JAC Milestone 2001

Lionel Seinturier*, Renaud Pawlak**, Laurence Duchien***, Gérard Florin**

* Université Paris 6, Lab. LIP6, 4 place Jussieu, 75252 Paris cedex 05, France
** CEDRIC-CNAM, 292 rue Saint-Martin, 75141 Paris cedex 03, France
*** USTL-LIFL, Bâtiment M3, 59655 Villeneuve d'Ascq cedex, France

Lionel.Seinturier@lip6.fr, {pawlak, florin}@cnam.fr, Laurence.Duchien@lifl.fr

Abstract

JAC (Java Aspect Components) is a framework for aspect-oriented programming in Java. It is developed as a joint research project between the CEDRIC-CNAM and LIP6 computer science laboratories. This report gives a snapshot of the project as of September 2001.

Unlike languages such as AspectJ which are mostly class-based, JAC is object-based and does not require any language extensions to Java. It uses the Javassist class load-time MOP. An aspect program in JAC is a set of aspect objects that can be dynamically deployed and undeployed on top of running application objects. Aspect objects may define three kinds of aspect methods: wrapping methods (that wrap application methods and provide the ability to run code before and after the wrapped methods), role methods (that add new functionalities to application objects), and exception handlers. The aspects composition issue is handled through a well-defined wrapping controller that specifies for each wrapped object at wrap-time, runtime or both, the execution order of aspect objects.

1 Introduction

Separation of concerns in software engineering has always been a very natural means to handle complexity of software developments [Par72]. However, modularizing concerns can be a very tricky task for the programmer and rise some issues such as performance, crosscutting, or redesigning when the software is used in a context that is quite different from the overseen one. By handling crosscutting within the language or system, the recent approach of Aspect-Oriented Programming (AOP) [KLM+97] seems to be a very promising way for helping developers to handle separation of concerns and to overcome the drawbacks of traditional design approaches.

However, if AOP introduces a new programming paradigm that complements existing ones, it is clear that it brings a new bunch of difficult problems. The composition of an aspect to an application (the aspect is said to be woven) is one of them. Several approaches exist: AspectJ [KHH+01] which is a general language for AOP, the composition filter object model [BA01] where aspects are defined as filters applied upon application objects, aspectual components [LLM99] that define patterns of interaction with roles and connectors to map these roles to application objects, subject-oriented programming [HO93][OKH+95] which decomposes an application into subjects and provides composition rules to recompose them, domain specific languages to define crosscutting concerns based on patterns of events [DMS01]. We review some of them in the related works section of this paper.

Nevertheless, when several aspects have to be composed to an application, a given aspect not only crosscuts the application, but may also crosscut others aspects. Indeed, aspects may not be orthogonal to each others. We call this issue the *inter-aspects composition* aspect. Some solutions exist, e.g. precedence rules in AspectJ [KHH+01] or composed connectors in aspectual components [LLM99] (see section 6 for a discussion of these features), but as far as we know, this is still an open issue for the AOP community. Most of the time aspect programmers still have to invent some *ad hoc* means to handle it. This problem deeply affects the potential power of AOP by making aspects less re-usable that they should be and dramatically limits the simplicity of using a set of aspects.

In this paper, we present a framework called JAC (for Java Aspect Components) [JAC] that is a proposal to cleanly deal with *inter-aspects composition* within an aspect-oriented application, making by this way aspects more reusable. JAC is the continuation of A-TOS [PDF+00], a previous work developed in Tcl. The paper is structured as follows. First, we point out that one of the main problem in AOP is to be able to easily compose several aspects coming from different sources, during the development process, and while the application is running. In section 3, we present JAC and the way we deal with the *inter-aspects composition* issue at weave-time and at runtime. Section 4 presents a case study. Section 5 discusses JAC performances. We compare JAC with other related works in section 6. Finally, section 7 concludes this paper and presents some future works.

2 Important issues in AOP

When programming aspects with an aspect-oriented language, framework, or system (AOS) the main problems programmers have to face is to handle the consistent composition of aspects. In a general development process, aspects can be programmed by different programmers and can conflict if nothing is done to avoid it. We call this problem the *inter-aspects composition* issue [PDF99]. This issue can occur at weave-time but also at runtime and can be split in several sub-problems of different natures. The following list shortly depicts some of the most currently encountered. Most of them remain open issues that are discussed in the AOP community. We don't assume an AOS to handle them automatically. We rather think that a neatly designed AOS should provide solutions (e.g. API, language constructs, ...) to support programmers in addressing them.

2.1 Weave-time issues

The weave-time issues occur when the weaver weaves the aspects (or a particular aspect) into the base program. Depending on the AOS, weaving can be at compile-time or at runtime (when an aspect is dynamically added or removed from the application). We name the weave-time issues *WIn* to be able to refer them later.

2.1.1 Checking for aspect compatibility with the application (WI1)

Assuming that we know that our application should never contain a given aspect, the underlying AOS should be able to check its type (assuming the aspect programmer provides a formal or semi-formal type system) and refuse to weave it. For instance, if the base program already implements an authentication policy (by choice or because you add some aspect to an existing application), then you should add a composition constraint that prevents an authentication aspect to be woven.

2.1.2 Checking for inter-aspect compatibility (WI2)

If we know that two aspects are incompatible (e.g. some redundancy and fault tolerance algorithms) the program can refuse to weave one aspect if the other is already woven.

2.1.3 Checking for inter-aspect dependence (WI3)

If an aspect is woven and needs another aspect (e.g. a binding aspect may need a naming aspect), the AOS should be able to automatically weave the needed one (or report an error).

2.1.4 Checking for aspect redundancy (WI4)

If we know that two aspects implement the same concern in two different ways (e.g. two different authentication algorithms or two different persistence implementations), then the program can refuse to weave one of both aspects or unweave the previous one to replace it by the new one.

2.1.5 Ordering the aspects at weave-time (WI5)

Regarding a join point (a location in the base program where a set of aspects can intercess their behaviors), some aspects must always be called before others (e.g. authentication) and some must always be called after others (e.g. persistence). The AOS should place the different aspects so that they are correctly ordered and so that the aspect programmer does not care anymore about the others.

For the five above described issues, the AOS must be able to define some checking and/or ordering rules. This code can be seen as an aspect that rules how aspects behave regarding each others at weave-time. In several works, this aspect has been called a composition aspect. At weave-time, the weaver will refer the composition aspect to decide how to weave an aspect to an application program.

2.2 Runtime issues

By runtime issues, we mean the issues that may occur when the aspect is already woven, and that can arise when the execution of a join-point that is intercessed by the aspect occurs. In some cases, the set of advices that is defined in the aspects for a given join-point may change regarding the context. Since it depends on the execution context, the weave-time composition aspect can hardly deal with the runtime issues. Using a few examples, we will show that the AOS can take advantage of using a runtime composition aspect to deal with these issues. We name the runtime issues *RIn* to be able to refer them later.

2.2.1 Checking for intra-aspect consistency (RI1)

Some aspects need to perform context-dependent tests to remain semantically consistent. Let take the example of a counting aspect, that counts the number of calls to a given method. The following pseudo code is inspired from the AspectJ [KHH+01] syntax. It increments the *counter* variable before each execution of method *A.m1*.

```
aspect CountingAspect {
    private int counter;
    joinpoint jp1 = ( class A, method m1 );
    before jp1 advice1 {
        counter++;
    }
}
```

Now imagine that method *m2* calls *m1* 10 times. The aspect programmer (that is aware of the base program code) can optimize the application by adding another *before* advice for method *m2*.

```
aspect WrongOptimizedCountingAspect {
    private int counter;
    joinpoint jp1 = ( class A, method m1 );
    joinpoint jp2 = ( class A, method m2 );
    before ( jp1 ) advice1 {
        counter++;
    }
```

```
    before ( jp2 ) advice2 {
        counter += 10;
    }
}
```

However, this aspect is wrong because, if the user of class A calls method $m2$, then the counter will be incremented by 20 (first by 10 by *advice2*, and next 10 times by 1 by *advice1*) instead of 10. Thus, the aspect code needs to perform a contextual test to skip the first *before* advice when the second has already been applied (this is informally expressed in the following pseudo code by the `advice2.alreadyApplied()` method call that returns true if *advice2* has already been applied).

```
aspect OptimizedCountingAspect {
    private int counter;
    joinpoint jp1 = ( class A, method m1 );
    joinpoint jp2 = ( class A, method m2 );
    before ( jp1 ) advice1 {
        if ( advice2.alreadyApplied() ) { skip }
        counter++;
    }
    before ( jp2 ) advice2 {
        counter += 10;
    }
}
```

These kinds of aspects has been pointed out in [BMV00]. Brichau and al. call them *"jumping aspect"* since *the join points seems to be jumping around the code depending on the context in which a component is used.* This aspect code supposes that the aspect system is able to memorize the aspect advices that have been already applied (note that AspectJ [KHH+01] can support this feature with *CFlows*). The *skip* operation means that the current advice is skipped. We will see later on that this solution is not totally satisfying.

2.2.2 Skipping an aspect (RI2)

Depending on the state of the application (and of the context), some aspects may be skipped for optimization matters. This can be a generic optimization, e.g. a persistence aspect may be called only one time out of ten so that the objects states are saved less often; or an authentication aspect may be skipped if we know that the client has already been authenticated. It can also be a more application semantics dependent optimization. For example, the GUI aspect could be skipped if we know that the action performed on the base object will not affect its graphical representation.

The following pseudo code skips a persistence aspect advice when the load of the system goes over a given threshold.

```
    after ( jp ) advice1 {
        if ( System.getLoad() > THRESHOLD ) { skip }
        // serialize and write...
    }
```

2.2.3 Choosing an aspect (RI3)

We previously talked about checking for aspect redundancy. On the other hand the application programmer could deliberately weave several aspects that implement the same concern so that, depending on the program context, the AOS could use the aspect that seems to be the most efficient. For instance, we can choose a different image compression algorithm whether the client is locally or remotely connected or whether s/he asked for a real-time QoS.

Similarly to previous examples a simple means to deal with this issue is to apply some contextual tests.

```
aspect CompressionAspect {
    before ( jp ) advice1 {
        if ( Network.getLoad() > THRESHOLD ) {
            // low quality compress...
        } else {
            // high quality compress...
        }
    }
}
```

2.2.4 Ordering the aspects at runtime (RI4)

In some cases (that most of the time depend on the application semantics) the aspect ordering is not known at weave-time. In these cases, the AOS should be able to re-order the aspects for a given join point and a given context within the runtime system. Let us take the example of a logging aspect that can switch from a verbose mode to a very verbose one. In the verbose mode, the log traces only the successfully authenticated access to an object, and in the very verbose mode, the log traces also the non authenticated access tries. A simple way to implement the verbose mode is to apply the authentication aspect *before* the logging aspect, whilst the very verbose mode implies that the authentication aspect is applied *after*. Moreover, the application programmer may want to be in verbose mode for a set of trusted client hosts and in very verbose mode for all the other hosts. As a consequence, the aspect ordering depends on some runtime contextual information and the AOS should provide features to help the programmer to deal with this in a clean way.

This issue can be overcome by using two different advices with contextual tests so that it does not seem to be a problem anymore (here, we assume that the weave-time composition aspect order the advices so that *advice1* is before the authentication advices, and *advice2* is after).

```
aspect LoggingAspect {
    private boolean veryVerbose;
    joinpoint jp = ( class A, getAllMethods );
    before ( jp ) advice1 {
        if ( ! veryVerbose ) { skip }
        write ( logFile );
    }
    after ( jp ) advice2 {
        if ( advice1.allreadyApplied() ) { skip }
        write ( logFile );
    }
}
```

2.2.5 Inter-aspect dependence at runtime (RI5)

Assuming that we can skip an aspect at runtime, the WI3 issue is also applicable at runtime. For instance, if we skip a tracing aspect, all the aspects that depend on it must also be skipped.

2.3 Discussion

As one can see in figure 1, the aspect composition issue is a critical point in Aspect-Oriented Programming. In most of the existing languages or systems that more or less support aspect-orientation, handling these issues is mostly part of the aspect programmer task (like shown in the previous examples). This is a very important limitation to AOP since the composition issue is a *crosscutting* concern in the sense that some code should be added in the whole set of aspects to deal with it, especially when the composition problem can not be entirely solved at weave-time but must also be handled at runtime.

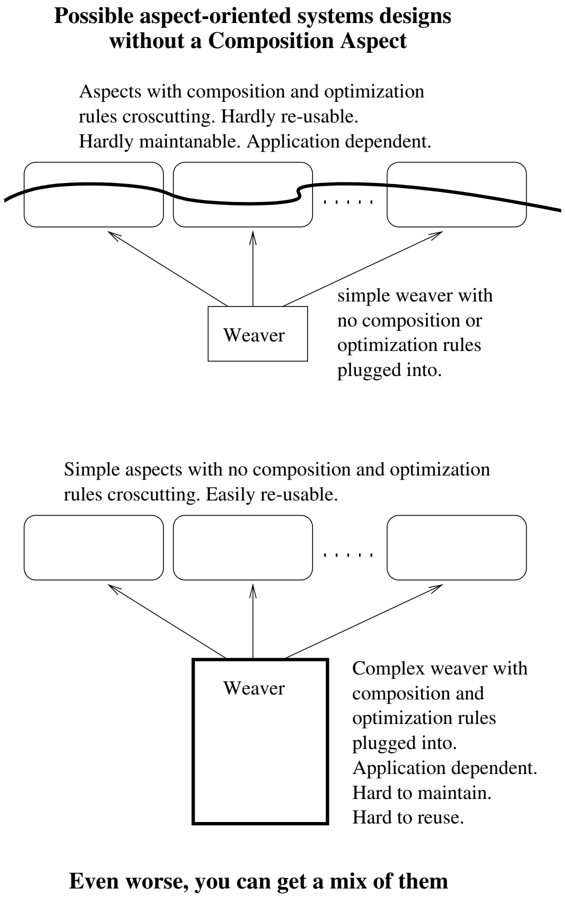

Figure 1: With or without a Composition Aspect ?

To illustrate this, let us take again the counting aspect depicted in section 2.2.1. The applied optimization (incrementing the counter by 10 when *m2* is called) skips the default increment. However, for other base programs, such optimization can be applied for several other kinds of join-points. Thus the skipping test is difficult to generalize and the aspect can hardly be reused as this. Moreover, another aspect could use the same kind of contextual test (for instance, a security aspect that also needs to count the calls to *m1*). The same problem occurs for the other examples when the aspects check the system or network loads to know if they have to be applied. Regarding this, it becomes clear that the contextual tests crosscut the aspects and make them hardly maintainable and reusable. In addition, since the same contextual tests may appear in a set of aspects for the same join-point, they can finally lead to performance issues.

For us, the only way to solve the runtime issues is to externalize the contextual tests into a well-modularized *inter-aspect composition* aspect. If we can do this, the AOS can apply a global contextual test for each join point and handles all the contextual tests at once. As shown in figure 1, the aspects remain free from runtime issues pollution and are more efficient and easier to maintain in comparison with other approaches without contextual part and aspect composition. Regarding our examples, it appears that we can classify the runtime issues into two categories: the issues that involve only one aspect (RI1, RI2, and RI3), and the ones that involve several aspects (RI4, RI5).

In the next section, we present our framework called JAC (for Java Aspect Components). With JAC, programmers develop applications in an aspect-oriented fashion and have support for the composition issue so that they are able to solve most of the previously described issues. Indeed, JAC programmers can cleanly describe how the composition of the aspects will be handled by the application within a well-bounded part of the program called a wrapping controller.

3 The JAC framework

This section presents our framework. Section 3.1 gives an overview. The four software entities of an JAC aspect-oriented program are introduced. Each of them is described with more details in the remaining subsections. Section 3.6 presents some additional elements from the JAC API.

3.1 Overview

JAC is a Java framework that provides support for dynamic aspect-oriented applications. By dynamic, we mean that an aspect can be woven and unwoven at runtime. An aspect-oriented program in JAC is entirely written in regular Java and is composed of four main parts.

3.1.1 Base program

A JAC base program is composed of a set of Java objects that implement the core functionalities of the application. These objects run on a regular JVM.

3.1.2 Aspect programs

A concern that refines or modifies the base program behavior is implemented in an aspect program that can be woven to the base program. An aspect program is implemented by a set of aspect objects (also called dynamic wrappers) that can hold three kinds of aspect methods *wrapping methods*, *role methods*, and/or *exception handlers*. Contrary to AspectJ [KHH+01] that mixes join-points definitions and advice definitions within the same aspect code, we separate the core functionalities of the aspects within independent aspect objects that can be compared to pure advice sets. The join points are defined by the weaver and the link between a join-point and a pure advice is done at weave-time. This choice makes the advices more reusable and, for instance, several aspects can reuse the same advice (for instance, a counting advice can be reused for a security or for a debugging aspect). This design choice is similar to the one made by aspectual components [LLM99]. Compared to this approach, a JAC aspect program corresponds to an aspectual component, and a JAC aspect object corresponds to a participant.

- **Wrapping methods**: A wrapping method can wrap any method of any base object and seamlessly executes some code *before* and *after* this method (they are thus equivalent to the *before*, *after*, and *around* advices of AspectJ and to the *replace* keyword of aspectual components). A base method can be wrapped by as much wrapping methods as needed and a wrapping method can be added and removed at runtime (contrary to regular wrappers, JAC wrappers do not change the base object reference when wrapping it and implement the wrapping by using an internal and dedicated MOP).

- **Role methods**: A role method can be attached to one or many base objects at runtime and extend the base object interface (similarly to the *introduce* statement in AspectJ and to the participant methods of aspectual components). A role method can be invoked by calling the *invokeRoleMethod* on a base object.

- **Exception handlers**: An exception handler is a method that is notified when an exception is raised within (or from) the base object method it is attached to. For instance, this is very useful when the invoked object is wrapped by a wrapping method that raises an exception that is not defined by the base object.

3.1.3 Weaver

The weaver is responsible for deploying the aspect objects on the appropriate base objects so that a set of functionalities crosscut the base program and implement a new concern. The weaving code implicitly defines pointcuts and links the advices (wrapping methods, role methods and exception handlers) to them. Since the composition rules are externalized within the composition aspect, the weaver does not take this issue into account. The weaving process is configured by a property file that says where and when aspect objects are to be deployed. This file is described in more details in section 3.4.

3.1.4 Composition aspect

This part defines rules about how the different aspects of the program are composed at weave-time (to solve WI1, ..., WI5) or at runtime (to solve RI1, RI2, RI3). The composition aspect provides an *WrappingController* interface that is a MOP dedicated to aspect composition and that is implicitly upcalled by the system each time some composition issue can occur. For performance issue, the composition MOP is not activated by default (since it is sometimes not needed). The weaver is responsible for activating the composition MOP for a given base object.

Figure 2 sums up the architecture of JAC programs. Table 1 compares the JAC programming model with the AspectJ [KHH$^+$01] and the aspectual components [ML98] ones. Contrary to these two examples, the JAC model externalizes the Runtime composition rules so that the aspects are more reusable. Only AspectJ integrates its pointcut definition within the aspect entities. Thus, reusing aspects implies the definition of abstract aspect classes.

3.2 Base objects

Base objects are regular Java objects whose classes are automatically adapted to be able to dynamically add a wrapper, attach a role object, or an exception handler. This translation is done at class load-time with the Javassist [Chi00] tool. Since Javassist works on the bytecode and during load-time, the source code is not needed and the class files of the program are never changed on disk. Programmers can still use them to build regular Java applications.

Figure 3 zooms on a JAC object that has been translated by the JAC class loader and that initially offers two methods *m1* and *m2*. The Javassist meta-classes we developped add the following elements:

- a wrapping chain (a vector of references to wrapping methods) for each initial method,
- a set of references to exception handling methods (methods called if the kind of exception they catch is raised) for each initial method,
- a field that contains a reference to the wrapping controller (upcalled to handle the composition, as it will be explained later),
- a vector that contains references towards the role objects attached to the JAC object.

These data are manipulated by the weaver that uses the JAC Object Manipulation Interface that is also added at load-time and that furnishes methods to manipulate wrappers (*wrap*, *unwrap*, *nextWrapper*), role objects (*addRoleObject*, *rmvRoleObject*, *invokeRoleMethod*), exception handlers (*addExceptionHandler*, *rmvExceptionHandler*).

In the example of figure 3, the configuration applied by the weaver (at runtime) is the following: *m1* is wrapped by {*ao1, wm1*} and {*ao1, wm2*}, it is attached to one exception handling method {*ao1, h1*} while *m2* is wrapped by {*ao2, wm3*} and is attached to {*ao1, h1*} and {*ao2, h2*}.

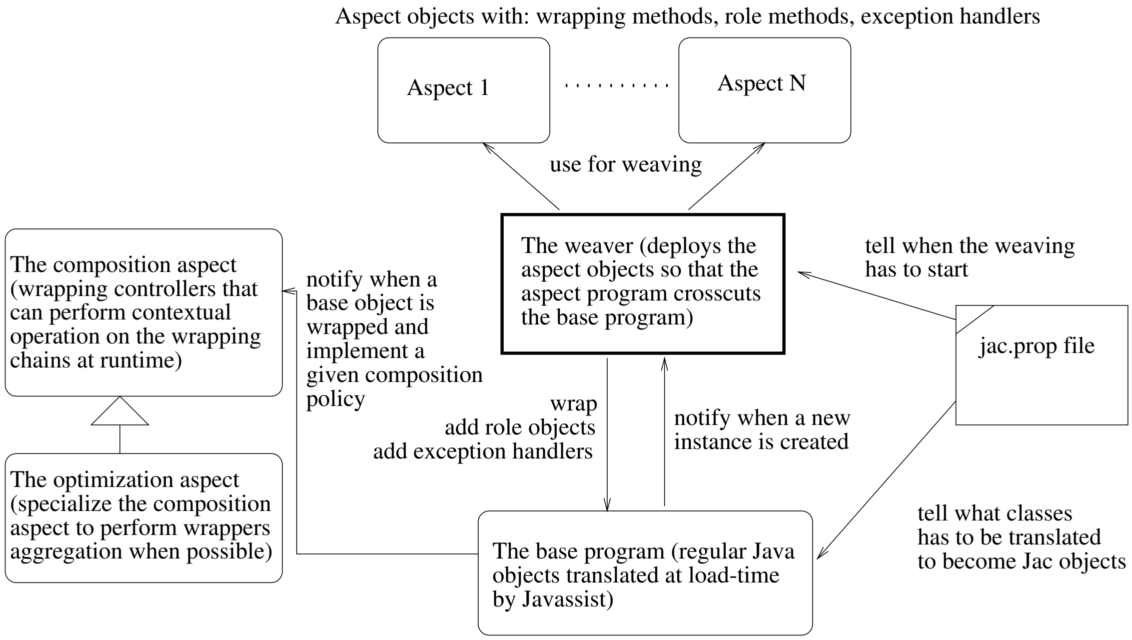

Figure 2: JAC applications architecture.

Features	JAC	AspectJ	Aspectual Components
Aspect definitions	aspect programs	aspect classes	aspectual components
Aspect entities	aspect objects	aspect classes	participants
Aspect entities members	OO model members, wrapping methods, role methods, exception handlers	OO model members, pointcuts, advices (before / after / around / introduce)	OO model members, replace
Pointcut definitions locations	weavers	aspect classes	connectors
Weave-time composition	wrapping controllers (before/afterWrap)	precedence rules	composite connectors
Runtime composition	wrapping controllers (getNextWrapper)	aspect classes (CFlows)	

Table 1: Comparison of JAC, AspectJ, and aspectual components programming models.

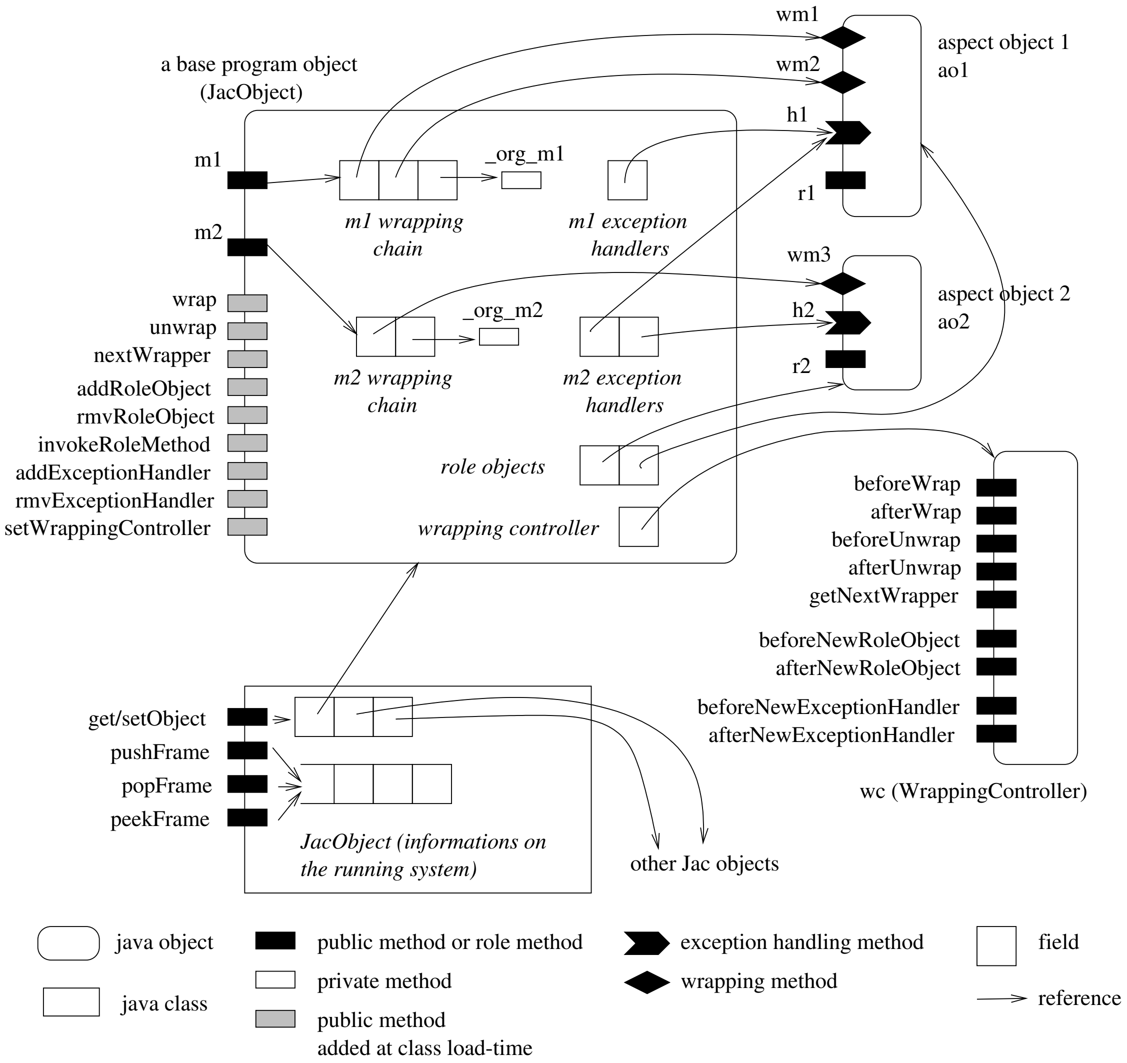

Figure 3: A JAC object overview.

3.3 Aspect objects

As described in section 3.1.2, aspect objects in JAC are containers for wrapping methods and/or role methods and/or exception handlers. There is no dedicated declaration nor configuration needed to distinguish between these three types. They are dynamically infered and checked from the method signature. When trying to use a method as a wrapping method, either the signature comply to the one of a wrapping method and the binding is accepted, either it does not and in this case, an exception is raised. The process is the same for role methods and exception handlers. This section presents the three legal signatures for aspect methods.

3.3.1 Wrapping methods

Wrappers in JAC are not like regular wrappers. Instead of sharing the same interface as the wrappees, all JAC wrappers has the same MOP-like interface with a reference to the base and an array of parameters. Nevertheless, they are not meta-objects in the sense that:

- their granularity is method-based (instead of object-based for meta-objects),

- several wrapping methods (defined in the same or different classes) can wrap a given base method,
- a wrapping method can wrap one or several base methods (defined in the same or different classes).

A wrapping method is required to be public and to return an `Object` instance. This object will be in most cases the one returned by the base method. Nevertheless, its value can be changed if any post treatments are part of the wrapper semantics. Its class can also be changed even if this situation should be less common. Indeed, from a client point of view, the wrapping mechanism must remain transparent. Thus, if on the server object side, a wrapper modifies the return parameter class, then on the client object side, another wrapper will be required to restore the expected class.

```
public Object wrappingMethodExample
( Wrappee wrappee, String methodName, Object[] methodArgs, int rank );
```

The first argument of a wrapping method is a `Wrappee` reference. This is the reference of the base object currently wrapped by these wrappers. Note that this value may change during the life of a wrappee as wrappers can be dynamically attached and removed from a base object. The `Wrappee` interface corresponds to the JAC Object Manipulation Interface mentionned in section 3.2. The second argument of a wrapping method is the wrapped method name. The third one is the array of arguments that are to be transmitted to the base method. As for the return argument, these values can be changed provided that the base method can retrieve some semantically correct values from them. The last argument gives the rank of the wrapper in the wrapping chain.

Apart for logging purpose, the value of the *rank* argument is mainly used when calling the *nextWrapper* method of the `Wrappee` interface. This method either calls the next wrapper in the chain, or the base method if the current wrapper is the last one in the chain. Note that this default behavior provided by the wrapping controller that can be redefined. Note also that the call to *nextWrapper* is optional: if omitted, the execution returns to the caller without going neither through the rest of the chain, nor the base method. In most cases, the call to *nextWrapper* looks like the following:

```
Object ret = wrappee.nextWrapper( methodName, methodArgs, rank );
```

The four above described arguments are automatically set by JAC whenever a wrapping method is called. A wrapping method signature can also contain any list of exceptions required by the wrapper semantics.

3.3.2 Role methods

The only thing that is required for a method to qualify as a role method is to be public. The return type may be an object, a basic type (such as `int`), or `void`. Programmers can define any number of arguments in its signature. If the first one class is `Wrapppee`, it will be interpreted by JAC as a reference to the wrappee object to whom this role method is attached. Role methods can also raise exceptions.

```
public Object roleMethodExample( Wrappee baseObject, ... );
```

Role methods are invoked with the *invokeRoleMethod* method provided by the `Wrappee` interface. The signature of the *invokeRoleMethod* method is the same as the *java.lang.reflect.Method.invoke* method (ie a string for the name of the role method to invoke and some arguments as an array of objects). A typical call will look like:

```
aWrappee.invokeRoleMethod( "aMethodName", new Object[]{ /** some arguments */} );
```

3.3.3 Exception handlers

Exception handling methods must be public and return `void` (they can return something else but this is useless as this value will be discarded in all cases). They require as an argument an object that represent the exception raised. The class of this object is usually a subclass of *java.lang.Exception* although JAC does not require it (ie any class will fit).

```
public void catchExample( ExException e );
```

The following sample sums up the signatures of the three possible aspect method types with JAC.

```
public class AspectObjectExample {

 /**
  * Wrapping method example.
  * All arguments are set by JAC.
  */
 public Object wrappingMethodExample
   ( Wrappee wrappee, String methodName, Object[] methodArgs, int rank )
   throws ExException {

  /** before code ... */
  if ( [...] ) { throw new ExException(); }

  /**
   * Call to the next wrapper or
   * to the base object if there is no more wrappers.
   */
  Object ret = wrappee.nextWrapper(methodName, methodArgs, rank);

  /** after code ... */
 }

 /**
  * Role method example.
  * The first (optional) parameter is automatically set by JAC
  * as a reference on the base object.
  * The method can contain any list of user specified parameters.
  */
 public Object roleMethodExample( Wrappee baseObject, ... ) {
  /* ... */
 }

 /**
  * Exception handler example.
  * ExException is the class of the exception caught by this method.
  * Usually this is a subclass of java.lang.Exception
  * although JAC does not require it (ie any class will fit).
  */
 public void catchExample( ExException e ) {
 /* ... */
 }
}
```

3.4 Weavers

Weavers are responsible for deploying aspect objects and wrapping controller on top of running base objects. A weaver must extend the *jac.core.Weaver* class and redefine the

```
public void weave();
```

method. Section 4.2 gives a running example of such a weaver. The way the *weave* method is upcalled during the base program execution is defined in a property file (`jac.prop`) that resides in the current working directory. Two properties are required:

- `jac.weaver`: gives the weaver class (ie the subclass of *jac.core.Weaver* that provides the customized *weave* method,
- `jac.startWeavingPlace`: defines where and when the weaver must be called. The value of this property is a space-separated list of four arguments.

From a generic point of view, a weaver can be called whenever a method is called on a JAC base object[1]. The first two arguments define which class and which method within this class is to be monitored. The `any` keyword is a shortcut to monitor all the methods of the class. Polymorphic methods are not distinguished. The remaining two arguments of the `jac.startWeavingPlace` property are numbers: the first one is an instance count, and the second one a method call count. Both are 0-starting counters and allow to specify policies such as "*upcall the weaver on the* i*th call to method* m *on the* j*th instance of class* c". Instances are considered in the order in which they are created in the base program. The following sample upcalls the *weave* method of class *AgendaWeaver* on the first call of any method on the first instance of the *AgendaClient* class:

```
jac.weaver: AgendaWeaver
jac.startWeavingPlace: AgendaClient any 0 0
```

Other more complex policies may also be specified. For instance:

```
jac.startWeavingPlace: AgendaClient printAll 1 5
```

upcalls the *weave* method of class *AgendaWeaver* on the 2nd call of any method on the 6th instance of the *AgendaClient* class. As a weaver must deploy aspect objects on top of running application objects, one of its key characteristics is to be able to access them. Unfortunately there is no standard way of doing this with the reflection API of Java (eg there is no such thing as accessing all the instances of a given class). To solve this issue, JAC provides a API that does just that. Basically, a shadow reference to each new JAC object instance is automatically registered in a hashtable. Then, method

```
public static Object[] getObjects( Class c );
```

in class `JacObject` returns an array of references to instances of class `c`.

3.5 Wrapping controllers

Wrapping controllers are JAC software entities that deal with the composition aspect both at weave-time and at runtime. A default wrapping controller is provided but can be customized by subclassing `jac.core.WrappingController`. Any base object can be attached to a dedicated wrapping controller, or the same controller can be shared among several base objects. Section 4.3 gives a running example of a wrapping controller.

The important point to notice is that wrapping controller externalize the composition issue of aspect-oriented programs. Thus aspect and base programs can stay free of dependancies resolving code between aspects (that occur at weave-time) and of contextual tests on the effective running of an aspect (that occur at runtime).

[1] Although this could seem to be a limitation (eg a weaver can not be called when a object reads a variable), this is a rather fair compromise as most of the important stuff in an object-oriented program is supposed to happen through method calls. More complex triggering scheme (eg groups of methods calls, code patterns, ...) could also be envisioned but are out of the scope of this document.

3.5.1 Weave-time issues

Weave-time issues mentionned in section 2.1 are adressed by reifying the wrapping mechanism. Thus each time the *wrap* method is invoked on a JAC object the call is intercepted by the following method of the wrapping controller:

```
public int beforeWrap( Wrappee wrappee, String wrappee_method_name,
                       Wrapper wrapper, String wrapper_method_name,
                       int default_rank );
```

Solutions to weave-time issues can be implemented by saying that either a wrapper must be rejected (ie not wrapped), or inserted at a given position in the wrapping chain. The *beforeWrap* method is called with the reference of the object to be wrapped (so called *wrappee*), the method name to be wrapped, the wrapper and the wrapping method name. The last argument *default_rank* is the position at which the wrapper will be inserted in the chain if nothing is done to change it (by default at the end of the chain). This method returns an integer which is the effective position (the same or a new one) choosen for this wrapper. By default, the *beforeWrap* method does nothing (more precisely it simply returns `default_rank`). This mechanism allows us to implement precedence rules (WI5). Since rank -1 means that the wrapper must not be added, we can also implement exclusion rules (WI1 to WI4).

A *afterWrap* method with the same signature than *beforeWrap* is also available in the *WrappingController* interface. This method is upcalled after the effective insertion of the wrapper in the chain and gives the ability to implement any post-treatment required. *beforeUnwrap*, *afterUnwrap*, *beforeNewExceptionHandler*, *afterNewExceptionHandler*, *beforeNewRoleObject*, *afterNewRoleObject* methods are also available. They perform the same job for respectively the unwrapping mechanism, the adding of an exception handler, and the adding of a role method.

3.5.2 Runtime issues

Section 2.2 shows that the execution of a wrapper is subject to some contextual tests. To produce better maintainable code, we think that a good design principle is to externalize these tests so that they do not crosscut and pollute the aspect codes.

This externalization is adressed by reifying the dispatch mechanism between each wrapper of the chain and between the wrappers and the base object. Thus each time the *nextWrapper* method is invoked on a JAC object the call is intercepted by the following method of the wrapping controller:

```
public int getNextWrapper( Wrappee wrappee, String wrappee_method_name,
                           Object[] wrappers, Object[] methods,
                           int default_rank );
```

Solutions to runtime issues can be implemented by saying that either a wrapper must be executed at its current rank in the chain, skipped or executed later. The *getNextWrapper* method is called with the reference of the wrapped object, the wrapped method name, the array of wrappers of the chain, and the array of wrapping methods of the chain. The last argument *default_rank* is the position of the wrapper in the chain. This method returns an integer which is the rank of the next wrapper to effectively run. By default, the *getNextWrapper* method does nothing (more precisely it simply returns `default_rank`).

The sequence diagram shown in figure 4 sums up the dispatch process and focuses on the objects interactions within the JAC system when a method is called. This protocol allows the wrapping controller to entirely control the way wrappers are applied at runtime. The figure shows the (simplified) case of a method m that is called on a base object o and that is wrapped by two wrapping methods (*wm1* and *wm2*). One can see that the wrapping controller is called when the wrapping method has finished its *before* work and calls the next wrapper. Thus, the wrapping controller can choose the next wrapper (in this example, it chooses to call *wm1* before *wm2*).

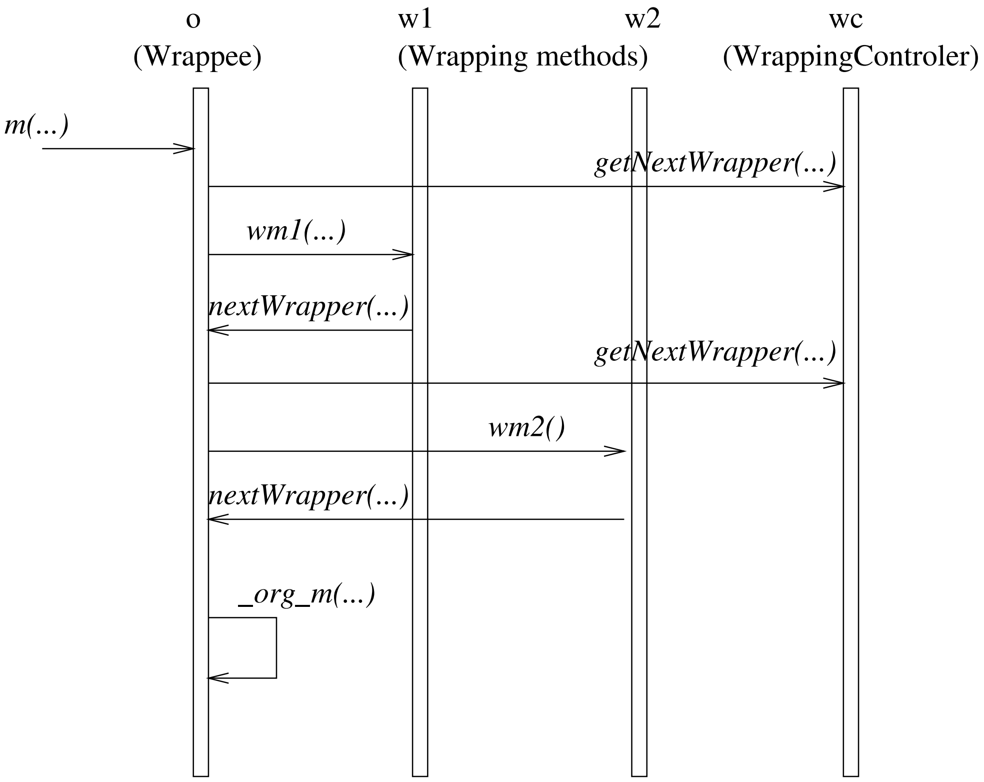

Figure 4: The wrapping control mechanism.

3.6 Additional JAC API

JAC introduces two additional sets of API to handle the execution stack and the execution context of a Java program. These features are not directly related to aspect-oriented programming. Rather they are general features that for many reasons (among other performance) should be included in the core API of the JDK. As they are not and as JAC aspect programs use them extensively, we felt the need to provide them.

3.6.1 Program stack

The idea here is to be able to query the execution stack of a Java thread. Each frame on this stack contains three elements: the called object, the called method, and the execution context (see section 3.6.2). These data are automatically pushed by JAC. Obviously, the first frame pushed onto the stack corresponds to the first method called on the first JAC object of the program. The stack can be queried with the following static method of the *jac.core.JacObject* class:

```
public static Object[] peekFrame( int index );
```

The index is an integer value between 0 and 2 corresponding to the three above mentioned elements pushed onto the stack. This method returns the elements pushed as an array of objects. Hence, `JacObject.peekFrame(1)` returns the chain of called method names as an array of objects (elements then need to be casted to strings by programmers).

Note that some kind of stack management is provided in the API of JDK 1.4. Nevertheless its use is a bit cumbersome as it requires to raise and catch an exception before calling the *getStackTrace* method on the exception object.

3.6.2 Execution context

Execution contexts in JAC are hashtables that map attribute names to values. They are thread dependant and allow us to propagate data along a call graph. The JAC API provides methods to get the current context (method *getContext*), and to add (method *addAttribute*) and get (method *getAttribute*) attributes. Each attribute is a pair composed of a key string and of value (that can be any Java object).

```
/** The following method is defined in the jac.core.Jacobject class */
public static jac.core.Context getContext();

/** The following methods are defined in the jac.core.Context class */
public void addAttribute( String name, Object att );
public Object getAttribute( String name );
public void setAttribute( String name, Object att );
```

4 Case study

This section presents the way an aspect-oriented application is developped with JAC. In a first step for clarity sake, we will not externalize the composition code. It will be done in section 4.3. We take the simple example of a agenda application. In this application, the agenda class is defined as follows:

```
public class Agenda {

   /** The agenda ID (must be unique) */
   public String id;

   /** The appointment list */
   public Vector appointments = new Vector();

   /** The repository for the agendas */
   public static AgendaRepository repository;

   /** Creates a new agenda and registers it into the repository */
   public Agenda( String id, AgendaRepository repository ) {...}

   /** Make an appointment with other users */
   public void makeAppointment(
           String agenda_id, String date, String object, String[] with ) {
      // resolve the agendas of the with array with the repository
      // call the makeAppointment method on all these agendas
   }

   /** Print the appointments */
   public void printAppointments() {
      System.out.println( "** Appointment list for " + id + ":" );
      for (int i = 0; i < appointments.size(); i++) {
         printAppointment( i );
      }
   }

   /** Print an appointment */
   public void printAppointment( int i ) {
```

```
        System.out.println( i + 1 + ". " + appointments.get(i) );
    }
}
```

It is easy to figure out that the agenda repository class implements some methods to register and to resolve agenda with a string ID. We also assume that an `AgendaClient` class that calls `Agenda` instances is defined. Thus, a client program of an agenda can, for example, use a *makeAppointment* method that takes an ID and that can be implemented as follows:

```
public class AgendaClient {

   /** The repository for the agendas */
   public static AgendaRepository repository;

   /** Make an appointment with other users */
   public void makeAppointment(
         String agenda_id, String date, String object, String[] with ) {
      Agenda a = repository.resolve( agenda_id );
      a.makeAppointment( date, object, with );
   }

   // Same principles can be applied
   // to the printAppointments methods ...
}
```

4.1 Aspects

4.1.1 Counting aspect

This aspect illustrates the RI1 issue of section 2.2.1. The idea is to be able to count the number of times the *printAppointments* by the printing methods of the method is called. As depicted in section 2.2.1, a simple but not optimized way is to create a wrapper that simply increments a counter.

```
public class CountingWrapper {
  int counter;
  public Object incr( Wrappee wrappee, String meth, Object[] args ) {
    counter++;
    return wrappee.nextWrapper( meth, args );
  }
}
```

In this example, *incr* is the wrapping method. Notice, that, at this stage, you do not specify that the *incr* method will wrap the *printAppointment* method. This will be defined later in the weaver. For this reason, our aspect objects can be regarded as pure advices that can be reused for several aspects.

As depicted in section 2.2.1, this wrapper can be optimized to avoid several calls (one for each agenda) to the *incr* wrapping method when the *printAppointments* method is called. The following aspect defines two wrapping methods: *incr* for *printAppointment* like methods, and *multiIncr* for *printAppointments* like methods. *incr* takes advantage of the JAC API (call to *JAC.peekFrame* method) and checks whether the client calling method is *printAppointments*. If so, nothing is done as the update of *counter* is handled elsewhere. If not, *printAppointment* has been directly called and *counter* needs to be incremented. *multiIncr* increases the value of *counter* depending of the number of agendas in the application object (we use the Java reflection API to introspect the wrappee fields). The *field* and *callingMethod* fields connect this aspect to a base program.

```
public class CountingWrapper {

  int counter;

  // The following fields allow the wrapper to be generic and
  // customizable regarding the base program class it has to count.
  String field, callingMethod;

  public CountingWrapper( String field, String callingMethod ) {
     this.field = field;
     this.callingMethod = callingMethod;
  }
  public CountingWrapper() {
    field = "appointments";
    callingMethod = "printAppointments";
  }

  public Object incr( Wrappee wrappee, String meth, Object[] args, int rk ) {
    // we increment the counter only
    // if the calling method is "printAppointments".
    if ( ! ( JacObject.peekFrame(1)[0] == wrappee &&
             JacObject.peekFrame(1)[1] == method ) )
      counter++;
    return wrappee.nextWrapper( meth, args, rk );
  }

  public Object multiIncr( Wrappee wrappee, String meth, Object[] args, int rk ) {
    counter += wrappee.getClass().getDeclaredField(field).get(wrappee);
    return wrappee.nextWrapper( meth, args, rk );
  }
}
```

4.1.2 Authentication aspect

A simple means to implement an authentication aspect is to wrap the *AgendaClient* instances with a client-side wrapper that adds the agenda ID in the context and to wrap the agendas with a server-side authentication wrapper that will read the context to check if the client accesses the right agenda. Notice that the server-side authentication wrapper throws an exception in the case the authentication fails. This is a good example to illustrate the use of (1) exception handlers (defined here at the client-side), and (2) contexts. The following class defines the *addAuthInfos* wrapping method and the *catchAuthenticationException* exception handler for *AgendaClient* instances.

```
public class ClientAuthenticationWrapper {

   public Object
     addAuthInfos( Wrappee wrappee, String meth, Object[] args, int rk ) {
     JacObject.getContext().addAttribute( "clname", args[0] );
     return wrappee.nextWrapper( method, args, rk );
   }

   public void catchAuthenticationException( AuthenticationException e ) {
      System.out.println( e.printStackTrace() );
   }
}
```

At the server-side, another RI1 issue type occurs. Indeed, since the *Agenda* class also calls the *makeAppointment* method to notify the other agendas that a common appointment has been taken, it implies that the server-side authentication wrapper must be skipped if the *makeAppointment* is called by an agenda. Once this has been done by the *checkAuthInfos* wrapping method we check whether the *clName* attribute is present in the context transmitted by the *Agenda* client and if so, if the client is authorized to access this agenda. If not, we throw an exception that will be catch by the exception handler defined previously.

```
public class ServerAuthenticationWrapper {
   public Object
     checkAuthInfos( Wrappee wrappee, String meth, Object[] args, int rk )
       throws AuthenticationException {

     if ( ! (JacObject.peekFrame(1)[0] instanceof Agenda) ) {

        Object clientName = JacObject.getContext().getAttribute("clname");
        if ( clientName == null ||
             ! clientName.equals(wrappee.getFieldValue("id")) ) {
           throw new AuthenticationException(
                   clientName+" is not authorized to access "+meth );
      }

      return wrappee.nextWrapper( method, args, rk );
   }
}
```

4.2 Weaver

The weaver is the part of a JAC program that weaves the aspects to the base program, i.e. deploys the aspect objects on the base objects. For example, the following weaver weaves the counting and the authentication aspects to a base program that contains several instances of the *Agenda* and *AgendaClient* classes.

```
public class AgendaWeaver extends Weaver {

  // Some information to parameterize the join-points
  String calledMethod = "printAppointment";
  String callingMethod = "printAppointments";
  String field = "appointments";
  String authenticatedMethods =
   { "makeAppointment", "printAppointment", "printAppointments" };
  String clientMethods = { "makeAppointment" };

  public void weave() {

    Object[] agendas = JacObject.getObjects(Agenda.class);
    Object[] clients = JacObject.getObjects(AgendaClient.class);

    // create a server authentication wrapper
    ServerAuthenticationWrapper saw = new ServerAuthenticationWrapper();

    // create a server authentication wrapper
    ClientAuthenticationWrapper caw = new ClientAuthenticationWrapper();
```

```
    // wrap the agendas to add authentication and counting
    for ( int i=0 ; i < agendas.length ; i++ ) {

      // create one counting wrapper per agenda
      CountingWrapper cw = new CountingWrapper( calledMethod, callingMethod, field );
      agendas[i].wrap( cw, "incr", calledMethod );
      agendas[i].wrap( cw, "countWithField", callingMethod );
      agendas[i].wrap( saw, "checkAuthInfos", authenticatedMethods );
    }

    // wrap the clients to add authentication
    for ( int i=0 ; i < clients.length ; i++ ) {
      agendas[i].wrap( caw, "addAuthInfos", clientMethods );
    }
  }
}
```

Furthermore, we specify with the following configuration file that the weaving is to be done on the first method call on the first instance of the *AgendaClient* class (see section 3.4 for details about this property file).

```
jac.weaver: AgendaWeaver
jac.startWeavingPlace: AgendaClient any 0 0
```

4.3 Composition aspect

In most common cases, the authentication must be applied first. Indeed, it is quite clear that any request to an object must be authenticated before any job is perform and even before any other aspect is runned. Thus we need to implement a rule that forces the authentication aspect to be called before any other aspect. The wrapping controller that performs this weave-time check follows (see method *beforeWrap*). It also checks that the authentication wrapper is applied only once to the same base object.

This wrapping controller also externalizes the counting wrapper optimization rule described in section 4.1.1. By this way, we let the aspect code free from any pollution and the composition policy can be centralized within a well-bounded software entity.

```
public class MyWrappingController extends WrappingController {

  /** beforeWrap deals with weave-time issues related to the composition aspect */

  public int beforeWrap( Wrappee wrappee, String wrappee_method_name,
                         Wrapper wrapper, String wrapper_method_name,
                         int default_rank ) {

    if ( wrappers.length > 0 &&
         wrappers[0].getClass() == ServerAuthenticationWrapper.class ) {

      // refuse to wrap authentication twice !
      if ( wrapper.getClass() == ServerAuthenticationWrapper.class ) {
         return -1;
      }
      // put the new wrapper after the authentication wrapper
      return 1;
    }
    return default_rank;
```

```
  }

  /** getNextWrapper deals with runtime issues related to the composition aspect */

  String calledMethod = "printAppointment";
  String callingMethod = "printAppointments";

  public int getNextWrapper( Wrappee wrappee, String wrappee_method_name,
                             Object[] wrappers, Object[] methods, int rank ) {

      // check if the calling method is "print"
      if ( wrappee_method_name == calledMethod &&
           JacObject.peekFrame(1)[0] == wrappee &&
           JacObject.peekFrame(1)[1] == callingMethod &&
           wrappers[rank] instanceof CountingWrapper ) )

         // skip the wrapper!
         return rank + 1;

       else
         return rank;
  }
}
```

5 Performance issues

We now present some performance measurements about JAC. Section 5.2 discusses about an ongoing development that aims at speeding up the execution of an aspect-oriented program with JAC.

5.1 JAC performance measurements

This section studies the cost of running an aspect-oriented program with JAC. As illustrated in figure 4, we need one regular invocation to the wrapping controller before choosing each wrapper and two invocations per wrapping method (one to call it, and one when the *nextWrapper* method is called by the wrapper).

The following trace is the output of a benchmark program that creates an object and calls 100 times an empty method on it. It runs under JVM2 and Linux with a Pentium 300 MHz processor. The program performs nothing, objects are regular Java objects, they are not translated by Javassist, and there is no weaving by JAC. We just run it to get the amount of time taken by the JVM to do this job.

```
>>> start time: 981972594672 ms
>>> end of class loading: +639 ms [duration: 639 ms]
>>> end of benchmark: +1242 ms [duration: 588 ms]
```

To test the overload due to the wrapping mechanism, we implement an empty aspect that wraps the base object with 10 empty wrappers and adds a wrapping controller that is upcalled but does not change the wrappers ordering. Note that 10 wrappers is quite a huge number and we don't expect such a case to be encountered (1, 2 or 3 aspects seems to be a more adequate case for the near future). The same program running with JAC produces the following trace.

```
>>> start time: 981973419075 ms
>>> end of class loading/translating: +1484 ms [duration: 1484 ms]
```

```
>>> end of weaving: +1586 ms [duration: 102 ms]
>>> end of benchmark: +6249 ms [duration: 4663 ms]
```

The overhead of the dynamic application of 10 wrappers (even empty) when the base method is called 100 time is huge: 4663 ms against 588 ms. This is no surprise as we replaced calls to 1 base object with calls that go through 10 wrapper objects and 1 base object. This roughly gives us an overhead of 4 ms per base call per wrapper. The first duration gives the time needed to load the classes and adapt them with Javassist. Compared to the previous case where the classes where loaded with the standard classloader of the JVM, the duration is multiplied by 2.4. To reduce it, JAC proposes a starting option that writes the result of the class translation in a temporary directory so that, at the next start of the application, the class will not be translated anymore (unless we explicitly require it). When the translated classes are stored, the result is almost the same than when no translation occurs (655 ms against 639 ms).

```
>>> start time: 981973903301 ms
>>> end of class loading/translating: +655 ms [duration: 655 ms]
>>> end of weaving: +755 ms [duration: 100 ms]
```

This gives us a simple mean to optimize class translation. The second overhead already mentioned (4663 ms against 588 when 10 wrappers are inserted before 100 calls to 1 base object) can only be overcome if fewer wrappers are used. We are in the process of working on a mechanism that we call *wrapper aggregation* that dynamically aggregates the whole wrapping chain and the wrappee into one unique object. Each method of this aggregation is constructing by inlining the bytecode of wrapping methods and of the wrapped method (note that it also aggregates the states of the wrappers and of the base object; this mechanism induces some state consistency issues between the aggregations and the regular objects but we will not enter into details here). By this way, we can expect the cost of running a application weaved by 10 aspects to be greatly improved.

Next section explains how we can deal with the second overhead (which comes from the wrapping mechanism).

5.2 Optimizing aspects with JAC

To deal with the performance issue, we currently work on proposing a mechanism that we call *wrapper aggregation* and that dynamically aggregates the whole wrapping chain and the wrappee into one unique object. Each method of this aggregation is composed by inlining the bytecode of the set of the wrapping methods and of the wrapped method (note that it also aggregates the states of the wrappers and of the base object; this mechanism induces some state consistency issues between the aggregations and the regular objects but we will not enter into details here).

When an aggregation is available, the only reflective call is the one that delegates the work to the aggregation. Moreover, contrary to regular wrappers that use the java.lang.reflect package, the aggregation wrappers perform direct access to the wrappee state and thus run more efficiently. The aggregation creation mechanism is efficient since it does not require any bytecode loading because all the involved bytecodes are already loaded within the VM (we store them in our JAC class loader).

Since the JAC framework cleanly separates the composition aspect from the other aspects, these optimizations can be context-sensitive and be really efficient. For example, if it appears in the wrapping controller that the wrappers order does not depend on the context, then we can aggregate the wrapping chain once and never use the dynamic wrapping mechanism any more. Of course, if the wrapping chain changes at runtime (in this case the wrapping controller is notified thanks to the *beforeWrap* method) then the aggregation should be replaced by a new one.

The following code shows how to optimize the document example. Depending on the context (whether the calling method is *printAll* or not), the whole wrapping chain is replaced by an aggregation that contains or not the counting wrapper. With this method, the context test is done only once and the appropriate aggregated wrapping chain is called very efficiently (an aggregate does not perform reflective invocations and does not upcall the wrapping controller anymore).

Notice that this kind of optimization would not have been possible if the composition aspect was not centralized within the wrapping controller. Indeed, if the composition information is hardcoded within the aspect codes, then the context-sensitive tests would be always performed for all the aspects even if not needed.

```
public class EfficientCountingWrappingController
    extends CountingWrappingController {

  Object noSkipAggregate = null;
  Object skipAggregate = null;

  public int getNextWrapper (
               Wrappee wrappee, String wrappee_method_name,
               Object[] wrappers, Object[] methods, int rank ) {

    if ( skipAggregate == null ) {
      // create the aggregate and memorize it in the wrappee
      skipAggregate = wrappee.createAggregate(
         getWrappersWithSkip(wrappers, CountingWrapper.class) );
    }

    if ( noSkipAggregate == null ) {
      // create the aggregate and memorize it in the wrappee
      noSkipAggregate = wrappee.createAggregate(
         getFullWrappingChain(wrappers));
    }

    // check if the calling method is "print"
    if ( wrappee_method_name == calledMethod &&
         JacObject.peekFrame(1)[0] == wrappee &&
         JacObject.peekFrame(1)[1] == callingMethod ) {
      // call the aggregate (this stops the current wrapping
      // chain evaluation)
      wrappee.callAggregate(skipAgregate);
    } else {
      wrappee.callAggregate(noSkipAgregate);
    }
  }
}
```

The following trace is for the same benchmark as the one described in section 5.1 but we have added a wrapping controller that aggregates the wrapping chain of the ten wrappers into one unique aggregation[2]. It shows that the overhead (one regular call to the wrapping controller, one reflective call and one aggregation creation) makes the JAC program less than one and a half slower than the pure Java version. This is, to us, an excellent tradeoff regarding the flexibility JAC furnishes at runtime (moreover, once the aggregations are created, their creation overhead is not sensible any more).

```
>>> start time: 981983417636 ms
>>> end of class loading: +648 ms [duration: 648 ms]
>>> end of weaving: +747 ms [duration: 99 ms]
>>> end of aggregation creation: 848 ms [duration: 101 ms]
>>> end of benchmark: +1543 ms [duration: 695 ms]
```

[2]Note that our aggregating system is currently a work in progress and is not yet available. However, the program we made is a quite faithful simulation of what will happen when an aggregation occurs.

6 Related works

In [BW00], Büchi and Weck designed a mechanism called generic wrappers. Generic wrappers are type safe and support modular reasoning. Their focus is oriented towards the definition of reusable and composable components implemented by different vendors. Type soundness is one of their main concerns. Multiple wrappers for a single wrappee can be defined in their approach. Nevertheless, their system does not dynamically manage the execution order of wrappers in a wrapping chain as our wrapping control does.

The composition filter object model [BA01] (CFOM) is an extension to the conventional object model where input and output filters can be defined to handle sending and receiving of messages. This model is implemented for several languages, including Smalltalk, C++ and Java [Wic99]. The latter implementation is an extension to the regular Java syntax where keywords are added to declare, for instance, filters attached to classes. The goals of this model and ours are rather similar: to handle separation of concerns at a meta level. Nevertheless, JAC does not require any language extension (i.e. wrappers and wrappees are written in regular Java).

AspectJ [KHH$^+$01] is a powerful language that provides support for the implementation of crosscutting concerns through pointcuts (collections of principle points in the execution of a program), and advices (method-like structures attached to pointcuts). Precedence rules are defined when more than one advice apply at a join point. In many features (e.g. pointcuts definition) AspectJ has a rich and vast semantics. Nevertheless, we argue that in many cases that we have studied, simple schemes such as the wrapping technique proposed by JAC are sufficient to implement a broad range of solutions dealing with separation of concerns.

Aspectual components [LLM99] and their direct predecessors adaptative plug and play components [ML98][MSL01] define patterns of interaction, called participant graphs (PG), that implement aspects for applications. PGs contain participants roles (e.g. publishers and subscribers in a publish/subscribe interaction model) that, (1) expect features about the classes upon which they will be mapped, (2) may reimplement features, and (3) provide some local features. PGs are then mapped onto class graphs with entities called connectors, that define the way aspects and classes are composed. Aspectual components can be composed by connecting part of the expected interface of one component to part of the provided interface of another. Nevertheless, it seems that by doing so, the definition of the composition crosscuts the definition of the aspects, loosing by this way the expected benefits of AOP. The approach taken in JAC consists in modularizing this crosscuting concern (i.e. inter-aspects composition) in the so-called wrapping controller (see section 4.3).

Subject oriented programming [HO93][OKH$^+$95] (SOP) and its direct successor the Hyper/J tool [TOHS99], provide the ability to handle different subjective perspectives, called subjects, on the problem to model. Subjects can be composed using correspondence rules (specifying the correspondences between classes, methods, fields of different subjects), combination rules (giving the way two subjects can be glued together, and correspondence-and-combination rules that mix both approaches. Prototype implementations of SOP for C++ and Smalltalk exist, and a more recent version for Java called Hyper/J is available. This latter tool implements the notion of hyperspace [OT01] that *permits the explicit identification of any concerns of importance.*

JAC also shares some caracteristics with other research projects. [Bus00] propose an event/action model to compose aspects in a CORBA environment. Still related to the event/action paradigm, Douence & al. [DMS01] are interested in formalizing the notion of crosscut. [BSLR98] studies the notion of compatibility between aspects and meta-classes. The JavaPod [BR01] platform for adaptable components offers an heritance like mechanism to extend object with non-functional properties.

We think that JAC covers a field that, up to our knowledge, is not fully and cleanly addressed by any of these solutions: dynamic ordering of aspect programs and context-sensitive optimizations within the composition aspect. JAC is widely inspired from the Lasagne abstract model [TVJ$^+$01] that defines some concepts to achieve dynamic and context-sensitive selection of collaboration refinements depending on the client of the application (but that is less flexible since it does not allows runtime reordering of the wrapping chain).

7 Conclusion and future works

Separation of concerns is one of the major requirements for modern applications. Flexibility and dynamic evolution are also needed most of the time. In this paper, we present the JAC framework that meets both needs by using the notion of wrapping controller to implement a composition aspect. To fix the ideas, figure 2 summed up how all the JAC parts interoperate when building an aspect-oriented application. JAC takes advantage of the Javassist [Chi00] load-time MOP to transparently implement the needed glue between aspect and application programs.

It is to notice that, contrary to AspectJ [KHH+01] that focuses on the pointcuts expression with a new language, we mainly focus on the definition of a generic architecture for AOP. We believe that we have reached many of the desirable properties needed by aspect-oriented system and languages and that our framework can be later on coupled with a more high-level language in order to facilitate the programmer task. The following list summarizes the main features provided by JAC.

- The base program is written in regular Java, can be launched independently from the aspects, and the source code is not needed for the weaving.
- Aspect programming does not require any syntactical extension. Aspect objects with JAC contains methods that can be either wrapping methods (that provides the ability to run *before*, *after*, and *around* code), role methods (that introduce new features in application objects), or exception handlers. Compared to AspectJ they can be seen as pure advice entities. Aspect objects in JAC remain free from any deployment or composition issues that are completely handled by the weaver and the wrapping controller. In our sense, this makes aspect programs more generic and more reusable.
- The weaver is a regular Java program that uses introspection features so that any kind of crosscutting schemes can be implemented. The weaver is responsible for deploying aspect objects onto application objects. So it defines the way aspects are composed with applications. Several wrapping methods can wrap a given application method creating by this way, wrapping chains. Several role methods and exception handlers can also be attached to a given application object. Furthermore, the weaver is notified when a new instance of a base class is created so that the aspect can be extended in the mean time of the base program extension. Assuming some transactional features, aspects could also be smoothly added and removed at runtime, without stopping the application. The weaving mechanism is object-based and is well-fitted to distributed programming since the modification of a given instance do not necessary affect the other class instances (thus allowing heterogenous environments).
- The wrapping controller is a regular Java program that externalizes aspect composition. It allows context-sensitive modifications of the wrapping chains. This composition aspect should be programmed by a programmer that knows about the whole set of aspects that can be woven to the application. The main advantages of this feature are to greatly simplify the programming of the weaver (that just handles the deployment of aspect objects) and of aspect objects (that just perform core functionalities). This last property allows the aspect programmer to produce much more generic and reusable aspect code than it would be if s/he had to deal with the aspect composition issue.

The JAC framework is used by the "Ecole Centrale de Lille" Laboratory to implement the software part of the CarVia application that consists in controlling electronic devices via the power-line network. JAC is currently under evaluation by Alcatel Research to implement a security aspect within a network management platform. JAC has also been used to implement the Lasagne abstract model [TVJ+01] and a distributed agenda management application. These concrete projects tend to prove that the JAC framework allows the programmer to produce high quality and easily maintainable code. During the development process of these applications, the benefits of AOP are fully used and the concerns are developed independently from each other.

In the future, we will focus on the aggregation optimization so that JAC can be as efficient as less dynamic aspect-oriented frameworks. We will also study the composition aspect to find out some recurrent patterns and useful abstractions so that we can propose a more high-level programming interface to deal with this issue. The purpose of this work is to make the composition process as automatic as possible.

References

[BA01] L. Bergmans and M. Aksit. *Software Architectures and Component Technology: The State of the Art in Research and Practice*, chapter Constructing Reusable Components with Multiple Concerns Using Composition Filters. Kluwer Academic Publishers, 2001.

[BMV00] J. Brichau, W. De Meuter, and K. De Volder. Jumping aspects. Presented at the ECOOP 2000 workshop on Aspects and Dimensions of Concerns, June 2000. `http://trese.cs.utwente.nl/Workshops/adc2000/`.

[BR01] E. Bruneton and M. Riveill. Experiments with javapod, a platform designed for the adaptation of non-functional properties. In *Procedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 52–72. Springer, September 2001.

[BSLR98] N. Bouraqadi-Saadani, T. Ledoux, and F. Rivard. Safe metaclass programming. In *Proceedings of the 13th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'98)*, volume 33 of *SIGPLAN Notices*. ACM Press, October 1998.

[Bus00] L. Bussard. Towards a pragmatic composition model of corba services based on aspectj. In *Workshop on the Aspects and Dimensions of Concerns at ECOOP'2000*, June 2000. `http://trese.cs.utwente.nl/Workshops/adc2000/`.

[BW00] M. Buchi and W. Weck. Generic wrappers. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 201–225. Springer, June 2000.

[Chi00] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, June 2000.

[DMS01] R. Douence, O. Motelet, and M. Sudholt. A formal definition of crosscut. In *Procedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186. Springer, September 2001.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings of OOPSLA'93*, volume 28 of *SIGPLAN Notices*, pages 411–428. ACM Press, October 1993.

[JAC] JAC. The JAC project home page. `http://cedric.cnam.fr/caolac/jac/`.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, June 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997.

[LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University's College of Computer Science, April 1999.

[ML98] M. Mezini and K. Lieberherr. Adaptative plug-and-play components for evolutionary software development. In *Proceedings of OOPSLA'98*, volume 33 of *SIGPLAN Notices*, pages 96–116. ACM Press, 1998.

[MSL01] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In L. Bergmans and M. Aksit, editors, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2001.

[OKH+95] H. Ossher, K. Kaplan, W. Harrison, A. Matz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA'95*, volume 30 of *SIGPLAN Notices*, pages 235–250. ACM Press, 1995.

[OT01] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2001.

[Par72] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[PDF99] R. Pawlak, L. Duchien, and G. Florin. An automatic aspect weaver with a reflective programming language. In *Proceedings of Reflection'99*, volume 1964 of *Lecture Notes in Computer Science*. Springer, July 1999.

[PDF+00] R. Pawlak, L. Duchien, G. Florin, Laurent Martelli, and Lionel Seinturier. Distributed separation of concerns with Aspect Components. In *Proceedings of TOOLS Europe 2000*, June 2000.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, pages 107–119, 1999.

[TVJ+01] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, Joergensen, and N. Bo. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of ICSE'01*, 2001.

[Wic99] K. Wichman. ComposeJ: The development of a preprocessor to facilitate composition filters in the Java language. Master thesis, Dept of Computer Science, University of Twente, December 1999.
`http://trese.cs.utwente.nl/publications/paperinfo/wichman.thesis.pi.ref.htm`.

Contents

List of Figures

List of Tables