Removing indirection objects for non-functional properties

D. Hagimont[1], N. De Palma[2],
INRIA Rhône-Alpes, Sardes project[3]
655, Avenue de l'Europe
F-38334 Saint Ismier, France

Keyworks: aspects, components, middleware, Java

1 Introduction

Complex applications generally include many aspects. A general trend in software development is to separate, as long as possible, these different aspects in order to improve the quality of the software which is easier to maintain [9]. The past few years have seen the emergence of component-based programming which promotes the separation of aspects. Component-based programming aims at improving code evolution and reuse by enabling the configuration of complex component-based architectures and the association of different aspects with the components which compose this software architecture.

Classical component models such as the Corba Component Model (CCM [10]) define the notion of *ports* to exhibit components' provisions and requirements. These ports correspond to the entry and exit points of the components and they are generally implemented with indirection objects in the middleware systems which provide a runtime support for components. Indirection objects are then used for associating non-functional aspects with components as they allow capturing all the interactions of one component with other components.

However, indirection objects generate a performance overhead at execution time. This overhead is negligible for remote components interactions, but it becomes significant for local interactions. In this latter case, it is not acceptable to pay such a cost, even for separating aspects.

This paper proposes a solution based on code injection to avoid the use of indirection objects to manage non-functional properties. We have experienced our approach with two non-functional properties: replication and protection. These experiments have been conducted in the Java environment. We propose to use a bytecode transformation tool (such as *Javassist* [4][12] or BCEL [1]) to inject in the application code, the code of the aspect which is usually integrated in indirection objects. The non-functional code is injected in the application at the bytecode level and it can be injected at compile time or even at runtime. We report some preliminary measurements which show the validity of the approach.

The rest of the paper is organized as follows: Section 2 presents the approach in general terms. Sections 3 and 4 respectively describe our experiments with the replication and protection aspects. Section 5 presents a preliminary performance evaluation. Section 6 discusses the related work and we conclude in Section 7.

2 Injecting non-functional code

In a component-based middleware, an interaction between two components generally goes through two indirection objects, the first one being the exit port of the invoking component and the second the entry port of the invoked component. In the rest of the paper, we will refer to these indirection objects using respectively the well-known terms *stub* (exit port) and *skeleton* (entry port).

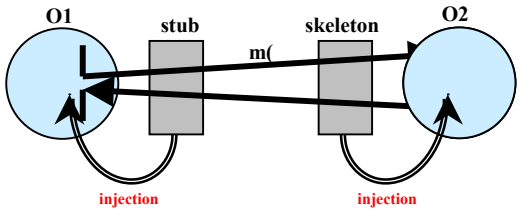

Figure 1: non-functional code injection

Stubs and skeletons capture interactions between components and allow the association of non-functional treatments with the invoking and invoked components. These treatments may consist in controlling the binding to a local component (as in our replication scheme - Section 3) or implementing access

[1] INRIA
[2] FranceTelecom R&D
[3] The work presented in this paper is supported by the *Réseau National des Technologies logicielles*.

control checks (as in our capability-based protection scheme - Section 4).

The general idea that we apply to the two examples described below, is to inject the non-functional code (which is usually integrated in indirection objects) within the functional code of the application (Figure 1). This code injection can be performed at compile time or at load time.

In order to present this approach in the rest of the paper, we will describe the transformed code in Java, even if the tools we are using perform code injection at the level of Java bytecode.

3 Replicated shared objects

In a recent experiment, we implemented a service (called *Javanaise* [8]) for the management of replicated shared Java objects. This service provides the abstraction of a distributed shared object space.

Even if Javanaise actually implements component replication, we will describe it as if it was implemented at the level of individual Java objects.

3.1 Non-functional aspect

In Javanaise, the objects managed by the applications are transparently replicated on the client nodes, so that the application developer can program as in a centralized setting. The service ensures the consistency of the replicated objects. Objects are brought on demand on the requesting nodes and are cached until invalidated by the coherence protocol. Thus, managing object replicas requires mechanisms for faulting on objects, invalidating and updating objects in order to ensure consistency. To handle object binding and consistency, applications may have to interact with a Javanaise server which allows object replicas to synchronize (i.e. to update/invalidate objects state). In the version we have used for this experiment, the service implements a multiple-readers/single-writer entry consistency protocol [2]. The programmer only has to specify, using an extended IDL (Interface Description Language), the lock which has to be taken (read/write) before entering each method.

In the following, we briefly describe the initial implementation of Javanaise using indirection objects (as described in [7]), and we then depict our implementation based on code injection.

3.2 Indirection-based implementation

A prototype of this service has been implemented on top of Java and relies on indirection objects (stubs and skeletons) to implement object references. For each object reference, a stub and a skeleton are transparently inserted between the referenced object and the object which contains the reference. Figure 2 illustrates the invocation of a method *m()* from the caller object *o1* on the callee object *o2*.

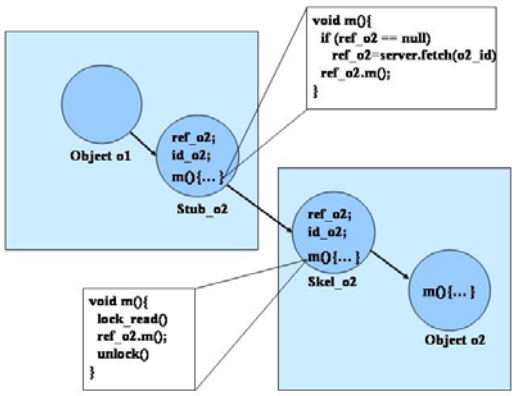

Figure 2: implementation of Javanaise based on indirection objects

The roles of the stub and the skeleton are respectively:

- *Stub*: the stub object (*Stub_o2*) is used to manage dynamic binding of references to objects that may be brought dynamically from remote nodes. When the Java reference in the stub object is null (*ref_o2*), a copy of the referenced shared object is fetched, either locally if the object is already cached or remotely from the Javanaise server by using a unique identifier associated with the object (*id_o2*). There is one stub object per reference pointing to object *o2*.
- *Skeleton*: The skeleton object (*Skel_o2*) is used to manage invalidates and updates of the shared object (*o2*) according to a consistency protocol. With the entry consistency protocol, the *lock_read/lock_write* procedures in the skeleton ensure that a lock and a consistent copy of the referenced object (*o2*) are present on the local node; they may fetch them from the server and update the Java reference (*ref_o2*) in the skeleton (if a new copy of the object is loaded).

The stub is copied with the object which includes the reference (*o1*). The reference *ref_o2* in the stub is set to null when the stub is copied on a machine (it is a transient Java reference). The skeleton is copied with the referenced object (*o2*) the first time it is loaded on a machine. The reference *ref_o2* in the skeleton may be null subsequently to invalidations by the consistency protocol. An invalidation in the skeleton avoids having to invalidate all the stubs which may reference the object. The full explanation of this protocol can be found on [7].

3.3 Injection-based implementation

In this section we show how code injection can be used to implement this replication service as a non-functional property and without requiring indirection (stub/skeleton) objects.

Injecting the stub

Injecting the stub in *o1* requires:

- For each reference variable (field or local variable) pointing to a shared object, we add the declaration of a new variable which contains the identifier of the shared object as was in the

stub (*id_o2*). This new variable has to be assigned when the reference variable is assigned (we inject the code which manages these assignments). Therefore, the reference to *o2* in *o1* is now a straight reference to object *o2* (since the skeleton is also merged within *o2*).

- To manage the dynamic binding of the reference to *o2* (as done by the stub) in the code of *o1*, we inject the code which checks the binding. If the reference is null, a binding primitive is invoked which communicates with the Javanaise server.
- When a reference to *o2* is transmitted (for example as parameter of a method), the identifier of *o2* is also transmitted (we add the identifier parameter in the signature of the method).

Here is the original code of the object o1:

```
public class O1 {
  Itfo2 o2=null;
 public void x() {
    o2.m();
}}
```

The resulting translated code is the following:

```
public class O1 {
  transient Itfo2 o2=null;
  Id id_o2; // the id of the o2 object

 public void x() {
    // managing the dynamic binding
    if (o2 == null)
       o2 = Server.fetch(id_o2);
    o2.m();
}
```

Injecting the skeleton

Injecting the skeleton code in o2 is straightforward. First, we have to add a new field called *id_o2* which contains the identifier of *o2* as held in the skeleton. We also inject the code of the *lock_read/lock_write/unlock* procedures which maintain consistency. These procedures manage a *lock* variable which tells whether a lock is cached on the local machine. When a lock is cached, no communication with the server is required and the executed code is a simple (synchronized) check of the lock variable, without any indirection.

If a lock in the required mode is not cached, it is fetched from the Javanaise server (the most recent state of the shared object is also fetched).

```
public class O2 {
  public int id_o2;
  private short lock;
  public void m() {
    // code of the lock_read method
    // code of the method m()
   // code of the unlock method
  }
}
```

In the next section, we show how this code injection based approach can also be applied to another non-functional property related to access control.

4 Capability-Based Protection

The purpose of protection is to control the actions of entities called agents, able to invoke operations on entities called objects. We assume that an object is defined by the operations that access it. The protection mechanisms must control, for each agent in the system, the objects it can access and the operations it can invoke on them. In a previous work, we have proposed a protection model based on *hidden software capabilities*[5]. A *capability* is a reference that identifies an object and contains access rights that define the allowed operations on that object. In order to access an object, an agent must own a capability on that object. The capability can thus be used to access the object, but can also be copied and passed to another agent, providing it with access rights on that object. Therefore, software capabilities allow access rights to be dynamically exchanged between mutually suspicious interacting agents.

In this paper, we can consider that the mutually suspicious interacting agents are interacting components or objects.

4.1 Non-functional aspect

In our hidden software capabilities model, protection definition is completely disjoined from the application code and described in an extended Interface Definition Language (IDL). Capabilities are invisible to the programmer, in the sense that it is possible to define the protection policy of an application independently from the application code.

The extended IDL allows defining views which are restrictions of Java interfaces. A view specifies the set of authorized methods. A capability is materialized by a Java reference which can be used by the programmer, and a view which is invisible to the programmer but specifies the access rights associated with the Java reference. In a view, when a method includes a reference parameter, a view may be associated with the reference parameter, specifying that a capability must be passed with the parameter. Therefore, a view describes how capabilities may be exchanged as parameters.

Two interacting agents both define their protection policy using views.

Let us consider the case of a printing server providing a *Printer_itf* interface that allows a client to print out a file. The printer interface is the following:

```
interface Printer_itf {
 public void init();
 public void print (Text_itf text);
}
```

Since the printing of the file executes asynchronously, the client provides a callback (the text object) that allows the printer to retrieve the text to print. The interface of the text object is the following:

```
interface Text_itf {
  public String read();
  public void write(String s);
}
```

In this example, the clients and the print server are mutually suspicious and they both specify their protection views. In the server view, the *init()* method can be executed by the printer administrator but this privilege is not granted to the clients. In the client view, when the *print()* method is invoked, a *reader* view is passed with the *text* parameter (a view which only grants access to the *read()* method).

4.2 Indirection-based implementation

A first implementation of this non-functional protection scheme has been developed using stubs and skeletons that encapsulate capability specific code.

A stub implements the protection policy of the caller (client) and a skeleton implements the protection policy of the callee (server):

- A skeleton prevents invocation of denied methods. It inserts stubs for onward parameters (imported capabilities) and skeletons for backward parameters (exported capabilities).
- A stub inserts skeletons for onward parameters (exported capabilities) and stubs for backward parameters (imported capabilities).

On our example in Figure 3, a reference from the client to the server includes two indirections through a stub object (*Printer_stub*) and a skeleton object (*Printer_skel*). The skeleton is responsible for preventing the invocation of the *init()* method. The stub must guarantee that only a reference to the text with the *reader* view is passed as parameter of the *print()* method. Therefore, the stub creates (and inserts) a skeleton object (*Text_skel*) which denies invocation of the *write()* method.

In Figure 3, *Printer_skel* would have inserted a *Text_stub* object if a method of *Text_itf* has had a reference parameter with an associated protection view.

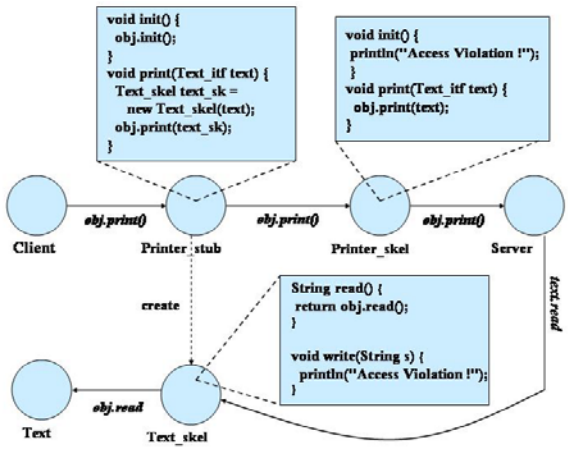

Figure 3: implementation of protection based on indirection objects

A detailed description of this protocol can be found in [6].

4.3 Injection-based implementation

In this section we show how code injection can be used to implement this protection scheme as a non-functional property and without requiring indirection (stub/skeleton) objects.

The protection code injection proceeds as follows:

- For each reference variable (field or local variable) pointing to a protected object, we have to inject the declaration of two new variables which represent the two views associated with the capability: the view specified by the object owner (the callee view) and the view specified by the owner of the capability (the caller view). These two variables are view identifiers (integers). They are assigned when the reference variable is assigned (we inject the code that does it).
- For each protected method invocation using a reference, we add the callee view associated with this reference, as parameter of the method. We inject at the beginning of the method the code which check whether this invocation is authorized in the definition of this callee view.
- When a reference to a protected object is passed as a method parameter, we inject the code which will initialize the caller view and callee view variables on the parameter receiver side. This initialization depends on the definitions of the view (caller and callee) associated with the capability used for the method invocation.

In the case of our printer example, when a client receives a reference to the printer (from a name server[4]), it also receives the capability caller and callee views. The method invocation to the printer takes two additional parameters, the callee view of the printer which allows checking access right on the printer side, and the callee view for the text parameter which specifies the access rights that are granted to the printer on the text (this callee view to pass with the text is defined in the view identified by pr_caller).

```
public class Client {
  public static void main(String args[]) {
    Printer_itf pr; // ref to the printer
    short pr_caller; // printer caller view
    short pr_callee; // printer callee view
    …
    Text text = new Text();
    …
    // pr_callee : passed for checking
    //    the capability on the callee side
    // view_reader_id : the view passed with
    //   the text, depends on the (local)
    //   definition of pr_caller
    switch (pr_caller) {
      …
      pr.print(pr_callee, text, view_reader_id);
    }
  }}
```

[4] The name server is extended accordingly.

On the server side, the printer checks that the access rights, associated with the view received as parameter, grant access to the method (for *init()* and *print()*).

In the *print()* method, the injected code initializes the view variables associated with the text received parameter. The callee view is received as parameter. The caller view for the text is defined in the callee view for the printer (depending on *pr_callee*).

```
class Server implements Printer_itf {

  public void init(short pr_callee) {
    // checking the capability … based on
    // the (local) definition of pr_callee
    if (pr_callee != … ) {
      println("Access Violation !!!!!  ");
      return;
    }
   …
  }
  public void print (short pr_callee, Text_itf
text, int text_view) {
     short text_caller; // text caller view
     short text_callee; // text callee view

    // checking the capability … based on
    // the (local) definition of pr_callee
    if (pr_callee != … ) {
      println("Access Violation !!!!!  ");
      return;
    }
    // initialize the views for the text
    text_callee = text_view;
    switch (pr_callee) {
       …
      text_caller = …
    }

    text.read(text_callee);
    return;
  }
}
```

5 Performance evaluation

In this section, we provide some preliminary measurements of the performance improvements we may expect from code injection for managing the two non-functional properties described above.

The implementation of the bytecode translators associated with these two aspects is in progress. The evaluation presented in this section is based on hand-written code, i.e. the code injection was performed at the application source level, applying our transformation patterns by hand.

For this performance evaluation, we consider the basic scheme illustrated on Figure 4.

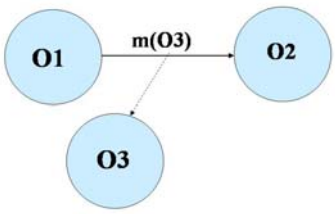

Figure 4: basic scheme for performance evaluation

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. These measurements have been done under three conditions:

- with the Javanaise service, implemented with indirection objects or with code injection[5],
- with the capability-based protection scheme, implemented with indirection objects or with code injection,
- on Java without integration if any non-functional property.

Here are the resulting performance figures using a 1Gh Pentium processor with 256 Mo of RAM. These results are given for 100000000 iterations over the method call.

	Capability		Javanaise		Straight invocation
	Object indir.	Injection	Object indir.	Injection	
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

These performance figures show a general speedup when using injection code technique. In the following we detail each case:

Capability

In the case of a single method call with no parameter, the speedup is 48%. This speedup is explained because we avoid two indirection calls.

In the case of a method call with a reference parameter, the speedup is 75%. In the version based on indirection objects, when we transmit a protected object reference (to *o3*) as parameter, *o2_stub* has to instantiate *o3_skel* to protect *o3*[6].

In the code injection version, we do not have to instantiate any stub since the protection code is embedded in the caller and callee objects (however we have to pass new parameters to implement the capability transfer). We give the cost of a straight

[5] The measurements were performed after all the involved objects have been replicated on the local machine.

[6] In a worth case, *o2_skel* could have to instantiate *o3_stub* to implement the protection policy associated with object *o3*.

method invocation in order to situate the cost of a non-functional property implemented with code injection, compared to the same application code without any non-functional property.

Javanaise

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%). This is explained by the fact that in the indirection objects version, passing a reference parameter implies passing a reference to a stub object which can be shared by both referencing objects. Therefore, no stub creation is required for passing a reference parameter.

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

6 Related work

We consider three ways to relate this approach to previous works.

From the point of view of non-functional aspect management, many projects addressed the issue of providing support for the integration of non-functional code within the functional code of applications. In two previous experiments, we addressed the issue of managing object access control and replication as non-functional aspects [5][8]. The proposed solutions were based on indirection objects and we studied in this paper the benefits of using code injection instead of object indirections. Other projects dealt with many different non-functional aspects (e.g. transactions or synchronization) and it would be interesting to study the application of the code injection approach to these aspects.

From a more technical point of view, many different projects experimented with Java bytecode transformation tools in order to weave additional code in applications' functional code. Were notably considered applications resource control [3] and thread migration [11]. We believe bytecode engineering tools such as Javassist [12] or BCEL [1] open many perspectives and will play an important role in future middleware systems which aim at allowing runtime adaptation of system services.

Finally, from the performance point of view, binary transformation is a technique which allows many optimizations. It was previously applied to manage software components sandboxing with the Software Fault Isolation technique [13] which 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. The approach presented in this paper shares many ideas with the software fault isolation proposal, especially the motivations and technical approach.

7 Conclusion

Component-based programming promotes the separation of aspects. Thus resulting software are easier to build, to reuse and to adapt. Middleware systems supporting component-based applications usually rely on indirection objects to separate functional and non-functional code. These indirection objects induce an overhead at execution time. This overhead is negligible in remote components interactions, but it becomes significant in local interactions. In this latter case we do not want to pay such a cost even for separating concerns. We proposed a solution based on code injection to avoid the use of indirection objects for managing non-functional aspects. We have shown the feasibility of our approach with two aspects: protection and replication. The preliminary measurements that we made show that the benefit on performance is significant.

These experiments have also allowed us to identify many similarities in the nature of the code injections we applied for implementing the two aspects presented in the paper. We believe that there exist few code transformation patterns which are required to perform non-functional code injection in a component-based system. Part of our future work is to clearly specify these patterns and could be to provide a high level language which would facilitate the definition of a specific code injector.

[1] BCEL, http://bcel.sourceforge.net/

[2] B. Bershad, M. Zekauskas, W. Sawdon, The Midway Distributed Shared Memory System, 38th IEEE Computer Society International Conference (COMPCON'93), February 1993.

[3] 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.

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

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

[6] D. Hagimont, L. Ismail, A Protection Scheme for Mobile Agents on Java, Third ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom), September 1997.

[7] 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.

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

[9] G. Kiczales, C. Lopes, Aspect-Oriented Programming with AspectJ, Technical Report, Xerox PARC, 1998.

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

[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] 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), LNCS 2072, Springer Verlag, 2001.

[13] 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.