On the design of a generic object adaptor

Øyvind Hanssen, Frank Eliassen
University of Tromso, Dept. of Computer Science
9037 Tromso, Norway
{ovindh|frank}@cs.uit.no

Abstract:

An object adaptor provides object view mapping mechanisms between existing information systems and a global object-based federative environment. This paper addresses issues concerning construction of mapping mechanisms supporting the integration of different data-repositories in a global object-based environment. We focus on the mapping of object identities and object activation management performed by the object adaptor component. A simple model and a generic framework that supports design implementation of object adaptors is proposed. We also present an approach to a notation for specifying application specific parts of object adaptors. This notation is useful as the basis for the automated generation of application specific object adaptor code.

1. Introduction

Data model transparency is the foundation for distributed object management systems (DOMS) integrating data objects of existing data repositories into a persistent uniform global object space. Transparency is achieved by constructing mappings between data and requests in each of the local data models and a single global model (the canonical data model). It is widely accepted that object-oriented models are particularly suitable as canonical models [Nicol93].

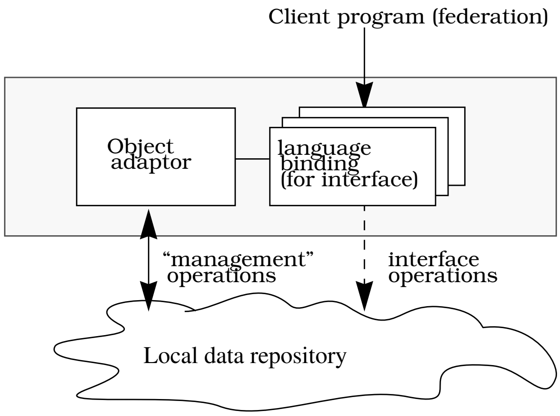
While a large body of work has been undertaken to provide clients of DOMS transparent language bindings to the generic services provided by the DOMS (for example CORBA [OMG.91.12.1]), relatively little work has gone into the area of providing tools for integrating pre-existing data repositories into DOMS as object implementations.

In this paper we present a framework for modelling, design and implementation of mapping mechanisms between different existing applications and data repositories (e.g. legacy information systems) and a global object-based federative environment. The mapping mechanisms perform two main tasks, represented by two software components: the object adaptor and the langauge binding.

The object adaptor is responsible for managing mappings between global object identity and local identities or corresponding data structures. The language binding provides access transparency and representation of object references and data values in the programming languages that the client application programmers use. This idea is illustrated by the figure below.

The remainder of this paper is organized as follows: In section 2, we discuss the management of object-identity and object activations. In section 3, we present the main ideas of a conceptual model of the object adaptor. In section 4, we present our design of a generic object-oriented framework for object adaptors, based on the conceptual model, and in section 5, we present an appraoch to a notation for specification of management policies and permanent objects.

FIGURE 1. Model of the mapping functions



2. Object identity and management

The design of object adaptor framework is based on a distinction between perma-nent and temporary object identity [ElKa91, HaeDitt93]. Objects with permanent identity can be used as "roots of persistency" [1], or they can participate in global associations.

The reliability of permanent identity has a (virtuallly) unlimited scope of time, but may be limited in scope of space. The reliability of temporary identity is limited both in scope of space and time (e.g. timely limited to a session). When there is a discrepency between the required reliability of object identity at the global level, and the reliability that can be supported by an existing local data repository, this gap must be filled by the object adaptor. For instance, it may be necessary to "simulate" permanent identities for local objects which have temporary identities only.

Enhancing the reliability of local identity in this way, is done by management-mechanisms of the object adaptor. Management involves activation, passivation and update of mappings between global and local identities.

2. 1. Object activation and passivation

An object in a local data repository can be in either a passive or active state. An active object has a representation in primary storage, and is ready to respond to operation invocations. A passive object needs to be activated before it can carry out any operations.

In some local data repositories, activation and passivation is done transparently, and objects are identified with reliable oids (e.g. surrogates). In others, such management is done more explicitly by the application programmer. In the latter case, the object adaptor must perform these tasks, because we want them to be transparent at the federation level.

The identities of activated objects are usually temporary. Therefore, the manage-ment of object identity mappings (e.g. between surrogate based global oids and activated objects) is clearly needed for objects that need to have permanent iden-tity at the federation level.

1. In the federative object model, persistency is defined by reachability from another persistent object [Banc87, KhVa90]

2. 2. Management strategies

Management is primarily needed when objects are passivated (typically by some policy in the local data repository which the federation have not control over), or when identities are changed for some other reason (e.g. because of address relocation or change of primary key value).

Key issues are how the object adaptor discover that something has happened, and how it reactivates or finds information about a more recent identity. There are many possibilities, depending on the local system and the application of interest. Generally, we can distinguish between three types of management strategies:

- Management by exception: The object adaptor takes some action when the client attempts to access an object and an exception occurs because the identity-map is invalid (c.f. [Blair92]). The action taken is to find a more recent representation of the object or activate one, if neccessary.
- Notification: The local system notifies the object adaptor if the local identity is changed or if objects of interest are passivated. This can for example be done by use of trigger mechanisms or active database system services.
- Monitoring. The object adaptor periodically checks if maps need to be updated, for example by testing the validity of maps or by reading the system log.

The choice of management strategy, and especially the method for activating objects, will strongly depend on the characteristics of the local system and on the degree of local autonomy required. The design of management policy should be done with care, because it can seriously affect performance, scalability and local autonomy. Optimizations are possible, if we have knowledge about certain dependencies between objects, e.g. systems where objects are grouped in clusters, and where clusters are units of activation/passivation (c.f. [RM-ODP.1]).

3. Conceptual model

To assist the design of object adaptors, we sketch a simple conceptual model that capture the essence of how object adaptor software works. The concepts described here could be mapped to fairly different designs.

3. 1. Proxy objects

In our model of the object adaptor, proxy objects are representatives of local objects in a federative environment. They are responsible for identification and management of local objects and their identities. The object adaptor provides (to client programs), access to objects through object references, that in essence reference proxy objects.

The purpose of introducing this concept is to encapsulate information about a given object, in a single logical unit. This information encompasses identification of the current activation, identification of passive object (in permanent storage) and methods for activation, passivation and comparision.

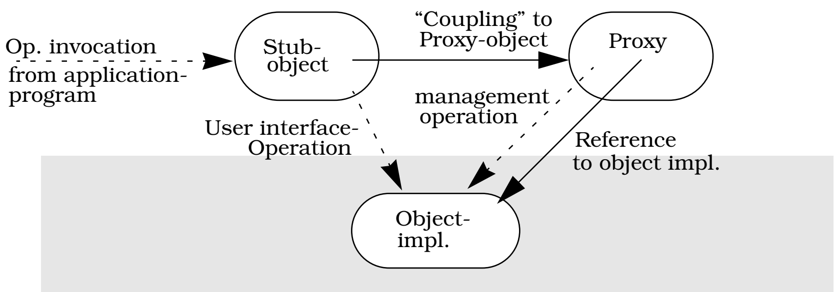
A proxy object is, from a logical point of view, a tuple *(oid, e)*, where oid is the (global) object-identity and *e* is an identifying expression that denote the corresponding local object (c.f. [ElKa91]).

Since proxy objects reflect object identity, it is relevant to distinguish between permanent and temporary proxy objects. Permanent proxies are able to (re-) activate object representations, and have the information neccessary to do this. A temporary proxy object exists as long as a client use it, and the local object is active. If an object is passivated, the corresponding temporary proxy is no longer valid and should be discarded or garbage collected.

3. 2. Stub-objects

We do not want to view the user language binding as a integral part of the proxy objects. Proxies should be defined and exist independently of the language binding in use. The language binding will need to use the proxy object to be able to perform its operations. We can then recognize two cooperating types of objects: the stub objects, which maps user operation invocations to corresponding actions against the local data repository, and the proxy objects which represents the object identity map and management mechanisms.

FIGURE 2. The role of the stub- and proxy-objects



3. 3. Interaction model

The interface offered by the object adaptor to its clients, encompasses three basic operations. These are conceptual in the sense that they are not neccessarily explicitly represented in an implementation. This simple model is able to capture the semantics of programmer interfaces for a wide range of languages. The operations are as follows:

- Bind. Defines a binding between a client and the object. Argument to the operation is an identification of the object, (e.g. a name) and it returns an object reference to an object which is active in the sense that it is able to accept operation invocations (through the basic operation invoke). Binding is usually done implicitly when invoking operations which return object references.
- Unbind. The opposite: cancels a binding between a client and the given object.
- Invoke. For invoking operations.

Object references are values which are available to the client programmer through the language binding, as unique object identifiers. Semantically, object references are quite like pointers. They can be used as arguments and results in procedures, they can be copied, deleted and assigned to variable names, the same way as pointers are.

The figure below illustrates how these ideas can be mapped to C++. Object-references for a particular object-type (*Account*) are represented as a C++ class with methods that implements stubs for user operations (*credit, debit* and *balance*). The class also defines a copy constructor, a destructor and an assignment operator, which include the *bind/unbind* semantics.

FIGURE 3. Example of object-reference class in a C++ language binding

```
class AccountRef public ObjectRef
{
public:
    AccountRef(AccountRef&);                    // bind
    ~AccountRef();                              // unbind
    AccountRef& operator = (AccountRef&);       // unbind + bind
    ...
    int Credit(Real);                           // invoke "Credit"
    int Debit (Real);                           // invoke "Debit"
    Real balance();
};
```

4. The framework

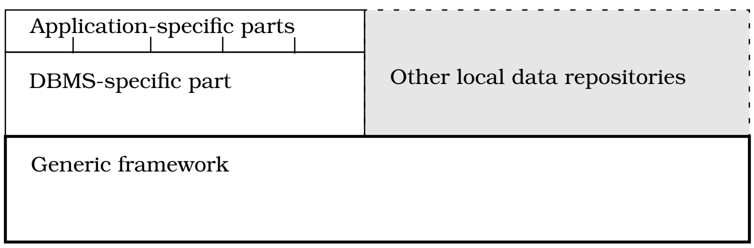
An object-oriented framework [Wirf90a, Wirf90b] represents an abstract design of an application or a subsystem in a specific domain. Such a generic application is refined to a specific one, by defining subclasses of the abstract classes, where the abstract methods of these classes are implemented. These methods are mainly called from the framework itself. The benefits of this approach are not only reuse of code, but also reuse of design. Much of the global flow-control can be implemented in prefabricated code.

[HaeDitt93] shows how the framework idea can be applied to the problem of integrating different component-databases in a common global data-model. Integration is supported by a "homogenization layer" over the different component-databases, which is implemented as a "integration framework" that consist of hierarchies of cooperating (C++) classes.

We have adopted this idea, and are developing a prototype of a reusable framework, which represents design and a partial implementation of an object-adaptor. Specific object adaptors are produced by refining the generic code, i.e. by implementing subclasses of the abstract classes of the framework. The task of this extra code will mainly be management of objects that is made visible at the global level with permanent identity.

The figure below illustrates how a generic object adaptor framework can be extended with building blocks for different local data repositories. These extensions can be made for specific applications, or for specific architectures or DBMSs[2] that can support different applications. Then we can develop DBMS-specific frameworks, based on the generic framework.

FIGURE 4. Building blocks for object-adaptors



2. The use of the word "DBMS" here is a simplification. In fact, frameworks can be made for a wide range of programming environments or underlying architectures (middleware), that supports the application. The design described here, originally focused on distributed object management systems like ANSAware.

The most important parts of our design is as follows (the details of the design are described in [Hanssen94]):

- A representation of the proxy object concept. Our design include an abstract proxy class and an abstract meta proxy class. Included in this is a abstract way to represent local object activations.
- A representation of management policies and permanent identification of objects.

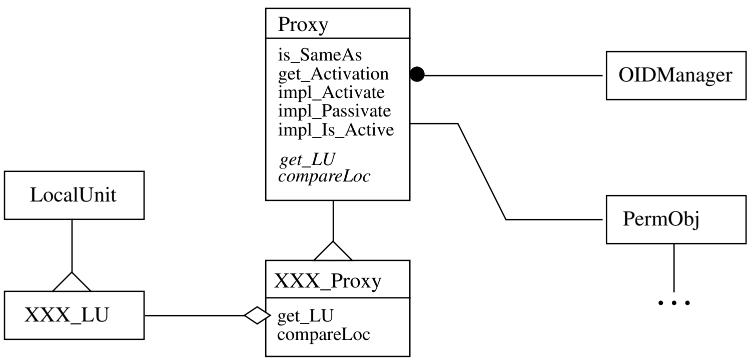
In addition, we need facilities for keeping track of active proxy objects (e.g. to be able to find already active proxies for a given object), for generating global object-identifiers (if neccessary), for garbage collection, etc.

4. 1. The proxy object

The figure below, which uses the OMT-notation (c.f. [Rumbaugh91]), illustrates how a class hierarchy of proxy objects can be designed. An abstract class *Proxy* implements the behaviour of proxy objects. We also need a corresponding *Meta-Proxy* hierarchy, to be able to instantiate proxy objects without knowledge of its concrete representations.

As in [HaeDitt93], we include a class *LocalUnit*, which represents the activated state of the object. By encapsulating such states in concrete classes which are subclasses of the abstract class *LocalUnit*, we are able to refer to local activations, independently of how these are represented. *LocalUnit* objects are part of proxy class objects, and they can be accessed (or referred to) through methods of the *Proxy* class.

FIGURE 5. Example of a proxy-class hierarchy



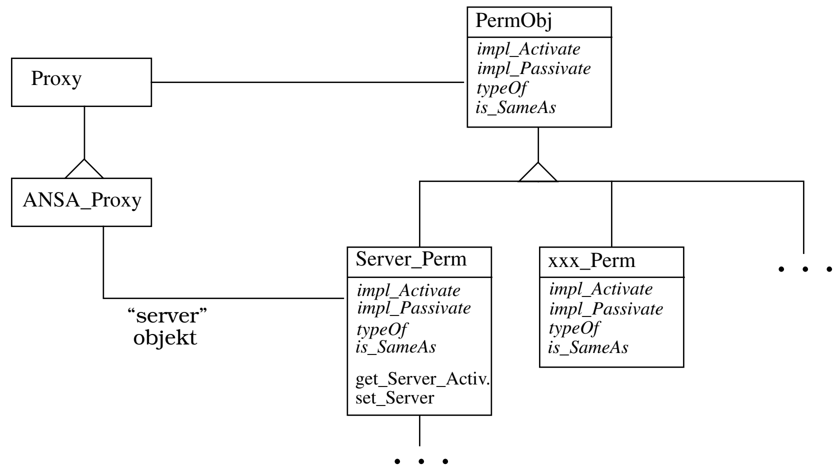
4. 2. Management of permanent objects

In our design, permanent proxy-objects are associated to *PermObj* objects, which represents management policy and permanent identification. *PermObj* is an abstract class. The *PermObj* hierarchy is illustrated in the figure below. It is separated from the proxy-hierarchy, because we require the choice of management-policy to be orthogonal to the object's type, and the permanence and management-policy to be a dynamic property of proxy objects.

The *Server_Perm* class can be used as a base class when the management is done by invoking operations of another object (server object) that is managed by the object adaptor. This class defines an association to a proxy object representing the server.

The management policies can be implemented by defining subclasses of the *PermObj* class (or the *Server_Perm* class).

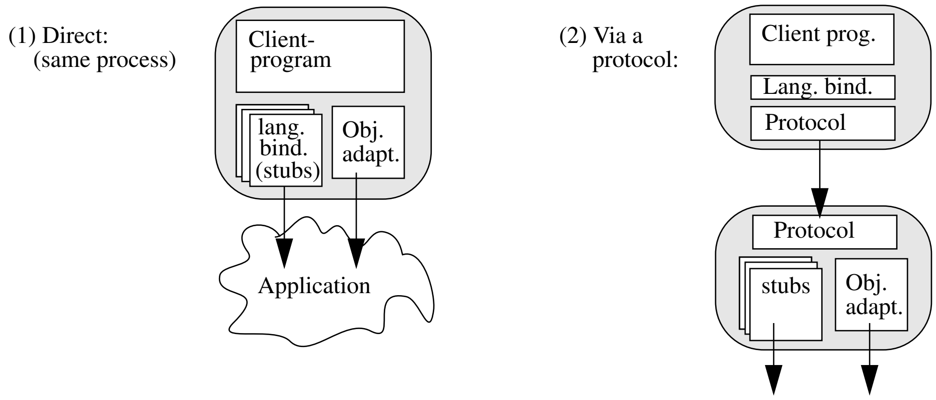
FIGURE 6. The class hierarchy for management



4. 3. Language bindings

Another feature of the protoype object adaptor framework is its support for two kinds of language binding policies. The first one is a direct language binding, where client program and object adapter run in the same process. The second one is a "canonical binding" where client and object adaptor are spacially separated and interacts through a language independent protocol. In this case, specific language bindings must be done in the clients address space (c.f. CORBA). The figure below illustrates these two alternatives.

FIGURE 7. Language binding alternatives



5. Specification of permanent objects and their management

When implementing the mapping mechanisms for a specific application, this is mainly based on two categories of information:

Export schemas (or component schemas) define the application programmers view of the local data repository. This information is used to generate application-specific language binding software (mainly stubs). Schemas can, if specified properly, be used to generate stub code automatically (typically by use of stub compilers tailored to the specific DBMS and programming language in use).

Management information defines which objects are permanent, how these are identified and how they are managed. Such information is usually not found in component schemas, and not specified in a formal notation.

We are developing a simple notation (or language) for specifying permanent objects and their management policies [Hanssen94]. A goal of our future work is to develop tools that automatically generate management code from such specifications. Benefits of using such a notation and associated tools are:

- Management policies and methods can be specified quite detailed, independently of implementation language and how the object adaptor framework is implemented.
- By use of automatic code generation, the implementation work is simplified and less error prone, because the implementor can ignore some implementation details.

5. 1. Management classes

The management policy and methods can be specified in an object oriented style, as "management classes". In the design described in this paper, management classes are mapped to specializations of the *PermObj* class. Management classification is orthogonal to object types. One management class can be applied to one or more objects, which can be of one or more types. We also allow objects of the same type to have different management classes (e.g. if they are managed differently in different contexts). The main parts of a management class specification are as follows:

Declaration of instance variables: Each variable has a name and a type (e.g. chosen from the set *{STRING, INTEGER, BOOLEAN}*). These variables hold values that serves as identification of permanent objects.

Specification of management policies and methods: First, we need an activator method, which tells how the object adaptor activate an object (or find the current activation). The choice of method can be done by using DBMS-specific predefined procedure names that denote standard activation methods, or by embedding hand written source-code (e.g. C++ or SQL). There can also be necessary to specify if a kind of notification or monitoring strategy is used. The instance variables can be referred to in management specifications.

We can optionally specify a passivator method, possibly with a passivation policy which tells under which conditions the object adaptor should initiate passivation (e.g. after a certain amount of idle time).

The figure below illustrates how a management class can be specified. In this example we assume ANSAware [AR.000.00, AW-APM] as the underlying architecture supporting the applications of interest. ANSAware has certain standard architecture services for notification and for activating and passivating objects (e.g. *Factory, Capsule* or *Object*). The keyword *SERVER* is used to indicate that management is done by operations of another managed object (c.f. the *Server_Perm* class mentioned in section 4.2)

FIGURE 8. Example of management classes

```
MGMT-CLASS PlanetServer =
BEGIN
   VAR nodename  : STRING;
   VAR path      : STRING;
   VAR program   : STRING;
   VAR arg       : STRING;

   ACTIVATOR     = Factory (nodename, path, program, arg);
   PASSIVATOR    = Capsule;
   NOTIFICATION;
END;

MGMT-CLASS Planet =
BEGIN
   SERVER    : PlanetServer;
   VAR name  : STRING;
   VAR pos   : INTEGER;

   ACTIVATOR = Capsule ("Planet", "%name, %pos");
   PASSIVATOR = Object;
   PASSIVATION AFTER 120;
END;
```

5. 2. Permanent object instances

We also need to specify each object instance that is used as “root of persistency” in client applications. The specification of a permanent object can consist of the following parts:

- A name, so that the application program can refer to the object.
- The name of the management class.
- Class specific parameters that assign values to the variables declared in the management class.
- An indication if the object is exported to the federation (e.g. by the keyword “EXPORT”). Objects that are not exported are only used by the object adaptor itself in managent of other objects.

The figure below illustrates how permanent objects can be defined using the management classes from the figure above.

FIGURE 9. Example of permanent objects

```
PERMANENT pserv  = PlanetServer ("odslab2", "/usr/local/bin", "pserver", ""
PERMANENT world1 = Planet (pserv, "Jupiter", 5)EXPORT;
PERMANENT world2 = Planet (pserv, "Saturn", 6) EXPORT;
```

6. Conclusions and future work

In the work described here, we have made a first step towards a set of general tools for designing and implementing object adaptors. This includes a general design and implementation framework that can be extended with system and application specific parts.

In our future work, we will develop further the notation for specifying object management policies, and ultimately realize a prototype tool that can automatically generate application and management code from such specifications. The resulting framework will be evaluated in experiments developing refinements for accessing relational and object oriented databases as well as special purpose data repositories.

References:

[AW-APM] ANSAware 4.1 *Application Programmers Manual*

[AR.000.00] R. J. van der Linden, *An Overview of ANSA*, ANSA architecture report 000.00, APM 1993

[Banc87] F. Bancilhon, T. Briggs, S. Khoshafian, P. Valduriez, *FAD, a Powerful and Simple Database Language*, Proc. 13th VLDB conference, Brighton 1987.

[Blair92] G. Blair, P. Dark, N. Davies, J. Mariani, C. Snape, *Integrated Support for Complex Objects in a Distributed Multimedia Design Environment*, Lancaster University, 1992.

[ElKa91a] F. Eliassen, R. Karlsen, *Interoperability and Object Identity*, CS RR 91.12, October 1991

[ElKa91c] F. Eliassen, R. Karlsen, *Interoperability using Functional and Object Oriented Programming Concepts: Problems and Solutions*, CS RR 91-13, October 1991

[HaeDitt93] M. Härtig, K. R. Dittrich, *An Object-Oriented Integration Framework for Building Heterogenous Database Systems, Interoperable Database Systems* (DS-5) (A-25) 1993 IFIP

[Hanssen94] Ø.Hanssen, *Objekt adapter og programmeringsgrensesnitt for ANSA ware applikasjoner*, Cand Scient thesis, University of Tromsø, november 1994 (in norwegian)

[KhVa90] S. Khoshafian, P. Valduriez, *Sharing, persistence and object-orientation: A database perspective*, In: Advances in Database Programming Languages, F. Bancilhon & P. Buneman (eds), ACM Press

[Nicol93] J. R. Nicol, C. T. Wilkes, F. A. Manola, *Object Orientation in Heterogenous Distributed Computing Systems*, Computer, june 1993, pp. 57-67

[OMG.92.12.1] *The Common Object Request Broker: Architecture and Specification*, OMG Document number 91.12.1, Revision 1.1, Draft, 10. Dec. 1991

[RM-ODP.1] ISO/IEC JTC/SC21/WG7, *ITU-T X.901 ISO/IEC 10746.1, ODP Reference Modeo Part 1*, N. 885 Working document, November 1993

[Rumbaugh91] J. Rumbaugh, M. Blaha,W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall 1991

[Wirf90a] R. J. Wirfs-Brock, R. E. Johnson, Surveying Current Research in Object-Oriented Design, CACM, Vol. 33, No. 9, September 1990

[Wirf90b] R. J. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990