On the Role of Interceptors and AOP in Adapting CORBA Applications*

Nélio Cacho, Thaís Batista, Fabrício Fernandes

Departamento de Informática e Matemática Aplicada (DIMAp)
Universidade Federal do Rio Grande do Norte (UFRN)
Campus Universitário – Lagoa Nova – 59.072-970 - Natal - RN

cacho@consiste.dimap.ufrn.br, thais@ufrnet.br,
fabricio@consiste.dimap.ufrn.br

Abstract. *In this paper we describe two meta-programming strategies that have been used to extend CORBA-based applications with minimal or no impact on existing application code: CORBA interceptors and aspect-oriented programming (AOP). We compare the benefits of using AOP with those of exploiting interceptors to extend CORBA-based applications. We present the main issues in which using AOP in this context is different from taking advantage of the existing CORBA interceptor mechanism. In order to illustrate our discussion we use a dynamic aspect-oriented language, AspectLua, and a meta-object protocol, LuaMOP, that supports dynamic weaving of CORBA components and aspects.*

1. Introduction

CORBA has been used as an underlying middleware platform for distributed application development for over a decade. The dynamic nature of current applications requires support for reconfiguration at runtime. Meta-programming mechanisms [Kiczales et al. 1991] have been widely used to support the runtime adaptability of distributed applications with minimal or no impact on existing application code [Wang et al. 2001].

CORBA portable interceptors [OMG 2004] support meta-programming by providing hooks in which developers can register their code. The ORB will automatically execute this code upon the occurrence of relevant events [Baldoni 2003]. The code can extend the behavior of both applications running over CORBA and also the CORBA platform itself. This mechanism provides meta-objects to be invoked at predefined interception points. As operation invocations pass through these meta-objects, application behavior can be adapted transparently by simply modifying the meta-objects.

Another mechanism that has gained popularity as an approach to support the adaptability of applications is aspect-oriented programming (AOP) [Elrad et al. 2001]. AOP defines a modularization mechanism that provides a high degree of *separation of concerns* in software development. The dynamic aspect-oriented approaches [Bouraqadi and Ledoux 2002][Sullivan 2001] use meta-programming as an underlying technology

* This work is partially supported by the Brazilian Agency CNPq project 552007/2002-1.

to support runtime aspect definition and dynamic weaving. Defining an aspect consists of specifying points of a component where a code should be inserted (join points), the moment that such code should be inserted (after, before or around the join point) as well as the code to be inserted (advice). A dynamic weaving process does the integration between components and aspects at runtime. Thus, it makes feasible to adapt an application dynamically.

In this work we compare the benefits of using these two approaches to adapt CORBA-based applications. In order to illustrate the use of dynamic AOP in the development of CORBA-based application we combine the following dynamic tools: (1) a dynamic aspect-based language, AspectLua [Fernandes and Batista 2004a][Fernandes and Batista 2004b] where AOP is built on top of the reflective features of an interpreted language, named Lua [Ierusalimsky et al. 1996]; (2) a meta-object protocol, LuaMOP [Fernandes et al. 2004], which provides operations to inspect the internal structure of the language and to modify its behavior in order to glue components and aspects; (3) a binding between Lua and CORBA, LuaOrb [Cerqueira et al. 1999].

Thus, we compare the support for application adaptability provided by the CORBA interceptors with those provided by the use of LuaMOP and AspectLua in a CORBA platform.

This paper is organized as follows. Section 2 presents CORBA interceptors highlighting their support for dynamic adaptation of applications. Section 3 comments about aspect-oriented programming. This section also presents a set of tools that use AOP to handle dynamic adaptation of CORBA-based applications as well as some examples. Section 4 focuses on the comparisons between the two approaches. Section 5 comments about some related works. Section 6 contains the final remarks.

2. CORBA Interceptors

CORBA *Portable Interceptors (PI)* are objects invoked by an ORB in the path of an invocation to adapt its behavior transparently [Wang et al. 2001]. It provides support for a developer to define some code that will be automatically executed upon the occurrence of relevant events such as request/reply communication.

CORBA specification [OMG 2004] defines two types of PI: *request interceptors* and *interoperable object reference interceptors (IOR interceptors)*. The goal of the *Request Interceptors (RI)* is to intercept the flow of a request/reply sequence through the ORB at specific points on clients and servers. Using RI it is possible to verify information about a request and to manipulate the service context propagated between clients and servers. An *IOR interceptor* inserts information into IORs in order to describe objects. Since this interceptor is not used for adaptability purposes, we will only focus on *Request Interceptors*.

Figure 1 shows interception points of a *Request Interceptor*. Those points are divided in two groups according to their location: *client-side* and *server-side*. At the *client-side* there are five interception points: *send_request, send_poll, receive_reply, receive_exception* and *receive_other***.** *send_request* is used to get information about a request. It also allows the modification of the service context before sending the request to the server. *send_poll* is used specifically in an asynchronous method invocation following the *polling model*. In this model the client invokes a method and receives a *Poller valuetype* from the server. *receive_reply* is used to intercept the reply of an

invocation. This point can get information about a reply sent by a server before it reaches the client. *receive_exception* is invoked upon the occurrence of an exception at a method invocation. Thus, it is possible to get information about an exception before it reaches the client. Finally, *receive_other* is used to get information when the result of an invocation is of a different type from the previously mentioned invocation type (reply and exception). Thus, this interceptor is invoked in case of *Poller* objects or replies regarding to a *LOCATION_FORWARD* exception.

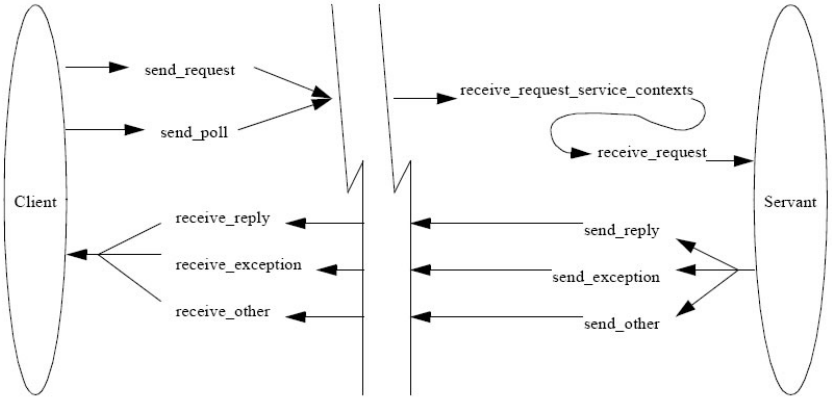

Figure 1: Request Interception Points from [OMG 2004]

At the server-side there are also five interception points as follows: *receive_request_service_contexts*, *receive_request*, *send_reply*, *send_exception* and *send_other*. The two first points act together in the same solicitation. *receive_request_service_contexts* is invoked before the *servant manager*. After that, *receive_request* is used. The difference between them is related to the data they provide. *receive_request_service_contexts* provides data about the *service context* while *receive_request* provides other data about the invocation such as the parameters. *send_reply*, *send_exception* and *send_other* are used to access and modify data related to the service context of a reply at three different moments: when a reply is sent to a client, in the occurrence of an exception during the remote invocation, and when any other thing happens, for example, a GIOP Reply with a *LOCATION_FORWARD*.

To illustrate the use of CORBA interceptors we have implemented a case study of a banking application that allows objects replication. In this case, the replication will be inserted in the application using interceptors. The application has been designed with two different components: a client and a server.

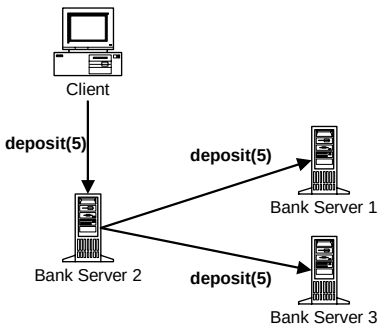

Figure 2: Replication Process

Figure 2 shows how the replication process works. A client invokes the *deposit* method in *BankServer2*. *BankServer2* processes the invocation and, through an

interceptor, forwards the request to *BankServer1* and *BankServer3*. This replication implies that all the servers must contain the same value. This approach is interesting for fault tolerance because if *BankServer2* is unavailable, the client-side interceptor can be invoked to forward the request to *BankServer1* or *BankServer3*.

In order to illustrate how this application has been implemented, we show the interceptor code of the client and the server as well as the code to initialize the server interceptor. Client and server implementations will not be detailed because the replication code is at the interceptors. Servers implement a *MyBank* interface with a *deposit* method.

To apply an interceptor at a client or server class it is not necessary to insert code inside these elements. It is necessary to register a class as an interceptor to invocations of a given class. For instance, to register `ClientInitializer` class as an interceptor to invocations of *demo.client* class the following command must be used: `java -Dorg. omg. PortableInterceptor. ORBInitializerClass. ForwardInit. demo. interceptors.ClientInitializer demo.client`

```
1 public class ServerInitializer extends org.omg.CORBA.LocalObject
implements ORBInitializer
2 {
3   public ServerInitializer() {   }
4   public void post_init(ORBInitInfo info)
5   {
6    NamingContextExt nc=NamingContextExtHelper.narrow(info.resolve_initial_references
                                                          ("NameService"));
7  info.add_server_request_interceptor(new ServerInterceptor(nc, (info.arguments())[0]);
8   }
9   public void pre_init(ORBInitInfo info) { }
}
```

Figure 3: Source code of the server interceptor

`ClientInitializer` class must implement *post_init* and *pre_init* methods of *ORBInitializer* interface. An example of the class used to register a server-side interceptor is illustrated in Figure 3.

```
1 public class ClientInterceptor
2    extends org.omg.CORBA.LocalObject
3    implements ClientRequestInterceptor{
4    private NamingContextExt namer = null;
5    public ClientInterceptor(NamingContextExt nc){namer = nc;}
6    public String name() {return " ClientInterceptor";}
7    public void destroy(){}
8    public void receive_exception(ClientRequestInfo ri)
       throws ForwardRequest{
9       MyBank bank = null;
10       BindingIteratorHolder bi = new BindingIteratorHolder();
11       BindingListHolder bl = new BindingListHolder();
12       BindingHolder b = new BindingHolder();
13       namer.list(0, bl, bi);
14       if (bi.value != null){
15        while ( bi.value.next_one(b) ){
16        bank=MyBank.narrow(namer.resolve(namer.to_name(b.value. binding_name[0].id)));
17          break;}
18        throw new ForwardRequest( bank );
19    }
20 public void send_request(ClientRequestInfo ri) throws ForwardRequest {
21       byte[] data = {(byte)(1)};
22       ServiceContext sc = new ServiceContext();
23       sc.context_id = 1;
24       sc.context_data = data;
25       ri.add_request_service_context(sc,false);
       }
. . .
}
```

Figure 4: Source code of the client interceptor

In this example *post_init* method is invoked at ORB initialization time: when *ORB.init(args)* method is invoked. *post_init* method first gets the reference for the naming service and then invokes the *add_server_request_interceptor* method to register with the ORB the *ServerInterceptor* interceptor. Such interceptor receives, as parameters, the naming server reference and a number (passed as command-line argument) that identifies the server.

The client-side of the fault tolerance process is done by the interceptor shown in Figure 4. In this interceptor the *receive_exception* method is invoked whenever a client method cannot finish an invocation. In this case study we have assumed that this problem occurs due to server unavailability. Then, the code from line 10 to 15 searches for available servers. Their references are obtained at line 16. Finally, the invocation is forwarded to the first available server by using the *FowardRequest* exception.

In this particular application, servers need to distinguish between client calls and server calls to the *deposit* method. This is achieved by adding a byte with value one to the context of the call. This byte, which is added by the *send_request* method, will be used to indicate to the server the receipt of a client invocation, and then, it must be replicated. If this byte is 0, the request is from a server and thus it does not need to be replicated.

```
1 public class ServerInterceptor
2    extends org.omg.CORBA.LocalObject
3    implements ServerRequestInterceptor
4 {
5    private NamingContextExt namer = null;
6    private String servernum;
7    private int flaginvocation = 0;
8    public ServerInterceptor(NamingContextExt nc, String servernum)
9          { namer = nc;  this.servernum = servernum;}
10    public String name(){ return "ServerInterceptor";}
11    public void destroy(){}
12
13    public void send_reply(ServerRequestInfo ri )
14
15       {
16          String operation = new String(ri.operation());
17          if ((operation.equals("deposit"))&&(flaginvocation == 1))
18           {
19                BindingIteratorHolder bi = new BindingIteratorHolder();
20                BindingListHolder bl = new BindingListHolder();
21                BindingHolder b = new BindingHolder();
22                namer.list(0, bl, bi);
23                if (bi.value != null)
24                   while ( bi.value.next_one(b) )
25                   {
26                     String cond = new String(b.value.binding_name[0].id);
27                     if (!(cond.equals("bankserver"+servernum)))
28                      {
29                       MyBank bank = MyBankHelper.narrow( namer.resolve(
30                                  namer.to_name( b.value.binding_name[0].id)));
31                              bank.deposit(5);
32                      }
33                   }
34             }
35        }
36    public void receive_request_service_contexts(ServerRequestInfo ri )
        throws org.omg.PortableInterceptor.ForwardRequest
37      {
38         ServiceContext sc = ri.get_request_service_context(1);
39         byte[]data = sc.context_data;
40         flaginvocation = data[0];
41      }
42}
                                        . . .
```

Figure 5: Source code of the server interceptor

Figure 5 illustrates the interceptor code applied to the server. Lines 36 to 41 show the implementation of *receive_request_service_contexts* that obtains the *service_context* from the invocation and sets the variable *flaginvocation* with the value of *context_data*. At line 13 there is the definition of the *send_reply* function. This function replicates the client invocation to others servers. Since an interceptor captures all invocations, it is necessary to define which methods will be replicated and to check if the replication has not applied yet. At line 17 we define that only invocations to the *deposit* method with *flaginvocation* set to one should be replicated. The code from line 19 to 27 gets all servers registered with the naming service and then searches for a server different from the one that received the invocation. When the condition defined at line 27 is satisfied, the server reference is obtained and the *deposit(5)* method is invoked to replicate the client invocation.

3. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) emphasizes the need to decouple concerns related to coding computational components from those related to non-functional aspects of an application. This decoupling is often supported by proposing the use of different languages for programming these two types of activities [Papadoupoulos and Arbab 1998]. For instance, Java programmers write the functional code in Java and the aspect code in a Java extension for aspect-oriented programming such as AspectJ [Elrad et al. 2001]. A special compiler does the integration between Java and AspectJ programs.

Although there is no consensus about the terminology of the elements that make part of AOP, in this work we use the terminology used in AspectJ because it is the most traditional aspect-oriented language. *Aspects* are the elements designed to encapsulate crosscutting concerns and take them out of the functional code. *Join Points* are well-defined points in the execution of a program. *Advices* define code that runs at join points. They can run at the moment a joint point is reached and before the method begins running (before), at the moment control returns (after) and during the execution of the joint point (around).

3.1 LuaMOP

LuaMOP is a meta-object protocol that offers a meta-object for any instance in the Lua environment. Lua is an interpreted extension language developed at PUC-Rio [Ierusalimsky et al. 1996]. It is dynamically typed: variables are not bound to types although each value has an associated type. Lua includes conventional features, such as syntax and control structures similar to those of Pascal, and also has several non-conventional features, such as the following: (1) Functions are *first-class* values, which means they can be stored in variables, passed as arguments to functions, and returned as results. Functions may return several values, eliminating the need for passing parameters by reference; (2) Lua *tables* implement associative arrays, and are the main data structuring facility in Lua. Tables are dynamically created objects and can be indexed by any value in the language, except nil. Lua stores all elements in tables generically as *key-value* pairs. Many common data structures, such as lists and sets, can be trivially implemented with tables. Tables may grow dynamically, as needed, and are garbage collected.

LuaMOP provides three categories of meta-objects: variables, functions and tables. Such categories are organized in a hierarchical way where *MetaObject* is a base

meta-class. Derived from this meta-class is the *Variables* meta-class. Then, derived from this meta-class there are two other meta-classes: *Table* and *Function* meta-classes.

To start using LuaMOP the method *getInstance(instance)* should be invoked. This method returns the meta-object corresponding to the object with name or reference described by the *instance* parameter. This meta-object is an instance of one such meta-classes: *Variable*, *Table* or *Function*. For each meta-class there are methods to describe and modify the behavior of a meta-object. LuaMOP methods will be described in this paper when used in AspectLua.

3.2 AspectLua

AspectLua explores the power of Lua tables and uses it for the definition of aspects, pointcuts and advices. These elements are defined using the Lua features and no special commands are needed. AspectLua defines an *Aspect* object that handles all aspects issues. This object defines a function to create a new aspect: the *new* function. After creating an aspect, it is necessary to define a Lua table that contains the aspect elements (name, pointcuts and advices). Figure 6 illustrates the generic code for aspect definition. The first parameter is the aspect name. The second one is the Lua table that defines the pointcut elements: its name, its designator and the functions or variables that must be intercepted. The designator defines the pointcut type. The extension supports the following type: *call* for function calls, *callone* for those aspects that need to be executed just once, *introduction* for introducing functions in tables (objects in Lua) and *get* and *set* applied upon variables. The list field defines functions or variables that will be intercepted. This list can use wildcards. For instance *Bank.** means that the aspect should be applied for all methods of the *Bank* class. Finally, the third parameter is a Lua table that defines the advice elements: the type (after, before, around) and the action to be taken when reaching the pointcut. In Figure 6, the *logfunc* function will act as an aspect to the *deposit* function. For each *deposit* function invocation, *logfunc* function will be invoked *before* it in order to print the deposit value.

```
function deposit(amount) . . . end
function logfunc(a) print('It was deposited: ' .. a) end
a = Aspect:new()
a:aspect( {name = 'logaspect'},
          {pointcutname = 'logdeposit', designator = 'call',  list = {'deposit'}},
          {type = 'before', action = logfunc} )
```

Figure 6: Generic code to aspect definition

A dynamic weaver is responsible for integrating aspect code and the application code. These two codes are weaved by the LuaMOP protocol. Then, the final code is executed.

```
function interceptFunction(list, adType, adAction)
for _,fn_name in ipairs(list) do
   local name = fn_name
   metaobject=LuaMOP:getInstance(name)
   if adType == 'before' then   metaobject:addPreMethod(adAction)
   elseif adType == 'after' then metaobject:addPosMethod(adAction)
   elseif adType == 'around' then metaobject:setAroundMethod(adAction) end
end
end
```

Figure 7: interceptFunction function

Aspect creation consists of defining aspects properties. Aspects can intercept functions, and also intercept set and get functions applied upon variables. It also allows the introduction of new methods in a program. Function interception specification is done through the definition of the function to be intercepted and the action (advice) that will take place when such function is invoked. This issue is supported by an AspectLua function (Figure 7) that uses the following LuaMOP functions: *getInstance*, *addPreMethod*, *addPosMethod* and *setAroundMethod* . *getInstance* function returns the metaobject correspondent to the *name* passed as a parameter. For a metaobject of type *function* the following methods can be invoked: *addPreMethod*, *addPosMethod* and *setAroundMethod*. These methods insert a function before, after or around the target method.

It is also possible to introduce new functions in a running program. Lua tables containing methods and properties represent functions. An aspect can be defined using the designator *introduction* to specify the function to be introduced in an object. The *introductionFunction* in AspectLua implements this facility getting the instance of the destination object using the LuaMOP *getInstance* function. After that, the *setField* function is called to introduce the function.

```
function checkRights() ... end
a:aspect( {name = 'secaspect'},
          {pointcutname = 'verifyRights', designator = 'call', list = {'deposit'}},
{type = 'before', action = checkRights } )
local order = Aspect:getOrder('deposit')
Aspect:setOrder('deposit', { order[2], order[1]})
```

Figure 8: Defining order to aspects invocations

To control the execution order of aspects in a given pointcut, AspectLua offers *getOrder* and *setOrder* functions. *getOrder* is used to get the list of aspects associated with a variable or function. It receives as a parameter the name of the variable or the function. It returns a list with the current aspect invocation order. *setOrder* is used to modify this order. This function receives the following parameters: variable or function name and the new execution order. In Figure 8 the *deposit* method has two aspects that will be executed before it. By default, the execution order is the order of aspect definition. Therefore, *logfunc* will be executed before *checkRights*. To modify this order, *setOrder* can be used with the following parameters: *deposit* and a table defining a different order. In order to get information about a variable or function, *getOrder* function is invoked receiving as a parameter its name.

3.3. Combining CORBA components and Aspects

As we have already mentioned, AspectLua uses the same language constructions of Lua. So, there is no need to change or use another interpreter. The Lua interpreter is used to execute the component part as well as the aspect code. This adds a great flexibility for using AOP via Lua.

A configuration file is used to configure the application. This file contains invocations to components and aspects that compose the application. It is submitted to the Lua interpreter that interprets the aspects definitions and does the association (pointcut and join points that must be intercepted) between what was defined and the component. AspectLua achieves this association using the LuaMOP functions.

Applications developed in Lua can use CORBA components due to the support of LuaOrb binding. This binding relies on the CORBA dynamic invocation interface (DII) to handle dynamic access to CORBA components in the same way as using any Lua object. LuaOrb also uses the CORBA dynamic skeleton interface (DSI) to handle dynamic installation of Lua objects as a CORBA server. When the interpreter finds a code related to CORBA objects, it invokes LuaOrb to handle this invocation. Then, LuaOrb calls the proper object in the CORBA platform. To invoke a CORBA method via LuaOrb it is necessary to create a proxy to the remote object. This is done by using the *createproxy* method. This method receives two parameters (a remote object reference and its interface) and returns a proxy to be used to invoke methods of such remote object.

Figure 9 illustrates the role of each component (Lua, LuaORB, AspectLua and CORBA) as well as the way they are combined to compose an application development environment.

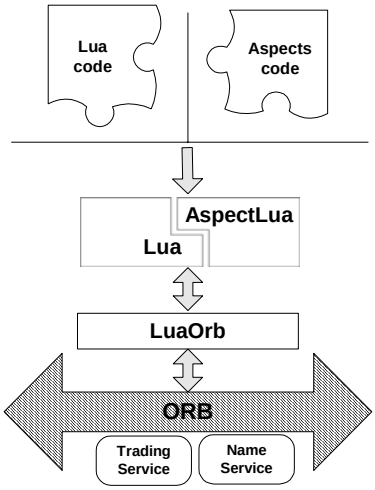

Figure 9: Architecture of AspectLua with LuaOrb

The adaptation capability provided by the integration of components and aspects will be illustrated in this section using the same case study discussed in section 2. This will allow us to compare interceptors and AOP support for adapting CORBA applications.

Figure 10 illustrates the *deposit* method invocation of the *bank* object. In this case, if the bank object becomes unavailable, the application will fail.

```
bank = luaorb.createproxy(readIOR("./account.ref"),"IDL:Account:1.0")
bank:deposit(5)
```

Figure 10: Client code

To overcome this problem, we insert an aspect in the *bank* object for searching other implementations instead of returning an error to the client.

Figure 11 shows the code of *trynewreferences* function. At line 2 the client creates a generic connector [Batista et al. 2000]. At line 4, it invokes, , the *deposit* method. The function of the generic connector is to find an implementation that offers the *deposit* method. At line 7, an aspect that will act on the *deposit* method of the *bank*

object is defined. It will execute the *trynewreferences* function instead of executing the *deposit* method.

```
1 function trynewreferences(self, ...)
2        newbank = Generic()
3        table.remove(arg, 1)
4        newbank:deposit(unpack(arg))
5  end

6 client = Aspect:new()
7 client:aspect({name = clienteIntercept'},
  {name = 'replicationMethods', designator = 'call', list = { 'bank.deposit'} },
  {type = 'around', action = trynewreferences })
```

Figure 11: Client code

Figure 12 shows *bank_impl* table that represents the implementation of the *IDL:Account:1.0* CORBA interface. The *deposit* method is described by the code at line 4. Finally, at line 6, the *bank_impl* implementation is registered at the searching mechanism.

```
1 bal = 0
2 bank_impl = {
3   deposit = function(self,amount)
4        bal = bal + amount
5   end}
6 source_server,id = ls_createservant(bank_impl, "IDL:Account:1.0")
```

Figure 12: Server code without replication – coreServer.lua

```
1 os.execute("idl  --feed-ir -ORBIfaceRepoAddr inet:localhost:15000 account_rep.idl")

2 dofile("AspectLua.lua")
3 function replication_deposit(self,amount)
4   local proxy_Discovery=luaorb.createproxy(readIOR("./search.ref"),
                          "IDL:CosDiscovering/SearchComponents:1.0")
5  proxy_Lookup = proxy_Discovery.getLookup
6  search_result = {}
7  search_result = proxy_Lookup:search("(operationame =='deposit_rep')
                                    and(offerId != "..id..")")

8  replication = Generic()
9  for i,refid in ipairs(search_result) do
10    replication:deposit_rep(amount)("offerId == "..refid)
11 end
12 end

13 function deposit_rep(self,amount)
14   bal = bal + amount
15 end

16 a = Aspect:new()
17 a:aspect({name = 'AccountDeposit'},
  {name = 'replicationMethods', designator = 'call', list = { 'bank_impl.deposit'} },
  {type ='after', action = replication_deposit})
18 a:aspect({name = 'AccountDeposit_rep'},
  {name = 'replicationMethods', designator = 'introduction', list = {
   'bank_impl.deposit_rep'} },
  {type ='after', action = deposit_rep})
```

Figure 13: Replication aspects – aspectServer.lua

In order to dynamically adapt the server to include replication we define two aspects. The first aspect defines that all invocations to the *deposit* method of *bank_impl*

should be followed by the execution of the *replication_deposit* method. This method performs information replication. Another aspect is necessary to introduce the implementation of the *deposit_rep* function. This function will be invoked to replicate the information. This method is necessary to avoid recursion among several servers. Since *deposit_rep* method does not exist in the original *Account* interface previously defined, the IDL with this function must be loaded. This is done in the first line of the code of Figure 13. The *replication_deposit* function represents the replication process. Lines 4 and 5 get the reference of the discovery service [Cacho et al. 2004]. At line 7 the *search* function is used to find out servers that have the *deposit_rep* operation and whose id is different from the current server id (expressed by the *offered* variable). Next, the *deposit_rep* method is invoked for all servers described at the *search_result* table.

After defining the aspect code, the next step is to specify the application configuration file. This file defines the elements that compose the application: the server code (coreServer.lua) and, optionally, the replication code. The user can choose to include replication or not (Figure 14). In this example, dynamic reconfiguration takes place via an aspect that can be inserted in the application according to user selection. The configuration file can also implement a more automatic approach that does not request user intervention.

```
dofile("coreServer.lua")
print("Would you like to use replication ?(y,n)")
answer = io.read()
if (answer == "y") then
  dofile("aspectServer.lua")
end
```

Figure 14: Application configuration file

4. Interceptors versus AOP

In the previous sections, two approaches for handling replications and fault tolerance have been presented. In both cases, it was not necessary to modify the application code. Both provide support for adapting CORBA applications in a non-intrusive way. In this section we present a comparison of these approaches, in particular their advantages and disadvantages.

Figure 1 illustrates interceptors points defined by OMG. As we illustrated in section 2, each point requires different information such as: method name, parameters, *service_context* and so on. This introduces a great complexity in using interceptors. In contrast, the aspect definition using the environment we propose in this paper is very simple.

Although OMG has opened the ORB a bit by allowing the interception of messages at certain defined places within an ORB [Wegdam 2000], an important drawback is that the interceptors points defined in the current specification [OMG 2004] are only implemented by few ORBs. For instance, Mico implements another sequence of interceptors points that does not match with the current specification. JacORB does not support access to arguments, results, and exception related to an invocation. Some ORBs have similar proprietary mechanisms (for example filters in Orbix and Visigenic) [Wedgam 2000]. In this way, portability, that could be an advantage of interceptors, is not achieved.

Another drawback is that using interceptors it is not possible to constraint objects and methods in which an interceptor will act. Besides, since interceptors are defined inside the ORB, every invocation mediated by the ORB is captured by interceptors. For instance, the interceptor described in Figure 4 is invoked to handle the remote functions called by the *receive_exception* method, such as: *list, next_one, to_name, resolve* and *deposit*. To avoid problems regarding to infinite recursions, the implementation constraints remote invocation by using the *ForwardRequest* exception.

Using AspectLua it is possible to insert aspects *before*, *after* or *around* functions executions or operations upon variables. The place where the aspect will act is defined using a pointcut definition such as *bank_impl.deposit* or using wildcards, such as *bank_impl.**. So, it provides more flexibility than interceptors.

Since it is possible to define a lot of aspects or interceptors for a function, sometimes it is necessary to define the precedence order among aspects or interceptors. To handle this feature, AspectLua offers the *setOrder* function. This function can be used to define the execution order of aspects for a certain method. On the other hand, interceptors do not address this issue. Then, using interceptors the execution order is unpredictable.

Besides, while separation of concerns is one of the main goals of AOP, it is not an issue in the interceptors proposal. Interceptor code is inserted into the ORB. AOP code is independent.

5. Related Work

There are other mechanisms that support adaptability in the context of CORBA, such as smart proxies and servant managers.

Smart proxies are application-defined stub implementations that transparently override the default stubs. The disadvantage of smart proxies is that, to take advantage of their support for modifying the behavior of an interface, it is necessary to implement smart proxy class and register it with the ORB. Besides, smart proxies only address client applications. In contrast, interceptors and AOP handle both client and server applications.

Servant Managers, provided by the CORBA POA specification [OMG 2004], allow server applications to register objects that activate servants on demand. Unlike interceptors and AOP, servant managers are tightly coupled with POAs and servant implementations [Wang 2001].

[Wang 2001] also discusses a comparison among meta-programming facilities to adapt applications. It focuses on smart proxies and interceptors. Considering that smart proxies act only at the client side, we choose to analyze approaches that handle both client and server adaptation.

There are some works [Pichler 2002, Zhang 2003] that address the use of AOP in the context of middleware platforms. Zhang [Zhang 2003] mentions that AOP is a promising way to handle complexity of middleware architecture because it addresses separation of concerns and avoids scattering phenomena in the code. In this work we also use AOP in middleware platform, but analyzing it as an adapting mechanism.

Baldoni [Baldoni 2003] analyzes a particular class of interceptors: CORBA request interceptors. The paper focuses on the request redirection and piggybacking that

are the main mechanisms provided by PIs. In this paper we focus on the role of interceptors and AOP in supporting application adaptation.

6. Final Remarks

In this paper we have discussed two meta-programming facilities to support adaptation of CORBA-based applications: interceptors and aspect-oriented programming. Using these mechanisms, it is possible to improve the adaptability of distributed applications with minimal or no impact on existing application code. We have chosen to compare interceptors with AOP for some reasons. First of all, interceptors are standardized mechanism available inside the ORB. Second, AOP is an approach that has gained attention as an important mechanism for handling adaptability.

We have presented the CORBA interceptors mechanism as well as the aspect-oriented programming concepts. In order to exploit AOP to dynamic adaptation, we have defined an AOP dynamic environment composed of the following dynamic tools: the Lua language, the LuaOrb binding, AspectLua and LuaMOP. We have applied the two approaches in a same case study in order to illustrate the power of each of them.

Although interceptors are useful facilities inside ORB, it presents some disadvantages when compared with dynamic AOP: they do not allow the definition of constraints regarding to objects or method affected by them; they capture every invocation at pre-defined ORB interception points regardless of what operation is invoked; interceptors execution order is unpredictable. The AOP environment discussed in this paper is more flexible than the interceptor approach and it overcomes these drawbacks of CORBA interceptors. Besides, it is easier to use because it is not necessary to know the internal details of the middleware.

While interceptors are tightly coupled with ORB, the AOP environment is independent and located on a high abstraction level. On one hand, this independence is an advantage because it avoids that developers deal with the burden of middleware internal mechanisms. On the other hand this independence can be seen as a disadvantage because it is not a standardized mechanism of the middleware platform. So, portability is not achieved. As previously mentioned, interceptors portability is not a reality yet because most middleware implementations do not offer all interceptors points proposed by the current CORBA specification.

7. References

Baldoni, R. and Marchetti, C. and Verde, L. (2003) CORBA request portable interceptors: analysis and applications. Concurrency and Computation: Practice and Experience, pp. 551-579, 2003.

Batista, T., Chavez, C. and Rodriguez, N. (2000) Conector Genérico: Um Mecanismo para Reconfiguração de Aplicações baseadas em Componentes e Ambientes de Software (IDEAS'2000), Cancun, México, April 2000.

Bouraqadi, N. and Ledoux, T. (2002) Aspect-Oriented Programming using Reflection. Technical Report. Ecole d'Ingénieurs Center de Recherche Mines de Douai. 2002.

Cacho, N., Batista T. and Elias, G. (2004) Um Serviço CORBA para Descoberta de Componentes. 18th Brazilian Symposium on Software Engineering (SBES'2004). Brasília, DF. 2004.

Cerqueira, R., Cassino, C. e Ierusalimschy R. (1999) Dynamic Component Gluing Across Different Componentware Systems. In: DOA'99 — International Symposium on Distributed Objects and Applications, Edinburgh, Scotland, 1999.

Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., and Ossher, H. (2001) Discussing Aspects of AOP. Communications of the ACM, Vol. 44, No. 10, pp 33-38, October 2001.

Fernandes, F. A. and Batista, T. (2004a) Dynamic Aspect-Oriented Programming: An Interpreted Approach. Proceedings of the 2004 Dynamic Aspects Workshop (DAW 04), Aspect Oriented Software Development Conference 2004 (AOSD'04), Lancaster, England, March 2004, 44-50.

Fernandes, F. and Batista T.(2004b) A Dynamic Approach to Combine Components and Aspects. 18th Brazilian Symposium on Software Engineering. Brasília, DF, Brazil. 2004.

Fernandes, F., Cacho, N. and Batista, T. (2004) LuaMOP – A Meta-object Protocol for Dynamic Weaving. First Brazilian Workshop on Aspect-Oriented Software Development (WASP'04), 18th Brazilian Symposium on Software Engineering (SBES'2004). Brasília, DF. 2004.

Ierusalimsky, R., Figueiredo, L. H., and Celes, W. (1996) Lua – an extensible extension language. Software: Practice and Experience, 26(6):635-652. 1996.

Kiczales, G., des Rivieres, J. and Bobrow, D. (1991) The Art of the Metaobject Protocol. MIT Press.

OMG (2004) Common Object Request Broker Architecture: Core Specification Technical Report Revision 3.0.3,

Papadoupoulos, G. and Arbab, F. (1998) Coordination Languages and Models, In: Advances in Computers. Academic Press.

Pichler, R. and Ostermann, K. (2002) On aspectualize component models. Software Practice and Experience.

Sullivan, G. T. (2001) Aspect-Oriented Programming using Reflection. Communications of the ACM, vol 44, n. 10, pages 10-14. 2001.

Wang, N., Parameswaran, K., Schmidt, D. and Othman, O. (2001) The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware. Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), San Antonio, Jan/Feb, 2001.

Zhang, C. and Jacobsen, H. (2003) Re-factoring Middleware Systems: A Case Study.