Can Aspects Be Injected ?
Experience with Replication and Protection

Sara Bouchenak[1], Fabienne Boyer[1], Noel De Palma[2], Daniel Hagimont[3]

[1] Université Joseph Fourier
[2] INPG (Grenoble Institute of Technology, France)
[3] INRIA (French National Institute for Research in Computer Science and Control)
INRIA – Sardes Research Group
655, avenue de l'Europe, 38334 Montbonnot St Martin France
{ Sara.Bouchenak , Fabienne.Boyer, Noel.Depalma, Daniel.Hagimont, }@inria.fr

Abstract. Separation of concerns, which allows programming the nun-functional aspects of an application in a more or less orthogonal manner from the functional code, is becoming a general trend in software development. The most widely used architectural pattern for implementing aspects involves indirection objects, raising a performance overhead at execution time. Thus, it appears as being an attractive challenge to be able to inject the code of aspects within the business components of an application in order to avoid indirection objects. With regard to two aspects (replication and protection), this paper replies to the following question: being given the code of an aspect *as with an indirection-based implementation*, is it possible to use a generic (aspect-independent) tool which would automatically inject this code within the application components ? The results show that this injection process is feasible and can be automated through the handling of a specific injection pattern.

1 Introduction

A general trend in software development is to separate, as long as possible, the development of different *aspects* of an application in order to improve the quality of the software which is easier to maintain. This principle of aspect separation has been addressed in several contexts, such as the AoP domain (Aspect-Oriented Programming]) [6] and the component based programming research community. In the last case, this lead to the development of middleware environments such as CCM [8] and EJB [13], where aspects are described separately from applications' business code, and associated at runtime with applications through configurable objects (containers) that link aspects to business components.

Whatever the context is, most implementations rely on *indirection objects* for the implementation of aspects. These objects capture the interactions between the application components and allow executing additional treatments during these interac-

tions. However, they generate a performance overhead at execution time, which may become significant for applications which involve a large number of interactions between components.

This paper examines the feasibility of injecting the code of aspects within the business components of an application in order to avoid indirection objects. A main goal of our work is to let the aspect programming model stay unchanged. The proposed injection process is thus applied to the code of an aspect programmed as with an indirection-based implementation. With regard to an experimentation performed with two aspects (replication and protection), the contribution of this paper is to reply to the following question : being given the code of an aspect as with an indirection-based implementation, is it feasible to automatically inject this code within the application components ?

In a previous work on aspects, we implemented two prototypes, managing respectively replicated and protected components through indirection objects [3][4]. In order to improve the performance of the applications, we implemented a new version of each prototype, based on aspect-injection. These new prototypes were however implemented in an aspect-specific manner. The proposition that is made in this paper corresponds to the lessons learned from these experiments, which have shown that it is possible to implement a generic injector which takes as input the classes of the indirection objects managing an aspect.

The rest of the paper is organized as follows. The following section presents the basic principle of the injection process. Sections 3 and 4 respectively describe our experiments with the replication and protection aspects. Section 5 summarises the lessons learned from the presented experiments. The performance aspect is treated in in section 6. Finally, in Section 7 we describe the related work and in section 8 we present our conclusions.

2 Approach

In the indirection-based implementation of aspects, several successive indirection objects[1] may be involved, forming a chain between an invoking object and an invoked object. In the rest of the paper, we respectively use *caller* for the invoker object and *callee* for the invoked object. These terms are also used to denote a *potential* caller (an object having a reference to another object) or a *potential* callee.
In a general framework, two indirection objects are involved: the first one, associated with the caller, which we call the Client Indirection Object (CIO) and the second, associated with the callee, called the Server Indirection Object (SIO)[2], as shown in Figure 1. In more complex frameworks, the cardinality of the CIO or SIO may vary

[1] The term object may refer to simple Java objects as well as components.

[2] CIO and SIO can be considered as stub and skeleton.

from 0 (either the CIO or the SIO may be absent) to n (an aspect may involve multiple CIO and SIO, as for the protection aspect considered further in this paper).

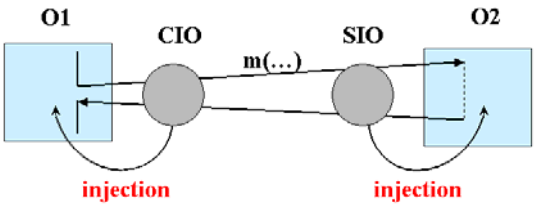


Fig.1. Principle of aspect injection

The CIO and SIO capture interactions between objects and respectively allow the association of aspect-related code with the caller and callee objects. The general idea that we have experimented is to inject the aspect-related code within the code of the application.

Code injection techniques that we experimented apply at the level of Java bytecode. In the rest of the paper, and for clarity purpose, we will illustrate our experiments through Java source code.

3 Replication aspect

The Javanaise replication service [4] [5] that we designed and implemented previously, is taken as reference. The Javanaise objects are transparently replicated, so that the application developer can program as in a centralized setting. (S)he only has to specify the access modes associated with its application's objects, i.e., read/write modes of the business (i.e. application-specific) methods. This is done through a Java-based extended IDL (Interface Description Language). Javanaise ensures the consistency of the replicated objects, which are brought on demand on the requesting nodes and are cached until invalidated by the coherency protocol.

The implementation of Javanaise using indirection objects is presented in 3.1 while the injection-based implementation is described in 3.2.

3.1 Indirection-based implementation of replication

Javanaise has been implemented on top of Java and relies on indirection objects. For each object reference, a pair of indirection objects (*CIO* and *SIO*) is transparently inserted between the callee (the object identified by the reference) and the caller (the object that contains the reference). This transparency is ensured by the fact that the caller views the CIO as being the callee (the CIO implements the same interface as the callee).

The Object1 and Object2 classes given in Figure 2, illustrate a sample program where an object *o1* calls a method *foo* which itself calls a method *m* on an object *o2*. In a centralized and non-replicated environment, the caller *o1* would directly reference the callee *o2*. When implementing replication using indirection objects, and in the case the callee *o2* is replicable, the caller *o1* references a CIO which itself references a SIO that references the callee *o2*.

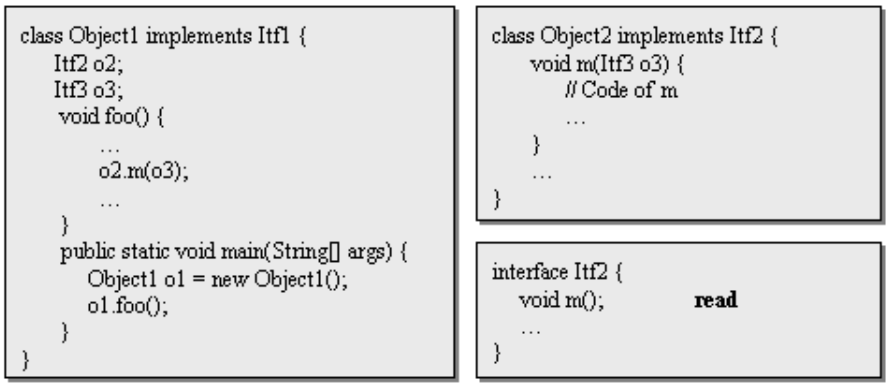


Fig.2. A sample program with replicated objects

Let's focus on the call of the method *m* by the object *o1* on the object *o2*. The associated CIO and SIO are implemented as illustrated by Figure 3 :

- *o2_CIO* manages the *dynamic binding* to the callee *o2*. It contains the unique identifier of *o2* (*id_o2* in *o2_CIO*'s code), and a reference to the associated SIO (*o2* in *o2_CIO*'s code). If this reference is null, it means that it is the first time that *o1* accesses *o2*. In this case, a copy of *o2* is fetched, either locally if *o2* is already cached or remotely from a Javanaise server using *o2*'s unique identifier.
- *o2_SIO* manages the *synchronization*, i.e., invalidation and update, of an *o2* replica. It contains the unique identifier of *o2* (*id_o2* in *o2_SIO*'s code), and a reference to a local replica (*o2* in *o2_SIO*'s code). According to the access modes specified in the object's interface, the methods of a replicated object are bracketed with *lock_read*/*lock_write* and *unlock_read*/*unlock_write* calls. The full explanation of the protocol can be found in [4].

This implementation using indirection objects significantly increases the number of method calls. In the given example, the call of method *m* from *o1* to *o2* is transformed into three method calls: first *o1* calls method *m* on *o2_CIO*, then *o2_CIO* calls method *m* on *o2_SIO*, and finally *o2_SIO* calls the effective method *m* on *o2*.

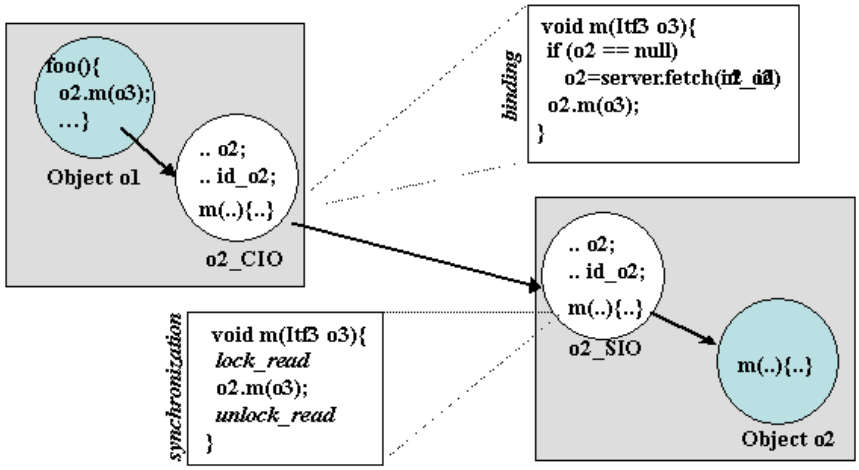


Fig. 3. The invocation of O2 by O1 is performed on a local replica of O2

3.2 Injection-based implementation of replication

In this section, we detail how we used code injection to implement more efficiently the replication service. The basic idea is to inject the CIO (resp. the SIO) within the caller (resp. callee) object.

Injection of CIO into a caller object
Injecting the CIO in the caller object implies to inject both the CIO's code and the CIO's data within the caller's class.

The CIO implements the methods defined in the interface of the callee object. It forwards the invocations of each method towards the SIO of the callee object, but it executes some pre and post treatments. These treatments are injected in the caller before and after the concerned invocations. In the *o1* object for example, the code that checks the binding to *o2* is injected before any call on *o2* (see Figure 4).
Injection of the CIO's data is difficult, because these data are strongly coupled with the reference of the callee object. Any reference manipulation should keep into account the CIO's data in a way that ensures that these operations provide the same semantic than with an indirection-based implementation. With regard to the operations that can be performed on a reference (assignment, transfer as parameter and comparison), there are two aspects to consider to manage the CIO's data :

- this data should follow the reference when transmitted from one object to another,
- the consistency of this data with regard to the reference should be ensured.

The first aspect is required to provide the semantic of the indirection-based implementation, in which a receiver of a reference obtain a way to access the CIO's data associated with the reference. Each time an object reference is transferred as a method parameter or as a result, some code should be injected to transfer, as additional parameters, the CIO's data associated with the reference.[3] This may be performed by modifying the signature of the business methods of the application objects. In the case of object *o2*, the signature of the business method *m* has been modified in order to include the data *(id_o3)* associated with the received reference to *o3* (see Figure 4).

In the same way, each time an object reference is assigned, some code should be injected to assign accordingly the CIO's data associated with the reference. In the case of *o1*, the CIO's data associated with the *o2* reference are composed of the unique identifier of *o2* (see the declaration of *id_o2* in Figure 4). Any assignment of the *o2* reference should also raise the according assignment of the *id_o2* data.
The management of reference comparison depends on the application programming model, which may allow or disallow the direct comparison of references (through the use of the == operator). For centralized applications, direct comparison may be authorized. In this case, the injection process does not perturb this model because it also permits a direct comparison between object references.

In the case only indirect comparisons are authorized by the programming model, through the use of a specific method defined in the CIO's code (such as the *equals*() method), then the injection process operates as for any CIO's method, by expanding the *equals()* invocation in the code of the application.

Injection of SIO into a callee object
SIO injection implies the injection of both the SIO's data and the SIO's code within the associated callee object replica.

The injection of the SIO's data is more or less straightforward. In the case of the o2 object, both the declaration of id_o2 (unique identifier of o2) and of the lock field (for the synchronisation of o2) of the SIO have to be injected in o2's class.

As for the CIO objects, the SIO's code is mainly composed of methods which perform actions before and after calling the effective business methods on the callee. The injection of the SIO's code consists in placing this code at the right place in the callee (see figure 5).

Finally, in accordance with the injection of CIO objects, the signatures of the business methods of a callee object may have to be modified in order to take into account the transfer of the CIOs data. Practically, in the case of *o2*, the signature of the busi-

[3] In fact, the receiver of a reference has at least to be able to retrieve the CIO's data associated with the received reference. The most simple way to perform this is to transfer a copy of these data along with the reference, which is possible in the most common cases because the CIO's data size is often small.

ness method *m* has changed in order to take the identifier associated with the received reference of *o3*.

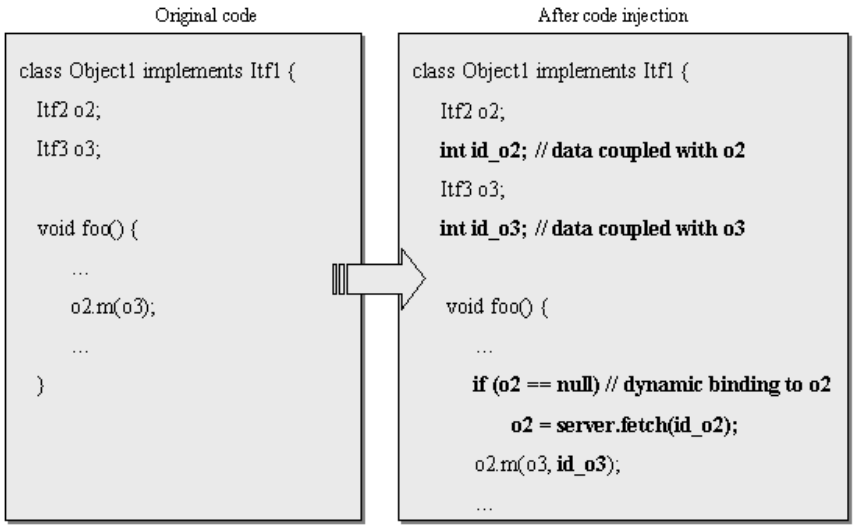


Fig. 4. Injection of the CIO in the case of the sample program of Figure 3

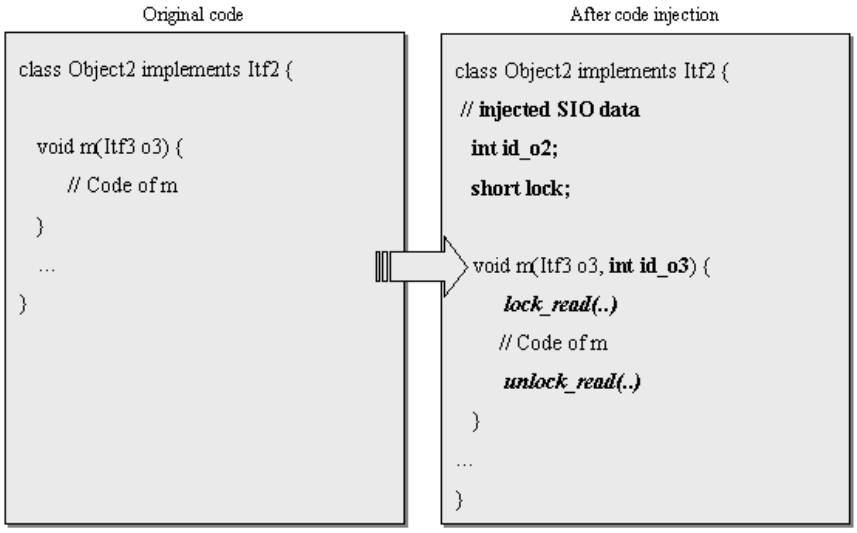


Fig. 5. Injection of the SIO in the case of the sample program of Figure 3

The experience on using the code injection techniques described in this section has shown that injection is feasible in the case of replicated distributed objects. The main injection pattern which has been highlighted by this experience is the need to manage *data strongly coupled with object references*, in a way that ensures that these data follow the reference all along its road. In the present case, these data correspond to the CIO's data which are composed of the unique identifier of the object identified by a reference.

4 Protection aspect

The second aspect that we addressed is protection, which purpose is to control interactions between mutually suspicious objects. We take as reference a protection model that we proposed in a previous work, based on *hidden software capabilities* [3].

4.1 The considered capability-based protection aspect

The protection of a given application is expressed in an extended Interface Definition Language (IDL) through the concept of *views* which are restrictions of Java interfaces. A view specifies a set of authorized methods. A *not* before a method declaration means that the method is not permitted. When an object reference is passed as parameter in a view, the programmer can specify the view to be passed with the reference using the key-word *pass*. If no view is specified, it means that no restriction is applied to this reference.

Let's consider the example of a printing service that allows a client to print out a file; this service provides a *print* operation for printing a text, and an *init* operation, for resetting the underlying printer. Figure 6 shows the protection views related to the printer example. The *Printer_client_view* view represents the protection policy from the client point of view: it specifies that the view *Text_server_view2* must be associated with the text parameter when the *print* method is invoked. The *Text_server_view2* view only authorizes *read* operations on texts, because a client allows a printer to read the text but not to modify it.
Similarly, *Printer_server_view1* and *Printer_server_view2* represents the protection policies from the printer point of view. The printer protects itself with the *Printer_server_view2* when the caller is a client. This view prevents clients from calling the *init* method on printers because only administrators are allowed to reset the printer. On the other side, the printer uses the *Printer_server_view1* to protect itself when the caller is an administrator.

During the execution, the protection views are managed through *capabilities*. A capability is an object reference coupled with the identification of the views controlling its use. In order to be allowed to invoke a particular method on an object, an application must own a capability to that object with the required access rights.

Client protection

```
view Printer_client_view
implements Printer_itf {
  void init();
  void print(Text_itf text pass
                        Text_server_view2);
}

view Text_server_view1
implements Text_itf {
  String read();
  void write(String s);
}

view Text_server_view2
implements Text_itf {
  String read();
  void not write(String s);
}
```

Printer protection

```
view Printer_server_view1
implements Printer_itf {
  void init();
  void print(Text_itf text);
}

view Printer_server_view2
implements Printer_itf {
  void not init();
  void print(Text_itf text);
}

view Text_client_view
implements Text_itf {
  String read();
  void write(String s);
}
```

Fig. 6. The Printer views

More precisely, a capability identifies two protection views : the client view and the server view. The client view defines the caller protection while the server view defines the callee protection, being given that the caller is the application owning the reference. With the Printer example, a client may own a capability on a printer identifying respectively *Printer_client_view* and *Printer_server_view1* for the client and server views.

A capability can be passed from an application to another application (through method invocation). The views associated with the capability may be redefined, according to the definition of the protection in the extended IDL.

In the printing example, when the *Client* calls the *print* method on the *Printer* object, a read-only capability (*Text_capacity*) on the *Text* object is passed from the *Client* to the *Printer* according to the *Printer_client_view* view (see step 1 in Figure 7). The client and server protection views associated with this capability are respectively *Text_client_view* and *Text_server_view2*. This capability allows the *Printer* object to read the content of the text (see step 2 in Figure 7).

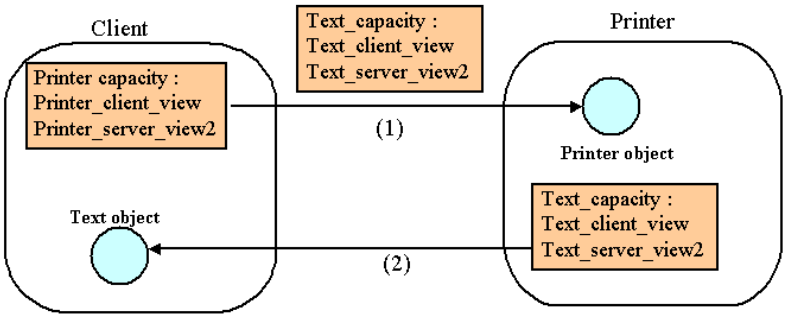


Fig. 7. Protection in the printing example

4.2 Indirection-based implementation of protection

The indirection-based implementation of the protection model presented above relies on indirection objects which raise an exception when an unauthorized method is called, while forwarding an invocation to the callee object in the other case.

This implementation uses both CIO and SIO indirection objects as described in section 0. CIO and SIO respectively implement a client protection view and a server protection view.

When a caller invokes a method on a callee, both the CIO and the SIO associated with the invoked reference control the parameters which are transferred according to the IDL views. An outgoing object reference may be replaced by a SIO reference. Conversely, an ingoing reference may be replaced by a CIO reference.

In the case of the printer application, when a *Client* object invokes a *Printer* object, it passes a reference to a *Text* object as a parameter of the *print* method. The CIO of the Client (of class *Printer_client_view*) replaces the reference of the Text object with a reference to a SIO object of class *Text_server_view* according to the definition of the IDL views. Figure 8 shows the code of the indirection objects used in the printer example.

4.3 Injection-based implementation of protection

In this section, we show how code injection has been used to implement the capability-based access control model without indirection objects. There is a major difference with the previous experiment on replicated objects, which makes the protection experiment complementary with the previous one: being given the class of a callee object, both the CIO and the SIO classes were unique with the replication aspect, while they can be multiple with the protection aspect.

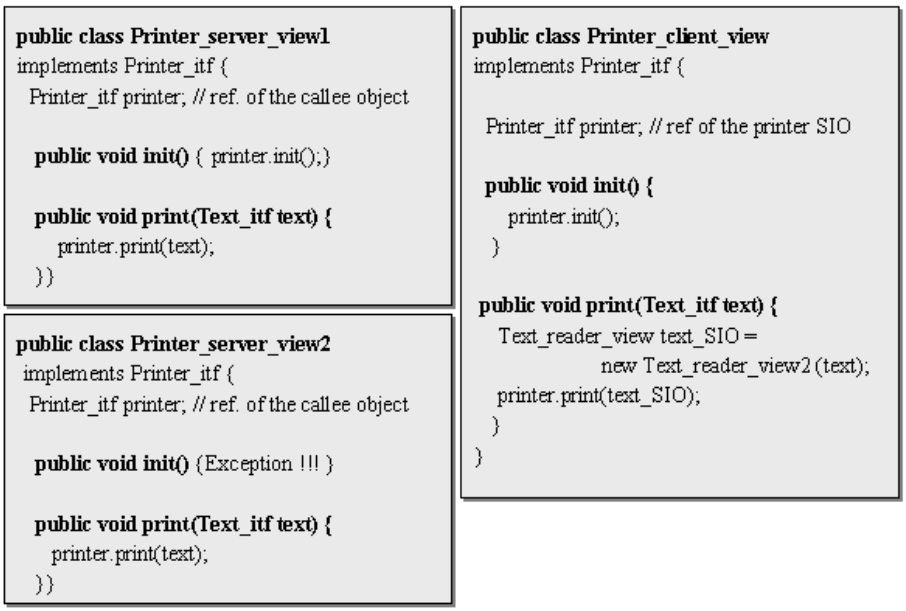


Fig. 8. Code of some indirection objects used in the printer example

As illustrated in Figure 9, being given the Printer class, there exists at least two SIO classes implementing two different views of the printer: *Printer_server_view1(*for administrators) and *Printer_server_view2 (*for clients), and two CIO classes*: Printer_client_view* and *Printer_adm_view* (although not shown in the Figure 9, a same caller object could use both CIO classes in the case it interacts with different printers with different roles).
The two main principles that have been applied to deal with the multiplicity of CIO/SIO classes are the following (the details of injection are given subsequently):

(1) to inject in any *callee* object (resp. *caller* object) the code of all the possible SIO classes (resp. CIO classes), being given that a CIO code has to be executed on a caller side while a SIO code has to be executed on a callee side.

(2) to associate to any object reference, the information that will allow to select, during a method invocation, both the right CIO and SIO code to execute. In Figure 9, this principle results in associating with the Printer reference used within the Client, the *indexes* that allow identifying CIO Printer_client_view and SIO Printer_server_view1 for the calls made from the Client to the Printer. *These indexes are considered as data strongly coupled with object references*[4].

[4] In the case of replication, both the CIO and SIO were unique for a given couple of interacting objects. Thus, there was no need for using CIO/SIO indexes.

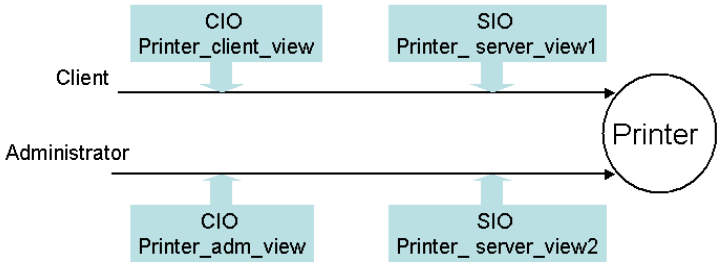


Fig. 9. Basic architecture for protection management

Injection of CIO into a caller object

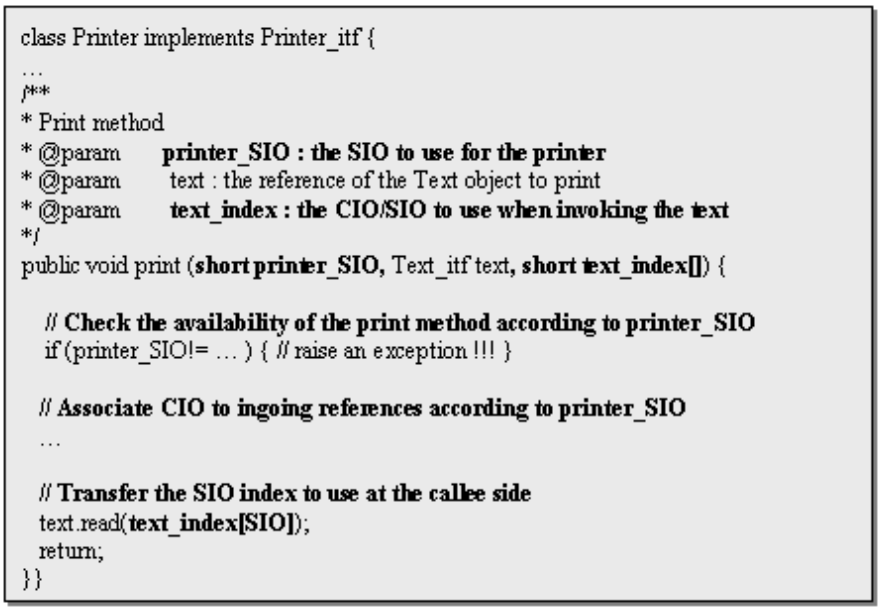
As for the replication aspect, the CIO mainly implements the methods defined in the interface of the callee object. Within a caller object, the injection process has to expand the invocations of these methods by their corresponding CIO code. Because multiple CIO can be injected within a caller, the selection of the right CIO code to execute is performed by injecting a *switch* statement applied on the indexes that are associated with the reference of the callee object.

Injection of SIO into a callee object

As for CIO, injecting several SIO within a callee object requires the indexing mechanism which allows to select the right code to execute on a given method invocation. During a particular method invocation, the index of the SIO code to execute should be received as a parameter of the method invocation. In the case of the printer example (Figure 10), the parameter *printer_SIO* has been added to the interface of the *print* method. According to the value of this parameter, the printer checks that the access rights grant access to the invoked *print* method.

Data strongly coupled with object references

In the case of the indirection-based implementation of protection, an object reference identifies a CIO object, which itself contains a reference to a SIO object through which the invocations should cross. CIO as well as SIO objects do not contain additional data. Thus the data strongly coupled with an object reference are limited to the indexes of both the CIO and SIO code. The management of these data takes into account the following kinds of operations on references: invocation, assignment, transfer as parameter and comparison.



```
class Printer implements Printer_itf {
...
/**
* Print method
* @param      printer_SIO : the SIO to use for the printer
* @param       text : the reference of the Text object to print
* @param       text_index : the CIO/SIO to use when invoking the text
*/
public void print (short printer_SIO, Text_itf text, short text_index[]) {

   // Check the availability of the print method according to printer_SIO
   if (printer_SIO!= ... ) { // raise an exception !!! }

  // Associate CIO to ingoing references according to printer_SIO
   ...

  // Transfer the SIO index to use at the callee side
   text.read(text_index[SIO]);
   return;
}}
```

Fig. 10. Code of the server after protection injection

Concerning object invocation, the data strongly coupled with an object reference may be divided into two parts : the data used at caller side and the data used at callee side[5]. In the present case, the caller-side data are composed of the CIO index, and the callee-side data of the SIO index. During an invocation, the SIO index has indeed to be known at the callee side in order to select the right SIO code to execute. To keep the semantic of an indirection-based implementation of protection, the callee-side data have to be transferred to the callee each time the callee is invoked. Thus, for any invocation on a protected object, code injection should add as a new parameter the index of the SIO to use at the callee side (either index of SIO printer_server_view1 or of SIO printer_server_view2 in Figure 11). The injection process modifies the signatures of the business methods of the application objects to take into account this additional parameter.

About reference assignment, each time a reference is assigned in the application's code, code should be injected to assign accordingly the data coupled with the references (CIO/SIO indexes). In the case of the printer reference contained in the *Client* class as shown in Figure 10, any assignment of *printer* should also raise the assignment of the *printer_index* variable (which include both CIO and SIO indexes).

[5] This distinction was not introduced in the case of the replication aspect, because the data strongly coupled with object references were only composed of data used at callee side.

In the same way, each time a reference is transferred as a method parameter or as a result, the CIO/SIO indexes associated with the reference should also be transferred. This is once again performed by modifying the signature of the business methods of the application objects. In the case of the *Client* class, the signature of the business method *print* has changed in order to transfer the indexes *(text_index)* associated with the reference *text*.

Finally, concerning reference comparison, the protection programming model only authorises indirect comparison of object references, through calls to an *equals()* method provided by CIO objects. The injection process operates as for any other CIO method, by expanding the method's code within the caller object.

```
public class Client {

public static void main(String args[]) {

  Printer_itf printer;
 // index of the CIO/SIO views to use when invoking the printer
  short printer_index[2];

  Text_itf text;
  // index of the CIO/SIO views to use when invoking the text
  short text_index[2];

   …

  // Invoke print method
  // the management of outgoing references depends on the
  // current CIO view associated to the printer
  switch (printer_index[CIO]) {
    …
    text_index[SIO] = Text_itf.Text_server_view2;
    …
  }
  printer.print(printer_index[SIO], text, text_index);
  }
}
```

Fig. 11. Code of the client after protection injection

This section has described a way to inject a protection aspect within the objects of an application. A way to manage the multiplicity of CIO/SIO classes is firstly to inject the whole collection of CIO code (resp. SIO code) within the corresponding applica-

tion objects[6], and secondly to use indexes to allow the selection of the right code to execute during a particular method invocation. We consider CIO/SIO indexes as data strongly coupled with object references. Moreover, in the protection example, these data include both a caller-side and a callee-side parts. The callee-side part has to be transferred to the callee when the reference to which it is coupled is invoked.

5 Lessons learned

The lessons learned from the injection experience which is described in this paper are firstly a list of injection patterns that should be implemented by a generic aspect-injector tool :

- Define additional data and methods within a class.
- Add instructions at the beginning and at the end of a particular method definition.
- Add instructions before and after a particular method invocation.
- Modify the signature of a method (add parameters).
- Enhance the components references with aspect-related data (detailed just after).

The highlighting of the last injection pattern, that we call the *extended reference injection pattern*, is the first main result of our experience. It allows to associate with a reference, the data that allows to manipulate the referenced component according to a given aspect. These aspect-specific data are associated with references in a way that provides the same semantic as if they were part of references.

Providing this pattern implies to inject the convenient code to ensure that for any extended reference transferred as parameter or as result of a method invocation, the whole extended reference content is transferred. Similarly, reference assignment and comparison should consider the whole extended reference content.

Concerning the invocations that are performed on an extended reference, we distinguish between two kinds of data composing the reference: the data used at caller side and those used at the callee side. An injector tool providing the extended reference pattern should inject the convenient code to ensure that the caller-side (resp. callee-side) data are actually present at the caller (resp. callee) side during an invocation.

The way to ensure the previous properties is to modify, by injection, the components interfaces. On a method invocation, the *data of the invoked reference, which are used at callee side,* have to be transferred to the callee as additional parameters. Moreover, for any reference transferred as parameter (or as result), the transfer of the whole extended reference content also involves additional method parameters.

[6] This is possible because both the size and the number of CIO/SIO classes associated with a given component is often small.

The following table illustrates the use of the extended reference injection pattern, by detailing the composition of an extended reference according to the composition of the indirection objects used for a given aspect. The index parts (CIO index and SIO index) are required as soon as there are multiple classes of CIO/SIO that can be associated with a given component. It should be noted that the SIO's data do not appear in this table because they can be injected within the callee object.

Extended reference content

Data used at caller side	**Data used at callee side**
CIO data	
CIO index	SIO index

In the case of the replication aspect, the extended reference was only composed of data used at caller side, because of the uniqueness of the SIO indirection object (only one SIO object can be associated with a duplicated object). This uniqueness has allowed to avoid the use of SIO index.

The other result of our experience is the fact that, for the considered aspects, their injection can be automated in the sense that it can be performed by a generic (aspect independent) injector tool. Being given the classes of both the CIO and SIO indirection objects managing a given aspect, an injector tool should be able to automatically determine the composition of the extended reference, as shown by the previous table. The injection process may then be entirely driven by the classes of the indirection objects and the extended reference composition.

6 Performance evaluation

This section provides the performance evaluation results in the case of a basic method invocation scheme illustrated in Figure 12.

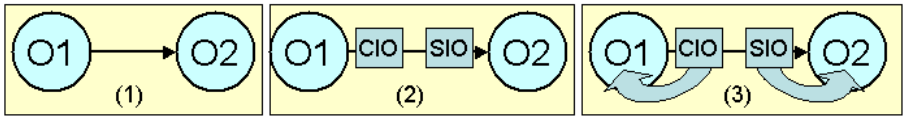


Fig. 12. Basic schemes for performance evaluation

In this scheme, object *o1* invokes a method *m()* on object *o2*. We consider the case where method *m()* does not take any parameter and the case where it takes a reference to another object *o3* as parameter. The code of method *m()* is empty. The measurements have been done under three conditions:

- on Java (JDK1.3) without integration of any aspect (case 1 in Figure 12).
- with the aspects (replication[7] and protection[8]) implemented with indirection objects (case 2 in Figure 12).
- with the aspects implemented with code injection (case 3 in Figure 12).

The following table presents the resulting performance figures using a 1Gh Pentium processor with 256 Mo of RAM. These results are given for 100 millions of iterations over the method call. We detail each case in the following.

	Protection		*Replication*		Java
	Indir.	Inject.	Indir.	Inject.	
Method m()	6458 ms	3354 ms - 48%	13889 ms	7591 ms -45%	1552 ms
Method m(o3)	14400 ms	3565 ms -75%	16022 ms	8453 ms -47%	1713 ms

Protection

In the case of a single method call with no parameter, the speedup is 48%. This speedup is explained because two indirection calls are avoided. In the case of a method call with a reference parameter, the speedup is 75%. In the version based on indirection objects, when a protected object reference (to *o3*) is transmitted as parameter, the SIO associated to the transmitted reference has to be instantiated. In the injection version, the indirection objects do not have to be instantiated since the protection code is embedded in the caller and callee objects (however we have to pass new parameters to implement the capability transfer).

Replication

In the case of a single method call with no parameter, the speedup is near 45%. Like for the capability experimentation, this speedup is explained because we avoid two indirection calls. In the case of a method call with a reference parameter, the speedup is about the same (47%). The contrast with the protection aspect is explained because with the replication aspect, the indirection objects are shared by the application components (only one CIO is created per object duplica).

The implementation of the replication service has a higher cost than the implementation of the protection service, because the replication service requires costly synchronization operations.

[7] In case of the replication aspect, the measurements were performed after the callee object has been replicated on the local machine.

[8] In the case of the protection aspect, four views were used (two client views and two server views).

7 Related Work

In the domain of component-based middleware such as EBJ [13] or CCM [8], the integration of non-functional properties such as security, transactions or persistence generally relies on indirection objects which allow capturing the interactions between components, and therefore to reify invocations. In the context of AoP [9] [10], the focus has rather been on the definition of language support for programming aspects (e.g., AspectJ [6]). The integration often relies on extra method calls (on aspect objects) which are injected in the code of the applications before, after or at entry of business method calls. Some optimizations are performed to reduce the performance overhead. In particular, AspectJ inlines some calls to aspect methods, but this is performed for limited cases only (statically known final methods in particular). In the same way, the JIT uses techniques that, in some cases, expand the called code in the calling code (In-lining). As soon as an aspect involve either indirection objects containing data, or multiple indirection objects, such inlining techniques are no longer sufficient. Indeed, in these cases, injecting aspects imply to modify the signatures of the business methods of the application components, as synthesized in 5.
Our proposal should be considered as complementary with the solutions proposed either by tools such as Aspect/J, or environments such as EJB or CCM. The injection process that we propose is indeed applied on the classes of the indirection objects, *after* these have been generated by these tools or environment.

Another point that should be considered is the domain of reflexive environments, which instrument and reify applications' behaviour during execution [12]. Such environments may be used for managing aspects [7]. However, these environments usually use additional objects at runtime (meta-objects), leading to the same observation as the one we addressed in this paper.

From a technical point of view, many different projects have experimented with Java bytecode transformation tools such as Javassist [2] or BCEL [17], in order to inject additional code in applications' functional code. The Software Fault Isolation technique [15] injects binary code that verifies that a component does not address the memory region allocated to another component. Software fault isolation allows component confinement without having to manage components in separate address spaces, which would be costly due to address space boundaries crossings. Other projects uses binary transformation techniques with different objectives, considering applications resource control [1], distribution [14], and thread migration [11].

8 Conclusion and perspectives

Separation of aspects is promoted by several domains, such as component-based environments and AoP languages and runtime supports. Resulting software is easier to build, reuse and adapt. The main motivation of these environments is to provide programmers with the flexibility of integrating orthogonal aspects into their applications. But this flexibility may be obtained to the detriment of performance, as the

implementation of aspects through indirection objects incurs an overhead on applications.
In this paper, we experimented a complementary approach: we investigated the issue of combining the flexibility of separation of aspects with efficiency. In the proposed approach, rather than implementing aspects using indirection objects, code injection techniques are used in order to optimise the integration of aspects into applications' code. One main objective of our work was to let the aspect programming model stay unchanged. This implies that the injection process be applied to the code of aspects programmed as with the indirection-based implementation.

Relying on an experimentation performed with two main aspects (replication and protection), this paper has the following results.

1. Being given the code of an aspect as with the indirection-based implementation, it is it feasible to use a generic (aspect-independent) injector which automatically injects the aspect code within the application components.
2. Aspects, even complex, can be injected with a benefit on the execution performances.
3. The injection process uses a specific injection pattern that we called *extended reference*. It basically allows to enhance usual object references with strongly coupled data accompanying references all along their road.

There are two important issues that we plan to address in a future work. Firstly, the injection of multiple aspects within an application has to be studied. Secondly, the monitoring and the instrumentation of an aspect-injected application becomes complex, because the code of aspects is distributed all over the application's components rather than localized within indirection objects. The identification of the code specific to the management of aspects becomes more difficult and should be helped by the provision of suitable monitoring features.

Another issue to investigate is dynamic aspect integration [2][10]. Managing aspects in indirection objects provides a means to dynamically add/remove aspects. Injecting aspects in the business code of the application makes it more difficult to dynamically modify aspects, since it may modify the structure of the application objects. We are currently working on the possibilities of making the components of an application evolve dynamically, by capturing (serializing) their state and rebuilding (de-serializing) a new version of their state with the newly integrated aspects. Such flexibility could be provided by implementing primitives which capture (serialize) the state of the application's objects and rebuild (de-serialize) a new version of these objects with the new integrated aspects. This is also a perspective to our work.

References

[1] W. Binder, J. Hulaas, A. Villazón, R. Vidal. Portable Resource Control in Java: The J-SEAL2 Approach, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2001), October 2001.

[2] S. Chiba. Javassist - A Reflection-based Programming Wizard for Java, ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java, October 1998.

[3] D. Hagimont, J. Mossière, X. Rousset de Pina, F. Saunier. Hidden Software Capabilities, Sixteenth International Conference on Distributed Computing Systems (ICDCS), May 1996.

[4] D. Hagimont, D. Louvegnies. Javanaise: Distributed Shared Objects for Internet Cooperative Applications, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), September 1998.

[5] D. Hagimont, F. Boyer. A Configurable RMI Mechanism for Sharing Distributed Java Objects, IEEE Internet Computing, Volume 5, number 1, January 2001.

[6] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. Aspect-Oriented Programming. European Conference for Object-Oriented Programming (ECOOP '97), Jyväskylä, Finland, June 1997.

[7] M. O. Killijian, J. C. Ruiz, J. C. Fabre. Portable Serialization of CORBA Objects: a Reflective Approach. 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2002), Seattle, WA, USA, November 2002.

[8] Object Management Group, CORBA Components: Joint Revised Submission, OMG TC Document orbos/99-08, August 1999.

[9] R. Pawlak, L. Duchien, G. Florin. An automatic aspect weaver with a reflective programming language. 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), Saint-Malo, France, June 1999.

[10] A. Popovici, T. Gross, G. Alonso. Dynamic weaving for aspect-oriented programming. 1st Aspect Oriented Software Development (AOSD'02), Enshede, The Netherlands, April 2002.

[11] T. Sakamoto, T. Sekiguchi, A. Yonezawa, Bytecode Transformation for Portable Thread Migration in Java, International Symposium on Mobile Agents (MA'2000), September 2000.

[12] B. Smith Reflection and Semantics in a Procedural Language. Technical Rapport, Laboratory for Computer Science, Massachussets Institute of Technology, 1982.

[13] Sun Microsystems, Enterprise Java Beans Specifications, Version 2.0, 2001.

[14] M. Tatsubori, T. Sasaki, S. Chiba, K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. European Conference on Object-Oriented Programming (ECOOP'2001), Budapest, Hungary, June 2001.

[15] R. Wahbe, S. Lucco, T. Anderson, S. Graham, Efficient Software-Based Fault Isolation, 14th ACM Symposium on Operating System Principles (SOSP'93), pp. 203-216, December 1993.

[16] Charles Zhang, Hans-Arno. Jacobsen, Quantifying Aspects in Middleware Platforms, Conference on Aspect-oriented software development (AOSD'03), pp. 130-139, Boston, Massachusetts, March 2003.

[17] BCEL, 2002. http://bcel.sourceforge.net/