mens [17] describe a solution that tackles this problem.

Yet another cause of conflicts is having weaving scripts of different aspects perform "contradictory" actions (e.g. setting some class variable to different values). We address this issue by allowing developers choose aspects precedence as in AspectJ. That is, developers choose the order of weaving. However, the resolution of this kind of conflicts deserves further investigations we defer to future work.

4 Examples

Figure 8 shows part of the distributed diary system built using our solution after weaving [4]. Application core includes various classes: Person, Supervisor, and Diary. However, these classes does not define any instance variable or method related to aspects such as absence management, log or authentication.

4.1 Example of static cross-cutting: the "absence management" aspect

The absence management aspect introduces two new roles: *absenceRequester* and *absenceManager*. An *absenceRequester* is supposed to store an available vacation days. It is also supposed to understand the requestVacation: message which argument is the vacation duration in days. As a response to this message an *absenceRequester* checks if the duration is less than or equal to available vacation days and then requests the confirmation of an *absenceManager* (message acceptVacation: *duration* for: *anAbsenceRequester*). When the

[4] The full code is available on-line at http://csl.ensm-douai.fr/MetaclassTalk

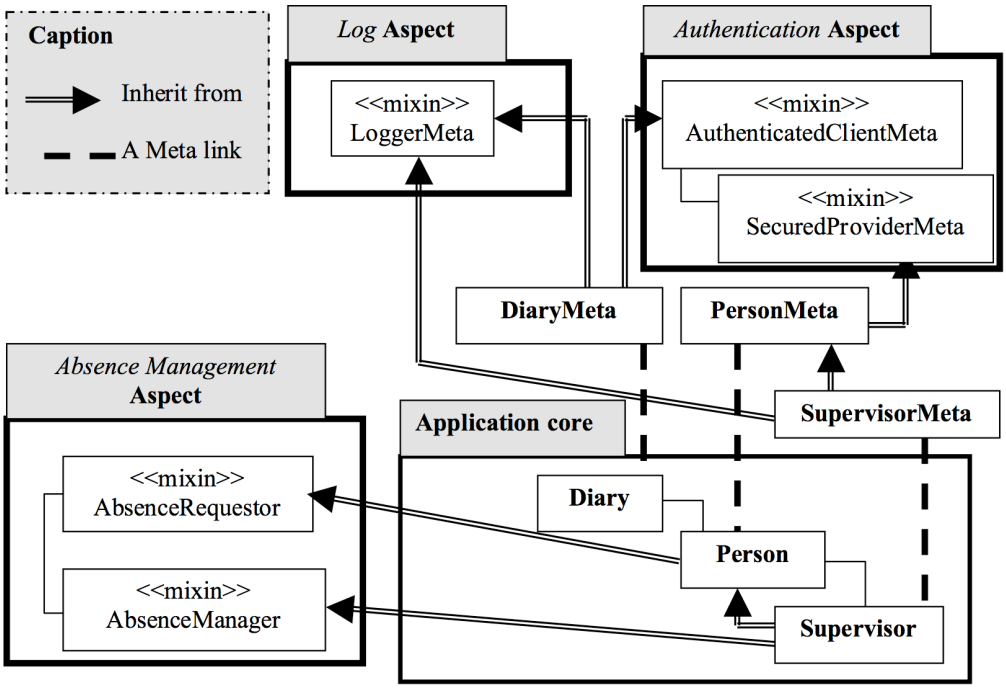

Fig. 8. A Subset of the distributed diary system built with our solution

absenceManager accepts the request, the *absenceRequester* decrements available vacation days counter and inserts an event describing the absence into a diary.

```
Aspect subclass: #AbsenceManagementAspect
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Unified AOP-Diary Example-Aspects'.

AbsenceManagementAspect >> initialize
super initialize.
self addAllMixins: {AbsenceRequester. AbsenceManager}
```

Fig. 9. Definition of the "absence management" aspect

Each one of the above described roles is implemented using a mixin. So, mixins AbsenceRequester and AbsenceManager define appropriate instance variables and methods for handling vacation requests. So, the description of the "absence management" aspect includes only these two mixins. Figure 9 shows that this description consist in defining a class which instances have two mixins: AbsenceRequester and AbsenceManager. Pre-weaving and post-weaving scripts are implemented as methods in the aspect's class. Because here we don't need any special processing, we don't override the existing empty implementations provided by class Aspect. It worth noting that there is no direct reference to application core. Therefore, this aspect can be reused in other applications.

```
| absenceAspect |
absenceAspect := AbsenceManagementAspect new.
absenceAspect map: AbsenceRequester to: Person.
absenceAspect map: AbsenceManager to: Supervisor.
absenceAspect weave.
```

Fig. 10. Weaving the "absence management" aspect

To weave the "absence management" into our application, we need first to map each of its mixins to application core classes. In our implementation, an aspect is but an object that can be parameterized with a map describing which mixins to link to which application core classes (see figure 10). Then, by sending the weave message to the aspect, the weaving is finished. First, pre-weaving script is performed. Then, the AbsenceRequester mixin is added to the superclasses list of class Person. Next, the AbsenceManager mixin is added to the superclasses list of class Supervisor. Last, the post-weaving script is performed.

4.2 Example of dynamic cross-cutting: the authentication aspect

As mentionned above, an aspect definition can include mixins that can go either to the base-level or to the meta-level. The "absence management" presented in the previous subsection is an aspect which definition relies on mixins that are to be linked to base-level classes. Here we present the "authentication" aspect which implementation relies on changing the semantics of message dispatch. So, it defines mixins that are to be linked to meta-object classes.

The authentication aspect introduces two roles: *authenticatedClient* and *securedProvider*. An *authenticatedClient* holds a login and a password that grant him access to services of some *securedProvider*. So, when an *authenticatedClient* needs to send some message to a *securedProvider*, *authorizedClient* first sends the pair login and a password to the *serviceProvider*. A *securedProvider* accepts processing only messages sent by client with valid login and password.

The authentication aspect implements these two roles using two meta-level mixins AuthenticatedClientMeta and SecuredProviderMeta. Figure 11 provides the actual code of these two mixins. We can see that mixin AuthenticatedClientMeta extends the semantics of message sending. It overrides method send:from:to:arguments: introduced in the MetaclassTalk MOP to perform first authentication before actually sending messages. Mixin SecuredProviderMeta extends the semantics of message reception. It overrides method receive:from:to:arguments: introduced in the MetaclassTalk MOP to actually perform received message from only authenticated clients.

The obtained authentication aspect is reusable since it does not refer to any

```
Mixin named: #AuthenticatedClientMeta
  instanceVariables: 'login password '
  category: 'Unified AOP-Diary Example-Aspects'.

AuthenticatedClientMetat ≫ login: loginString password: passwordString
  login := loginString.
  password := passwordString

AuthenticatedClient ≫ send: selector from: sender to: receiver arguments: args
  receiver metaObject authentify: sender login: login password: password.
  ↑super send: selector from: sender to: receiver arguments: args
```

```
Mixin named: #SecuredProviderMeta
  instanceVariables: 'passwordDict authenticatedClients'
  category: 'Unified AOP-Diary Example-Aspects'.

SecuredProviderMeta ≫ initialize
  super initialize.
  passwordDict := Dictionary new.
  authenticatedClients := Set new

SecuredProviderMeta ≫ authentify: client login: login password: tentativePass-
word
  | actualPassword |
  actualPassword := passwordDict at: login ifAbsent: [↑self].
  actualPassword = tentativePassword ifFalse: [↑self].
  authenticatedClients add: client

SecuredProviderMeta ≫ acceptMsg: selector from: sender to: receiver
  ↑sender == receiver or: [
    (authenticatedClients includes: sender)]

SecuredProviderMeta ≫ receive: selector from: sender to: receiver arguments:
args
  (self acceptMsg: selector from: sender to: receiver) ifFalse: [
    ↑self error: 'Access restricted'].
  ↑super receive: selector from: sender to: receiver arguments: args
```

Fig. 11. Mixins for the "authentication" aspect

core application class. Now, let see how to weave it. In our diary application accesses to a given diary have to be restricted to only some persons (*e.g.* its owner). Hence, instances of Person should be authenticated before message sends to instances of Diary. And, instances of Diary should check authoriza-

tions on message receptions. To get this behavior, we map mixin AuthenticatedClientMeta to class PersonMeta and mixin SecuredProviderMeta to class DiaryMeta. After weaving we get the mixins and meta-object classes linked.

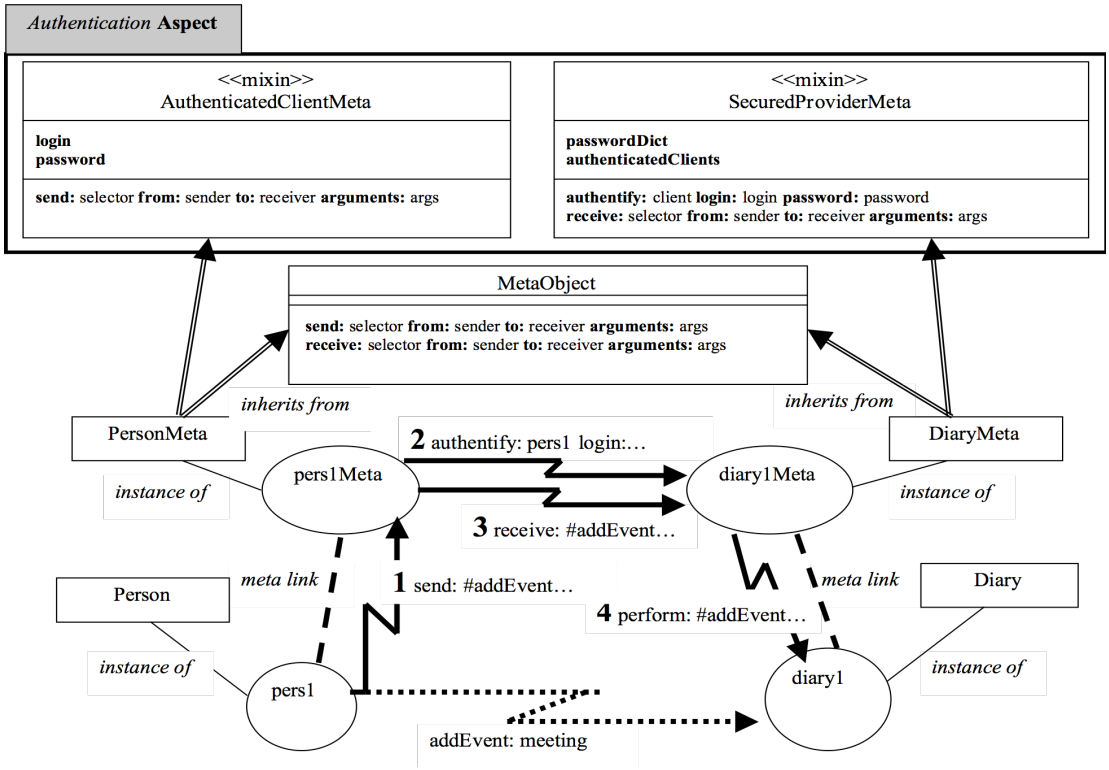

Fig. 12. Example of an authentication aspect in action

Figure 12 provides an example showing how actually authentication is performed. Every instance pers1 of class Person is linked to a meta-object pers1Meta instance of PersonMeta. And, every instance diary1 of class Diary is linked to a meta-object diary1Meta instance of DiaryMeta. When pers1 sends some message, say addEvent:, to diary1, the message sending is intercepted by the pers1Meta meta-object. This interception translates into a message send:. . . [5] implicitly dispatched (*i.e.* by the reflective infrastructure) to pers1Meta (step 1). Arguments of this message are informations about the addEvent: message (*e.g.* sender, receiver, selector,. . .). The meta-object pers1Meta attempts to do authentication by sending message authentify: *pers1* login: *loginOfPers1 password: passwordOfPers1* to diary1Meta, the meta-object of the receiver (step 2). Then, pers1Meta delivers the addEvent: message to perform to diary1Meta (step 3). The diary1Meta meta-object does check if the sender (*i.e.* pers1) has been granted access. If pers1 is not allowed to add an event to diary1, an exception is thrown. Otherwiser, the addEvent: message is performed by diary1 (step 4).

[5] The actual selector of this method is send:from:to:arguments:superFlag:orginClass:.

5 Related Work

5.1 AspectJ

AspectJ [16] mainly focuses on dynamic cross-cutting aspects. Nevertheless, using *inter-types declarations*, it does support to some extent the definition of static cross-cutting. Indeed, AspectJ allows the *introductions* of methods and instance variables into existing classes. However, no conflict support is provided when two aspects requires the introduction of homonymous instance variables or methods in the same classes.

Aspects reuse is also an issue with AspectJ. As demonstrated in section 2.1.2 (page 6) AspectJ does not encourage reuse. Building reusable aspects mainly relies on developers discipline and results into complex code even for simple aspects.

Last, AspectJ is complex. Dynamic and static cross-cutings are defined using different language constructs.

5.2 Hyper/J

Hyper/J stemmed from work on *Multi-Dimensional Separation of Concerns* (MDSOC) [20]. It allows developers choose arbitrary dimensions to carve up and modularize applications. Every dimension is implemented as a set of classes. Composition rules allow developers express how to merge classes defined in different dimensions.

Hyper/J shares with our work uniformly define aspects. However, our solution supports incremental dynamic weaving and unweaving. In Hyper/J weaving is a static operation that relies on program transformation. No information about original dimensions are available in the resulting application.

5.3 AspectS

AspectS is a dynamic infrastructure supporting AOP in Smalltalk [13]. It allows expressing uniformly both static and dynamic cross-cuttings. However, because AspectS' implementation relies on method wrappers, only cross-cuttings related to message dispatch can be expressed. Accesses to existing instance variables can not be captured.

Besides, new instance variables can not be introduced in application classes. Nevertheless, aspects can hold dictionaries that associate state to application objects. This solution has two limitations. Access to dictionaries is slow compared to direct access to instance variables. And, the code of aspects holding such state is rather complex.

5.4 ClassBoxes

ClassBoxes are modules allowing the definition of scopped class extensions [1]. They can be used to implement static cross-cutting [2].

This approach supports dynamic weaving/unweaving of aspects. Besides, visibility control helps resolving some potential conflicts. However, the use of ClassBoxes to implement dynamic cross-cutting is still to be studied.

5.5 Traits

Traits can be viewed as mixins without structure (no instance variables), but with a powerful composition mechanism [22]. The trait model indeed provides different operators to compose traits at methods granularity level. Developers can for example hide or rename some trait's method.

Traits can be used instead of mixins to implement unified aspects. Their composition operators can be helpful for solving conflicts among aspects. Moreover, the absence of instance variables definitions in traits reduces conflicts. However, it also restricts their expressive power.

6 Conclusion and Future Work

We described in this article foundations for a platform allowing a unified description of dynamic and static cross-cutting. To this end, we make use of mixins as aspects building blocks. In this context, weaving relies on mixin-based inheritance and reflection. Static cross-cutting is implemented using mixins that are inserted into base-level class hierarchies by the weaver. While dynamic cross-cutting is implemented using mixins are inserted into meta-level class hierarchies by the weaver. This solution helps building reusable aspects since mixins can be easily implemented without any connexion to application classes.

One possible perspective of this work is to improve conflict resolution support.

Our current solution mainly relies on explicit mixins linearization. Application developers can only reorder mixins linked to a given class. Granularity of aspect conflict resolution can still be finer to allow even more conflicts resolutions. Traits compositional operators introduced by Schärli et al. [22] is a possible solution to deal with conflicts at the method level. Another alternative is to use the mixin model introduced Van Limberghen et al. [17] which provides operators to deal with both methods and instance variables conflicts.

Reuse is yet another intersting direction to follow. In this paper, we mentioned mixin reuse to build different aspects. We also, presented aspect reuse to build different applications. A third possibility yet to explore is aspect reuse to build new aspects out of existing ones.

References

[1] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. In *Research Track of the ESUG 2004 Smalltalk Conference*, Köthen (Anhalt), Germany, September 2004. Selected for publication in the special issue on Smalltalk Language of the Elsevier international journal "Computer Languages, Systems and Structures" 2005.

[2] Alexandre Bergel and Stphane Ducasse. Dynamically applying static aspects with classboxes. *Journes Francophones de la Programmation Par Aspects, JFDLPA'04*, 2004.

[3] N. Bouraqadi. Efficient support for mixin-based inheritance using metaclasses. In *Workshop on Reflectively Extensible Programming Languages and Systems at The International Conference on Generative Programming and Component Engineering (GPCE'03)*, Erfurt, Germany, September 2003.

[4] N. Bouraqadi. Metaclass composition using mixin-based inheritance. In *Research Track of the ESUG 2003 Smalltalk Conference*, Bled, Slovenia, August 2003. European Smalltalk Users Group (ESUG).

[5] N. Bouraqadi and T. Ledoux. *Aspect-Oriented Software Development*, chapter 12 – Supporting AOP using Reflection. Addison-Wesley, 2005.

[6] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311. ACM Press, 1990.

[7] Richard Cardone and Calvin Lin. Using mixin technology to improve modularity. In Filman et al. [11], pages 219–241.

[8] Olivier Caron, Bernard Carre, Alexis Muller, and Gilles Vanwormhoudt. Mise en oeuvre d'aspects fonctionnels rutilisables par adaptation. *Journe Francophone sur le Dveloppement de Logiciels Par Aspects, JFDLPA'04*, September 2004.

[9] Brian de Alwis and Georg Kiczales. Apostle: A simple incremental weaver for a dynamic aspect language. Technical Report TR-2003-16, University of British Columbia, Vancouver, Canada, March 2003.

[10] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[11] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[12] Adele Goldberg and David Robson. *Smalltalk 80*, volume 1 – The Language and its implementation. Addison-Wesley, 1983.

[13] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[14] Sonya E. Keene. Object-oriented programming in common lisp: A programmer's guide to clos. *Addison-Wesley,Reading, Massachsetts, USA*, 1989.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Loingtier Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[16] Ramanivas Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.

[17] Marc Van Limberghen and Tom Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object Oriented Systems*, 3:1–30, 1996.

[18] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, pages 147–155, Orlando, Florida, 1987. ACM.

[19] Sean McDirmid and Wilson C. Hsieh. Aspect-oriented programming with jiazzi. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 70–79, New York, NY, USA, 2003. ACM Press.

[20] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):44–50, October 2001.

[21] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. Jac: a flexible solution for aspect oriented programming in java. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of Reflection 2001*, number 2192 in LNCS, pages 1–24, Kyoto, Japan, September 2001. Springer Verlag.

[22] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*, LNCS. Springer Verlag, July 2003.

[23] Brian C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, POPL'84*, pages 23–35, Salt Lake City, Utah, USA, January 1984.

[24] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.