

## Distribution and aspects

Hafedh Mili, Hamid Mcheick, and Salah Sadou†

Dépt d'Informatique, UQÀM, C. P. 8888, Succ. Centre-Ville, Montréal (P.Q.) H3C 3P8, Canada  
Laboratoire VALORIA, Université de Bretagne Sud, Vannes, France  
{Hafedh.Mili@,{mcheick,sadou}@larc.info.}uqam.ca

### Abstract

The separation of concerns, as a conceptual tool, enables us to manage the complexity of the software systems that we develop. Such was the intent behind the OORAM [Reenskaugh et al., 1995]. When the idea is taken further to software packaging, greater reuse and maintainability are achieved. There have been a number of approaches aimed at modularizing software around the natural boundaries of the various concerns, including subject-oriented programming [Harrison & Ossher, 93] aspect-oriented programming [Kiczales et al., 97], and our own view-oriented programming [Mili et al., 99]. The same applications that warrant the kind of separation supported by the above techniques tend also to be distributed where different users may be interested in different aspects of the application at different times. In this paper, we look at distribution in the context of the separation of concerns, and present an approach to distributing objects that embed different aspects.

## 1 Introduction

In the real world, objects change roles during their lifetime. Generally speaking, we need a mechanism for allowing objects to change behavior during their lifetime, specifically when that change takes place within the *same* program run. In the context of a distributed application, different sites, and different users within the same site, may see different aspects of the same objects, including different functionalities, different access rights and privileges, different quality of service parameters, and so forth. The transition from analysis to design consists of deriving an implementation of the functionalities specified at analysis time in a way that satisfies design-level constraints and properties and addresses design-level concerns. Addressing these concerns usually means adding code that crosscuts normal modularization boundaries, i.e. typically objects and methods.

These are but three of the most common situations requiring us to modularize programs along dimensions other than the traditional function, class, or method, inherent in both procedural and object-oriented programming.

There have been a number of approaches to providing language-level support for separation of concerns in the OO research community (*subjects* [Harrison&Ossher, 93], *aspects* [Kiczales et al., 1997], *views* [Mili et al., 99-01],

and others). Each one of these approaches was intended to solve one particular set of problems related to the three mentioned above.

It turns out that the same applications that warrant the use of separation of concern techniques also tend to be distributed and offer different sets of functionalities to different user communities. Thus, we have a situation where:

1. Objects acquire and lose behavior dynamically (the *dynamic behavior change problem*),
2. Objects offer different sets of behaviors to different client programs simultaneously (the *multiple interface problem*), and
3. (Server) objects and client programs are distributed (the *distribution problem*).

We could treat the different problems, if there are no interactions between the three aspects, or try to find a global solution that accommodates all three requirements in an optimal fashion. In this paper, we propose an approach that handles all three requirements in a unified framework.

We assume that readers are familiar with AOP, subject-oriented programming/HyperJ ([Ossher et al., 95], [Tarr et al., 98]). We provide a brief overview of view oriented programming in section 2. Section 3 explores distribution issues in the context of these methods. Section 4 describes the principles underlying our approach. We conclude in section 5.

## 2 View oriented programming in a nutshell

We view each object of an application as a set of core functionalities that are available, directly or indirectly, to all the users of the object, and a set of interfaces that are specific to particular uses, and which may be added or removed during run-time. The interfaces may correspond to different types of users with similar functional interests or to different users with *different* functional interests. We set out to provide support for the following:

- Enable client programs to access several functional areas or views simultaneously,
1. Support the addition and removal of views (functional slices) during run-time, making objects support different interfaces during run-time, and
  2. Have a consistent and unencumbered protocol to address objects that support views.

Figure 1 shows an aggregation-based implementation of this idea. The dashed object boundary (rectangle) represents our *abstraction* of an application object: it consists of the combination of the core instance and the views. In this example, the core object includes two state variables ('a' and 'b'), and supports three operations (f(), g(), and h()). The view objects, which point to the core object, may add state ('c' for view 1 and 'd' for view 2), behavior (i(...) for view 1, j(...) for view 2, and k(...) for views 1 and 2), and delegate shared data and behavior. In this case, each behavior invocation goes to the components that implement it, i.e. either the core object or the views. The application object is seen as supporting the *union* of behaviors of the core instance and of the currently attached views.

Roughly speaking, our approach consists of putting a wrapper around the core object and its views, which will intercept the various behavior invocations and dispatch them to the appropriate components. A major concern for our approach has been to not overburden developers who wish to take advantage of the view mechanism, with additional language constructs and novel thought processes: support for views has to be seamless. Accordingly, our implementation is based on:

- Providing an API for manipulating views during run-time (adding and removing, activating and deactivation)
- Transforming code that uses views by replacing simple (core object) references by references to the wrapper, when needed.

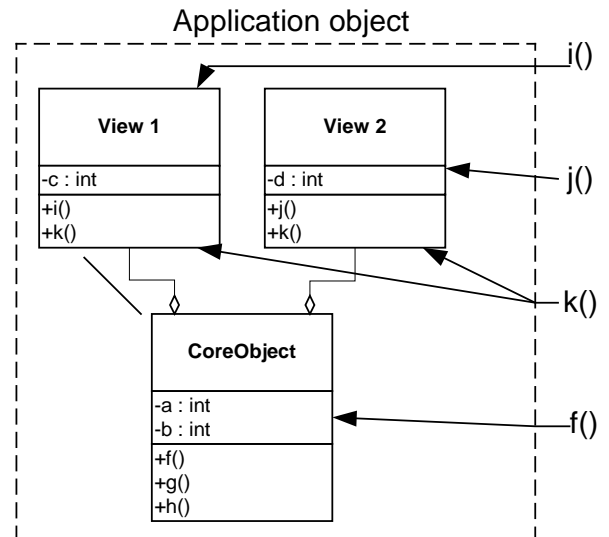


Figure 1. An object with two views.

The following code excerpts illustrate the idea. Assume that we are a merchandising company that manages a fleet of trucks. Different business functions view trucks differently. Operations see them as machine-like resources that need to be scheduled, and for which we need to schedule operators. Accounting view them as amortizable assets whose depreciation may be matched against earnings, reducing taxable income. And then, there is Inventory, which is concerned with the basic properties of a truck, such as the serial number, the make, model year, and the like. The following code excerpts show how a developer might write an application that supports different sets of functionalities at different times.

```

import com.walmart.core.Truck;
import com.walmart.finance.*;
import com.walmart.operations.*;
import com.walmart.maintenance.*;
(1) Truck myTruck = new Truck(id);
(2) myTruck.attach("Finance");
(3) myTruck.attach("Operations");

```

```
(4) float val =
    myTruck.getResidualValueOn(dt1);
(5) myTruck->releaseOn();
```

The truck is created by instantiating the core class (Truck, line (1)). Lines (2) and (3) add two functional aspects/concerns to the core object, corresponding to the views “Finance” (amortization, residual value, etc) and operations (scheduling, routing, etc.). Line (4) invokes a behavior that is supposed to be only part of the “Finance” view. Line (5) invokes a behavior defined by two views, and so a combination of the various implementations must be invoked.

In order for the above code to exhibit the behavior we just described, this code is transformed by replacing references to the core object by references to its “wrapper”, which takes care of dispatching calls according to the number and state (i.e. active or inactive) of the views attached to the core object.

It has been our experience that in business information systems, the roles played by domain objects often correspond to generic business processes, and do not depend on the business domain. Using our model of view programming, the different roles that an application object can play will be represented by views. We propose a kind of a template for functional roles/views that is parameterized by those elements of the interface of the core object that are *required* by the functional role. This template is called *viewpoint*. The *views* are then instantiations of the *viewpoint* for the *core class*, like Büchi & Weck’s generic wrappers [Büchi & Weck, 00].

### 3 Distribution issues

The combination of aspects and distribution is interesting for three reasons:

- 1) Distribution is, itself, one of those design aspects that crosscut implementation classes, and that clutter the code without bringing in any new user-defined functionality. It would thus seem to be a perfect fit for a technique such as aspect-oriented programming, which appears to be particularly well suited for separating design-level concerns.
- 2) Depending on the separation of concerns technique, objects that embody several

concerns may be fragmented, which may raise a number of issues for distribution,

- 3) Considering that different functional areas usually imply different data ownership and use privileges, to what extent can aspect, role, or view boundaries can be used as units for distribution—and thus duplication—in a distributed application context.

We look at the three questions in turn.

#### 3.1 Implementing distribution with separation of concerns techniques

The question here is whether distribution logic can be encapsulated in components along the boundaries of the various separation-of-concerns techniques. There are two sides to this issue. First, we have to figure out where, in an object-oriented program, does distribution makes a difference, i.e. what needs to be changed to turn a non-distributed application into a distributed one. Once we do this, we then have to analyze the various separation-of-concerns techniques to figure out which method’s *abstraction boundaries* [Mili et al., 01] best match the required changes to accommodate distribution.

Turning a regular application into a distributed one is, for the most part, a solved problem. Existing distribution frameworks all use a variant of the proxy pattern, and various compilers will automatically generate most of the code involved in “distributing” objects. There remain a few changes that need to be accommodated:

- Lifecycle issues: the “creation” of remote objects is different from that of local objects.
- Handling remote exceptions: remote method invocations may raise a number of exceptions that may either be related directly to the remoteness of objects, or that may be re-castings of user-defined exceptions.

Both of these changes are to take place in the client program, but they can occur anywhere within a method.

Both subject-oriented programming and view-oriented programming allow composition only at the method level. Only AOP supports composition at sub-method levels, with some restrictions. Thus, aspect-oriented programming seems to be the best fit for handling these kinds

of aspects, on demand. As we later see, we used AOP to introduce multi-aspect logic into distribution (CORBA) logic.

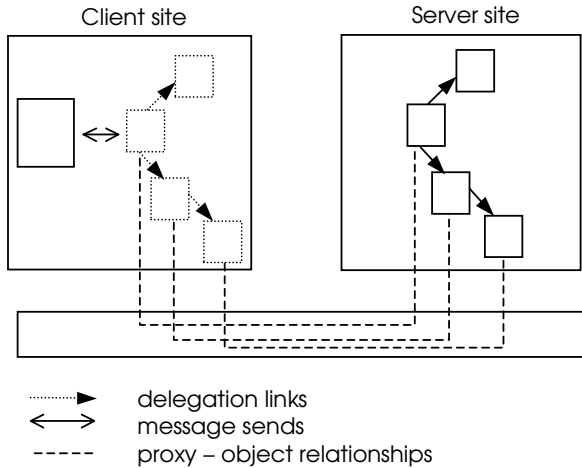


Figure 2. The entire delegation machinery is distributed

### 3.2 Distributing multi-aspect objects

The effect of distribution on objects that embody several aspects or concerns depends on the separation of concerns technique that we used in the first place. If the method involves compile-time integration of the various aspects, then there is no interaction between separation of concerns and distribution since the “multi-aspect” objects look no different from regular objects, and the same distribution issues will be raised.

In those methods where different concerns are represented as separate objects (our approach and dynamic adapters ([Knisel,99] and [Büchi & Weck, 00])), then there are two strategies, and corresponding host of issues to be addressed.

The first strategy consists of separating the multi-aspect “aspect” from the distribution aspect, and distributing multi-aspect objects like any network of related objects. In other words, a core object and all its appendages will be defined as remote objects, with their own interfaces, proxies, data holders, etc. Figure 2 illustrates this strategy.

While the simplicity of this approach may be appealing, it suffers from a lot of problems. Consider the following (naïve) delegation-based dispatch algorithm:

```

perform(Message m, Target o) {
    Method meth = o.lookup(m);
    if ( meth = null) then
        deleg <- o.getDelegates();
        while (meth != null) do
            meth=deleg.next().lookup(m)
        enddo
    endif
    if (meth != null) then
        meth.invoke(o,m)
    else o.doesNotUnderstand(m)
    endif
end perform

```

If we transpose this algorithm into the distribution context, it seems natural that the method invocation itself (`meth.invoke(o,m)`) would take place on the server side. However, it is not clear whether the method look-up itself happens on the client side or on the server side. If it happens on the client side, then a lots of network traffic will be generated to *resolve* a single message send. Further, the client-side proxies will not be “light clients”, but will have to implement some of the processing logic. Thus, it seems more reasonable to implement the delegation-based dispatch on the server side. If the delegates exist only as stores of state and behavior for the delegator, then we may want to forget about “remoting” the entire structure, and let the dispatching happen on the server side.

The second strategy for distributing multi-aspect objects consists of tackling both problems simultaneously. There are two major advantages to using this second strategy:

- *Conceptually*, we could use the abstraction mechanisms provided by the distribution infrastructure, such as the separation of interfaces from implementation, to hide some of the conceptual complexities of supporting multi-aspect objects,
- *Performance-wise*, combine distribution required dispatching with multi-aspect required dispatching, reducing dispatch complexity and performance overhead.

Figure 3 illustrates this second strategy. From the client side, the composite object looks like a single monolithic object.

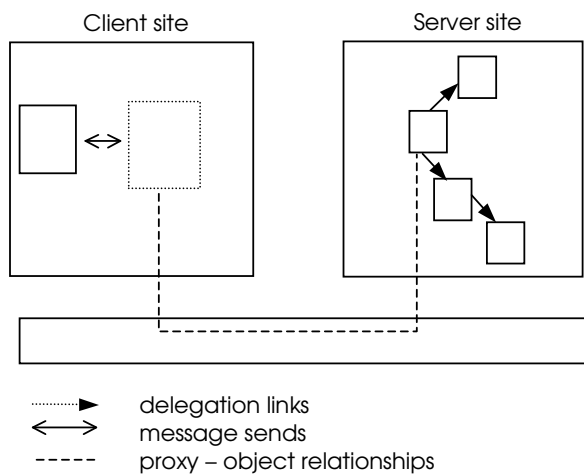


Figure 3. The client side sees a single object. The delegation machinery is hidden on the server's

This approach does have a disadvantage, though. We may be sacrificing the dynamic interface evolution of client-side proxies, unless we resort to using the dynamic invocation interface on the client as well. This is the approach that we have taken in our work, as explained in the next section.

#### 4 Views and distribution

In this section, we are interested in the situation where an object with views is distributed. In the most general case, it is conceivable that views, embodying function specific state and behavior, may be owned by different sites, and may thus reside in different sites. Further, different client sites may have access to different sets of functionalities.

We address our model of view programming from the perspective of a CORBA/RMI-like model where a single state-holding copy of an object is available over the network whereas different proxies/stubs route requests to that object through ORBs.

Because support for views and support for distribution *both* involve, 1) code transformations, and 2) providing proxies<sup>1</sup> for implementation objects, we decided to combine the two processes together. A *distributed object configurator* (DOC) enables architects to select,

<sup>1</sup> As mentioned in section 2, an object with several views is wrapped by a *composition view* that dispatches method calls appropriately

from a set of interfaces and a set of sites, which interfaces are going to be visible to which sites, and which sites will implement which interfaces. We next look at the simplest case where all the views reside in the same site.

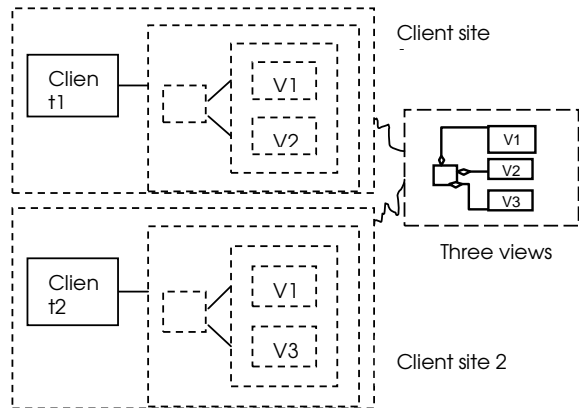


Figure 4. A single-server, multiple-clients.

#### 4.1 Single server, multiple clients

Figure 4 shows an example situation where several views reside on the same server, but different client sites see different subsets of those views. There are two issues that need to be addressed:

1. How to make the same server object implement two or more client interfaces, and
2. Where to handle the dispatch of methods implemented by several views or by the core class and one or more views.

We look at these issues in turn.

**Implementing several interfaces.** Existing CORBA products generate, from the same IDL interface, a client stub and a server skeleton that is supposed to be specialized or somehow refined/completed to provide the full implementation of the object. There are two approaches to server object implementation, one based on inheritance, and the other based on forwarding. With the inheritance-based approach, the object implementation must inherit from the generated skeleton. With the forwarding-based approach (also referred to as "tie approach"), a subclass of the generated skeleton forwards method calls to an object. In a language that supports multiple inheritance, both approaches allow a class to implement several interfaces. In Java, the same class can support several

interfaces, which makes the tie approach appropriate for implementing several interfaces with the same Java class. Figure 5 illustrates this.

A final point has to do with view creation and destruction. The first call to attach(<viewpoint name>) creates and attaches a view, and subsequent calls have no effect. The same goes for requests to detach (destroy) a view. With server objects handling multiple interfaces, we have to keep a count of the number of clients that access a given interface, much like COM does. However, we have to make sure that several requests to detach a view that originate from the same site will count as one. For the time being, we have adopted a simple and somewhat naïve two-stage reference count strategy: the tie object maintains its own reference count. That count is incremented whenever a new attach(...) request is forwarded from the client side, and decremented whenever a new detach(...) request comes from that site. Similarly, the shared object implementation maintains its own count of the various views, which indicates how many server interfaces need a particular view. A tie object no longer needs a view when *its* reference count goes to zero.

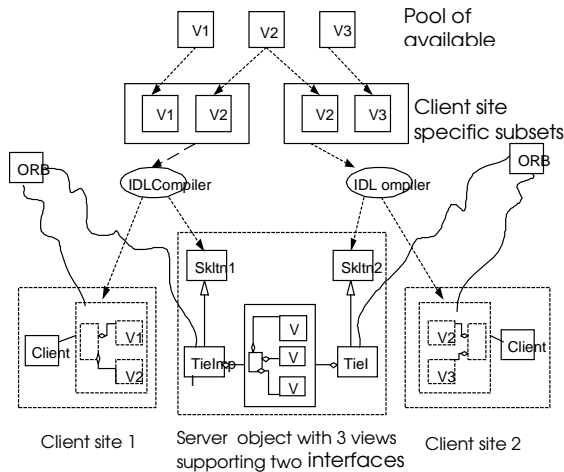


Figure 5. Server Object(s) implement several client interfaces.

Things get complicated when several client process use the same interface, and thus share the same server-side tie object. This could cause dangling references if some client site requests more 'detach(...)' than it had requested attach(...), thus inadvertently making a view unavailable to

other clients who still need it. This could call for a per-client site management of reference counts on tie objects. Further, calls to detach may fail to complete (remote exceptions), leaving the system in an incoherent state. We are currently studying some distributed garbage collection algorithms to handle view lifecycle management.

**Dispatching to multiply-implemented methods.** In a proxy-based implementation of distribution, the client side code only forwards requests to the server side code. When dealing with objects with views, we know that some method calls won't be answered if at the time the call is made, the server object does not have the corresponding view. We have the option of simply forwarding method calls to the server, and let the server side dispatch method calls or raise exceptions if a method is not currently supported. Alternatively, we could handle the dispatch on the client side, and then ensure that any call that goes to the server will get answered.

The first solution has the advantage of simplicity, but can be costly, performance-wise, depending on the relative frequency of failing method calls. The second alternative has the advantage of distributing the dispatching between client and server, and obviating the need for an expensive round-trip in those cases where the method called is not supported. The second reason why we might still need client-side view management, anyway: if we have two client programs that use the same interface (and thus, refer to the same server-side tie object) but that may use different view activations: we need to have a *per-proxy* view management.

With client-side method dispatching, the client side stub is similar to the "wrapper" described earlier in the sense that it has the combined interface of the core object and the available views; based on the views currently active on the object, it may dispatch to different server side method combinations. Those method combinations will have different names generated using some mangling scheme.

## 4.2 Distributed Object Configurator

We are currently implementing support for multi-aspect objects in a distributed environment using a combination of view programming and

CORBA/Java (Iona's OrbixWeb™). A tool called *Distributed Object Configurator* (DOC) enables application architects to configure a distributed application by specifying:

1. Which functional aspects (views) are needed overall? These views are generated by instantiating *viewpoints* for core objects.
2. Where do the various object components (core object and views) reside, and
3. Which combinations of views to support at which client sites,

Viewpoints are described using our own viewpoint syntax. The instantiation of a viewpoint for a core class generates two entities, 1) a CORBA IDL interface, used by DOC, and 2) a Java class representing the view.

As mentioned earlier, support for distributed multi-aspect objects involves the following changes, as compared to plain distribution, 1) the implementation objects are "regular" objects with views, i.e. using an aggregation-based simulation of delegation, and 2) some view-specific processing at both the stub and skeleton. This view specific processing consists, on the client side, of three changes:

- Addition of new infrastructure remote methods (e.g. attach/detach view).
- Addition of new local methods and variables
- Modification of dispatch of view-defined methods.

Server-side changes include support for attach/detach and the corresponding reference counting logic.

We considered rewriting the CORBA IDL compiler to generate the behavior we want, for both stubs (clients) and skeletons (servers). While such a compiler is fairly simple to implement, this seemed to be the kind of situation for which aspect-oriented programming is particularly suited. Hence, we decided to take the code generated by OrbixWeb's idl compiler, and weave into it aspects that handle our own brand of multi-aspect objects.

## 5 Conclusion

Our work addresses the problem of supporting several functional domains within the same application, by composing at will functional fragments developed by independent third

parties. Those same situations that require, or could use, decentralized development of functional domains also require distributed ownership of the functional domain data, and distributed execution of the resulting programs. View programming seems like a perfect fit to the extent that we have resolved most of the issues dealing with the uniqueness of object reference, and the multiple dispatch of methods. There remain a number of issues dealing with optimizing the implementation of distributed view programming which we continue to explore, both theoretically and empirically.

**Acknowledgments:** This work was sponsored the SYNERGIE initiative (Quebec), by NSERC (Canada), and by YAGO Tech.

## References

- [Büchi & Weck, 00] M. Büchi and W. Weck, "Generic Wrappers", in ECOOP 2000, LNCS 1850, pp. 201–225, 2000.
- [Gamma et al., 95] Erich Gamma et al. *Design Patterns –Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.
- [Harrison & Ossher, 93] W. Harrison and H. Ossher, "Subject-oriented programming: a critique of pure objects," in *Proc. of OOPSLA '93*, pp. 411-428.
- [Kiczales et al., 1997] G. Kiczales et al., "Aspect-Oriented Programming," in *Proc. of ECOOP 97*, Springer-Verlag LNCS 1241.
- [Kniesel, 99] G. Kniesel, "Type-safe Delegation for Run-time Component Adaptation," ECOOP'99, LNCS 1628, pp. 351–366, 1999.
- [Mili et al., 99] H. Mili et al. "View Programming: Towards a Framework for Decentralized Development and Execution of OO Programs," *Proc. of TOOLS USA '99*, Aug. 1-5, 1999, Prentice-Hall, pp. 211-221
- [Mili et al., 01] Hafedh Mili, Ali Mili, Sherif Yacoub, and Ed Addy, in *Reuse-Based Software Engineering*, John Wiley & Sons, Dec 2001.
- [Ossher et al., 95] Harold Ossher et al., "Subject-oriented composition rules," in *Proc. OOPSLA '95*, Austin, TX, Oct. 15-19, 1995, pp. 235-250.
- [Reenskaugh, 1995] Trygve Reenskaugh et al., in *Working with Objects*, Prentice-Hall, 1995
- [Shilling & Sweeny, 89] John Shilling and Peter Sweeny, "Three Steps to Views," *Proceedings of OOPSLA '89*, New Orleans, LA, pp. 353-361, 1989.