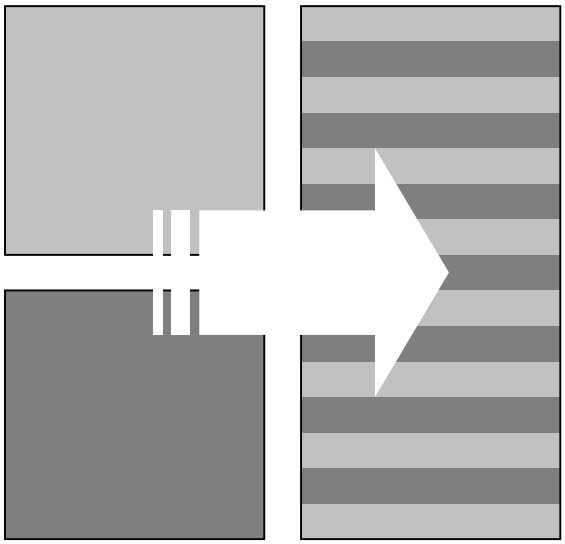
Towards a pragmatic composition model of Corba services based on AspectJ.

This report ends the year I spent in studying "Network and Distributed Systems" at the ESSI and working with the Rainbow Project's team (I3S).

DEA RSD
1999-2000



Supervisors: Mireille Blay & Anne-Marie Pinna
Author: Laurent Bussard

Rainbow Project
(**I3S – CNRS**)
Home page: http://www.essi.fr/~rainbow/

Laurent Bussard
I3S – CNRS UPRESA 6070 – Bât ESSI
930 Rte des Colles – BP 145
06903 Sophia-Antipolis Cedex, France

E-mail: bussard@essi.fr
Home page: http://www.essi.fr/~bussard/

Preface

This report ends the year I spent in studying *Network and Distributed Systems* at the ESSI and working with the Rainbow Project's team (I3S). Before writing it I have sent a paper [9] to the ECOOP's 2000 workshop on *Aspects and Dimensions of Concerns* and so have had the opportunity of presenting my last three months work. Moreover, we would like to reuse this report as a base pf another paper we are going to send to the conference *Reflection 2001*. It is the reason why this report, which ends my DEA, has not been written in French. Mr Philippe Nain and Mrs Mireille Blay have agreed with me about the uselessness of numerous translations.

Introduction

Nowadays the software are commonly distributed, they need transactions, synchronisation, security and numerous other high-level features. To let programmers deal with this increasing complexity, more and more powerful and abstract tools have been provided. Object Oriented Programming (OOP) seems to be the keystone of this evolution. Frameworks, libraries are used everywhere. Components, middlewares, and agents spread quickly. The Common Object Request Broker Architecture (Corba) is obviously the most used middleware. It offers a transparent access to remote objects and a set of powerful services allowing the development of large distributed applications. But the way services are defined by using the standard Object Oriented approach does not emphasise the reusability and simplicity of the code. It is the reason why letting collaborate specialists could be really difficult.

In fact these solutions often fail in capturing the abstract model of the application, limiting the adaptability and the reusability of this one. So, some works have been realised to deal with new models. In order to express micro architecture in abstract ways, design patterns are often described. The concept of meta-programming, that lets control the execution independently of the application description itself, seems yet to be an accepted approach (MOF, MetaUML, meta middlewares, Java reflect, interceptors, …). Our work presented in this report is just another little step in this direction.

The main reason why it is so difficult to understand, maintain and reuse the code, is the lack of separation of concerns. This is not due to clumsy programmers, but it is an inherent weakness of standard OOP. In fact, a lot of systems have properties that do not fit the object oriented model. Failure handling, persistence, concurrency and so on are aspects of the system that cut-across classes. By using standard object oriented programming those aspects are spread throughout the classes. For example, in order to use the transaction service, a programmer has to spread the application with lines of code related to this service (such as: "begin transaction", "rollback" or "restore initial state"). It becomes quickly a mess of mixed pieces of code and is almost impossible to spot the parts related to a given service. Moreover another adjacent problem is the combination of services that are not always independent.

After having more precisely introduced those limitations, the chapter number two will present some research results able to provide a better separation of concerns. The third chapter will focus on one of those possibilities: Aspect-Oriented Programming (AOP) and especially AspectJ that is an AO extension to Java. We will show how Corba Services can be viewed as aspects and the benefits of such an approach. The extensions necessary to deal with Corba IDL interfaces will be introduced too.

At last we will propose a way to compose aspects. It is a general problem that is independent of Corba services and is the core of our work. When two services are used together the programmer should have a way to define the expected behaviour. In case of transaction and event services composition, care should be given to avoid sending events related to an operation that could be cancelled by a rollback. It is the reason why we propose the definition of "*meta level composition aspects*" in order to choose how the composition has to be done. Moreover, an implementation of the minimal composition aspect compiler will be presented in order to validate those concepts. Finally the forth chapter will show a complete implementation of a simple example based on the new syntax.

Table of Contents

1 Motivations

The Common Object Request Broker Architecture (Corba) is more and more used [5], [9], [24], [28]. This ubiquitous middelware provides a transparent access to remote objects running on other platforms and that could have been implemented in other programming languages. The Corba standard defines a set of powerful services [2] and thus allows the development of large distributed applications. According to the OMG, *"The services are only as complicated as they need to be"*. In fact programmers spend a lot of time and effort in studying the needed services before being able to use them. It is obviously worse when they are not used to work with Corba.

In this chapter, we will show that programmers have to add numerous pieces of code in order to let an application benefit from one or more services. Those code pieces are spread on the application code so that it becomes quickly impossible to identify the parts corresponding to a given service. Moreover when two services are used together, the programmer has to think about composition while implementing the application. Finally we will explain that the Object Oriented approach is not sufficient to create easily reusable services and does not let several specialised programmers work together efficiently. In other words, it is difficult to write the core application without thinking about services and let a specialist add access to those services later.

This report presents a new viewpoint on the definition of services. Indeed, the code related to services is kept as entities instead of being spread in the application code. So that the reuse of services and the cooperation of programmers is increased. Our goal is the definition of an approach that lets encapsulate the services in order to create libraries of services. Moreover we want to define the entry points needed to allow the composition of those ones.

1.1 Overview of Corba

To be able to understand the problematic of Corba services composition, here is a short presentation of the Corba architecture and a list of the standardised services.

1.1.1 Corba core

The Common Object Request Broker Architecture (Corba) is an open distributed object computing infrastructure standardised by the Object Management Group (OMG). Corba deals automatically with many common network programming tasks such as object registration, location, and activation, request demultiplexing, framing and error-handling, parameter marshalling and demarshalling. Moreover Corba provides platform independence and language independence. Platform independence means that Corba objects can be used on any platform for which there is a Corba ORB implementation. Language independence means that Corba objects and clients can be implemented in just about any programming language. In other words, it means that a Java Client running on a Windows NT station can use transparently a service implemented in Smalltalk and installed on a Linux platform. Those features are not offered together by other middlewares such as DCOM, that is only usable on Windows platforms, or RMI, that can only be used from code written in Java.

A fundamental piece of Corba is the Object Request Broker (ORB). Its purpose is to facilitate communication between objects. It does so by providing some facilities, one of them is to locate a remote object, given an object reference. Another service provided by the ORB is the marshalling of parameters and returned values to and from remote methods invocations.

Another fundamental part of Corba architecture is the use of the Interface Definition Language (IDL), which specifies the interfaces between Corba objects. It ensures Corba's language independence because interfaces described in IDL can be mapped to any programming language.

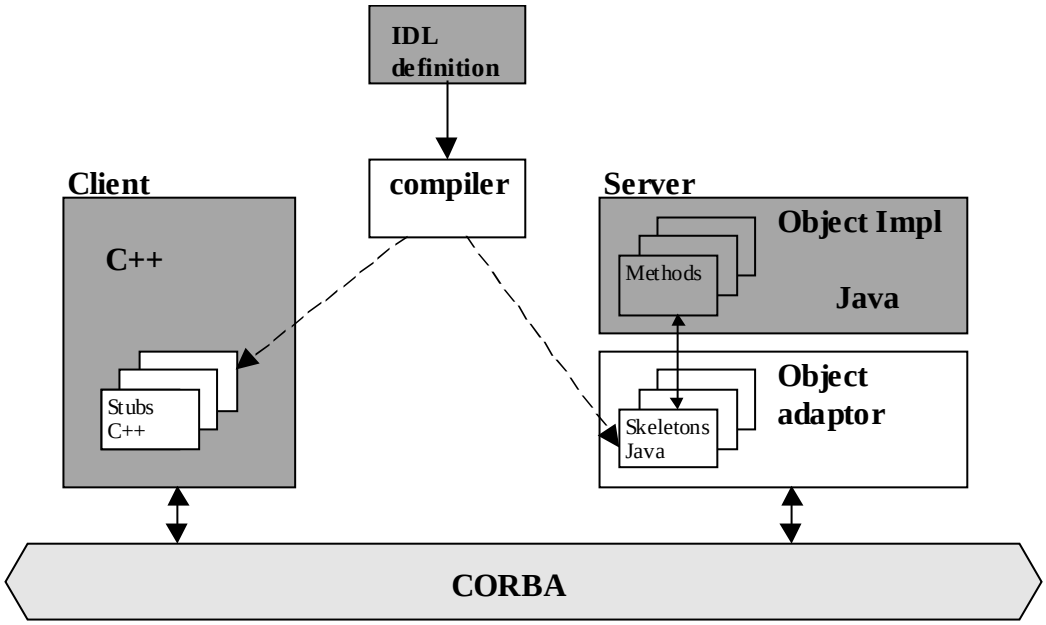


Figure 1 Principles of Corba applications

Corba defines the notion of object references to facilitate communication between objects. Those references are called Interoperable Object References (IOR). When a component of an application wants to access a Corba object, it first obtains an IOR for that object (for example, by consulting the Naming Service). Using the IOR, the component (client of that object) can invoke methods on the object (server). Corba ORBs usually communicate using the Internet Inter-ORB Protocol (IIOP). Other protocols for inter-ORB communication exists, but IIOP is fast becoming the most popular, first of all because of the popularity of TCP/IP.

In Corba, all communication between objects is done through object references. Thus visibility of objects is provided only through passing references to those objects. Objects are not passed by value. In fact it is possible since the release 2.3 and will be fully available in the upcoming Corba 3.0 release (expected for the end of 2000). However it is a deep philosophy change and is not really used yet. So we can say that remote objects in Corba remain remote.

Like the client/server architectures, Corba maintains the notion of client and server. In Corba, an object can act as both a client and a server. Essentially, a component is considered a server if it contains Corba objects whose services are accessible to other objects. Likewise, a component is considered a client if it access services from some other Corba objects. Of course, a component can simultaneously provide and use various services, and so can be considered a client or a server, depending on the scenario in question.

Corba IDL stubs and skeletons serve as the “glue” between the client and server applications, respectively, and the ORB. A stub is a small piece of code that allows a client component to access a server component. This piece of code is compiled with the client part of the application. Similarly, skeletons are pieces of code that have to be filled when the server is implemented. The stubs and skeletons are automatically generated according to the target programming languages when the IDL interfaces are compiled.

1.1.2 Standardised Corba Services

In addition to the Corba core that allows distributed objects to communicate with each other, numerous additional capabilities are provided. Corba Services and facilities offer both horizontal (generally useful) and vertical (designed for specific industries) services and facilities. These services are based on the basic ORB functionality. They deal with core system level functionality such as persistence and transactions, security, messaging and directory services. Here is the description of those services:

Naming - The mechanism by which an object can locate a target object by the target name. Every object has a unique reference IOR. The naming service maps an object name into a reference IOR.

Externalisation - Externalisation provides a standard way for getting data into and out of a component using a stream-like mechanism.

Persistence - Persistence provides a single interface to persistent data stores. This service may act as a front end to a variety of storage servers, such as Object Databases (ODBMSs), Relational Databases (RDBMSs), and simple files.

Events - Events allow components on the bus to dynamically register or de-register their interest in specific events. The service defines an intervening object called an event channel that collects events from their sources and distributes them among interested objects. In this way, the objects which trigger the events and the objects interested in the events do not need to know about each other.

Life Cycle - Life cycle defines operations for creating, deleting, moving, and copying components on the ORB.

Transaction - Transaction Service supports multiple transaction models, including the flat (mandatory in the specification) and nested (optional) models. The Object Transaction Service supports interoperability between different programming models. For instance, some users want to add object implementations to existing procedural applications and to augment object implementations with code that uses the procedural paradigm. To do so in a transaction environment requires the object and procedural code to share a single transaction. Network interoperability is also supported, since users need communication between different systems, including the ability to have one transaction service inter-operate with a cooperating transaction service using different ORBs.

Properties - Properties provide operations for dynamically associating named values (or properties) with any component.

Query - Query provides query operations for objects. It is a superset of SQL based on the upcoming SQL3 specification and the Object Database Management Group's (ODMG) Object Query Language (OQL).

Concurrency - Concurrency is used to handle situations where multiple client objects are attempting to use the same resource simultaneously. The concurrency service provides a lock manager that can obtain locks to resources in order to restrict their accessibility.

Relationships - Relationships provide a mechanism for creating dynamic associations (or links) between previously unrelated components. This service also provides a mechanism for traversing these links.

Collections - Collections provide a uniform way to create and manipulate common types of collections. Examples of common collections include groups, stacks, lists, arrays, trees, sets, and bags.

Time - Time is used by the ORB to maintain clock synchronisation across multiple machines. In the context of distributed systems, universal time is critical for correctly ordering events.

Security - Security deals with the security issues of components on an ORB. Issues include object authentication, permission to access resources, audit trails, and non-repudiation.

Change Management - Change management is used to track different component versions and their evolutionary history.

Trader - Trader provides a matchmaker service for objects: A new object will register the services it performs with the trader. Through registration, the object will give the trader, an object reference, so those clients will be able to connect to the service and invoke its operations. An object will also give the trader a description of the type of services offered (basically the function calls). Clients contact the trader to find out what services are available or to ask for a service by type. The trader will find the best match for the client based on the context of the requested service and the offers of the providers.

Licensing - Licensing provides a mechanism to measure a component use in order to be able to charge the customers. It supports charging per session, per node, per instance, and per site.

Own Corba services - In addition to those standardised Corba services everybody can create its own services and let use those ones world wide by delivering the corresponding IOR. The client will have to obtain the reference on the service and next to use it.

1.2 How the services affect the application structure

When a service is added to an application, the number of affected classes could be huge. In fact the Corba Services feature cut across several components.

In order to stay as clear as possible, let us take an example. We can define a simple bank application using the Corba Transaction Service. The bank client offers a method to make a transfer from an account to another one. In error case, the transaction ensures that the final result remains coherent by giving back the previous accounts states (rollback). If we compare the resulting code with the same application without transactions, it is obvious that the added code is spread on different classes and methods. Objects of both sides (client and server) are modified. Indeed the client has to start and stop the transaction and the server objects have to implement a rollback mechanism to recover the initial state in case of error. The following figure locates the code pieces related to the transaction service. It shows the affected methods and classes (left) and an extract of client side code (right). We see that the **transaction is mixed with the application logic**.

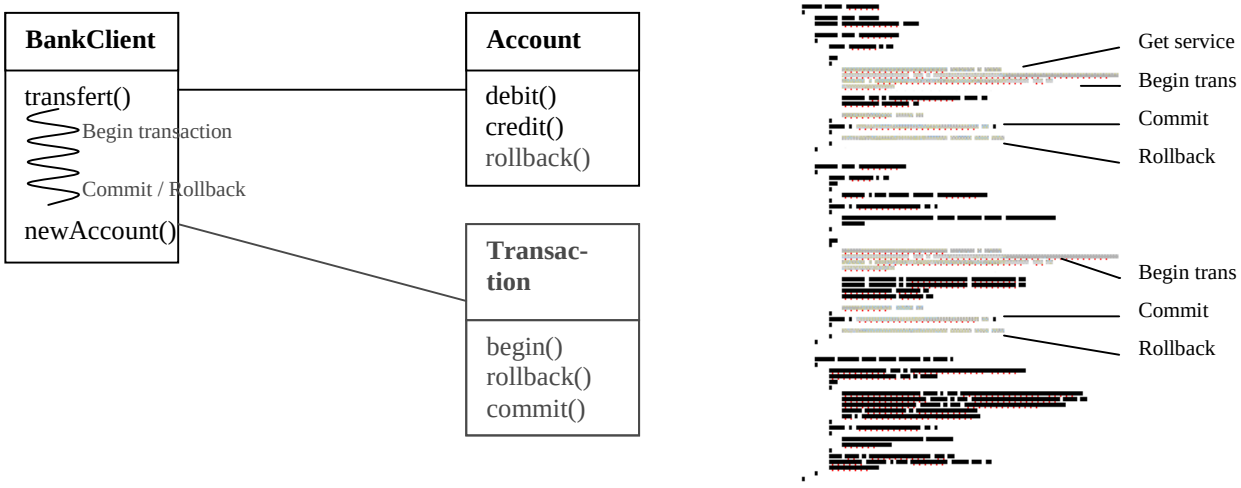


Figure 2 Transaction service integration (right: client side code. In grey: transaction code)

In this example we show a simple case where only two classes are affected. However each transactional part of the *BankClient* and each recoverable class such as *Account* have to be modified. Of course, when the number of used services increases, the situation becomes worse.

We can summarise the limitations of the standard OO approach:

- Each time programmers have to use a service they have to add pieces of code tangled with the application. In other words, the use of a service is **not reusable**.
- The pieces of code corresponding to the different services become quickly a tangled mess of lines of code and so it becomes very **difficult to maintain** or modify it.
- The **composition is hidden.** Because we cannot easily spot the pieces of code related to a service, it is almost impossible to see how some services are composed. Obviously composition cannot be reused.

To solve those problems we will try to define the services as separate entities:

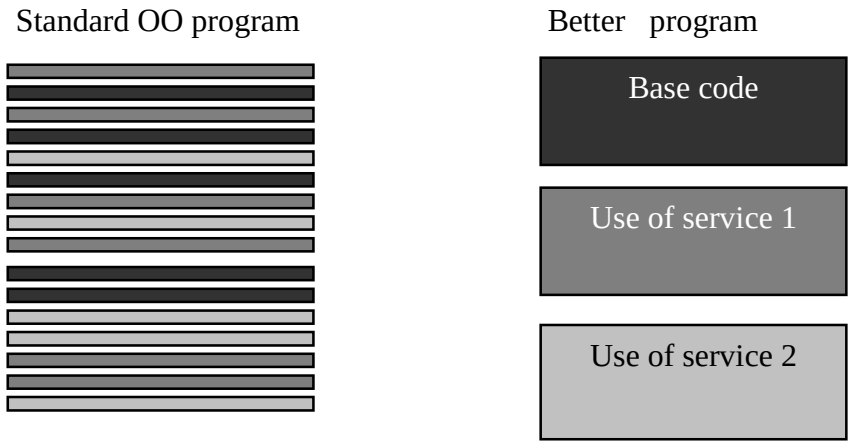


Figure 3 Tangled code problem

To be able to define a service that will modify the base behaviour of an application we need entry points on method calls or attribute value changes. Like this it should be possible to capture events that have to lead to Corba services calls. It seems obvious that **meta programming** will be needed.

1.3 Examples showing the abstraction needed to deal with services.

As a base to present our following examples we will take a well-known case: A bank client makes transfers from an account to another one. We will begin by using a simple example without transactions. To stay as clear as possible we have chosen to use UML diagrams, when possible.

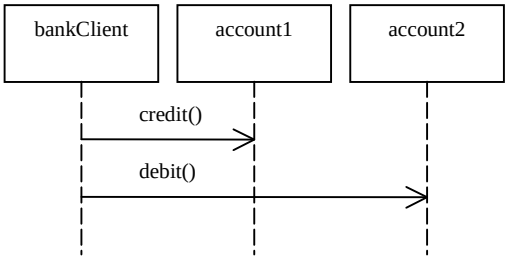


Figure 4 Sequence diagram of the base account example.

The bank client makes a transfer from the second account to the first one. It begins by crediting the first account and next debits the other one.

1.3.1 Transaction Service abstraction

The Transaction Corba Service allows to see a group of actions on different objects as atomic and to restore the initial state in case of error by using rollback mechanisms. Our goal is to be able to keep the code corresponding to the transactions as a separate piece of code. The following figure shows how the transaction service has to affect the *BankClient* and *Account* classes. The resulting application starts a transaction before the transfer in order to give the initial state back in case of error (compare Figure 4 and Figure 5).

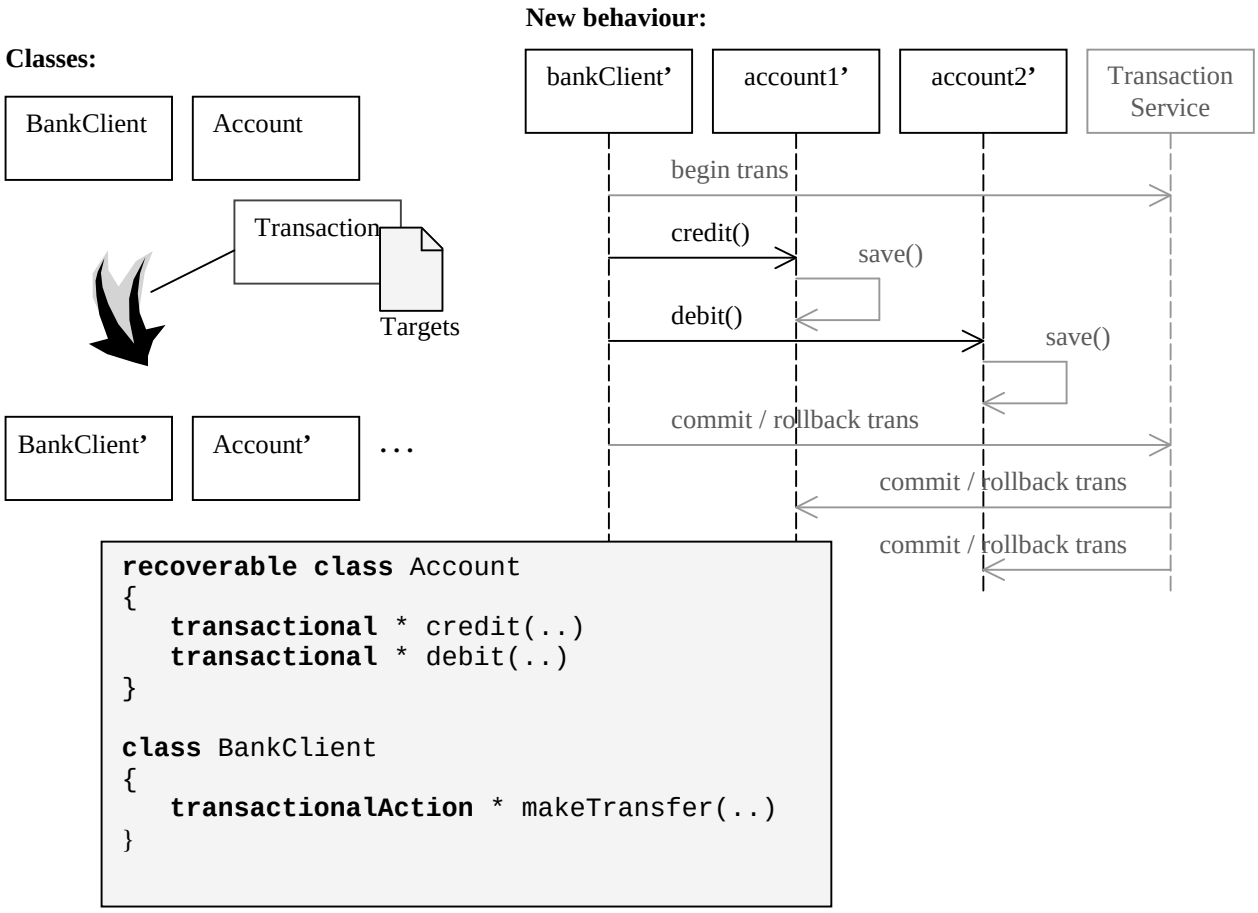


Figure 5 Sequence diagram of the example with the transaction service.

Our goal is too keep the code related to the transaction as an entity. So we will avoid having to spread the application with numerous pieces of code. Moreover there must be a way letting the programmer define which classes are recoverable and which client side method are transactional. It could be interesting to define a domain-specific programming language (DSL) for practical purpose.

The definition of such mini-languages to resolve a specific problem is more and more common. The aspect oriented programming language D [21] introduces three specific languages: Jcore expresses the basic functionality of the system, Ridl deals with remote access strategies and Cool is in charge of coordination of threads. In another context, IL has been defined to describe interactions in an easy way [26]. A language dedicated to transactions has been defined too [25] and JST provides a syntax about object synchronisation [22]. Finally, researches have been done in order to let the user define or specialise an aspect language fulfilling his specific needs [23] and [31].

Although those works are important, we will focus on the composition and assume that those languages are provided. Anyway we can deal with services composition without having such high-level syntaxes.

1.3.2 Event Service abstraction

We can imagine an event service letting the user define in which situations (error, trigger on attribute value, method execution, …) a Corba event has to be sent. If we choose to send an event each time a given method is executed without errors, we will have the following result:

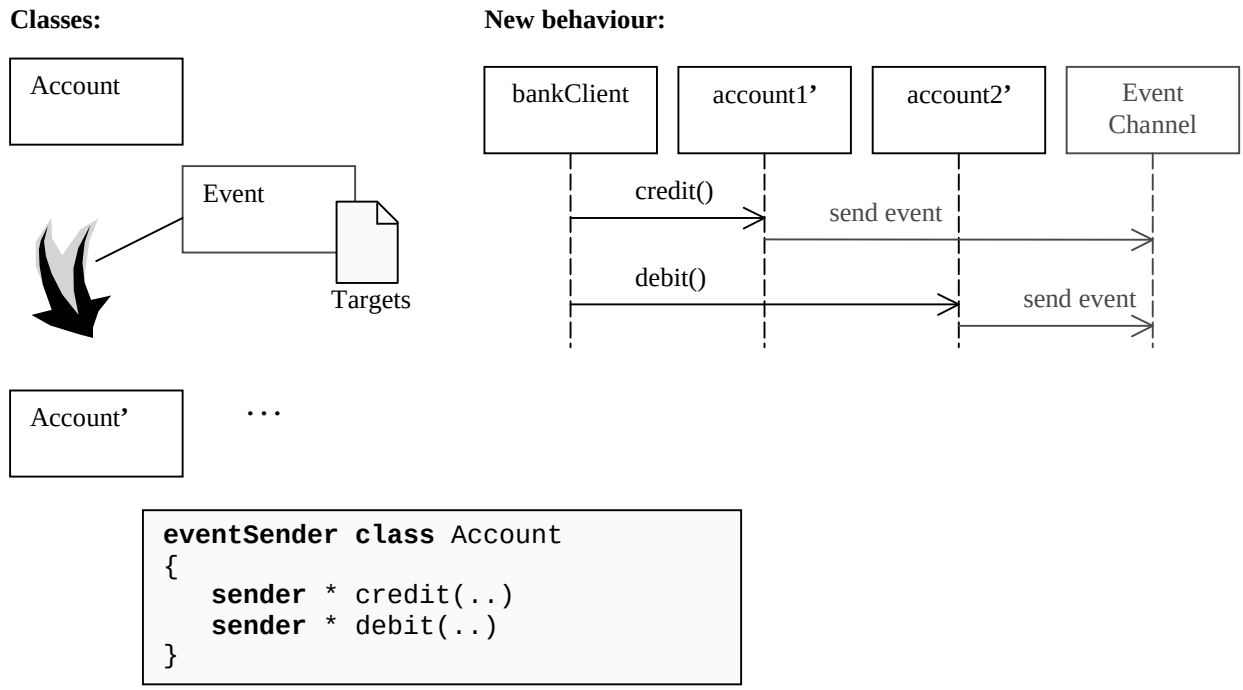


Figure 6 Sequence diagram of the example with the event service

A DSL lets the programmer define which methods have to send Corba events. The resulting application behaviour is shown in the right part of the figure.

1.4 How to deal with the composition of services

The service encapsulation presented previously allows transparent addition of services based on Corba to an application. But the user could have to use simultaneously two or more services and thus he will have to combine them.

Often it is possible to add the pieces of code of both services without problem. Unfortunately there are cases where the programmer has to deal with side effects between the added services. In other words, it happens that the resulting behaviour is not the expected one. With a standard OO approach, the services are directly called from the main code and the programmer has to deal with composition during implementation. In fact he will 'hard code' the composition into the application by choosing how and when the services have to be used.

As we can see, it is neither clean nor reusable. Anyway, if we want to extract the services from the base code we have no choice but finding a way to extract the composition too.

To have a better idea of the problem, let us look at two composition cases that do not work perfectly. First we will compose event and transaction services. Next duplication and log services will be added simultaneously.

1.4.1 A composition case: event and transaction

We begin with the combination of the two previously defined services (Events and Transactions Services). The *BankClient*'s *makeTransfer* method is transactional, the *Account* objects are recoverable. Moreover those objects send an event when *debit* or *credit* is called. So both services affect the same methods (*debit* and *credit* of *Account*).

The application combining Events and Transaction Services will work well excepted for the following case: when the transaction fails, the events credit has already been sent (look at the Figure 7) and displayed by the event viewers. In this case we notice that the behaviour is correct according to our

services definitions but it does not fit the expected behaviour. Indeed **as long as the credit or debit are not committed, the event should not be sent**.

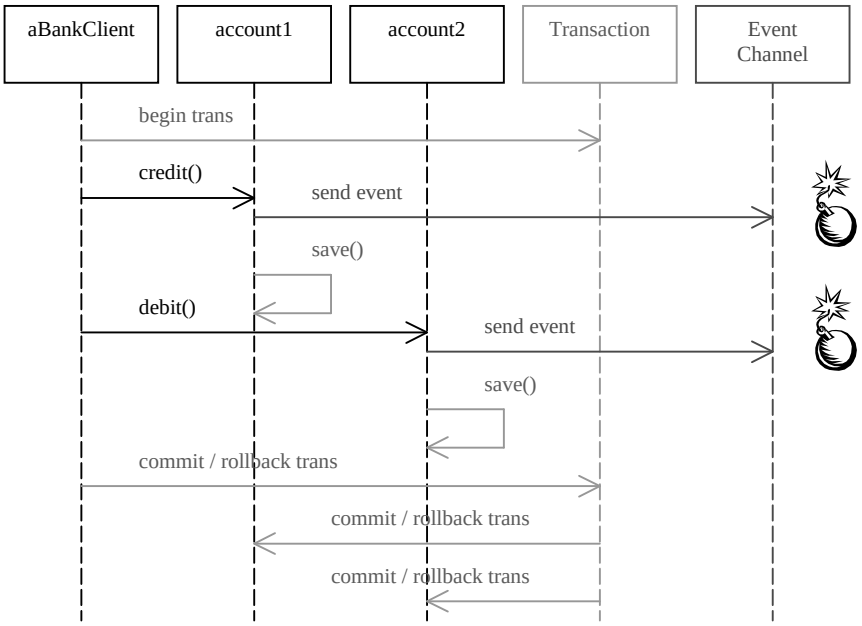


Figure 7 Sequence diagram of event and transaction services default composition.

The previous figure shows the resulting behaviour. However the expected behaviour is shown in the next figure. The difference is that in this case the debit/credit events are sent only in case of commit. It is important to understand that it is not a simple permutation of pieces of code but a **new behaviour**.

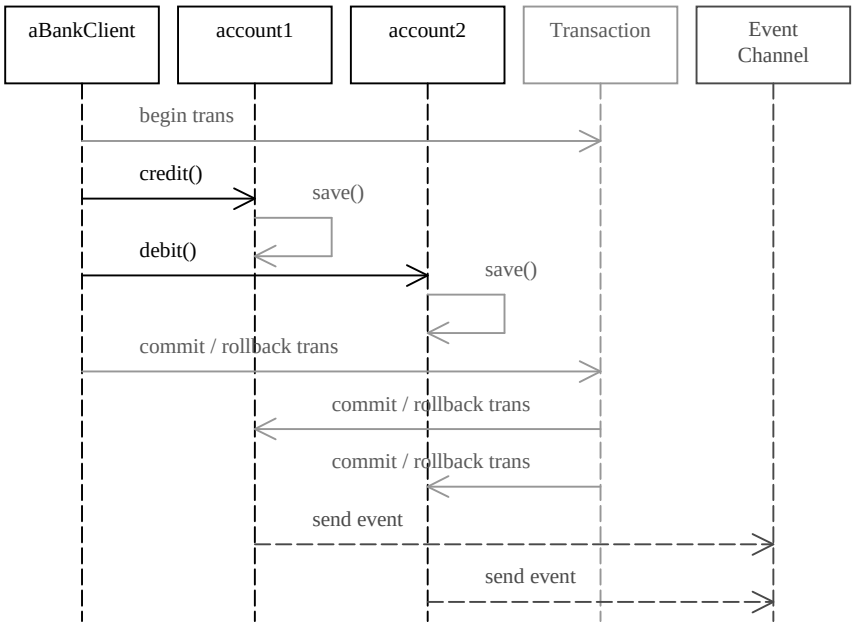


Figure 8 Sequence diagram of the example with the transaction aspect.

The programmer should be able to describe in which way the composition has to be fulfilled. A solution could be the definition of a service merging the two behaviours. However, it seems too laborious because it means the programmer must rewrite all the services when the composition does

not work properly. We have chosen another possibility. We want to find a way letting define the following rules:

Start of the transaction	→	**Save events instead of sending them.**
Transaction succeeded	→	**Send the pending events.**
Transaction failed	→	**Suppress pending events**

To implement such a behaviour, we need entry points on transaction and event services. In a more abstract way, **we need entry points on services**. The services being at a meta level, it should be a meta meta level.

1.4.2 Another composition case: duplication and log

Let us take another example of composition problem. We can define a service that duplicate an object in order to be fault tolerant. By letting run the copy on another server, the second object can be still available in case of network problem or crash of the first server. To benefit of this architecture, the second object has to be kept up-to-date.

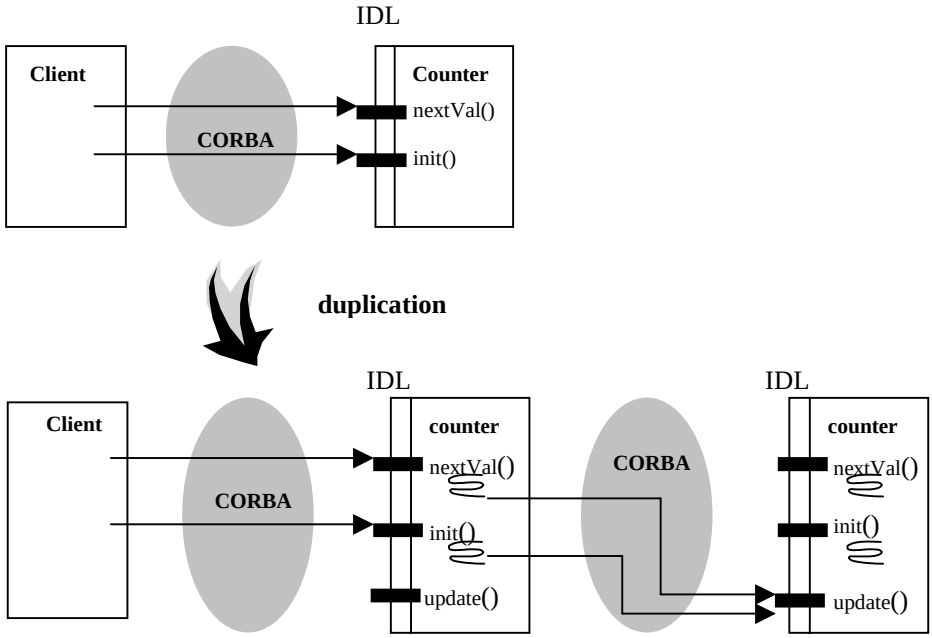


Figure 9 Duplication service

If the original counter crash or is no more accessible, the second one will be used instead. Of course a new backup object should be started.

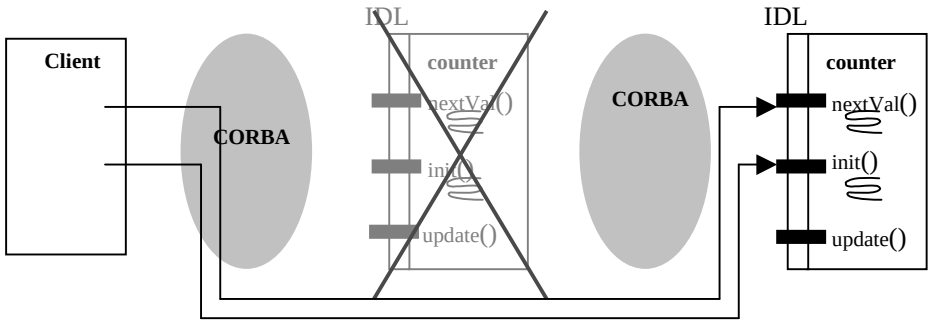


Figure 10 Behaviour when the base counter is no more available

We can define a log repository service that traces values of attributes. Each time an attribute value change, the modification is saved in the log.

If we combine this service with the duplication, each modification of the attribute will be sent twice (once from the original object and once from the copy). In our case it is useless and have to be changed. However it could be interesting during debugging time to keep a trace of the backup objects. So we see that **how services have to be combined depends on the application**. In this specific case we will have to block the repository actions when the modifications are done through the *update* method.

1.4.3 Towards a services composition model

We have seen that it could be essential to provide to the programmer a way to customise the composition of services. Moreover, to be as reusable as possible, the composition of the services should be independent of the application. In other words, the application determines what kind of composition is needed to obtain the expected behaviour but the composition should only update the services without modifying the application. It is the reason why we need a three levels architecture.

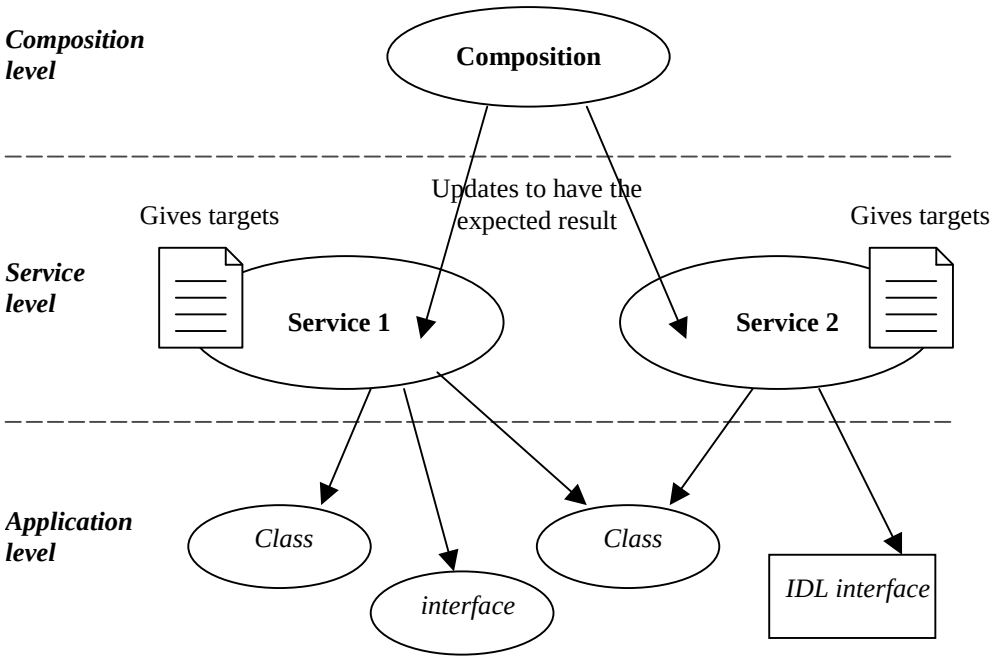


Figure 11 The three levels architecture.

The role of the composition level is the modification of existing services to ensure that the expected behaviour will be reached.

We need programming concepts replacing OOP in order to keep together the pieces of code related to a service. Of course it should be possible to combine those services and the base application in order to build a final application. Moreover we have to deal with IDL interfaces that could change when a service is added. Finally we should be able to customise the composition of services. The following chapter will describe the state of the art concerning those three points.

2 Related works

We have shown that the OO approach does not let encapsulate and compose Corba Services in a completely reusable way. It is the reason why we need to define a new approach of what is a service and how to encapsulate them. First we have to be able to separate concerns. We want to keep the pieces of code corresponding to a given service as an entity. Next we have to deal with IDL in order to modify the interfaces. Finally we have to provide a way to compose our services.

2.1 Separation of concerns

There are a lot of existing approaches to encapsulate system properties that crosscut application modules. Here we will quickly talk about such approaches that extend the Object Oriented Programming model.

Meta-Object Protocol (MOP) [27] is an Object Oriented interface for programmers to customise the behaviour and implementation of programming languages and other system software. Interesting MOP have been included in language such as Smalltalk, Lisp (CLOS [27], EuLisp), C++ (OpenC++ [1], Iguana [2]) and Java (OpenJava, Javassist [6], MetaXa [4]). This is the first approach that have been used to open the languages and so can be seen as the ancestor of what follows. However it is still an important research area because it is the most flexible and powerful solution. We have chosen not to use this approach because of its inherent complexity and low-level. Indeed our goal is to let Corba Services specialists encapsulate their pieces of code in an easier way. However we cannot yet ensure that it will never be necessary to use it.

Aspect-Oriented Programming (AOP) has been proposed by Xerox. It is an approach that allows programmers to first express each aspect of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using automatic tools. There are two types of aspects languages: specific ones such as D [21] that deal only with very specific aspects and generic ones such as AspectJ [8] that provide a lower-level syntax. **We have chosen to base our work on this approach** because it fits our requirement and is enough mature.

Subject-Oriented Programming (SOP) [19] has been proposed by IBM. It resolves the problem of handling different subjective perspective on objects. For example, the object representing a book for the marketing department of a publisher would include attributes such as short abstract. But the manufacturing department would be interested in rather different attributes such as kind of paper or kind of binding.

There are two goals in a subject approach: different development teams could work on the same object and it is possible to add unforeseen extensions to an existing system. Each perspective gives rise to a subject that is a collection of class fragments. Thus a subject is a partial or complete object model. The subjects can be composed by using composition rules:

- Correspondence rules: to define correspondences between classes, methods and attributes.
- Combination rules: to define how two classes have to be combined.

We have not used SOP in order to encapsulate Corba services because the code is not easily readable. In fact a file has to be generated to describe how the resulting application has been built. Moreover it is not as reusable as the aspects and the base code has often to be reworked in order to fulfil the SOP requirements.

Hyperspace [30] have been proposed by IBM too and will replace the SOP concepts. It supports multi-dimensional separation of concerns. It is possible to build composition rules, which describe the interrelationships and interactions among concerns and how to build new concerns out of existing ones. Hyperspaces are a generalisation of SOP, which is seen as a 2D space. By using Hyper/J, which is a Java implementation of hyperspaces, we have seen that there are the same complexity problems as

in SOP. Moreover, adding new dimensions increases it [11]. It is the reason why we did not choose this approach.

The **Composition Filters** (CF) [17] is motivated by the difficulties of expressing any kind of message coordination in the conventional object model. For example, expressing synchronisation at the interface level of an object requires something like injecting synchronisation into all its methods which need to be synchronised. The Composition Filters approach extends the conventional object model with a number of different message filters. The object consists of an interface layer and an inner object. The inner object can be seen as a regular object defined in a conventional OO programming language. The interface layer contains input and output message filters. Incoming and outgoing messages pass respectively through the input and output filters. Those ones can modify the messages or their target. We have not chosen this solution because it only lets use methods call as entry point. In our case we need more precise ones. For example we should be able to capture the modification of an attribute.

And, last but not least, **Adaptive Programming** (AP) [17] attempts to provide a better separation between behaviour and object structure in OO programs. This is motivated by the observation that OO programs tend to contain a lot of small methods which do none or very little computation and call other methods. For example, to compute the total salary in a company involves traversing some classes (Company, Department and Employee). Trying to understand the computation in such programs involves an endless pursuit through such small methods and wondering where the computation is done. Moreover a little change in the algorithm may require revisiting a large number of methods. The AP lets define how to go through a graph. AP is a special case of AOP. We have not used AP because it provides entry points on an application like other presented technologies but its aim does not fit our problem.

2.2 Separation of concerns in Corba environment

We have seen in the chapter 1.1 that one of the Corba goals is the separation of services, facilities and so on. However those paradigms are fused directly in the application code. In this paragraph we will focus on possibilities developed in order to improve the separation of concerns.

Numerous work are done to simplify the use of Corba. **Design Patterns** are proposed in order to standardise the way Corba should be used in specific cases [9].

The OMG has defined the **Corba Component Model** (CCM) [5] that provides a higher abstraction level than the Services. It is a language independent extension of the Enterprise Java Beans (EJB) and will be one of the main improvements of the Corba 3.0 release that should be available at the end of 2000. The components and our approach have the same goal: to simplify the work of the programmers by offering higher-level tools. The Corba components container environment provides persistence, transactions and security at a higher level of abstraction than Corba services do. This simplifies the application development but does not avoid tangled code because the programmer has still to define when a transaction has to begin and so on. In fact our approach and the component model should perfectly collaborate to let programmers work at a higher abstraction level.

Other works have been done associating Corba and meta-programming in order to build **Open middlewares** such as Open Corba [9], Reflective Middleware [13] and dynamicTAO [15]. Here, the goal is the creation of an open ORB allowing the redefinition of the ORB's behaviour. So it is possible to tune the middleware in order to provide a given Quality of Service. Like this Corba can be used to support real-time applications. The aim of those works is next to ours. They try to separate the control of the communication from the application core as we are looking for separating the services from the application.

2.3 Composition

Different works have been done to compose code pieces by using stacks [20] or rules [19], [16]. Those solutions provide a way to define which pieces of code (generally methods) correspond to another one and how they have to be combined (overwriting, concatenation, …). In our case we need to compose services and to define the resulting behaviour. We have to let define how the composition should be done.

Cases of incompatible services composition have been described [28]. We cannot add to an object two services having the same goal. For example the mapping of object names and references cannot be automatically done by using a file and simultaneously, by using the naming service.

We want to let the programmer define a new behaviour when services are composed. Moreover two given services could have more than one possible composition. It is the reason why none of those solutions solves our problem. This report presents a new approach based on meta-level composition.

3 Our approach to define the three levels model

Our aim is to provide a simplified way to use Corba Services and tools to define how to compose those ones. However to be able to compose services we have to see those ones as well-defined and more abstract entities. We have seen in the previous chapter that Aspect-Oriented Programming (AOP) seems to be the most appropriated way to solve that kind of problem. In this chapter we will present more precisely the AOP concepts and next a particular aspect oriented language: AspectJ. We have chosen to use AspectJ [8] because it is flexible enough to develop easily our own aspects. Moreover it is certainly the most mature solution in this research field and could reach quickly industrial applications. Finally, we want to focus on problems of composition and thus have to deal at implementation level.

3.1 Overview of Aspect-Oriented Programming

Nowadays software systems are broken down into modular units such as subroutines, procedures, objects and so on. Many systems have properties that cut across these abstraction mechanisms: failure handling, persistence, communication, concurrency, and many other aspects of a system's behaviour are not easily localisable to a single block of executable code - even though they can often be thought about relatively separately.

Because source code modules correspond so directly to blocks of executable code, and different aspects of concern must crosscut the executable code, modules become finally a tangled mess of lines of code for different purposes. This phenomenon is at the heart of much of the complexity in existing software systems.

3.1.1 Main goal of AOP

Aspect-Oriented Programming (AOP) is an approach that allows programmers to first express each aspect of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using automatic tools.

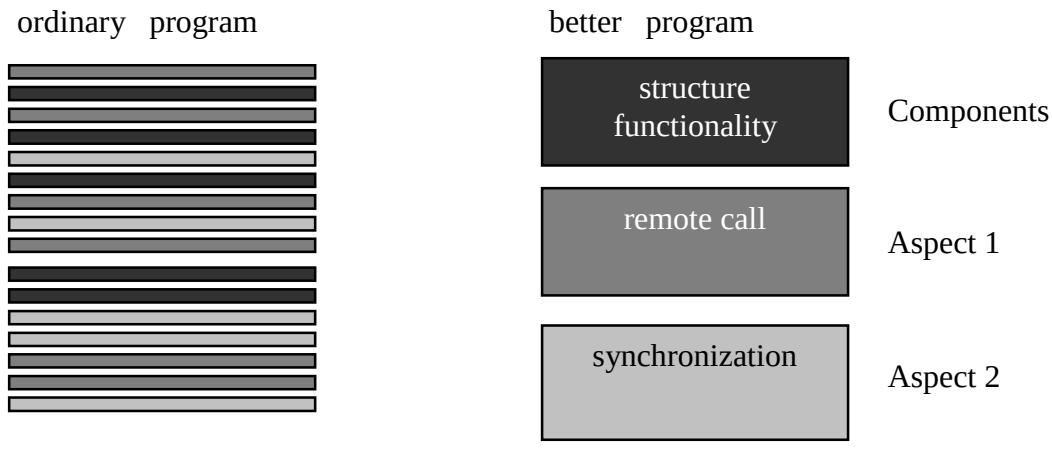


Figure 12 Differences between tangled code and AOP solution.

Generally the aspect compiler (or weaver) does not generate an executable but a combination of the classes and aspects. The result looks like an ordinary program and will be compiled by using a standard compiler.

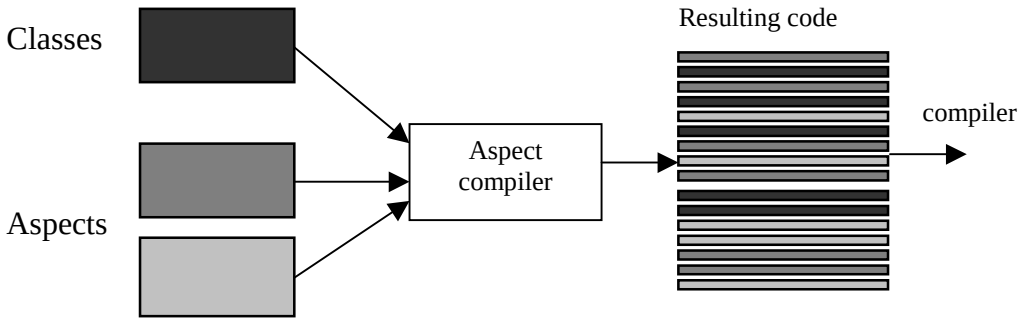


Figure 13 Weaver or aspect compiler.

3.1.2 Specific versus generic aspect-oriented languages

Early instances of aspect languages were limited in scope and dealt only with very specific aspects in very specific contexts. For example, D [21] consists of three specific languages:

Jcore: an object-oriented component language used to express the basic functionality and the activity of the system.

Ridl: an aspect language used to express remote access strategies.

Cool: an aspect language used to express coordination of threads.

The two aspect languages can be seen as meta-languages specially designed for implementing aspect programs that control the object-oriented program itself. Since each language is devoted to relatively few concerns, its syntax remains simple. Moreover the resulting software are more compact because we can define aspect by using a high-level syntax.

Recently, with AspectJ, AOP has taken a turn towards a more general aspect language applicable in broader context. In this case new aspects can be defined by using the same language.

The first solution gives a high level of abstraction avoiding specific implementation detail. In fact they are formulated in terms of concepts linked to their context (for example: mutual exclusion in the context of synchronisation). The second solution provides more generally applicable declarations. Of course to achieve such results, the language has to be less specific and so more low-level. Typical AspectJ declarations provide code to be executed upon entry and exit of methods. This approach is somewhere between Meta Object Protocols (such as Javassist [6]) and D. In other words AspectJ is less high-level than D but more powerful and flexible than it. Moreover AspectJ is more high-level than MOP but less powerful and flexible. Although it is a higher-level solution, the philosophy of AspectJ is really next to the MOP defined by Shigeru Chiba (openC++, openJava and Javassist).

D lets write code in an efficient way because of it aspect specific and high-level syntax but it is not flexible enough. However choosing AspectJ, which is a generic aspect-oriented language, does not forbid the definition of a specific mini-language to simplify the user job. For example after having defined a transaction aspect by using AspectJ, we can define a specific syntax to let the user choose which method will be transactional or which object will be recoverable.

The definition of service specific languages could be interesting if the services are part of a library. At least, we have to separate the crosscut definition of the aspect implementation [14].

3.2 Overview of AspectJ

AspectJ is a general-purpose aspect-oriented extension to Java. To give a first idea of the AspectJ syntax and to let you understand the pieces of code that will be shown latter in this report, here is a definition of the base possibilities offered by AspectJ.

3.2.1 crosscut

The crosscut allows capturing events such as method invocations, constructor invocations, and signalling and handling of exceptions. AspectJ's crosscuts capture collections of those events in the execution of a program. Crosscuts do not define actions. they simply describe events. For example,

```
crosscut setget(): MyClass & (void setValue(int) | int getValue());
```

This crosscut (named setget) describes the reception of *setValue* or *getValue* messages by any objects of type *MyClass*. Here's another example:

```
crosscut ioHandlers(): MyClass & * *(..) & catch(IOException);
```

This crosscut describes the catching of exceptions of type IOException in all operations of any objects of type MyClass.

NB: According to Gregor Kiczales, it will be soon possible to capture events such as attribute value modifications.

3.2.2 advice

Advice declarations define pieces of aspect implementation that have to be executed when the event corresponding to the associated crosscut happens. Here is an example of advice on a crosscut:

```
crosscut setget(): MyClass & (void setValue(int) | int getValue());

advice(): setget() {
   before { System.out.println("set or get value"); }
}
```

In this case the advice piece of code will be called before the code of *getValue* and *setValue.* It is possible to define different advice types:

- **before**: runs just before the execution of the actions associated with the crosscut.
- **after**: runs just after the successful execution of the actions associated with crosscut.
- **catch**: runs when the execution of the actions associated with crosscut return with a corresponding exception type.
- **finally**: runs just after the execution of the actions associated with crosscut, even in the presence of exceptions.
- **around**: traps the execution of the designated methods.

NB: More powerful abilities are available by passing parameters to the advice or by using *thisJoinPoint* that contains some important information about the event.

3.2.3 introduction

Introduction declarations introduce whole new elements in the given classes. Here is an example:

```
introduction MyClass
{
   String name;
   void setName(String _name)
   {
      name = _name;
   }
}
```

It introduces a new attribute and a new method in class *MyClass*. We can notice that pieces of code are written at aspect definition level and not at class definition time.

3.2.4 Comparison between standard OOP and AOP.

The next figure shows a simple example of client-side Corba Transaction Service. A client has two transactional methods that have to obtain the context, begin the transaction and next commit or rollback it. The left column shows the structure of the standard Java code (it is deliberately unreadable). The right column shows the same application written by using AspectJ. The pieces of code corresponding to the transaction (grey parts) are kept together as transaction Aspect. The second version is easier to understand and maintain because there is a clear separation of concerns. Finally the second version is smaller than the first one.

Standard Tangled Code **Aspect Oriented Code**

Figure 14 Differences between tangled standard Java code and cleaner AspectJ code.

With such an approach, the code reuse is improved because the aspect can easily be applied to other classes. Moreover, by defining a higher-level mini-language specific to the aspect, it is possible to increase the benefits. Indeed by using a dedicated syntax to implement an application aspect the result will be smaller and easier to understand. However we will not focus yet on the mini-languages definition because our main aim is to solve composition problems.

3.3 AspectJ in Corba environment

We have chosen to use Aspect-Oriented Programming and especially AspectJ in order to facilitate the manipulation of Corba Services. But we have encountered some limitations. Indeed AspectJ is not able to modify IDL interfaces and we have seen that those changes are mandatory in Corba environment. Moreover, AspectJ does not provide a way to define the aspects composition strategy. There is only one possible composition strategy: the concatenation of the aspects.

3.3.1 An example running without any problem : the event services

Interfaces modifications are principally needed when the added functionality acts as a server. In other words, the aspect adds or modifies the service offered by the object. However, when we add a piece of code having a client role generally no problem occurs. In the following example we will show how to emit Corba events from a standard object.

The Corba Event Service allows an object to send data to unknown objects. It has just to push the events in the Event Channel. The applications interested to those events will connect themselves to the Events Channel in order to receive those ones. They will be called when an event happens. In other words by using the Corba Events Service an event viewer can be connected or disconnected without any effect for the event sender. Moreover, the number of viewers is not restricted. The events will be broadcast to each ones.

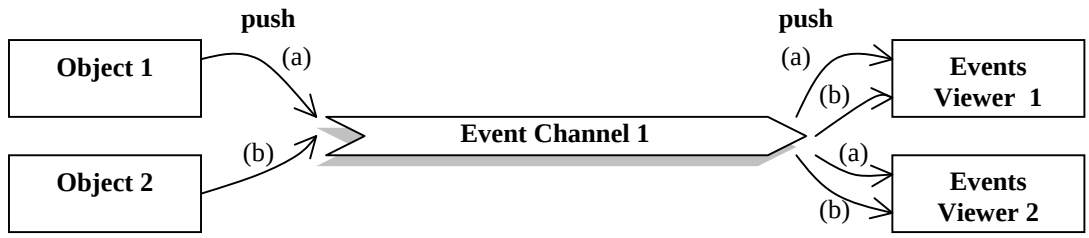


Figure 15 The push model of the Event Service.

For example we can choose to emit a specific event each time a given error occurs. To do that we define a new crosscut:

```
crosscut eventSender(): MyClass & * *(..) & catch(MyException);
```

This crosscut describes the catching of exceptions of type *MyException* in all operations of any objects of type *MyClass*. The corresponding advice will push data on the Event Channel.

```
static advice(): eventSender()
{
   after {
      supplier.sendEvent(thisJoinPoint.className + "."
                         + thisJoinPoint.methodName); }
}
```

Of course some other pieces of code have to be added by the aspect. It should obtain the ORB (singleton), initialise the event service, obtain the consumer (the channel) and, last but not least, create a supplier and connect it to the consumer.

We can see in this example that there is no need for interface changes. In fact the object modified could be a standard object having no idea of Corba and so no IDL interface.

NB: To stay as simple as possible we use the default event channel but it could be possible to create a specific event channel in order to let viewers select more precisely which events they are interested in.

3.3.2 Simple example needing IDL interface changes: instrumental aspect

An aspect corresponding to a Corba Service can modify or extend the interface of the affected object. Let us look at another example where the aspect part acts as a server side object. So, we can develop an instrumental aspect in order to do coverage, analyses or measures of objects. Each instrumented object can have to respond to specific messages such as get the statistics (*getStat)*. We can imagine other extensions such as *reinit* according to the user point of view. For example each time an exception happens or a method is called we could increment counters. Or we can imagine to keep a history or the average value of some attributes. A remote statistics analyser could obtain the statistics of an object by getting a reference on this one (by using the Naming Service) and using the introduced method *getStat*.

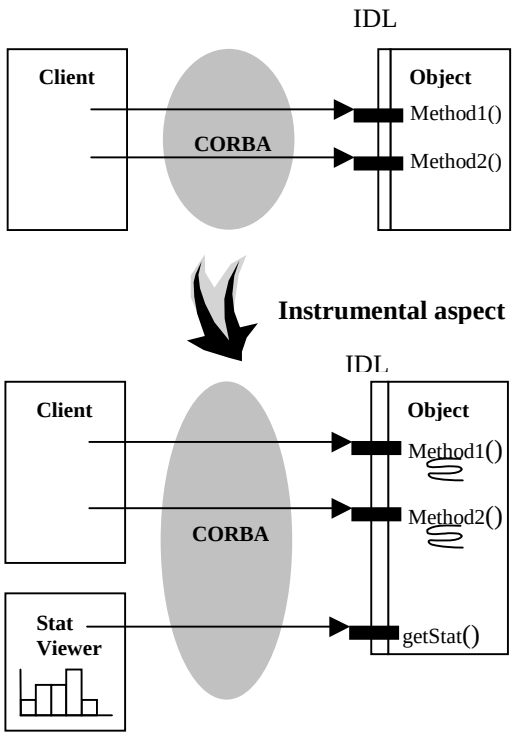


Figure 16 modification of IDL interface due to the aspect

NB: Here the IDL modifications correspond to the *introduction* of a new method in the implementation code. However we can introduce new methods without having to modify the interface. Indeed, only the methods that could be called remotely have to be added to the interface too.

3.3.3 Which entry points and modifications are needed

We have shown that it is sometimes possible to add the service without having to change the IDL interface of the affected class. The code added by the aspect has a **client role**. AspectJ can deal with those cases.

- Local changes: the modification does not have remote effects.
- Event sender: the object has to emit an event when a method is called or when an error occurs.
- Log repository: instead of sending event, another object method is called when something happens.
- Add of transactions to the client: begin transaction and commit/rollback have to be added to the transactional method.

In other cases, when the code added by the aspect has a **server role**, the interfaces have to be enlarged (add new method or inheritance). AspectJ cannot deal with such cases.

- Simple Statistics example: a new method is usable from another client which has to obtain a reference on the remote object.
- Do recoverable an object in Simple Transaction Service case: new methods are added to allow the commit and rollback operations by inheriting of "recoverable object" interface.

Of course, we can imagine cases that needs larger IDL modification or entry points that are not available in AspectJ.

NB: An aspect can add both client and server role to one or more objects. For example the transaction aspect modifies the client and server.

When an object has to become a client of another Corba object (or service), there is not any problem. A piece of Corba code has to be added to existing methods in order to call remote objects. However when we will add a client role to an object, new problems happens:

Feature	Comment	Availab-ility	Client	Server	Compo-sition
Extension of the implementation: If the interface is modified, the Corba object implementation has to be updated. Moreover new services need to be used and so client-side modification are required too.					
Methods behaviour changes	The behaviour can be changed by using crosscuts and advice.	ajc r0.6[1]	X	X	X
New attributes		ajc r0.6	X	X	X
New methods		ajc r0.6	X	X	X
New interface	Look at Annex A :	-			
Inheritance	We have to be careful because Java does not allow multiple-inheritance. So wrappers are certainly more appropriated in this case.	-			
Redirect access to an attribute		ajc future release		X	
Extension of IDL interface: In order to allow the management of the new behaviour, new methods have to be introduced in the existing class. If we want to use them remotely, the IDL interface will have to be updated.					
Add methods		aIDLc[2] 0.1		X	X
Add attributes		aIDLc 0.1		X	X
Inheritance	It is necessary that a defined interface inherit from another one to be able to use Corba Services.	aIDLc 0.2		X	X
Inclusion	It is often necessary to include new IDL files	aIDLc 0.2		X	X
Method modification	It could be useful to add parameters to a method or to replace a method by another one.	-		X	X
Extension of Aspects (aspect on aspect): In order to let compose the aspects we need entry points on those ones and on the associated objects and interfaces.					
Crosscut on action	It has been defined to customise the composition but could be used in another context.	aoac[3] 0.1			X

[1] ajc: AspectJ compiler. The release 0.6 is the last available version.
[2] aIDLc: Aspect IDL compiler. Look at 4.1.2.
[3] aoac: Aspect on aspect compiler. Look at 4.1.1.

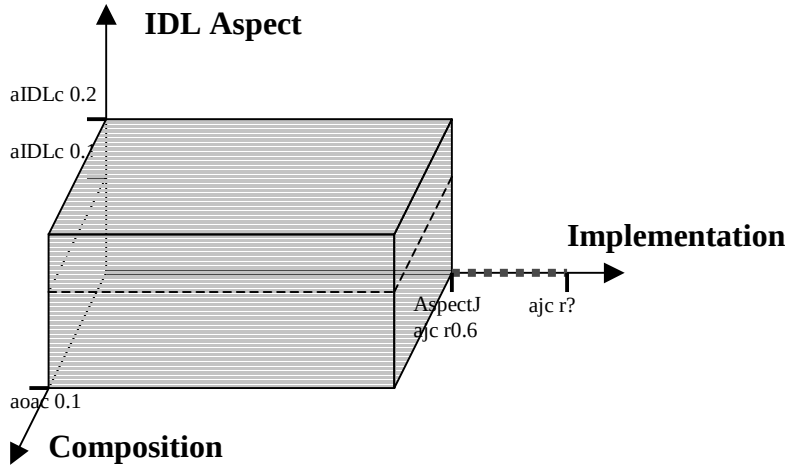


Figure 17 Extensions of AspectJ: two new dimensions (IDL and Composition)

The previous figure show the space covered by our extensions. Two new dimensions have been added and evolve independently of AspectJ. They let define aspects that touch IDL interfaces and allow the composition of aspects by defining meta-level composition aspects.

4 Basic implementation to validate the concepts

We have shown that the use of services together can lead to composition problems. The default result of the composition is not always the expected one and the programmer should have a way to define how the composition must be done. It is the reason why we think it is necessary to introduce *meta-level composition aspects* letting modify the services.

In this chapter we will present the implementation of a basic meta-aspect architecture in order to validate the concepts presented earlier. First we will show the syntax extensions needed to deal with meta-aspects and IDL interfaces modifications. Next we will present a complete implementation of the event and transaction aspects and their composition by using the extended syntax.

4.1 Definition of new compilers

After having describe the lack of AspectJ in Corba environment, we presents our solution. We can now precise the three levels architecture (look at Figure 11). The topic of this report corresponds to the bold parts of the following figure (meta-level composition aspect and IDL interfaces modification).

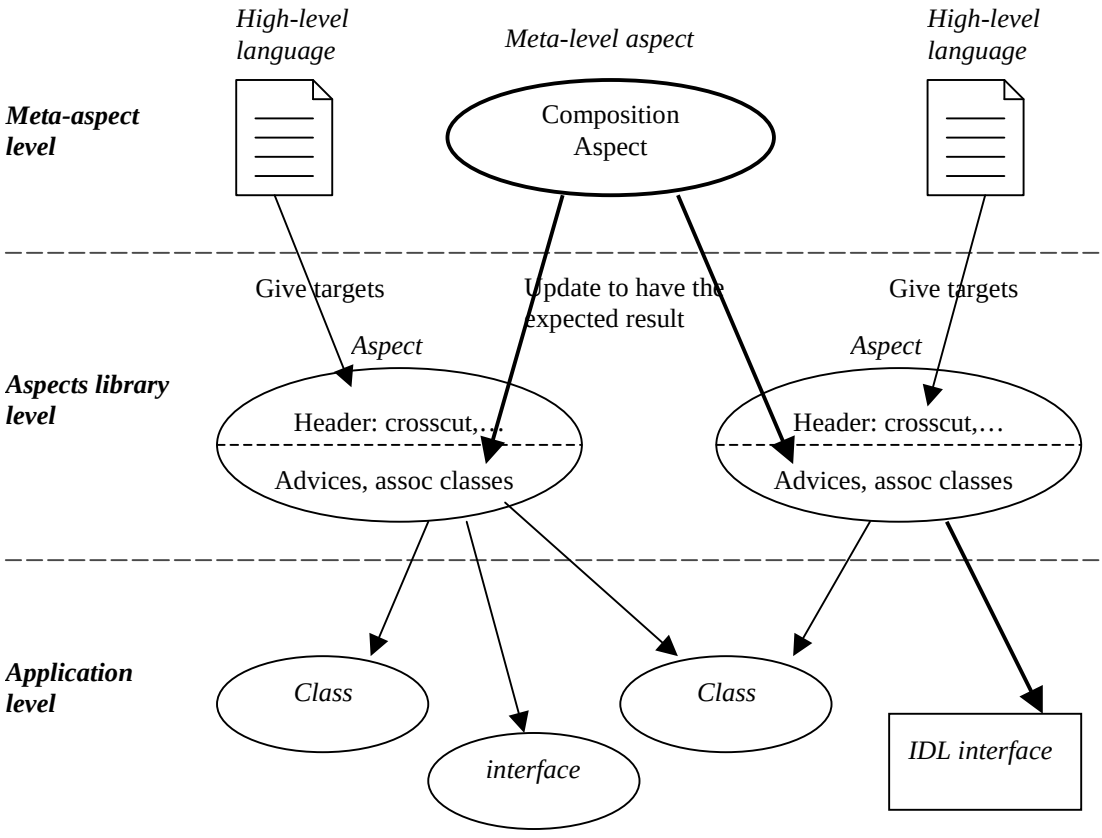


Figure 18 The three levels architecture.

To deal with the meta-Aspects, a syntax extension has been needed (look at chapter 4.1.1). It lets define crosscuts on aspects. Moreover, the syntax has been so extended that the aspects library can modify the IDL interfaces of the application (look at chapter 4.1.2).

4.1.1 Dealing with composition: Aspect on Aspect compiler (aoac)

Our goal is to be able to modify the aspects and associated classes independently of the application. The modification of classes associated to the aspect can be done by using AspectJ. But the

modification of the aspect themselves require entry points on the aspects. For example, instead of writing:

```
crosscut freezeEvent(): Account & (* debit(..) | * credit(..));
```

we would like to write:

```
crosscut freezeEvent(): EventAspect & sendAnEvent.after;
```

(meaning the block "after" of the advice *SendAnEvent* of the aspect *EventAspect*)

So we will be able to change the *sendAnEvent* crosscut of *EventAspect* without having to change the composition aspect. Moreover we will be able to add code before or after these action ("after" block of *sendAnEvent* advice).

We have defined a little extension of the AspectJ grammar to manage those cases.

Grammar extension

```
CrossCutOnActionDeclaration:                                          ← a)
    'crosscut' Identifier '():'
        CrossCutOnActionDesignator ';'

CrossCutOnActionDesignator:
    ConcreteTypeDesignator & ActionDesignator '.' ActionType          ← b)

ActionType:
   'around'                                                           ← c)
   'before'
   'after'
   'catch'  '(' FormalParameter ')'
   'finally'

AdviceOnActionDeclaration:                                            ← d)
    'static' 'advice' '():' Identifier '()'
       '{' AdviceOnActionStatements '}'

AdviceOnActionStatements:                                             ← e)
  AdviceOnAction
  AdviceOnActionStatements AdviceOnAction

AdviceOnAction:                                                       ← f)
    'around' Block
    'before' Block
    'after' Block
```

a), b) declaration of a crosscut on action; **c)** it is possible to capture the event corresponding to each action type; **d), e)** declaration of an advice on action event consumer; **f)** yet an advice on action allows only the definition of those three action types.

It will be easier to reuse this new aspect because when the crosscut of event and transaction aspects will be changed to fit a new application, *CompAspect* aspect will not have to change. Moreover by separating crosscut definitions and implementations we will have a more reusable solution [14].

We have implemented a precompiler generating AspectJ code. We will pass it the meta-level composition aspects using the extended syntax (*CompAspect*), the aspects to modify (*TransAspect,…*) and the associated classes (*EventSender,…*). However we will not give it the classes modified by the aspects (*Account,…*). Thus there is a clean separation between classes, aspects and meta-aspects (look at Figure 18).

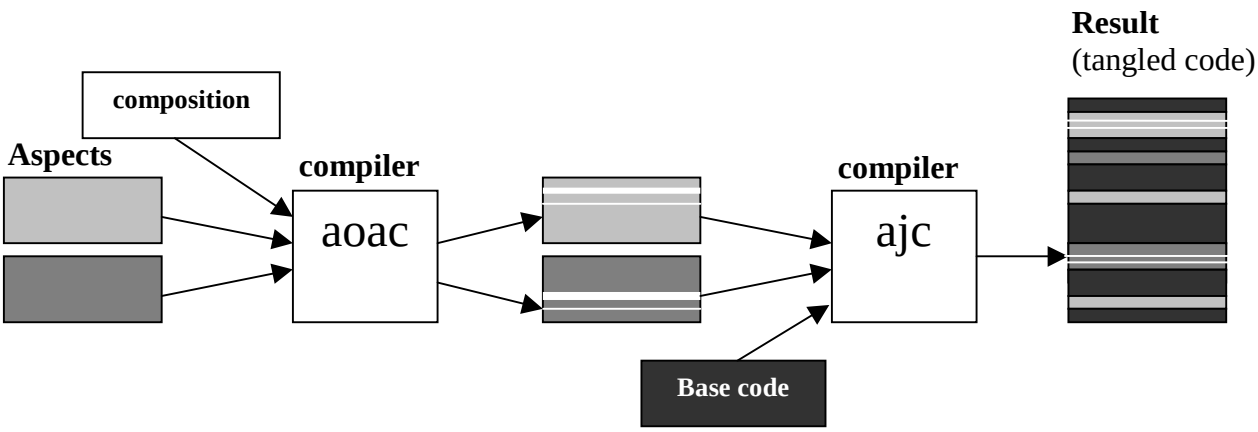


Figure 19 Composition process.

The advantages of such an approach are:

- **Crosscut precision**: We can define more precisely where the code added by the composition is put in the result. For example, we can add code before or after the action added by an aspect. Moreover it is possible to replace the code added by the aspect.
- **Reusability**: The modification of the aspect are independent of the application and so the composition is reusable if the same case happens in another application.

4.1.2 Dealing with interfaces: Aspect IDL compiler (aIDLc)

The second step is the modification of the IDL interface. As AspectJ does not take into account IDL interfaces we have to find a way to do it. We think that the best solution is the modification of IDL code and not the translation in Java interface. As a first step, we have chosen to define the IDL interface modifications directly in the AspectJ files. To ensure that the files remain usable by the AspectJ compiler, we add IDL parts as comments. After having defined the methods to introduce in the implementation, we can choose which ones have to be introduced in the IDL interface. The following example add a method *init* to the implementation class and to the IDL interface.

```
introduction CounterImpl
{
   public void init()
      { System.out.println("Init counter");   counterVal = 0; }
}
// @IDL@ introduction Counter { void init (); }  @IDL END@          ← IDL piece of code
```

Grammar extension

```
Idl:                                                                    ← a)
   '@IDL@' IdlAspect '@IDL END@'

IdlAspect:
   IdlIntroduction
   IdlInheritance

IdlIntroduction:
   'introduction' InterfaceIdentifier '{' IdlCode '}'                   ← b)
   'introduction' FileIdentifier '{' IdlCode '}'                        ← c)

IdlInheritance:                                                         ← d)
   'inherit' InterfaceIdentifier ':' InterfaceIdentifier
```

a) begins and ends the IDL part; **b)** adds pieces of code to an interface: method, attribute; **c)** adds pieces of code to a file: include, types; **d)** definition of a new inheritance;

NB: We have been interested in the possibility of having only Java code and interfaces instead of Java and IDL. So it should have been easier to modify the existing code and get entry point in this one. A well-known solution to define Corba objects without having to use IDL is **RMI over IIOP**[5]. It is a new feature of Corba that is part of the soon available 3.0 release. In fact, the programmer has to define RMI objects and corresponding IDL interfaces are automatically created. Like this it is possible to access those remote objects from a standard Corba client. It is easier to use Java interfaces instead of IDL ones but the server side code is not Corba but RMI. It is the reason why, after some tests, we have not followed this approach up.

4.2 A complete example based on the new syntax

After having presented the two compilers we have implemented, we will present the Corba Event Service and Corba Transaction Service encapsulated as aspects. Next we will presents the solution we propose to compose those ones.

4.2.1 Base Code

Let us look at the implementation of our previous example. The account have the following interface:

Account.idl

```
exception AccountException
{
    string reason;
};

interface Account                                                    ← a)
{
    void credit ( in long value );                                   ← b)

    void debit ( in long value ) raises ( AccountException );

    long getBalance();
};
```

a) : the interface; **b) :** three methods;

The class corresponding to this interface is implemented in *AccountImpl*.

AccountImpl.java

```
Public class AccountImpl extends _AccountImplBase                    ← a)
{
    private int balance = 0;

    public AccountImpl(int initialBalance)
    {
        balance = initialBalance;
    }

    public void credit(int value)
    {
        balance = balance + value;
    }

    public void debit(int value) throws AccountException
    {
        if ( balance - value < 0)
            throw new AccountException("balance < 0");
        else
           balance = balance – value;
    }

    public int getBalance()
    {
        return balance;
```

```
    }
}
```

a) : inherit of the skeleton class;

A server or account factory creates some accounts and register those ones by using the naming service. Now let's have a closer look at a simple client example:

BankClient.java

```
Public class BankClient
{
    public static void main( String args[] )
    {
        // Initialize the ORB
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);          ← a)

        // Resolve the NameService
        org.omg.CORBA.Object obj = null;                                    ← b)
        org.omg.CosNaming.NamingContext naming = null;
        try {
            obj = orb.resolve_initial_references("NameService");
            naming = org.omg.CosNaming.NamingContextHelper.narrow(obj);
        }
        catch ( org.omg.CORBA.ORBPackage.InvalidName name )
            . . .

        org.omg.CORBA.Object obj = null;

        // Build Account path
        org.omg.CosNaming.NameComponent [] name =                           ← c)
            new org.omg.CosNaming.NameComponent[1];
        name[0] = new org.omg.CosNaming.NameComponent();
        name[0].id = "Account1";
        name[0].kind = "Example";

        // Resolve Account reference from NameService
        try {                                                               ← d)
            obj = naming.resolve(name1);
        }
        catch  . . .

        // Narrow the object reference
        Account account = AccountHelper.narrow(obj);

        // Use the Account object
        try
        {
            System.out.println("balances: " + account.getBalance());       ← e)
            Account1.credit(400);
            System.out.println("balances: " + account.getBalance());
        }
        catch ( AccountException ex )  . . .

        catch ( org.omg.CORBA.SystemException ex )  . . .
    }
}
```

a) initialise the ORB; **b)** get the naming service; **c)** build the path corresponding to the account; **d)** get a reference on the account; **e)** use of the debit and credit methods.

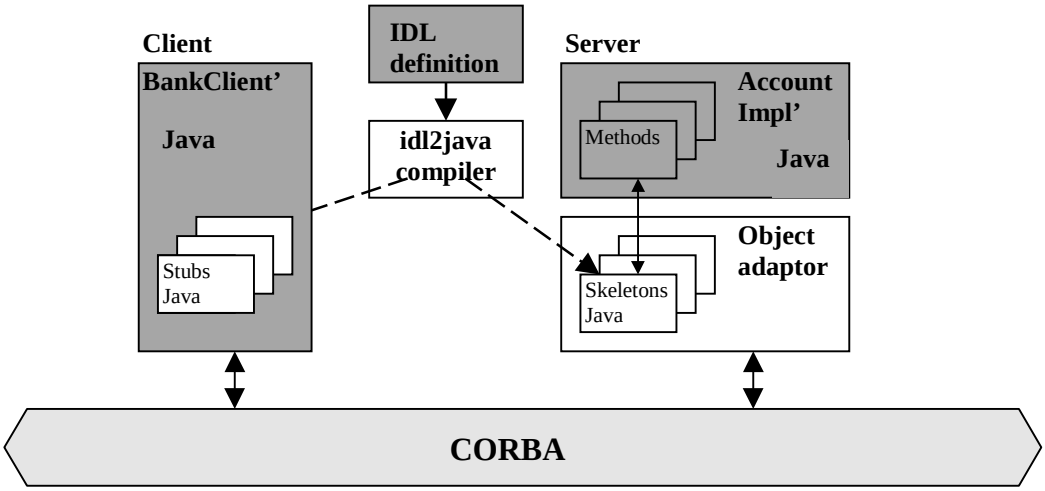


Figure 20 Implementation diagram of the base account example

The previous figure shows how the application is built. The application pieces that have to be written by the programmer are dark printed in the previous figure. The following figure shows how it works.

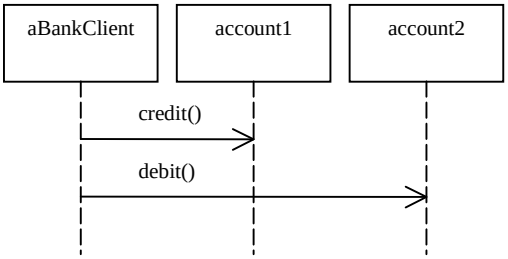


Figure 21 Sequence diagram of the base account example.

The bank client makes a transfer from the second account to the first one. It begins by crediting the first account and next debits the other one.

4.2.2 Event aspect

Let us describe more precisely the event service. The implementation of the Corba account is given in the following frame.

We could be interested to send an event in some particular cases such as the transferred amount is higher than 2000 $, the resulting balance is lower than 100 $ or an error has occurred. To stay as simple as possible we choose to send an event each time the debit or credit operation has correctly worked. The aspect *EventAspect* will be in charge of that. Thus the access to Corba Event Service is totally invisible for the application programmer.

EventAspect

```
public class EventAspect                                                    ← a)
{
   crosscut sender(): AccountImpl;

   // each constructor of AccountImpl...
   crosscut initEvent(): sender() & new(..);                                ← b)

   // each method having to send events...
   crosscut sendEvent(): sender() & (* debit(..) | * credit(..));           ← c)
```

```
    // add an attribute to send events                                  ← d)
    introduction sender() { EventSender eventSender = null;}

    // init the event sender object                                     ← e)
    static advice(): initEvent()
    {                                                                   ← f)
        before {
            EventSenderSingleton.instance();
        }
    }

    // send the event                                                   ← g)
    static advice(): sendEvent()
    {                                                                   ← h)
        after {
            EventSenderSingleton.instance().sendEvent
                (thisJoinPoint.className +
                    "." + thisJoinPoint.methodName);
        }
    }
}
```

a) an aspect is defined as a class; **b)** crosscut capturing the constructor of the account class; **c)** crosscut capturing the *debit* and *credit* methods of the account; **d)** introduction of a new attribute to the class *AccountImpl*; **e)** advice corresponding to the crosscut b); **f)** piece of code to add before the base code of the account constructor; **g)** advice corresponding to the crosscut c); **h)** piece of code to add after the base code of the account's *debit* and *credit* methods.

The *EventSender* Corba object hides the Corba code to initialise the Event Channel and send events. The *EventSenderSingleton* gives an access to this one. The aspect adds a piece of code to obtain the reference on the service in the constructor (*initEvent*). Moreover it adds another piece of code that sends an event by using the event sender at the end of *debit* and *credit* methods (*sendEvent*). The resulting behaviour is to emit an event "AccountImpl.debit" each time the method *debit* is used successfully and the same with the method *credit*.

EventSender.idl

```
Interface EventSender
{
    // send an event
    void sendEvent(in string data);
};
```

EventSenderImpl.java

```
public class EventSenderImpl extends _EventSenderImplBase
{
    myPushSupplier supplier = null;

    public static void main( String args[] )
    {
        // Initialize the ORB
        // Instanciate the event server service object
        // Bind service into NameService
        // Wait for invocations
    }

    public EventSenderImpl()
    {
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();               ← a)
        org.omg.CORBA.Object obj = null;
        org.omg.CosEventChannelAdmin.EventChannel channel = null;
        org.omg.CosEventChannelAdmin.ProxyPushConsumer consumer = null;

        try {
```

```
            obj = orb.resolve_initial_references("EventService"); }          ← b)
        catch ( org.omg.CORBA.ORBPackage.InvalidName ex ) {
            . . .

        channel = org.omg.CosEventChannelAdmin.                              ← c)
            EventChannelHelper.narrow(obj);

        org.omg.CosEventChannelAdmin.SupplierAdmin supplierAdmin =
            channel.for_suppliers();
        consumer = supplierAdmin.obtain_push_consumer();
        supplier = new myPushSupplier(orb,consumer);                         ← d)
        orb.connect( supplier );                                             ← e)

        try {
            consumer.connect_push_supplier( supplier ); }                    ← f)
        catch ( java.lang.Exception ex_connect ) . . .
    }

    public void sendEvent(String data)
    {
        supplier.sendEvent(data);
    }                                                                        ← g)
}
```

a) get a reference on the ORB (Singleton); **b)** get a reference on the event service; **c)** channel **d)** event consumer (the channel); **e)** the event supplier (account side); **f)** connect supplier and consumer; **g)** send an event.

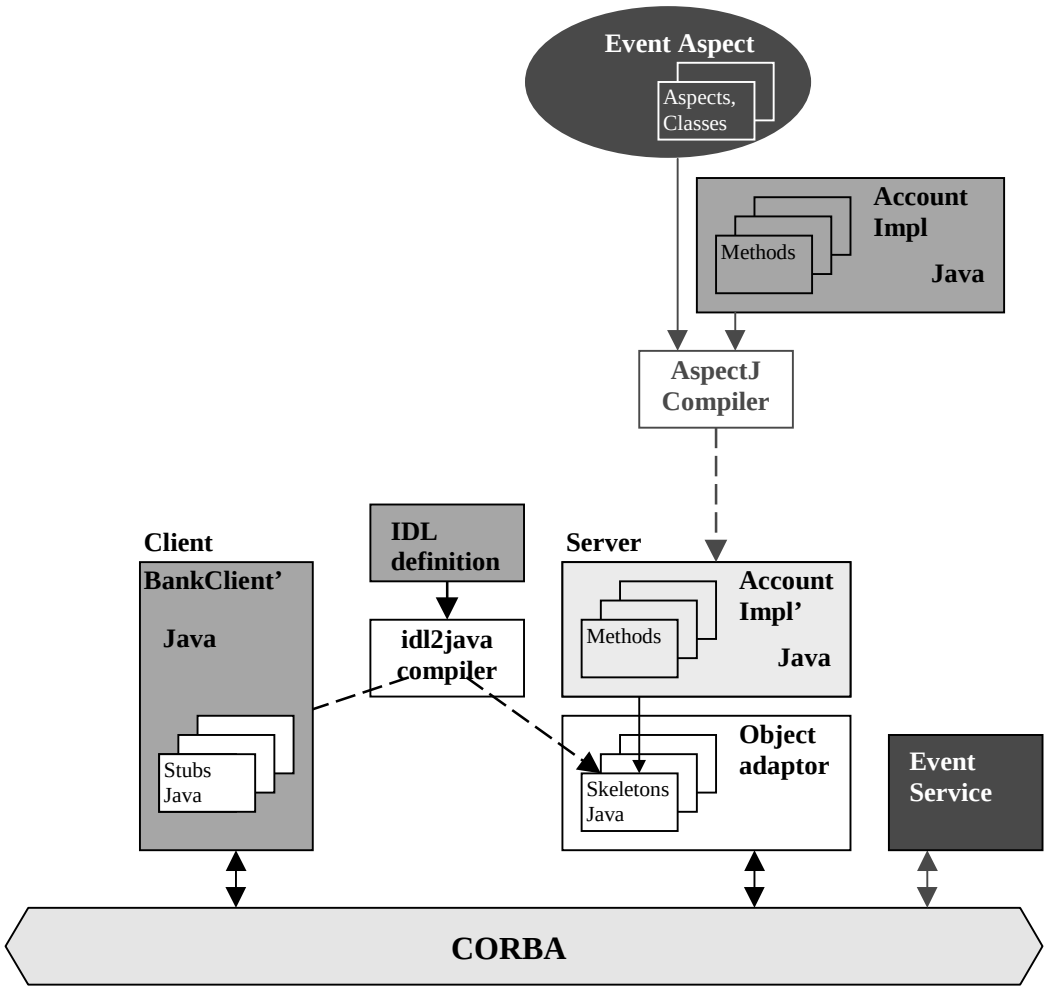


Figure 22 Implementation diagram of the example with the event aspect.

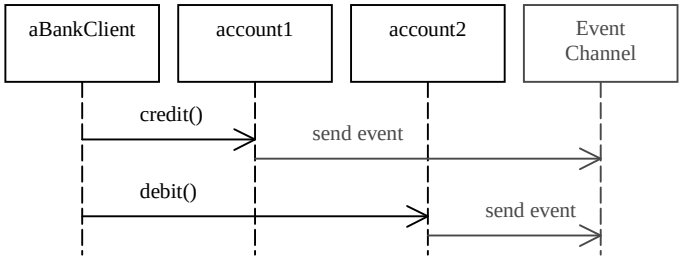


Figure 23 Sequence diagram of the example with the event aspect.

4.2.3 Transaction Aspect

Another interesting Corba Service is the Transaction one. It allows to see a group of actions on different objects as atomic and to restore the initial state in case of error by using rollback mechanism. To stay as simple as possible in this paper, we have chosen to redefine a simple transaction service. Indeed the Corba Object Transaction Service is really large and its use needs deep modification of the IDL interfaces and of the implementation (redirection of attribute accesses). Thus it is not completely definable by using AspectJ.

Our simple transaction aspect allows the user to develop the application without thinking about transactions. Once it is done, he can choose which methods will be transactional (client side) and which object will be recoverable (server side). Here is the code of the client side method making a transfer from an account to another one.

BankClient.makeTransfer

```
Public class BankClient
{
    public static void main( String args[] )
    {
        . . .

        try
        {
            makeTransfer(account1, account2, 400);                          ← a)
        }
        catch (TransferException ex)
        {
            System.out.println("Transfer cancelled: " + ex.reason);
        }
    }

    protected static void makeTransfer(Account accFrom, Account accTo,      ← b)
                int amount) throws TransferException
    {
        try
        {
            accTo.credit(amount);
            accFrom.debit(amount);
        }
        catch ( AccountException ex )
        {
            throw new TransferException("account exception : " + ex.reason);
        }
        catch ( org.omg.CORBA.SystemException ex )
        {
            throw new TransferException("Corba problem");
        }
    }
}
```

a) use the method *makeTransfer;* **b)** *makeTransfer* transfers amount from the account *accFrom* to the account *accTo*.

Now let us look at the aspect adding transactional behaviour to our base example.

SimpleTransServiceAspect.java

```
import java.lang.reflect.*;

public class SimpleTransServiceAspect
{
   // PART I : do transactional one or more given client methods.
   // ------------------------------------------------------------
   crosscut doTransactionnal(): BankClient & * makeTransfer(..);

   crosscut rollbackHandler(AccountImpl ac): ac & * debit(..);

   static advice(): doTransactionnal()
   {
       before {                                                          ← a)
           System.out.println("begin trans");
           SimpleTransServiceSingleton.instance().begin();
       }
       after  {                                                          ← b)
           System.out.println("commit trans");
           SimpleTransServiceSingleton.instance().commit();
       }
   }

   static advice(AccountImpl ac): rollbackHandler(ac)
   {
       catch(AccountException ex) {                                      ← c)
           System.out.println("rollback trans");
           SimpleTransServiceSingleton.instance().rollback();
       }
   }


   // PART II : do recoverable given server side classes.
   // ---------------------------------------------------------
   crosscut recoverableClasses(): AccountImpl;

   crosscut backupMethods(AccountImpl ac): ac & ( * debit(..) |
       * credit(..) );

   static advice(AccountImpl ac): backupMethods(ac)
   {
       before
       {
           if (SimpleTransServiceSingleton.instance().pendingTrans())
           {                                                             ← d)
               SimpleTransServiceSingleton.instance().addRecovObj(ac);
               Ac.backup();
           }
       }
   }

                                                                         ← e)
   // @IDL@ inherit Account: RecovObj @IDL END@ //
                                                                         ← f)
   introduction recoverableClasses() {
       Object[] savedValues = null;

       Public void commit()
       {
           System.out.println("commit");
           SavedValues = new Object[0];
       }

       public void rollback()
       {
           System.out.println("rollback");
```

```
            Restore();
            SavedValues = new Object[0];
        }

        protected void backup()                                    ← g)
        {
            // backup each field by using java.lang.reflect
            . . .
        }

        protected void restore()
        {
            // restore each field by using java.lang.reflect
            . . .
        }
    }
}
```

a) piece of code to add in order to begin the transaction; **b)** to commit the transaction; **c)** and to rollback the transaction; **d)** register the object; **e)** modification of IDL code. The *Account* interface should inherit of *RecovObj* in order to become a recoverable object; **f)** introduce implementation of methods corresponding to the *RecovObj* interface; **g)** add methods based on *java.lang.reflect* to save and restore the object.

RecovObj.idl

```
interface RecovObj
{
    // ends the transaction with success
    void commit();

    // an error has occurred --> restores the previous state
    void rollback();
};
```

This aspect adds to the transactional methods of the client the following calls: *begin* transaction before, *commit* transaction after and *rollback* when an exception happens. The operations done to make an object recoverable are more sophisticated. A new attribute *savedValues* containing a list of objects is added to the target class. It allows the class to keep a copy of its attributes. Moreover methods *commit*, *rollback*, *restore* and *backup* are added. *Restore* and *backup* use the power of *java.lang.reflect* to save in or to obtain from *savedValues* a copy of the attributes values. The objects referenced by another one have to be recoverable too. Finally the attributes are saved at the beginning of *debit* and *credit* methods.

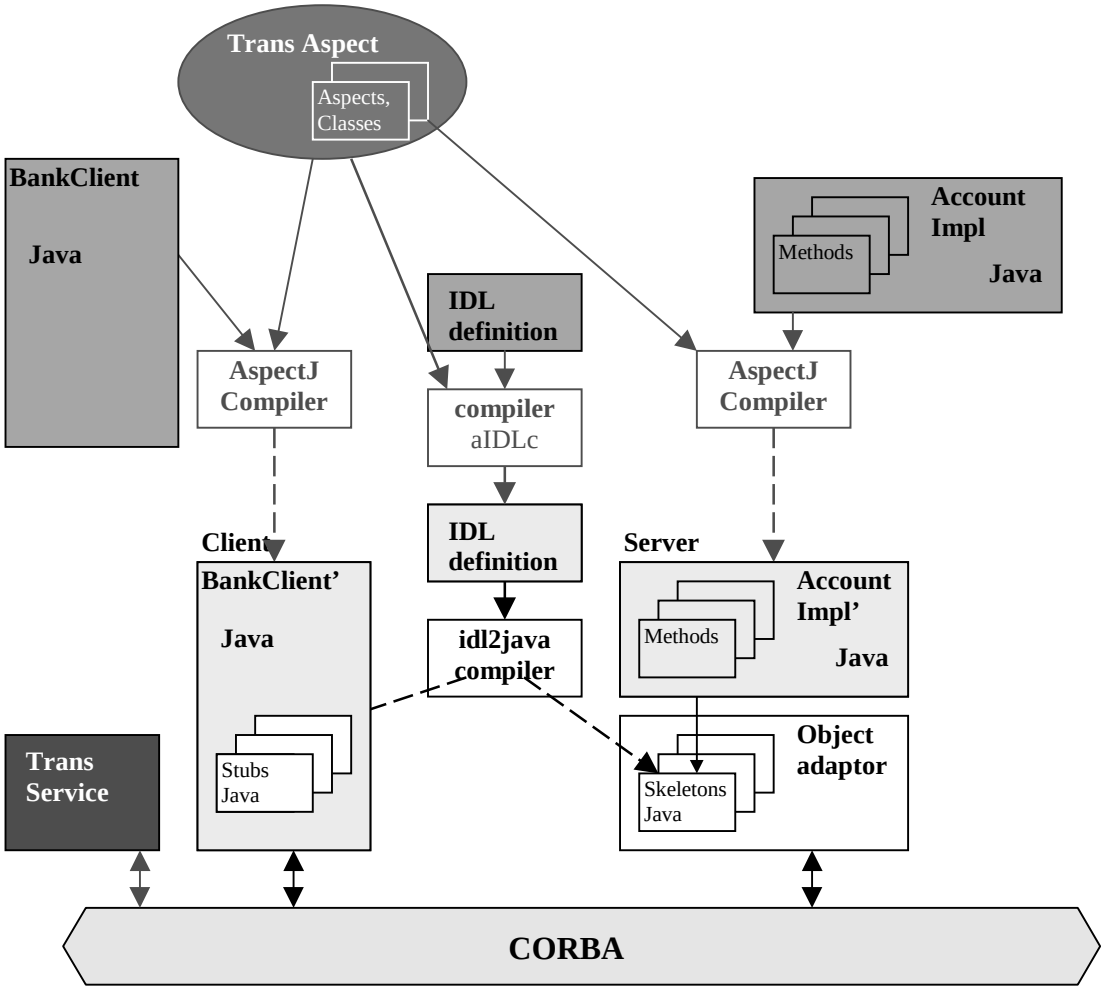


Figure 24 Implementation diagram of the example with the transaction aspect.

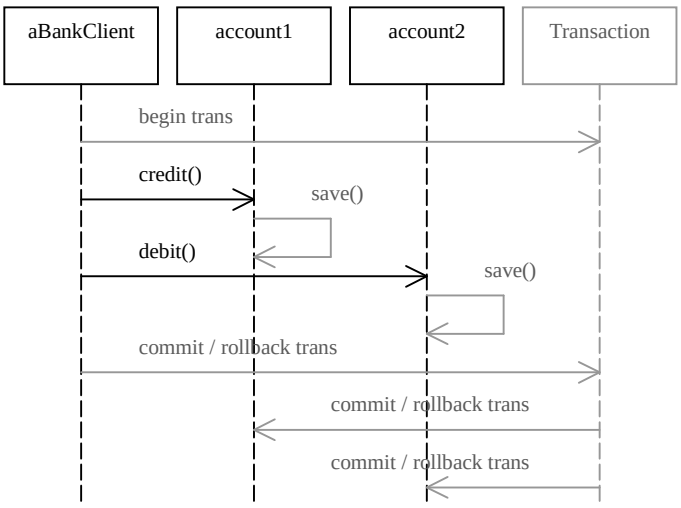


Figure 25 Sequence diagram of the example with the transaction aspect.

4.2.4 Meta level composition aspect

Here is an example of the syntax extension letting define crosscuts on actions. We will show how to compose the two previously defined aspects (event and transaction services).

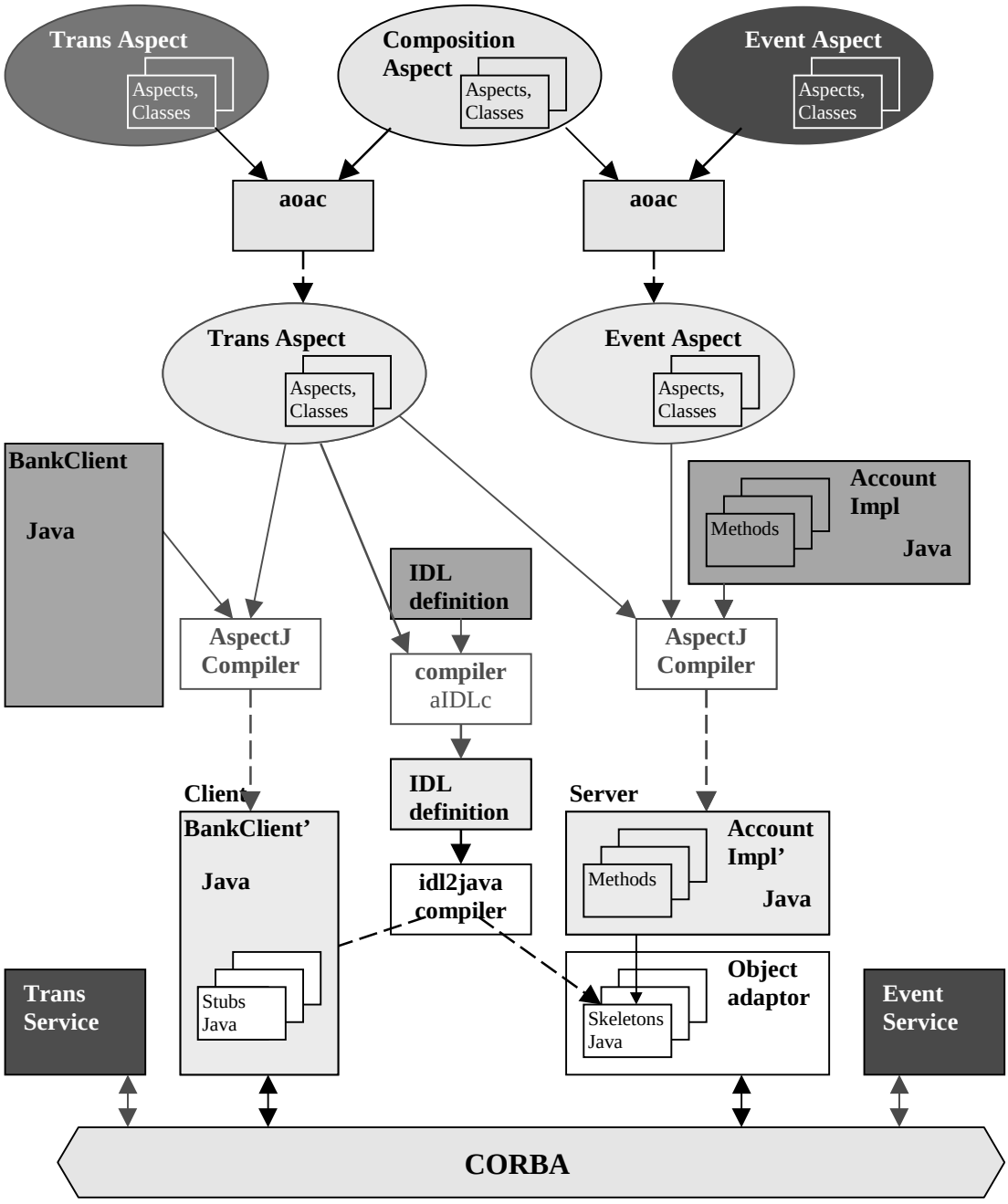


Figure 26 Implementation diagram of the example with both aspects and composition.

Now we can implement the aspect that helps to compose event and transaction aspect. We use the syntax extension of AspectJ defined previously.

CompositionAspect.java (part 1/2)

```
Public class CompositionAspect
{
    // ASPECT ON ASPECT: modify the Transaction aspect to freeze
    // events during the transaction

    crosscut blockEvents(): SimpleTransServiceAspect &                    ← a)
        doTransactionnal.before;

    crosscut sendBlockedEvents(): SimpleTransServiceAspect &              ← b)
        doTransactionnal.after;

    crosscut deleteBlockedEvents(): SimpleTransServiceAspect &            ← c)
        rollbackHandler.catch(..);

    static advice(): blockEvents() {
        before {
            EventSenderSingleton.instance().freezeEvent(); } }            ← d)

    Static advice(): sendBlockedEvents() {
        after {
            EventSenderSingleton.instance().unfreezeEvent(); } }          ← e)

    Static advice(): deleteBlockedEvents() {
        after {
            EventSenderSingleton.instance().removeEvent(); } }            ← f)

    . . .
```

a) captures the event corresponding to the beginning of a transaction **b)** commit; **c)** rollback; **d)** advice corresponding to the crosscut a). Block the events; **e)** send the blocked events; **f)** suppress the blocked events

This first part correspond to the modification of the aspects. The next part is less interesting. It is only needed to modify the event sender to let it keep the events instead of sending them.

CompositionAspect.java (part 2/2)

```
    . . .

    // ASPECT ON CLASSES ASSOCIATED TO ASPECTS: modify the event server
    // by adding a mechanism that can block the events.

    Crosscut initPendingEvent(EventSenderImpl es): es & new(..);          ← g)

    Crosscut replaceSendEvent(EventSenderImpl es, String data): es &      ← h)
        void sendEvent(data);


    // @IDL@ introduction EventSender {void freezeEvent();} @IDL END@     ← i)
    // @IDL@ introduction EventSender {void unfreezeEvent();} @IDL END@
    // @IDL@ introduction EventSender {void removeEvent();} @IDL END@

    introduction EventSenderImpl                                          ← j)
    {
        public void freezeEvent() { . . . }
        public void unfreezeEvent() { . . . }
        public void removeEvent() { . . . }

        String pendingEvent[] = null;                                     ← k)
        boolean frozen;
        void init() { . . . }
    }

    static advice(EventSenderImpl es): initPendingEvent(es)
    {
        before { es.init(); }                                             ← l)
```

```
    }

    static advice(EventSenderImpl es, String data) returns void:                ← m)
        replaceSendEvent(es, data)
    {
        around
        {
            if (es.frozen) {                                                     ← n)
                // save the event
                es.pendingEvent[es.nbPendingEvent] = data;
                es.nbPendingEvent++; }
            else {                                                               ← o)
                // call the initial code to send the event
                thisJoinPoint.runNext(); }
        }
    }
}
```

g) event sender constructor; **h)** sendEvent: the method to modify; **i)** modifications of the eventSender IDL interface; **j)** implementation of the three methods introduced in the interface; **k)** other introductions to keep events instead of sending them; **l)** initialises the introduced attributes; **m)** modify the *sendEvent* method; **n)** saves the event in a table; **o)** uses the initial code of the method → send the event.

We can see that there are **no reference to *Account* or *BankClient* classes** but only modifications of the aspects (*TransAspect*) and classes associated to those ones (*EventSender*). The new grammar allows the definition of new code before or after the "after" block of *doTransactionnal* advice.

4.2.5 Three composition types

We could think that the number of meta-level composition aspects would explode if we provide a library of Corba Services encapsulated in aspects. Indeed it seems that a composition aspect will be needed for each possible aspects couple. In fact there are only few cases where we have such problems. Thus we will provide an aspects library and a tool to let the user customise composition when the default result is not successful. Moreover the solution developed to resolve a specific composition is reusable.

Composition type	Example
Impossible composition	Two services cannot work together because they provide the same service. **Examples**: • Mapping name-IOR: Naming Services and file.
Simple composition	Numerous default compositions fit perfectly to the needed behaviour. Often the aspects have not any side effect and thus can be easily composed. In this case we can use AspectJ without problem. **Examples**: • Event sender and Naming Service • Log repository and Event Sender
Composition needing a "composition aspect"	Sometimes we have to define more precisely the needed behaviour. When the default composition behaviour does not fit the expected one, it is important to have tools allowing the modification of the composition. In this case the aspects are modified by aoac and next AspectJ is used. **Examples**: • Transaction and Event Sender • Log and Duplication

Open points

- The Aspect Oriented Programming is scalable and can be used to define large systems. Thus there is no reason to think that it will be impossible to use this concepts in wider Corba applications. However the mapping of a standardised Corba Service onto an aspect is not easy. When we defined our own services it was easier because during their development we kept in mind that they will be used as aspect.
- The composition of given services can differ according to the aimed applications. However we know that there are not many ways to combine services. Moreover incompatibilities are not common: we can use Corba Event Services and Log Repository or Naming Service without any side effect. We can imagine to develop a library of composition aspects in order to compose the different Corba services defined as aspects. To define how the aspects have to affect an application, the programmer has to use high-level languages, which are specific to the aspects. The composition aspect will only be needed to modify the aspect behaviour at a deeper level.
- The three compilers, Aspect on Aspect compiler (aoac), Aspect IDL compiler (aIDLc) and AspectJ compiler (ajc) should be merged in one. Indeed *aoac* adds principally new crosscut types and *aIDLc* adds only comments. Thus those ones could be introduced in the AspectJ syntax.
- The next release of AspectJ (0.7) should introduce *dominant* that is a way to assign priority to an advice. It is only a little step but could already solve some composition problems.
- It could be interesting to be able to modify only one objects and not all the instances of a class. So it could be possible to have in the same application accounts that send events and other ones that do not send events. It is related to dynamic weaving [17], a way to add dynamically aspects to an object.
- We will try to define the services and entry points in a more abstract way in order to facilitate the composition mechanisms.

Conclusions and future work

We have shown that the Object Oriented paradigm is not the most appropriated way to encapsulate Corba Services. Indeed, this approach does not let define easily reusable services. The reason why we encounter such difficulties is that services crosscut the base application. In other words the services and the base code are tangled and it becomes quickly impossible to have a clear view of which piece of code is related to a given service. The resulting tangled code is difficult to understand, maintain and reuse.

An emerging approach solving that kind of problems is Aspect Oriented Programming. By using this one, it is possible to keep all pieces of code concerning a service as a well-defined entity: an aspect. A weaver (or aspect compiler) is used to build the final application. So it is possible to plug-in services at compile time. Moreover those services can be reused on other applications. We have successfully used AspectJ as a base to encapsulate Corba services in aspects but we have had to define a way letting an aspect modify IDL interfaces. Indeed it is often necessary to add methods to an existing class. We have shown that viewing services (such as: naming, event or transaction, …) as aspects was possible and fruitful.

When the programmer has to use simultaneously some services, a composition problem can occur. Indeed, the result of the default composition is not always the expected one and the programmer should have a way to define how the composition must be done. To do that we have proposed *meta level composition aspects*. In other words we have defined new entry points on aspects themselves and so it becomes possible to define *'meta-aspects'* or '*aspect on aspect*' letting modify other aspects. Thus it is possible to define how the composition of services has to be done. Moreover, the aspects composition problem is not due to Corba environment but is a general lack of aspects. It is the reason why we think it is necessary to introduce a new mechanism like meta-level aspects in order to solve this recurrent problem.

We have presented this case study and our solution during the ECOOP's 2000 workshop on *Aspect and Dimensions of Concern*. The problems related to the composition of aspects being not specific to Corba, a task force have been created to deal with this case. After some hours we foreseen that another promising way to compose aspects and services is the multi-dimensional separation of concerns (Hyper/J [30]). The main idea is to see each concern as a dimension. Like this, the OOP approach has one dimension (objects), the AOP approach has two dimensions (objects and aspects that crosscut those ones) and the hyperspaces has N dimensions. So that we can define the main application, the events and transaction services as three dimensions crosscutting each other. This approach seems promising too but we have to study it more carefully.

There are still numerous searchers (such as Doug Lea) waging a war against AOP concepts because we lose the global view by trying to separate the concerns. So, they say that AOP is not usable in the real world where there are interdependencies between aspects of an application. In other words, it is easy to decompose an application but often too complex to recompose it. We have seen that our solution lets separate the concerns and give a flexible way to compose them. It is at the composition time that the interdependencies between aspects should be expressed.

Finally, Mr E.Hilsdale and Mrs C.Lopes, two researcher of the AspectJ team, told me that meta-level aspects is an interesting idea but not yet specified because their first aim is to provide a complete AOP solution in order to increase the number of AO programmers. Moreover there are not enough potential users of meta-level aspects yet and there are still a lot of challenging problems in AOP (fusion of AOP, SOP and CF, …).

At this point we know that the composition of aspects is really a challenging problem that is far of being completely solved and we think that the meta level approach to deal with the composition of aspects is promising and could be a PhD subject. In conclusion, our arguments are that aspects could only be composed if they are projected in a same meta object protocol.

References

[1] S. CHIBA. 'A Metaobject Protocol for C++'. ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pages 285-299, October 1995.

[2] OMG '*Corba Services book*'. Complete formal/98-12-09., 1998 http://www.omg.org/library/csindx.html

[3] B. GOWING, V. CAHILL. '*Meta-Object Protocols for C++ : The Iguana Approach*' in Proceedings of Reflection '96.

[4] M.GOLM, J.KLEINÖDER. '*metaXa and the Future of Reflection*' . Proc. of the OOPSLA '98 Workshop on Reflective Programming in C++ and Java, Vancouver, BC, 1998

[5] R. ORFALI, D. HARKEY. 'Client/Server Programming with JAVA and Corba'. Second Edition, 1998

[6] S. CHIBA. '*Javassist - A Reflection-based Programming Wizard for Java*' In Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java October, 1998.

[7] DISTRIBUTED OBJECT GROUP. '*JavaORB version 2.2 Programmer's Guide*' DOG, 1999 http://dog.exoffice.com/Projects/JavaORB/javaorb.html

[8] G. KICZALES, C. LOPES. '*Aspect-Oriented Programming with AspectJ, tutorial*' Xerox Parc http://www.parc.xerox.com/csl/projects/aop/

[9] T.J.MOWBRAY, R.C.MALVEAU '*Corba Design Patterns*' Wiley Computer Publishing, 1997

[10] L. BUSSARD, *'Towards a pragmatic composition model of Corba services based on AspectJ'*, ECOOP's 2000 workshop on Aspects and Dimensions of Concerns position paper, May 2000.

[11] K. OSTERMANN, G. KNIESEL, *'Independent Extensibility - an open challenge for AspectJ and Hyper/J'*, , ECOOP's 2000 workshop on Aspects and Dimensions of Concerns position paper, May 2000.

[12] T. LEDOUX. '*OpenCorba: A Reflective Open Broker*'. Reflection'99, Springer-Verlag, LNCS Saint-Malo, France, July 1999

[13] F. M. COSTA, G. S. BLAIR, G. COULSON. '*Experiments with Reflective Middleware*', Proc. ECOOP'98 Workshop on Reflective Object Oriented Programming and Systems, Springer Verlag, 1998

[14] A. BEUGNARD '*How to make aspect reusable, a proposition*', Proceeding of the aspect_Oriented Programming Workshop at ECOOP'99

[15] M. ROMAN F. KON, R. H. CAMPBELL. '*Design and Implementation of Runtime Reflection in Communication Middleware: The dynamicTAO Case*'. ICDCS'99 Workshop on Middleware. Austin, Texas. May 31 - June 5, 1999.

[16] C. PREHOFER. '*An Object-Oriented Approach to Feature Interaction*', Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems, 1997

[17] A.-M. DERY, *'Un haut niveau d'expression de la méta programmation'* ESSI's course, 2000.

[18] K. BÖLLERT, *'On weaving aspects'*, Proceeding of the aspect_Oriented Programming Workshop at ECOOP'99

[19] H. OSSHER, W. HARRISON, F. BUDINSKY, I. SIMMONDS. '*Subject-Oriented Programming: Supporting Decentralized Development of Objects*', IBM. http://www.research.ibm.com/sop/

[20] J.C. FABRE, T. PÉRENNOU '*A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach*', IEEE Transactions on Computers, vol 47, no 1, January 1998.

[21] C.V. LOPES, G. KICZALES '*D : A language framework for distributed programming*', Technical Report SPL97-007 P9710047, Xerox Palo Alto Research Centre, 1997.

[22] L. SEINTURIER '*JST : An Object Synchronisation Aspect for Java*' ECOOP'99 AOP Workshop Position Papers, 1999

[23] K. DE VOLDER, T. D'HONDT '*Aspect Oriented Logic Meta Programming*', Springler 1616 p. 250, Meta-Level Architectures and Reflection, Reflection'99, July 1999.

[24] D. SLAMA, J. GARBIS, P. RUSSELL '*Enterprise Corba'*, Prentice Hall, 1st edition, ISBN: 0130839639, March 1999.

[25] O. JAUTZY, *'Intégration de sources de données hétérogènes : une approche langage'*, Thesis, Ecole nationale des ponts et chaussées, March 2000.

[26] L. BERGER, A.-M. DERY, M. FORNARINO, *'Interactions between objects : an aspect of object-oriented languages'*, ICSE'98 Workshop on Aspect-Oriented Programming (Kyoto, Japan), April 1998

[27] G.KICZALES, J.DES RIVIÈRES, D.G.BOBROW, *'The Art of the Metaobject Protocol'*, MIT Press, Cambridge, MA (1991)

[28] M. FORNARINO, 'Le Standard CORBA', ESSI's course, 2000.

[29] A. SPECK, E. PULVERMÜLLER, M. MEZINI, *'Reusability of Concerns'*, ECOOP's 2000 workshop on Aspects and Dimensions of Concerns position paper, May 2000.

[30] P. TARR, H. OSSHER, *'Hyper/J User and Installation Manuel'*, IBM research, 2000, http://www.research.ibm.com/hyperspace

[31] D. DUGGAN, '*A Mixin-Based, Semantics-Based Approach to Reusing Domain-Specific Programming Languages'*, Springler 1850 p. 179, ECOOP 2000 - Object-Oriented Programming, June 2000.

Annex A : Why AspectJ should deal with interfaces

To avoid the recurrent problems related to multiple inheritance, it has been decided not to allow it in Java. However, to let some classes share characteristics, it is possible to use interfaces. It is a collection of constants and abstracts methods. A class can implement an interface by adding the interface to the class's *implements* clause and overriding the abstract methods defined in the interface. A variable can be declared as an interface type, and all the constants and methods declared in the interface can be accessed from this variable. All objects whose class types implement the interface can then be assigned to this variable.

In Corba, IDL lets define inheritance relationship between IDL interfaces. When C++ code is generated from IDL, the multiple inheritance is used. But when Java code is produced, interface implementation replaces the inheritance.

Unfortunately, AspectJ does not let add the implementation of an interface to an existing class. We think that this possibility should be given. In fact aIDLc resolve this problem in Corba environment but it should be generalised.

Example:

To implement the transaction service we have to find a way to define some classes as recoverable. To do that we choose to define an IDL interface (*RecovObj*) that defines *commit* and *rollback* methods. The aspect has just to define that a given class implements *RecovObj* to do this one recoverable.

If we implement this solution without Corba environment, AspectJ lets us introduce methods such as *commit* and *rollback*. However without having a common interface or ancestor, we have to pass references on those ones as *object*. To call their added methods it is mandatory to use *java.lang.reflect* that allows knowing if the method exists.

Annex B : Aspect on aspect compiler example

The aspect on aspect compiler (aoac) has been defined by using Java compiler compiler (javacc). To have more information about javacc, look at the annex D.3. The following example gives another idea of how it could be used.

B.1 Configuration :

The aspect on aspect compiler has to be produced by using javacc:

```
cd \bussard\javacc\javacc\examples\SimpleExamples\AspectOnAspect2
javacc AspectOnAspect.jj
```

For the next steps the following environment variables have to be set:

```
set PATH=%PATH%;C:\JDK1.2.2\BIN;C:\bussard\javaCC\javaCC\bin;
    C:\aspectj0.6\bin;C:\bussard\javaCC\javaCC\examples\SimpleExamples
    \AspectOnAspect2\bin
set CLASSPATH=.;C:\JDK1.2.2\lib\tools.jar;C:\aspectj0.6\lib\aspectj.jar;
    C:\bussard\javaCC\javaCC\examples\SimpleExamples\AspectOnAspect2
```

B.2 Simple counter example

Counter.java:

```
class Counter
{
   public int value = 0;

   public void init() { value=0; }
   public int inc() { return ++value; }
   public int dec() { return --value; }
}
```

MyTest.java:

```
class MyTest
{
   public static void main(String[] args)
   {
      Counter theCounter = new Counter();

      System.out.println("counter value: ");
      for (int i =0; i < 40; i++) {
         System.out.print(" " + theCounter.inc()); }
   }
}
```

When we try to use this example we have the following result:

cd \bussard\javacc\javacc\examples\SimpleExamples\AspectOnAspect2\example
javac MyTest.java Counter.java
java MyTest

```
counter value:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40
```

Now we will add an aspect which reinitializes the counter when the result is greater than 10.

anAspect.java:

```
public class anAspect
{
   static int valMax = 10;

   // increment of the counter
   crosscut myInc(Counter c): c & int inc(..);

   static advice(Counter c): myInc(c) {
      before {
         // init the counter when the value is equal to valMax.
         if (c.getVal() >= valMax)
         {
            c.init();
         }
      }
   }

   // create a new method in Counter to get access to the private value
   introduction Counter { int getVal() { return value; }}
}
```

we obtain the following result:

```
cd \bussard\javacc\javacc\examples\SimpleExamples\AspectOnAspect2\example
ajc -argfile ajc.lst
java MyTest

counter value:
1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6
7 8 9 10
```

Finally we will modify this aspect to warn the user each time the counter is modified. To do it we will use a new aspect modifying this one.

anAspectOnAspect.java:

```
crosscut warningWhenChange(): anAspect & myInc.before;

static advice(): warningWhenChange()
{
   before{
      int val = c.getVal();
   }

   after{
      if (val != c.getVal())
         System.out.println(" Has been changed !!!");
   }
}
```

we obtain the following result:

```
cd \bussard\javacc\javacc\examples\SimpleExamples\AspectOnAspect2\example
aoac anAspectOnAspect.java aoac.lstajc -argfile ajc.lst
cd result
ajc -argfile ajc.lst
java MyTest

counter value:
 1 2 3 4 5 6 7 8 9 10 Has been changed !!!
 1 2 3 4 5 6 7 8 9 10 Has been changed !!!
 1 2 3 4 5 6 7 8 9 10 Has been changed !!!
 1 2 3 4 5 6 7 8 9 10
```

NB: we cannot provide this new behaviour by adding another aspect modifying the counter class (before and after *inc* method). Indeed we have to test the modifications of the counter value done by the aspect but not the ones that are done by incrementing the counter.

The modified aspect file has the following look:

Result\anAspect.java:

```
public class anAspect
{
   static int valMax = 10;

   // increment of the counter
   crosscut myInc(Counter c): c & int inc(..);

   static advice(Counter c): myInc(c) {
      before {
         // **** this code has been added by AspectOnAspect ****
         // **** BEGIN ****
         int val = c.getVal();
         // **** END ****

         // init the counter when the value is equal to valMax.
         if (c.getVal() >= valMax) {
            c.init(); }


         // **** this code has been added by AspectOnAspect ****
         // **** BEGIN ****
         if (val != c.getVal())
            System.out.println(" Has been changed !!!");
         // **** END ****
      }
   }
   // create a new method in Counter to get access to the private value
   introduction Counter { int getVal() { return value; }}
}
```

B.3 Clean hierarchy

We recommend using aspect on aspect to modify aspect and associated classes. Often the aspects have to create new object and thus have to define new classes. The aspect on aspect can modify those classes. However the aspect on aspect should not directly affect base classes. Like this we can define a clean hierarchy with three levels: classes, aspects and aspects on aspect.

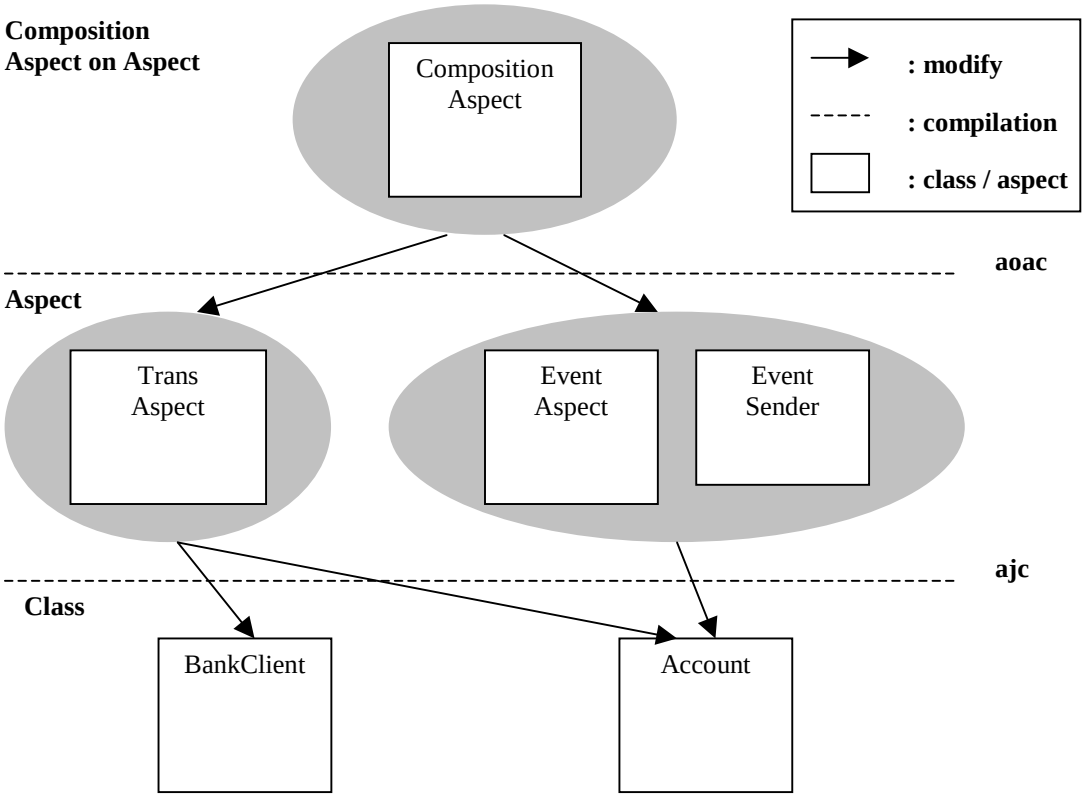


Figure 27 The three levels model: classes, aspects and composition aspects.

Annex C : RMI over IIOP example

I have tried to use RMI over IIOP because it is a possibility to avoid IDL interfaces and have only to deal with Java interfaces. Like this it is possible to change the interfaces from AspectJ.

This simple example is a counter with only one method *nextVal*, which returns an incremented value. We will add a new method *init*, which allows the initialisation of the counter. Finally we will implement this method and call it from the client.

Counter.java

```
interface Counter extends java.rmi.Remote
{
   public int nextVal() throws java.rmi.RemoteException;
}
```

Server.java

```
import javax.naming.InitialContext;

public class Server extends javax.rmi.PortableRemoteObject implements Counter
{
   public Server() throws java.rmi.RemoteException
   { }

   public int nextVal() throws java.rmi.RemoteException
   {
      counterVal++;
      System.out.println("nextVal returns " + counterVal);
      return(counterVal);
   }

   protected static int counterVal = 0;

   public static void main(String args[])
   {
      try
      {
         java.util.Properties env = new java.util.Properties ();
         InitialContext context = new InitialContext(env);

         Server counterObj = new Server();

         context.bind("counter", counterObj );

      }
      catch ( Exception ex )
      {
         ex.printStackTrace();
         System.out.println(ex.getMessage());
      }
   }
}
```

CorbaClient.java

```
public class CorbaClient
{
   public static corba_pkg.Counter counter = null;

   public static void main( String args[] )
   {
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      org.omg.CORBA.Object obj = null;
      org.omg.CosNaming.NamingContext naming = null;
      try
      {
```

```
         obj = orb.resolve_initial_references("NameService");
         naming = org.omg.CosNaming.NamingContextHelper.narrow(obj);
      }
      catch ( org.omg.CORBA.ORBPackage.InvalidName name )
      {
         System.out.println("Enable to resolve NameService");
         System.exit(0);
      }

      org.omg.CosNaming.NameComponent [] name = new
                                       org.omg.CosNaming.NameComponent[1];
      name[0] = new org.omg.CosNaming.NameComponent();
      name[0].id = "counter";
      name[0].kind = "";

      try
      {
         obj = naming.resolve(name);
      }
      catch ( org.omg.CosNaming.NamingContextPackage.NotFound ex )

      {
         System.out.println("Object not found");
         System.exit(0);
      }
      catch ( org.omg.CosNaming.NamingContextPackage.CannotProceed ex )
      {
         System.out.println("Cannot proceed");
         System.exit(0);
      }
      catch ( org.omg.CosNaming.NamingContextPackage.InvalidName ex )
      {
         System.out.println("Invalid name");
         System.exit(0);
      }

      counter = corba_pkg.CounterHelper.narrow(obj);

      try
      {
         System.out.println("CORBA client, current counter val:");
         for (int i=0; i<40; i++)
            System.out.print(counter.nextVal() + " ");
      }
      catch ( org.omg.CORBA.SystemException ex )
      {
         System.out.println("A CORBA System exception has been intercepted");
         System.out.println(ex);
      }
   }
}
```

ReInitAspect1.java

```
class ReInitAspect1
{
  // Counter

  introduction Server {
     public void init() throws java.rmi.RemoteException { System.out.println("Init
counter"); counterVal = 0; } }

  introduction Counter {
     public abstract void init() throws java.rmi.RemoteException; }
}
```

ReInitAspect2.java

```
class ReInitAspect2
{
   // RMI over IIOP client

   crosscut useInit(): CorbaClient & void main(..);

   static advice(): useInit()
   {
      after {
         try {
            CorbaClient.counter.init(); }
         catch ( org.omg.CORBA.SystemException ex )
         {
            System.out.println(ex);
         }
      }
   }
}
```

C.1 Configuration and environment variables.

```
set PATH=%PATH%;C:\WINDOWS;C:\WINDOWS\COMMAND;C:\;C:\JDK1.2.2\BIN;
    C:\JavaORB\bin;C:\JavaORB\idl;C:\aspectj0.6\bin
set CLASSPATH=.;.\corba_pkg;C:\JDK1.2.2\lib\tools.jar;
    C:\JavaORB\lib\JavaORBv2_2_4.jar;C:\JavaORB\lib\RMIoverJavaORB.jar;
    C:\javaORB\lib\jndi.jar;C:\javaORB\lib\providerutil.jar;
    C:\aspectj0.6\lib\aspectj.jar
set IDL_DIR=-IC:\JavaORB\idl
```

C.2 RMI over IIOP called by Corba (without aspect)

```
   cd \JavaORB\examples\javaToIDL\myCounter
   javac Server.java Counter.java
   java2idl Counter -tie
   javac *_Tie.java
   idl2java _Counter.idl -I/javaorb/idl/rmi
   javac corba_pkg\*.java
   javac CorbaClient.java

   naming
   java -DJAVA_ORB_DIR=C:\JavaORB
        -Djava.naming.factory.initial=JavaORB.jndi.CtxFactory Server
   java -DJAVA_ORB_DIR=C:\JavaORB CorbaClient
```

CORBA client, current counter val:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40

CORBA client, current counter val:
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

C.3 RMI over IIOP called by Corba (with aspect)

```
   cd \javaORB\examples\javaToIDL\myCounter
   ajc Server.java Counter.java ReInitAspect1.java
   cd ajworkingdir
   javac Server.java Counter.java
   java2idl Counter -tie
   javac *_Tie.java

   naming
   java -DJAVA_ORB_DIR=C:\JavaORB
        -Djava.naming.factory.initial=JavaORB.jndi.CtxFactory Server
```

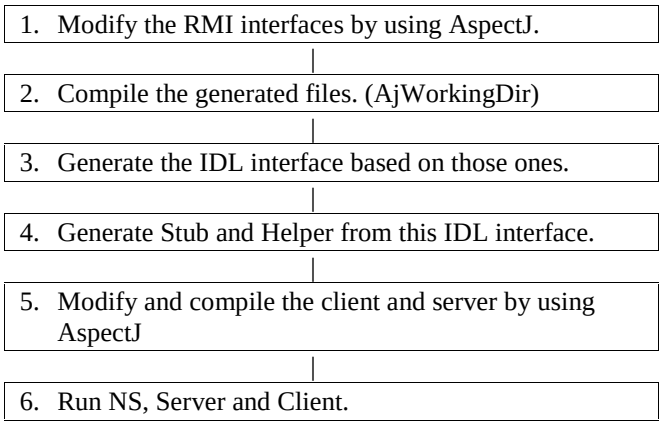
```
mv _Counter.idl ..
cd ..
idl2java _Counter.idl -I/javaorb/idl/rmi
javac corba_pkg\*.java
ajc CorbaClient.java ReInitAspect2.java

java -DJAVA_ORB_DIR=C:\JavaORB CorbaClient
```

```
CORBA client, current counter val:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40

CORBA client, current counter val:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40
```

C.4 two steps compilation



NB: we can see that AspectJ compiler (ajc) is needed twice in order to modify interfaces and next to modify and compile the client and server files. It's the reason why two aspect files are defined in this last example.

Annex D : Installation of the used software

I have chosen to work only with free software written in Java.

D.1 AspectJ

What is AspectJ?	AspectJ is a general-purpose aspect-oriented extension to Java.
Version	0.6
Where can we find AspectJ?	Xerox Parc : http://aspectj.org/
What are its features?	AspectJ extends Java with a new kind of module, called an *aspect*. Aspects have the ability to crosscut classes, interfaces and other aspects. This means that the code in a single aspect can span multiple classes (or interface or aspects). For example, a single aspect can contain all the code for a protocol in which multiple classes participate. Aspects improve separation of concerns by making it possible to cleanly localize crosscutting design concerns.

D.2 JavaORB

What is JavaORB?	JavaORB is a free implementation of CORBA 2.3 that provides numerous features and services. Its Java sources are available. [5]
Where can we find JavaORB?	Distributed Object Group (DOG) : http://dog.exoffice.com/Projects/JavaORB/javaorb.html
What are its features?	• Mapping 2.3 • POA and BOA • Portable stubs and skeletons (dynamic and stream based) • Interceptors (request and message levels) • DII, DSI, DynAny, • Object by value, • IIOP 1.2 (supports IIOP 1.0 & IIOP 1.1), • BiDirectional GIOP, • IDL reflection (re-use the JavaORB IDL parser), • Several threads policies, • Interface repository, • IDL compiler (supports JavaDoc Tags), • IDL to HTML , • IDL to RTF • supports user protocols, • provides a daemon for activation on demand, • Several CORBA services (*Interoperable Naming Service*, Event Service, Transactions Service, Property Service, Collection Service, Notification Service, Time Service and Persistent State Service) • RMI over JavaORB, a fully compliant implementation of RMI/IIOP and RMI/IDL. • JavaORB is compatible with JDK 1.1.x and 2.

D.3 JavaCC

What is JavaCC?	Java Compiler Compiler (JavaCC) is a parser generator.
Version	1.0
Where can we find JavaCC?	SUN Microsystems http://www.metamata.com/JavaCC/index.html
What are its features?	Java Compiler Compiler (JavaCC) is currently the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognise matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building, actions, debugging, etc. JavaCC comes with a bunch of grammars including both Java 1.0.2 and Java 1.1 as well as a couple of HTML grammars.