JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, No. 3
Special issue: TOOLS USA 2002 proceedings

CorbaViews – Distributing Objects that Support Several Functional Aspects

Hafedh Mili and Hamid Mcheick, University of Québec at Montréal, Montréal, Canada
Salah Sadou, Université de Bretagne-Sud, Vannes, France

Abstract

The separation of concerns, as a conceptual tool, enables us to manage the complexity of the software systems that we develop. A number of approaches have been proposed that aim at modularizing software around the natural boundaries of the various concerns, including subject-oriented programming (SOP) [Harrison & Ossher, 1993] aspect-oriented programming (AOP) [Kiczales et al., 1997], and our own view-oriented programming (VOP) [Mili et al., 1999]. Both SOP and AOP support compile-time composition. A major advantage of VOP is run-time behavioral composition, which comes at the expense of a cumbersome dispatching mechanism. The same applications that warrant the kind of separation supported by these techniques tend also to be distributed whereby different client sites see different compositions of aspects, simultaneously. The level of indirection provided by distribution middleware simplifies the programming model, and reduces the overhead of VOP.

1 INTRODUCTION

In the real world, objects change roles during their lifetime. From the time a person appears on the IRS records as a deductible expense, that person will keep changing roles until well beyond its death, regularly acquiring and relinquishing attributes and behavior. 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. Further, we should be able to support this behavioral change, *while* the program is running, and we should be able to accommodate new behaviors that were *not* anticipated

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.

Cite this article as follows: H. Mili, H. Mcheick, and S. Sadou: *CorbaViews – Distributing Objects that Support Several Functional Aspects*, in Journal of Object Technology, vol. 1, no. 3, Special issue: TOOLS USA 2002 proceedings, pages 207-229.
http://www.jot.fm/issues/issue_2002_08/article12

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 addresses design-level concerns. Such concerns include error handling, synchronization, logging, access to lower-level services, and the like. Addressing these concerns usually means adding code that crosscuts normal modularization boundaries, i.e. typically objects and methods.

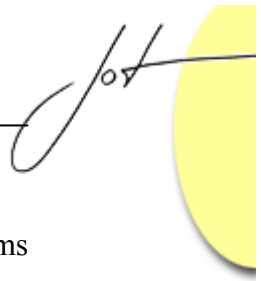
These are but three of the most common situations requiring us to modularize programs along dimensions other than the traditional function, class, or method, inherent in both procedural and object-oriented programming. There have been a number of approaches to providing language-level support for separation of concerns in the OO research community. Each one of these approaches was intended to solve one particular set of problems related to the three mentioned above.

The concept of views in OOP was first introduced by Shilling and Sweeny [Shilling & Sweeny, 1989] as a *filter* of a global interface of the class, but the views are not separable or separately reusable. Aksit et al. presented *composition filters* as a way of intercepting incoming and outgoing messages [Aksit et al., 1992]. However, the filters add no state, and can only modify *existing behavior* without adding new ones. Harrison and Ossher [Harrison & Ossher, 1993] proposed *subject-oriented programming* as a way to build integrated "multiple view" applications by composing application fragments, called *subjects*, which represent compilable and possibly executable functional slices [Harrison & Ossher, 1993]. However, the composition of subjects takes place at compilation time, and offers few degrees of freedom. Aspect oriented programming captures concerns that crosscut several entities in new constructs called *aspects* that are woven into the structure of functional code [Kiczales et al., 1997]. However, aspect "weaving" also takes place at compilation time, and aspects do not necessarily correspond to domain-level behavior.

The problem of dynamic adaptation has been addressed by a number of researchers. Earlier approaches were based on variations of the adapter/decorator design pattern. The problem with such approaches is that the various adapters (and the classes they adapt) have to be known beforehand [Buchi & Weck, 2000]. Two approaches attempt to address this problem in a type-safe fashion: Kniesel, with Darwin/Lava [Kniesel, 1999], and Büchi & Weck's generic wrappers concept [Buchi & Weck, 2000]. Presumably, both approaches enable an object to offer different interfaces to different client programs. With generic wrappers, the various interfaces have to be hierarchically composed. Further, with both approaches, the adapter and "adapted" have different object identities, and neither approach handles distribution explicitly.

It turns out that the same applications that warrant the use of separation of concern techniques also tend to be the kind of applications that are distributed and that offer different sets of functionalities to different user communities. In summary, 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 separately, 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. It relies on a (non-distributed) programming model called *view oriented programming* [Mili et al., 1999] that considers application objects as consisting of some core functionality to which state and behavior (*views*) are added and retracted on demand during run-time. To support this programming model in an existing typed language (first C++, and then Java), without unduly burdening programmers with new syntax and semantics, we set out to use a code transformation approach that trades performance and type safety for run-time flexibility. As it turned out, distribution actually simplifies this process, for two reasons, 1) a conceptually clean separation between interfaces (clients) and implementations (servers), 2) a built-in infrastructure for dynamic behavior invocation.

The next section includes a brief overview of the major separation of concerns techniques, and a more detailed presentation of our own view-oriented programming. Section 3 explores distribution issues in the context of these methods. Section 4 describes the principles underlying our approach. An ongoing implementation is described in section 5. We conclude in section 6.

2 SEPARATION OF CONCERNS TECHNIQUES

We start with some widely known methods, and then spend some time describing view-oriented programming because it is the basis for the approach described in section 4.

Subject-oriented programming

Subject-oriented programming views object oriented applications as the composition of several application slices representing separate functional domains or add-ons (features) to existing functional domains. Such a slice is called a *subject* and consists of a self-contained, declaration-wise, object-oriented program, with its own class hierarchy. Subject-oriented programming enables us to compose such hierarchies (*subjects*) into one that, generally speaking, consists of, 1) the union of the interfaces (signatures) emanating from the input subjects, and 2) the *composition* of the implementations of the methods that are defined in more than one subject. Default composition uses name matching to compose class definitions. Name matching may be overridden locally with composition expressions written in a powerful composition language [Ossher et al., 1995].

A major advantage of class composition *à la* SOP (i.e. by "merging" class hierarchies) over composition through multiple inheritance is that when two classes are "merged", all of their descendants (from both input hierarchies) will benefit from the

"merge". By contrast, adding an aspect (embodied in a class) to an existing hierarchy using multiple-inheritance requires that we create a subclass of each class in the hierarchy. SOP, and to some extent, its descendant MDSOC (Multi-Dimensional Separation of Concerns, [Tarr et al, 1999]) suffer from a number of limitations, including, 1) the compile-time binding of the various subjects, 2) the relative coarseness of the composition unit—the method—, because of which composability requires some pre-planning [Mili et al., 1996], [Mili et al., 2001a], and 3) the limited reusability of *subjects* whose composition relies on *extensional* name matching—renaming notwithstanding.

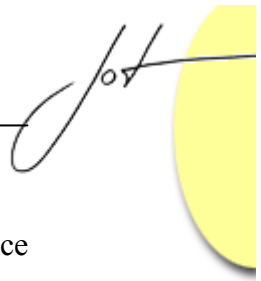
Aspect-oriented programming

Underlying AOP is the observation that what starts out as fairly distinct concerns at the requirements level, or at the design requirements level (non-functional requirements) end up tangled in the final program code because of the lack of support, both at the design process level, and at the programming language level, for keeping these concerns separate. With aspect-oriented programming (AOP), these concerns may be packaged as *aspects*, which can be woven into "any" application that has those concerns. AOP requires three ingredients:

- A general purpose programming language for defining the core functionalities of software components,
- An *aspect language* for writing *aspects*, i.e. code modules that address a specific concern and that cross-cut various components in the general-purpose language, and
- An *aspect weaver*, which is a pre-processor that "weaves" or "injects" aspects into the base software components to yield vanilla flavor components, coded in the general purpose programming language.

Kiczales et al. have proposed different forms of aspects. The simplest form of aspects, *advisories*, add some piece of code to specific methods identified by more or less complex <method,class> expressions (so-called *point cuts*), and may be used to instrument code or to handle some fairly generic functionality (logging, error handling, etc.). The language for describing *point cuts* enables us to specifying class names and method names *intentionally* (*property-based point cuts*) using more or less complex patterns. Further, the code embodied in the aspects can be inserted at a variety of points of control (so-called *join points*), as opposed to SOP's implicit before or after semantics. Finally, a third kind of aspects is proposed that handles associations between objects. Such aspects may have their own state variables, and may trigger the execution of a number of methods on the participating objects.

Aspect-oriented programming has gained wide acceptance in the research community in part because of its gradual learning curve: it is possible to do eminently and frequently useful things simply. However, much like SOP, it only supports compile-time composition of aspects. A number of proposals have been floated to support dynamic "weaving" of aspects. Such proposals rely on reflection, with two major



disadvantage, 1) lack of safety — too much uncontrolled power, and 2) performance penalty.

Dynamic adaptation methods

Adaptation techniques were first developed in the context of GUI frameworks. Typically, most graphical components would come in several flavors, a basic one, and a set of "decorated flavors" to include things such as borders, "scrollability", and the like. Rather than define a variant (subclass) for each combination of graphical attributes, we use the adapter/wrapper design pattern. To some extent, the filter-based approaches [Shilling & Sweeny, 1989] and *composition filters* [Aksit et al., 1992] are examples of such approaches. However, the traditional wrapper implementation suffers from a number of problems [Buchi & Weck, 2000]. Büchi and Weck defined a set of requirements that wrapping/adaptation methods must satisfy, including:

- *Run-time applicability*: the actual type and instance of the wrapped object must be decidable at run-time,
- *Genericity*: the wrapper must be applicable to any subtype of the declared interface of the wrapped object
- *Transparency*: the wrapper should be a subtype of the wrapped object
- *Overriding*: wrappers should be able to override methods of wrapped objects
- *Shielding*: a wrapper should be able to control whether clients can access directly the wrapped object.

The traditional decorator pattern fails the run-time applicability, transparency, and shielding conditions. Run-time applicability usually comes at the expenses of type safety. Both Kniesel [Kniesel, 1999] and Büchi and Weck [Buchi & Weck, 2000] proposed Java-based techniques for providing type-safe run-time bound decorators. Büchi and Weck defined the concept of *generic wrappers* which are represented using a class-like syntactic construct that specifies the static type of the objects to be wrapped — called *wrappee*. The wrappee is specified at run-time (wrapper creation time). In their model, they require that the wrapper be of a subtype of the *run-time type* of the wrapped object. However, in their first prototype implementation, they settled for the static type. With generic wrappers, a wrapper forwards method calls to the wrappee when those methods don't exist in the wrapper. Generic Wrappers support *conjunctive adaptation* in the sense that, if we want to define several wrappers on the same object, they have to wrap each other in a hierarchical fashion.

Kniesel proposed Lava as an extension of Java with a real delegation mechanism [Kniesel, 1999]. An object delegates to another object specified as a special attribute (an instance variable qualified as a *delegatee*) in its class definition. The delegatee may change during run-time, in the same way that *strategy objects* may change in the strategy pattern. Kniesel shows that his system is type safe. However, a major limitation of Lava is that the number and type of delegatees is fixed at compile-time: it is part of the class definition!

View-oriented programming

Basics: 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,
- Support the addition and removal of views (functional slices) during run-time, making objects support different interfaces during run-time, and
- Have a consistent and unencumbered protocol to address objects that support views.

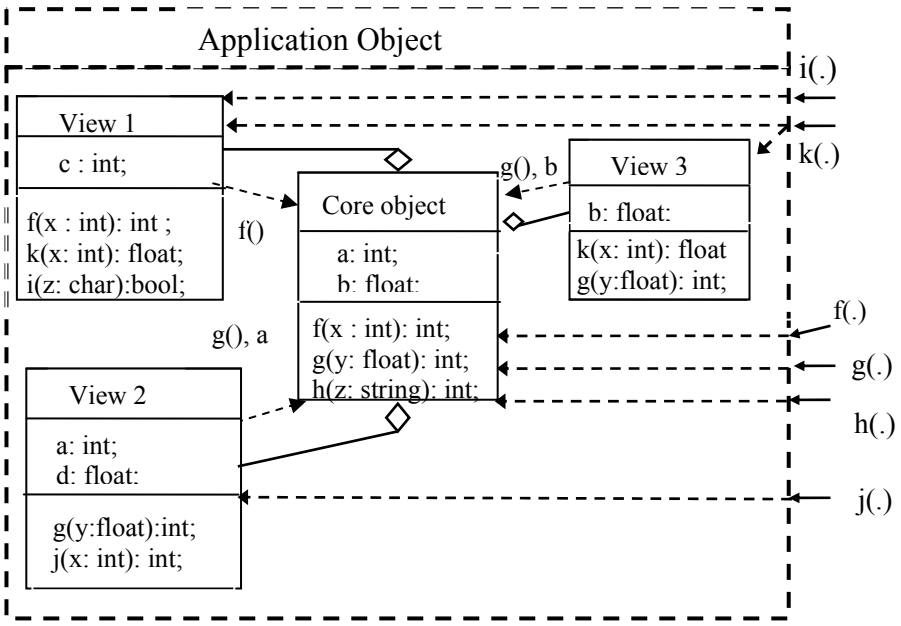
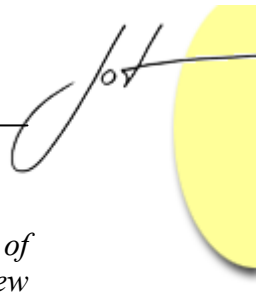


Figure 1. A model of an object with views.

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 3), and delegate shared data and behavior. In this case, upon invoking the operation f() on *view 1*, the



request is forwarded to the core object, and *the operation f() is executed in the context of the core object*. The same is true for references to the shared state variables ('a' for *view 2*, and 'b' for *view 3*). Practically, there will be a single copy of such variables, stored in the core object, and read/write requests will be forwarded to the core object. The application object is seen as supporting the *union* of behaviors of the core instance and of the currently attached views.

Viewpoints, reuse, and decentralized development. 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. For example, for the purposes of building an information system that supports the business, leasing *computers* is more similar (software infrastructure and functionalities) to leasing cars, then to selling *computers*. Using our model of view programming, the different roles that an application object can play will be represented by views. When those roles correspond to different business processes, then the *logic* of the code of the views should be reusable across business domains. 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, called *viewpoint*, can then be instantiated for different types of assets, be they trucks, buildings, machines, or computers. In the example of Figure 1, view 1 which uses the method *f(.)* of the core object, is the result of 'instantiating' a *viewpoint* that requires that the core object support a method *f()*.

Programming with views. Our approach consists of supporting view programming into a host language such as Java or C++ by adding a "views veneer" which language pre-processors will translate into vanilla flavor constructs from the host language. We will focus on the code transformations that need to take place, and on the run-time mechanics of our approach.

Consider the example of a *customer relationship management* (CRM) application. Let `Customer` be the core object (see Table 1). In addition to information about contact info and outstanding orders, we could support two additional functional areas, e.g. the customer credit profile (`CreditWorthinessCustomer` view), and a loyalty ("frequent miles") program (`LoyaltyCustomer` view). Table 1 shows *one possible implementation* of the views, i.e. as regular Java classes that contain view-specific data and functions, but that *forward* core data and functions to the core instance. The view classes are generated by instantiating a corresponding template (viewpoint) for the class `Customer` (much like Büchi & Weck's generic wrappers). Core objects store and manipulate their views through data structures and functions inherited from the class `Viewable`.

A `Customer` object is created by instantiating the core class. Later on, views may be added or removed to the core object dynamically, using an inherited API from `Viewable`. When we first add a view to a core instance, an instance of view class is created and linked with the core instance. That view object can later be deactivated, re-activated, or deleted. Deactivation turns off the behavior of a view, but preserves its state.

The behavior of an object with views depends on the set of views that are currently added (and active). When the object receives a message, its answer depends on whether its core functionality or one of the attached views supports the requested behavior. If no implementer is *currently* available for the requested behavior, the request is denied. In a reflective language such as Smalltalk, this behavior can be accomplished by modifying the dispatching mechanism. In a typed and (mostly) statically bound language such as C++, this behavior can be obtained by performing the appropriate compile-time code transformations. Java, which offers reflexive capabilities, does not support message *intercession* easily, and a code transformation approach is also used[1]

```
class Customer extends Viewable {
   private String name;
   private String number;
   private String address;
   private Collection orders;
   …
   public void addOrder(Order in){…}
   public Iterator getOrders(){…}
   …
}
class LoyaltyCustomer extends View{
   private Customer coreObject;
   private String loyaltyGrade;
   …
   public void printCustomer() {…}
   public void
      printListOrOrdersMade() {…}
   public String  getName() {
      return coreObject.getName();
   }
   …
}
```

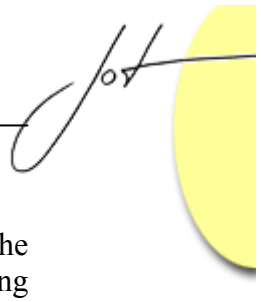
```
class CreditWorthinessCustomer
                extends View {
   private Customer coreObject;
   private String creditRating;
   private float creditLimit;
   private String accountState;
   …
   public void printCustomer() {…}

   public String  getName() {
      return coreObject.getName();
   }
}
```

Consider the following program excerpts:

```
          import com.walmart.core.Customer;
          import com.walmart.finance.*;
          import com.walmart.operations.*;
  (1)     Customer myCustomer = Customer.getInstanceWithID(id);
  (2)     myCustomer.attach("Loyalty");
  (3)     myCustomer.attach("CreditWorthiness");
  (4)     float val = myCustomer.getCreditLimit();
  (5)     myCustomer.printCustomer();
```

[1] AspectJ™ also uses a code transformation approach, instead of a reflexion-based implementation.



In line (4) the programmer invoked a behavior that is available in the `CreditWorthinessCustomer` view on the instance of `Customer`, without referring explicitly to the view instance. The underlying mechanism is a pre-processor that replaces line (4) with the following line,

```
(4')    float val=((CreditWorthinessCustomer)
    myCustomer.getView( "CreditWorthiness")).getCreditLimit();
```

because it knows that `getCreditLimit()` is available in the view class `CreditWorthinessCustomer`, but it does *not* know for sure that *at the time* that the call is made, an `CreditWorthinessCustomer` view is *attached* and *active*, and we cannot sort this out at compilation time.

Line (5) shows the method `printCustomer()` which is supported by both `LoyaltyCustomer` and `CreditWorthinessCustomer`. We adopted the approach advocated by Harrison & Ossher [Harrison & Ossher, 1993], which consists of *composing* the various method implementations. Our approach relies on a *universal composition view,* which is automatically generated to contain default implementations for the all the multiply defined methods:

```
class _CompView_Customer extends View {
   public void printCustomer() {
         // some combination of the implementations coming from
         // LoyaltyCustomer and CreditWorthinessCustomer
   }
   …
}
```

The actual code generated and the mechanics of composition views are slightly more complex, as they take into account the potential for broken delegation [Mili et al., 1999].

3 DISTRIBUTION ISSUES

The combination of aspects and distribution is interesting for three reasons. First, 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. Second, Depending on the separation of concerns technique, objects that embody several concerns may be fragmented, which may raise a number of issues for distribution. Third, 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 possibly for duplication — in a distributed application context. We look at these issues in turn.

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 make a difference (so called *join points*), i.e. what needs to be changed to turn a regular (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 technique's *abstraction boundaries* [Mili et al., 2001a] best match the required changes to accommodate distribution.

There have been a number of research efforts trying to categorize *concerns* in such a way that they can be matched against the panoply of techniques, both aspect-oriented and plain object-oriented techniques (see e.g. [Walker et al, 1999]). 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 (e.g. CORBA IDL compiler). There remain a few changes that need to be accommodated. One such change involves lifecycle issues. Indeed, the "creation" of remote objects is different from the creation of local objects. For instance, the former requires going through an object factory, which may itself be accessed using some sort of a naming service. We also need to handle remote exceptions. Indeed, 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 aspect-oriented programming supports composition at sub-method levels, with some restrictions (entry and return points, exceptions, etc.). 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.

Note that if we are interested in supporting a distribution infrastructure with a configurable set of services (e.g. transactions, security), then we are faced with a new instance of "multiple aspects" problem, this time concerning the distribution infrastructure implementation itself, instead of the application that executes *in the context of* the distribution infrastructure (see e.g. [Coady et al, 2001],[Joshi & Agrawal, 2002]).

Distributing objects embodying several aspects

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 (concerns), 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, and the same techniques used.

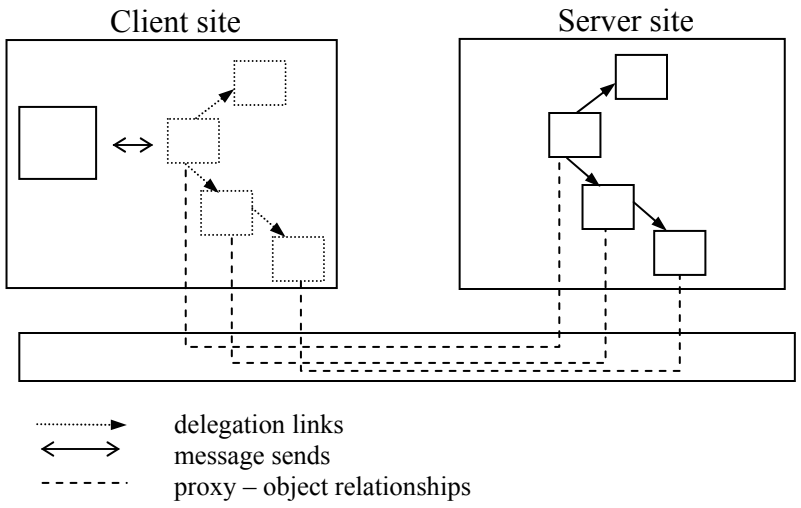


Figure 2. The entire delegation machinery is distributed

In those methods where different concerns are represented as separate objects (e.g. our approach and dynamic adapters (e.g. [Kniesel, 1999] and [Buchi & Weck, 2000]), then there are two strategies, and corresponding host of issues to be addressed. Roughly speaking, the first strategy consists of separating the multi-aspect "aspect" from the distribution aspect, and distributing multi-aspect objects like any network (aggregation) 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, and the like. 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, using the proxies of the various components, 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 duplicate some of the processing logic. It thus seems more reasonable to implement the delegation-based method 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.

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.

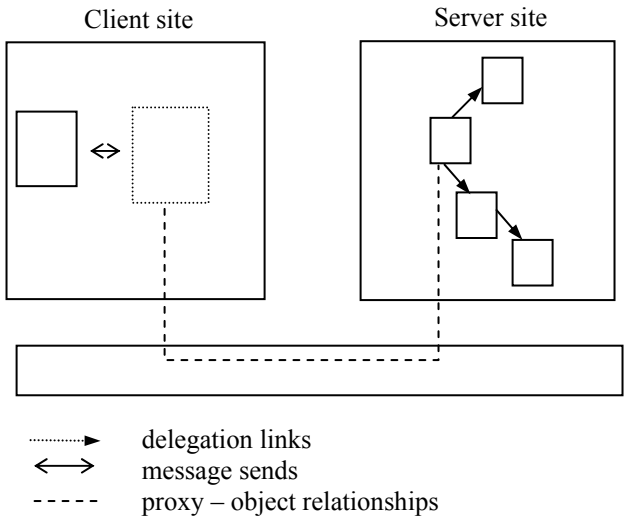


Figure 3. The client side sees a single object. The delegation machinery is hidden on the server's side.

4 VIEW PROGRAMMING AND DISTRIBUTION

Issues

In this section, we are interested in the situation where an object with views is distributed, and offers different sets of functions through different client sites. 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. Figure 4 illustrates such a scenario for the object in Figure 1. We assume in this example that site 3 is not aware of the existence of a core object behind the view, or of View 1 and View 2, and the behavior of these should not be available to it, except indirectly as a side effect of methods called on the core object. For the case of sites 1 and 2, they know about view 1 and view 2, but don't know about view 3, and any behavior invoked on the core object should only invoke the methods that are explicitly provided by view 1 and view 2 (or as side effects of such behaviors).

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. Figure 5 illustrates such a model. We assume for simplicity that a single ORB manages requests on behalf of all sites. We also assume for the time being that there is no object replication: each of the core instance and the views reside on a single site, with proxies representing them elsewhere.

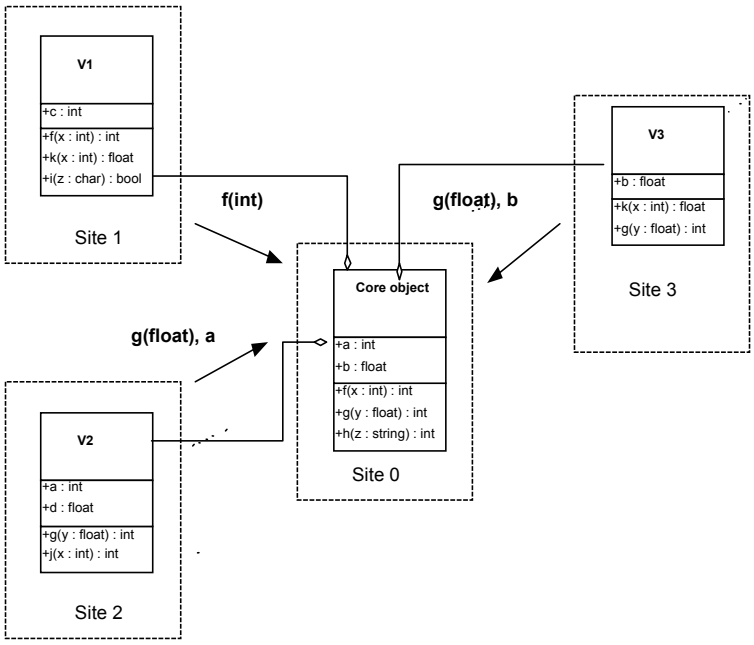


Figure 4. Distributed object with views.

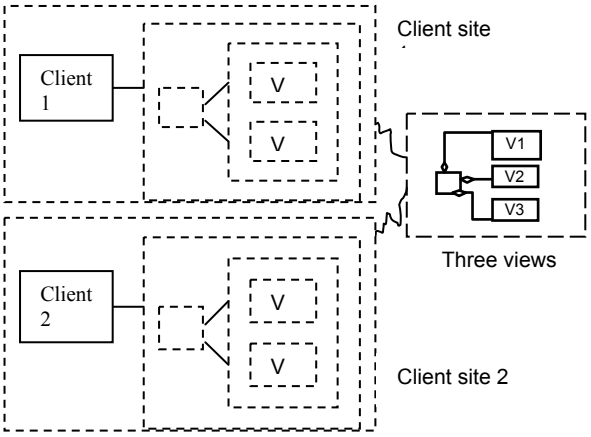


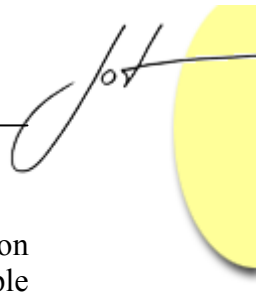
Figure 5. A single-server, multiple-client scenario.

Consider now the fact that our own parsing of client code of viewable objects also involves code generation, and a number of source code transformations (see section 2.4). For instance, a user program that uses several views of the same object will have messages to that object go through a *composition view*. However, a composition view is nothing but a "stub-like" class that implements the union of the interfaces (potentially) supported by the viewable object. This stub dispatches calls depending on which views are attached (and active) at that point in time (see section 2.4). Because support for distribution also involves generating a stub class that implements a *remote interface*, we could combine the two code generations. A *distributed object configurator* (DOC) enables architects to select, 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. DOC will be discussed in section 5. We next look at the simplest case where all the views reside in the same site.

Single server, multiple clients

Figure 5 shows an example situation where several views reside on the same server, but different client sites see different subsets of those views. In this case, each client site only sees the version of the server side object that corresponds to its views. There are two issues that need to be addressed. First, how to make the same server object implement two or more client interfaces, and second, where to handle the dispatch of multiply implemented methods (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. Whereas the client stub is supposed to be used as is, the server skeleton is supposed to be specialized or somewhat 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 (class `CustomerSkeleton` in the example above). 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 the inheritance-based and delegation-based object implementation approaches allow a class to implement several interfaces. In Java, multiple inheritance of classes is not supported, but the same class can support several interfaces, which makes the tie approach appropriate for implementing several interfaces with the same Java class. Figure 6 illustrates this.

We should stress that the usual "self problem" inherent in simple message forwarding is not an issue here: the "forwarder" does not perform any application specific processing whatsoever; it simply dispatches method calls coming over the wire. Thus there is no chance that a method on the server object would need to come back into the skeleton object. Note, however, that if a method on the server object needs to access a yet another remote object, the call is handled transparently (going through a proxy and the ORB) as if that other object were local.

A final point has to do with view creation and destruction. As mentioned in section 2.3, the first call to attach(<a view>) 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. One way of handling this is a 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. At the same time, the shared object implementation maintains its own count of the various views, which indicates how many server interfaces need a particular view. A server interface (tie object) no longer needs a view when *its* reference count goes to zero. Thus, whenever a tie object's count goes to zero, it asks the shared implementation to decrement *its* reference count. When that count goes to zero, the view is destroyed.

Things get complicated when several client processes 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.

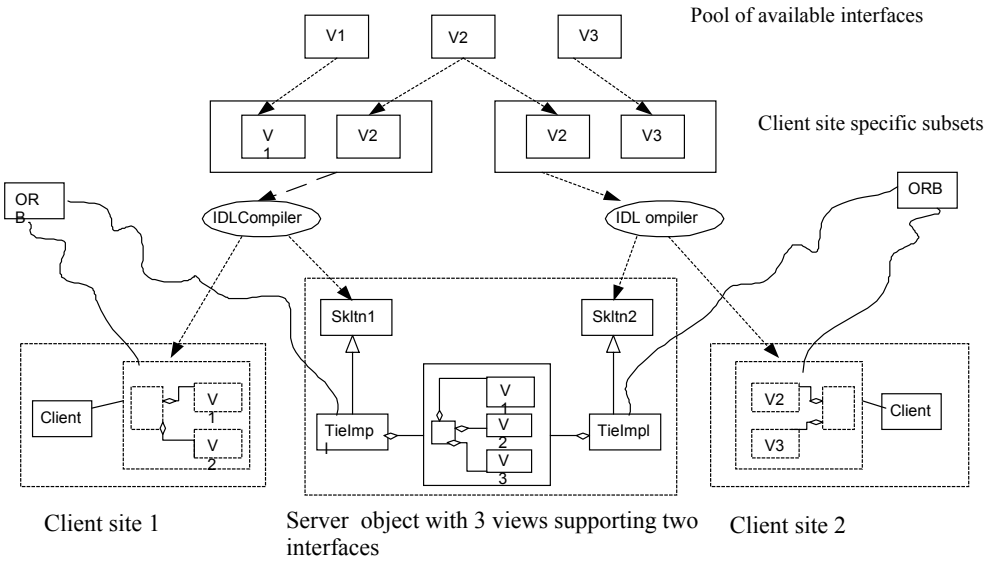


Figure 6. Server Object(s) implement several client interfaces.

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. Finally, with client-side view management, only view creation and destruction need to go the server; view activation and deactivation can be handled locally.

With client-side method dispatching, the client side stub is similar to the composition view 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 automatically using some mangling scheme.

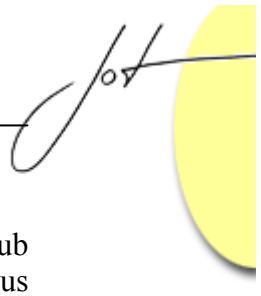


Figure 7 shows an example stub based on this implementation. In this case, the stub has an additional instance variable (`viewStates`) that contains the status of the various views—referred to by name on the client side. The stub supports methods to attach/activate and deactivate/detach views. Attaching a view will request the attachment to the server, and set the local variable (`viewStates`) accordingly. Deactivating a view is a local operation, and only works on viewStates. This stub code for the method '`printCustomer`()' illustrates the name mangling scheme used to dispatch to different method combinations on the server side: the method '`printCustomer()`', which is supported by the views `LoyaltyCustomer` and `CreditWorthinessCustomer`, has three versions, '`_LoyaltyCustomer_printCustomer()`', '`_CreditWorthiness-Customer_printCustomer()`', and the composition '`_CreditWorthinessCus-tomer_LoyaltyCustomer_printCustomer()`'. The prefix is generated by the method '`getActiveViewsSupportingMethod(String signature)`'.

```
package CustomerLoyaltyCustomerCreditWorthinessCustomer;
public class _CustomerStub extends ObjectImpl implements
   CustomerLoyaltyCustomerCreditWorthinessCustomer.Customer {
   private Hashtable viewStates;
   private static Hashtable methodsToViews;
   public boolean attachView(String viewName) {
   try {
        Request _req = _request("_attachView");
        req.add_in_arg().insert_string (viewName);
        _req.invoke();
    catch (ViewAttachmentException vae){  return false;}
    finally {
        viewStates.put(viewName,View.Active);return true;}
   }
   public void deactivateView(String viewName) {
   // View activation and deactivation are local
        viewState.put(viewName,View.Idle);
   }
   public void printCustomer() throws UnavailableBehavior {
        String prefix = GetActiveViewsSupportingMethod(
                                          "printCustomer##");
        if (prefix.length()==0) throw new
           UnavailableBehavior("printCustomer ()",viewStates);
        String reqName = prefix + "_" + "printCustomer";
        Request_req = _request(reqName);
        …
   }
   …
}
```

Figure 7. Client side dispatching

5 IMPLEMENTATION

Work on view programming has been ongoing for several years at the University of Quebec at Montréal. We have already implemented prototype support for view-oriented programming in C++ [Mili et al., 1999]. Our C++ implementation supports a small subset of C++, and does not address distribution but helped us identify a number of problems that we set out to solve in our Java-based distributed solution [Mili et al., 2001]. A number of the difficulties we had with the C++ implementation were related to difficulty of separating *interfaces* from *implementations*, both at the language level, and in terms of the programming model that needs to be supported in the context of a monolithic application. Java does support this separation. Further, the separation between interfaces and implementations is at the heart of the programming model for distribution, enabling us to concentrate view machinery in one place (see section 3).

In reference to the implementation strategies discussed in section 3.2, and to the issues raised in section 4.2, we decided to address distribution and multi-aspect issues in a single framework. As mentioned in section 3.2, and illustrated in some of the choices discussed in section 4.2, doing so enables us to simplify the overall scheme, and to enhance the overall performance of the combination of these two features.

As illustrated by our discussion in section 4.2, support for distributed multi-aspect objects involves the following changes, as compared to plain distribution, a) the implementation objects are “regular” objects with views, i.e. using an aggregation-based simulation of delegation, and b) some view-specific processing at both the stub and skeleton. This view specific processing consists, on the client side, of three changes, i) addition of new infrastructure remote methods (e.g. attach/detach view), ii) addition of new local methods and variables (activate/deactivate, and viewStates), and iii) modification of dispatch of view-defined methods (see example of ‘release()’ method). Server-side changes include support for attach/detach and the corresponding reference counting logic.

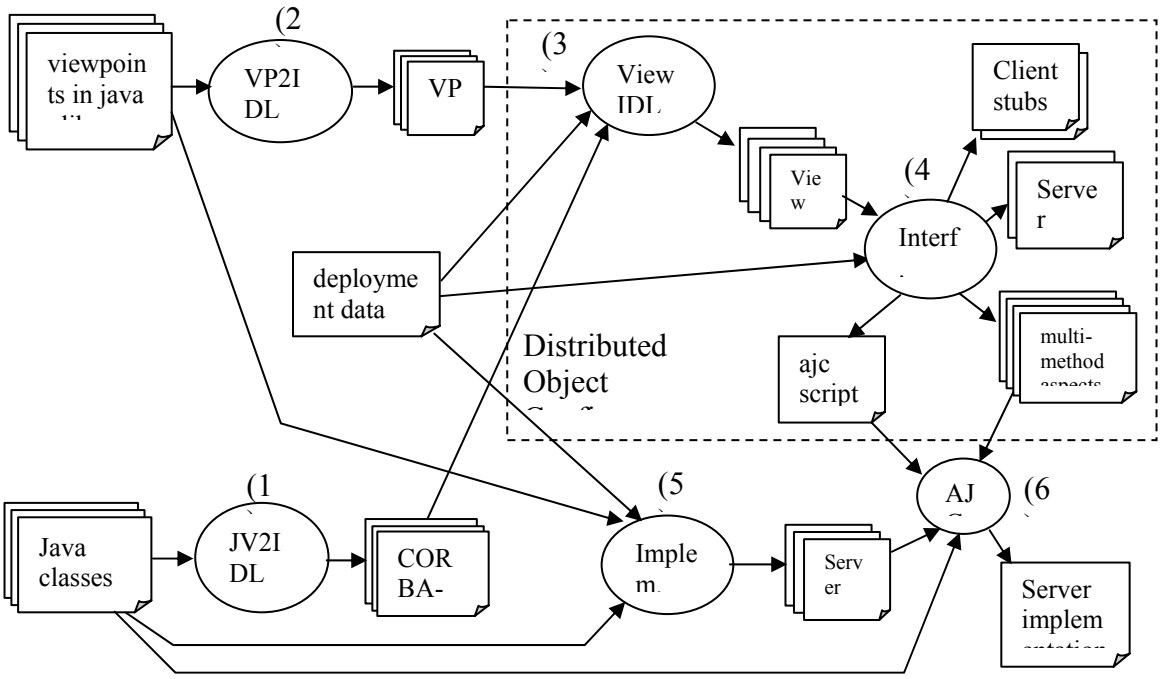


Figure 8. Structure of the toolset

Figure 8 shows the overall structure of our tool set. We numbered the various processes for easy reference. At the beginning of the process, Java classes (core objects) and viewpoints are translated to CORBA-IDL (process (1)) and CORBA IDL-like syntax (process (2)), respectively. The Distributed Object Configurator (DOC) uses these interfaces and deployment information to generate the view and distribution infrastructure code. Specifically, deployment data describes:

1) Which views will need to be supported, overall. These views will be generated by mapping the viewpoint IDL to core objects IDL, yielding view IDL (process (3)). It is assumed that each one of these views will be implemented somewhere—all by the same server for now,
2) Where do the various object components (core classes, view classes) reside. For the time being, we assume that the core object implementation and its views will reside on a single server. This will be generalized later to the case where core object and views reside on separate machines,
3) Which client site needs which interfaces. This will be used to generate a single IDL interface per client site, which, in turn will be used to generate a client stub and a server skeleton *per client interface* (one per combined interface). This is the work of process (4).

For those methods that have multiple implementations, we leave it up to the developers to specify the composition. For example, the method `printCustomer` is supported by both the credit worthiness view, and the customer loyalty view. Using the name mangling scheme discussed in section 4.2, we need to provide an implementation for:

```
public void _CustomerCreditWorthiness_CustomerLoyalty_print-
                                                    Customer();
```

We want to be able to provide such implementations but without editing either the source file (e.g. the Java core classes) or the automatically generated code (e.g. the server side skeletons). Thus, the tool (process (4)) generates, a) an empty method stub (with a single `return` statement), and b) an aspect that developers can edit, and that includes the actual implementation. The actual aspect looks as follows (simplified for presentation):

```
package ClientFideliteClientEstimationCreditClientarabica;
… // a bunch of imports

aspect _CreditWorthinesCustomer_LoyaltyCustomer_printCustomer{
  pointcut  _CreditWorthinessCustomer_LoyaltyCustomer_printCus-
tomer():args()&& call (String    _CompView_Customer._Credit-
        WorthinessCustomer_LoyaltyCustomer_printCustomer());
  before()    throws    Exception:    _pre_CustomerCreditWorthi-
ness_CustomerLoyalty_printCustomer () {
        // Developers edit this method
        try {
        ((CreditWorthinessCustomer)getView("CreditWorthiness"
                          +"Customer")). printCustomer();
        ((LoyaltyCustomer) getView("LoyaltyCustomer ")).print-
                                                    Customer();
        } catch(Exception ex){ex.printStack();}
    }
}
```

In this case, the implementation is included in a new method that will be called *before* the body of the multiple method. This is one of several flavours of aspects we are experimenting with. In addition to generating the aspects for the multiple methods, the processor (4) generates a script for the AspectJ compiler to weave all those aspects into the main body of the server-side code. The actual code for the server views is generated by the same tool we use for non-distributed applications (processor (5)).

Figure 9 shows a screendump of a preliminary implementation of DOC. This screen shows the selection of core classes, viewpoints, and the corresponding views.

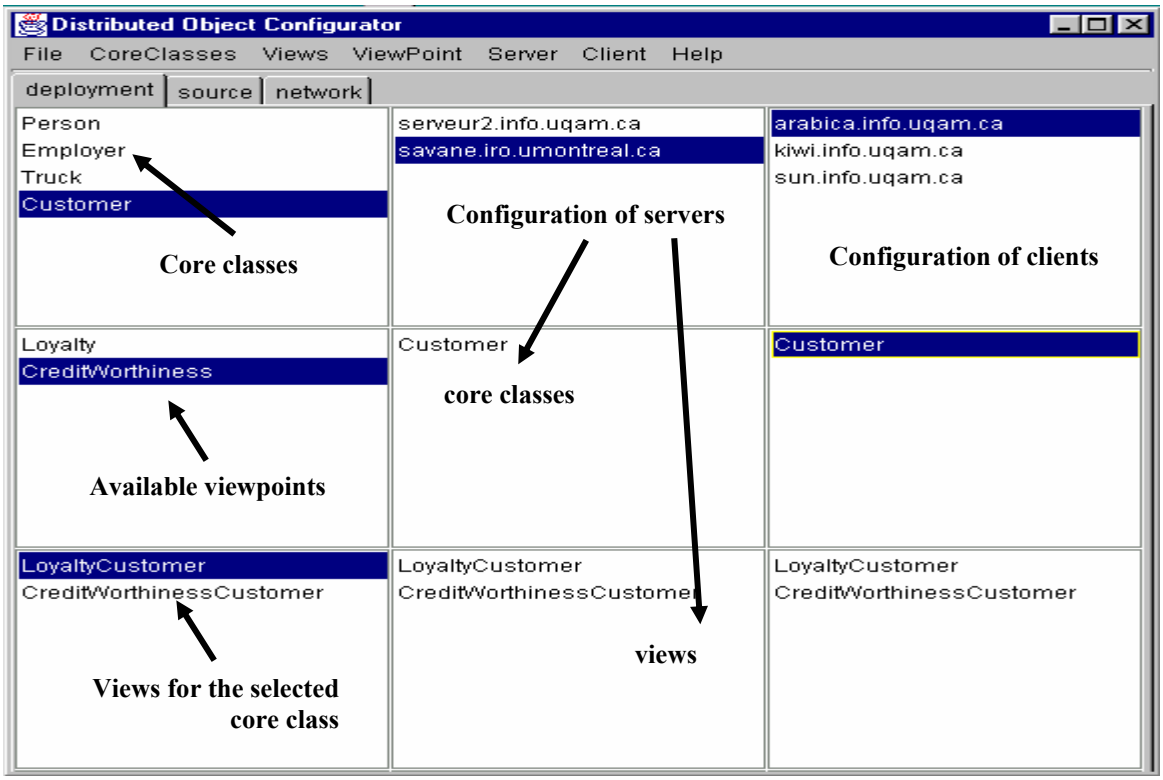


Figure 9. Specifying the functionality to be provided by servers to clients.

6 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—method supported by several views. 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 by Nortel, DEC, IBM, CAE Electronics, Machina Sapiens, the SYNERGIE program (Québec), and NSERC (Canada).

REFERENCES

[Aksit et al., 1992] M. Aksit and L. Bergmans, and L. Vural, "An object-oriented language-database integration model: the composition filters approach", *in Proc. of ECOOP 92*, Springer Verlag 1992.

[Buchi & Weck, 2000] M. Büchi and W. Weck, "Generic Wrappers", in ECOOP 2000, LNCS 1850, pp. 201–225, 2000.

[Coady et al, 2001] Y. Coady, G Kiczales, M. Feely, N. Hutchinson, and J. Suan Ong, "Structuring Operating System Aspects," *Communications of the ACM*, Special issue on Aspect-Oriented Programming, Oct. 2001, pp. 79-82

[Harrison & Ossher, 1993] W. Harrison and H. Ossher, "Subject-oriented program-ming: a critique of pure objects," *in Proc. of OOPSLA'93*, Washington D.C., Sept. 26-Oct 1, 1993, pp. 411-428.

[Joshi & Agrawal, 2002] R. K. Joshi and N. Agrawal, "AspectJ Implementation of a Dynamically Pluggable Filter Objects in Distributed Environment," *in Proc. of 2nd Int.Wshop Aspect Oriented Soft. Dev.*, Bonn, Feb 21-22, 2002

[Kiczales et al., 1997] G. Kiczales, J. Lamping, C. Lopez, "Aspect-Oriented Programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP),* Finland. Springer-Verlag LNCS 1241. June 1997.

[Kniesel, 1999] Gunter Kniesel, "Type-safe Delegation for Run-time Component Adaptation," ECOOP'99, LNCS 1628, pp. 351–366, 1999. eds Springer-Verlag Berlin Heidelberg 1999

[Mili et al., 1996] H. Mili, W. Harrison, and H. Ossher, "Subjectalk: Implementing Subject-oriented programming in Smalltalk," *Proc. of TOOLS USA 96*, Santa Barbara, CA, Aug 2-5, 1986.

[Mili et al., 1999] 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., 2001] H. Mili, H.Mcheick, J. Dargham, and S. Delloul, "Distribution d'objets avec vues", *in proc. of LMO'01*, special issue of *L'Objet*, Jan. 2001.

[Mili et al., 2001a] H. Mili, A. Mili, S. Yacoub, & E Addy, *Reuse-Based Software Engineering*, John Wiley & Sons, 2001.

[Ossher et al., 1995] H. Ossher et al., "Subject-oriented composition rules," *in Proc. of OOPSLA'95*, Austin, TX, Oct. 15-19, 1995, pp. 235-250.

[Shilling & Sweeny, 1989] John Shilling and Peter Sweeny, "Three Steps to Views," *Proc. of OOPSLA'89*, New Orleans, LA, pp. 353-361, 1989.

[Tarr et al, 1999] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, "N Degrees of Separation: Multi-dimensional separation of concerns," *in Proc. of ICSE'99,* Los Angeles, May 1999.

[Walker et al, 1999] R.J. Walker, E. Baniassad, and G. Murphy, An initial Assessment of Aspect-Oriented Programming," in *Proc. of ICSE'99,* Los Angeles, May 1999

About the authors

Hafedh Mili is a full professor, and the Associate Chair for Research at the Computer Science department of the Université du Québec a Montréal. He teaches, researches, and consults on object-oriented software engineering and software reuse. His book *Reuse-Based Software Engineering* with A. Mili, S. Yacoub, and E. Addy (John Wiley & Sons, 2002) explores the use of state of the art and the practice object techniques to help build more configurable software. He can be reached at hafedh.mili@uqam.ca.

Hamid Mcheick is a Doctoral student at the University of Montréal. His research interests include tools and environments for software reuse. His Master's research consisted of developing an intelligent C++ code browser for navigating reuse libraries.

Salah Sadou is an Associate Professor of Computer Science and the Univeristé de Bretagne Sud, in Vannes, France. Professor Sadou is interested in dynamic evolution of object-oriented programs. Parts of this work were performed while he was on sabbatical at the Université du Québec a Montréal.