Sunrise Project

Object Database Adapter

Programmer's Guide and Reference Manual

Francisco Reverbel

Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, NM 87545

August, 1996

Abstract

This document describes the Object Database Adapter (ODA) developed as part of the Sunrise Project at Los Alamos National Laboratory. Chapter 1 is an introduction to the adapter services and architecture. Chapter 2 is a programmer's guide for ODA users. Chapter 3 is a reference manual of the ODA API. While the first chapter addresses general Object Database Adapter issues, the remaining two chapters are aimed at the Sunrise ODA release for Orbix 2.x and ObjectStore 4.x.

Contents

1

Introduction

The Sunrise ODA is an Object Database Adapter that provides a general, application-independent way of using an ODBMS back-end to make CORBA objects persistent. Initially targeted at Iona's ORB, Orbix, and Object Design's ODBMS, ObjectStore, the ODA was later ported to a relational engine, mSQL, accessed in an object-oriented fashion (as an ODBMS), through a simple smart pointer-based object/relational mediator. Subsequently the ODA was ported to a second ORB, Visigenic's VisiBroker for C++ (formerly ORBeline). Currently there are ODA releases for:

- Orbix and ObjectStore;
- Orbix, mSQL, and a simple object/relational mediator;
- VisiBroker and ObjectStore.

Future plans include an ODA release for a full-fledged relational DBMS (initially Oracle), accessed through a commercially available object/relational mediator.

1.1 Why an Object Database Adapter?

An Object Database Adapter allows CORBA object implementations to be written in the database programming language of the ODBMS, a language that incorporates persistence into the programming environment. This is usually an extension of the C++ language that provides the programmer with a unified, single-level model of the memory hierarchy. "Single-level store model" means that no explicitly programmed

read/write calls are employed to copy objects back and forth between main memory and magnetic storage. The ODBMS performs these tasks automatically, allowing the application to access persistent objects in the same way it accesses transient objects.

The object implementation — a CORBA server — is still responsible for managing the persistent state of the objects it implements, but the object implementor's task is much simpler in ODBMS programming environment. Besides persistence, other database features – data consistency in the presence of concurrent accesses, crash recovery, and so forth – are available to the object implementation.

In existing database systems, regardless of their data model (object-oriented or relational), database clients must have some knowledge of the database schema. In the case of an object-oriented DBMS, clients need to know the object layout. In a relational DBMS, views can be used for data independence. But relational clients still need to know the external (view level) schema. By contrast, an Object Database Adapter makes database objects accessible to CORBA clients without exposing the database schema to these clients. The data members and the layout of a persistent CORBA object remains private, only its interface (a set of methods) is made public. This is specially interesting for web browser access to databases. With the integration of the Java language into the ORB environment, Java applets can interact with persistent CORBA objects through domain-specific interfaces, without any knowledge of how the objects are actually stored.

ORB/ODBMS integration, as realized by an Object Database Adapter, leads to ORB-connected multidatabases. Together with OTS, the Object Transaction Service[1] specified by the OMG, it enables the construction of distributed object databases that are truly heterogeneous, even with respect to the DBMS software running on the various database server nodes.

1.1.1 Persistence of CORBA Objects: the Issues

The gap between object access times in the ORB and in the ODBMS environment is the first issue an Object Database Adapter must address. Because CORBA clients access objects via remote method invocation, access times for CORBA objects are

[1]OTS implements the two-phase commit protocol in a CORBA environment. Its model can be viewed as an object-oriented extension of the X/Open DTP model.

expressed in milliseconds. Because ODBMSs keep an object cache at the client side, ODBMS clients can access objects much faster: access times for ODBMS objects are typically expressed in microseconds.

Due to the this performance gap, an Object Database Adapter cannot force all accesses to the objects in a database to be made through the ORB remote method invocation mechanism. In the case of a large collection of very small objects, the overhead would be unacceptable. Instead, the ODA should let the object implementor choose a suitable subset of database objects, presumably the higher level ones, to be accessed as CORBA objects. Since this subset may still be large, individual registration of its objects with the ORB is not practical. The ODA should allow a subset of database objects to be accessed through the ORB, without requiring an explicit registration call for each object.

Moreover, in the common case of an ODBMS that adds database features to C++, an object implementation (a CORBA server) cannot simply store in an object database the CORBA objects it implements:

1. It would be a waste of space: a C++ CORBA object has ORB-specific data members that should not be stored in the database. It also has a pair of hidden vbase and vtable pointers for each IDL interface in its inheritance chain up to `CORBA::Object`.

2. More importantly, the CORBA operations `duplicate` and `release` update the object's reference count. If the reference count were actually stored in the database, every operation on the object would have to be performed within an update transaction (because `duplicate` and `release` appear everywhere).

3. Yet more importantly, ORB implementations keep a hash table of active objects: a new entry is inserted into this table whenever the constructor of a CORBA object is invoked. In an ODBMS, the constructor of a persistent object is only invoked when the the object is added to the database. As far as the ORB is concerned, CORBA objects stored by other processes would not be active (the ORB keeps a table of active objects per process).

An Object Database Adapter has to solve these problems, ideally in a way that makes persistent CORBA objects appear exactly like ordinary database objects. As

much as possible, object implementors should be unaware that a persistent CORBA object does not really live in the database.

1.1.2 Persistence of CORBA Object References

Besides providing persistence to CORBA objects, an Object Database Adapter must also provide persistence to the corresponding object references. In CORBA, persistence of object references means that "a client that has an object reference can use it at any time without warning, even if the (object) implementation has been deactivated or the (server) system has been restarted".

With persistence of object references, it makes perfect sense for a client to store an object reference for later use. References to persistent CORBA objects implemented by server X can be stored by server Y (a client of server X) and vice-versa, thereby enabling the construction of ORB-connected multidatabases. In such a multidatabase, references to remote objects are used to express relationships between CORBA objects implemented by different servers. If distributed transactions are needed, they can be supported by a TP monitor that implements the Object Transaction Service specified.

1.1.3 Storability of CORBA Object References

Because an object reference is opaque and ORB-dependent, CORBA provides operations that convert an object reference to string and vice-versa. Object references are stored as strings; upon retrieval they must be converted back to their native form.

Translation to and from string format provides maximum flexibility, allowing object references to be kept in any media. In an ODBMS environment, however, object storage and retrieval are transparent to the programmer. The need for explicit translation of stored references does not prevent the construction of ORB-connected multidatabases, but is unnatural in the context of an object database: ODBMS users expect stored references that behave like any other database object.

Transparent storability of CORBA object references is yet another desirable feature of an Object Database Adapter. It eliminates the need for explicit object reference conversions, both before storage and after retrieval, and allows transparent use of stored references to invoke methods on possibly remote objects.

1.2 The Sunrise ODA

The present document describes the Object Database Adapter developed as part of the Sunrise Project and currently being used by the TeleMed and TeleFlex projects at Los Alamos National Laboratory. Henceforth referred to as ODA, this adapter

- provides an application-independent way of storing C++ CORBA objects in an object database, and
- makes CORBA object references persistent, but not transparently storable[2].

1.2.1 Object Persistence Approaches

To solve the problems of storing C++ CORBA objects in an object database, the ODA uses the delegation (tie) approach to interface implementation. Only implementation objects are stored in the database. The corresponding CORBA objects (tie objects) are automatically instantiated by the ODA whenever they are needed and deleted when not needed. CORBA references to such tie objects are generated and made persistent by the ODA, which

1. embeds a stringfied ODBMS reference to the corresponding implementation object into the `id` (`ReferenceData`) field of a CORBA reference to a tie object;
2. provides an object activation function that (re)constructs the tie object, using the information contained in its `id`.

Even though CORBA objects are not fully stored, clients can use and manipulate these objects as if they lived in an object database. This is called *pseudopersistence*.

In the pseudopersistence approach, all relationships between CORBA objects "stored" in a database are actually relationships between the corresponding implementation objects. Because the traversal of database relationships by a CORBA server involves only persistent implementation objects, not full CORBA objects, such a traversal is performed at ODBMS speeds. The only CORBA objects that need to

[2]A second and experimental adapter, with enhanced functionality, was also produced by the Sunrise Project. In addition to the ODA features, the enhanced adapter provides also transparent storability of CORBA object references. This document does *not* describe the experimental adapter.

be activated are the ones whose references are passed to CORBA clients. To this end, the ODA provides the CORBA server with a primitive that returns a reference to the CORBA object associated with a given implementation object. For efficient implementation of this primitive, the ODA maintains an in-memory table of active pseudopersistent objects. The pseudopersistence approach is supported by the ODA directives `ODA_def_persistent` and `ODA_Def_Persistent`.

The ODA also supports mere *persistence of implementation objects*. This is a simpler approach that makes implementation objects (not full CORBA objects) persistent, but does not make CORBA object references persistent. Hence, persistence of implementation objects is not an actual solution to the problems of object persistence. The usefulness of this option is limited to "top-level objects". Among the many objects implemented by a CORBA server, there is usually a top-level one, which represents the server itself. The top-level object of a server is known to its clients, which use it to start interacting with the server. A top-level object uses simple persistence of implementation and remains active as long as its server is active, because under this approach it can be assigned an `id` (marker, in Orbix terminology) arbitrarily chosen by its server[3]. With this scheme, a server can choose an `id` known to its clients, which in turn use this `id` to locate the server's top-level object[4]. Persistence of implementation objects is supported by the ODA directives `ODA_def_server` and `ODA_Def_Server`.

Persistence of implementation is employed for the top-level object in a CORBA server, pseudopersistence applies to other CORBA objects. Both approaches, however, are aimed at CORBA objects whose implementation objects live in an object database. In these approaches, neither the ODA nor the CORBA server has to deal with the activation and deactivation of implementation objects, performed automatically by the ODBMS as it transfers objects back and forth between main memory and magnetic storage. While this is the typical situation, sometimes it is desirable to have

[3]ODA-assigned object reference `id`s are used in the pseudopersistence approach.

[4]Locating servers by their `id`s is an interim solution. Such practice, supported by Orbix in substitution to an actual name service, is not in the spirit of the OMG architecture: according to CORBA, object `id`s should only be meaningful to the object implementations themselves. This use of `id`s will be dropped when implementations of the CORBA Name Service become available.

implementation objects activated under control of the CORBA server, so that their state can be dynamically retrieved from a variety of data sources. One of these data sources may be an object database kept by the CORBA server. Flat files and other CORBA servers are also possible data sources. The ODA addresses these situations with a third persistence approach, *pseudopersistence with activation of implementation objects*. This is a variation of the pseudopersistence approach, in which the ODA dynamically instantiates not only CORBA objects (tie objects), but also their implementation objects. The persistent state of an implementation object can be retrieved from various data sources, including an object database. Because implementation objects are application-specific, the server writer must provide an activation function for each implementation class. The ODA calls the activation function whenever it needs to create an instance of the implementation class. Pseudopersistence with activation of implementation objects is supported by the ODA directives `ODA_def_activated` and `ODA_Def_Activated`.

1.2.2 Support for Local Transactions

In an ODBMS, every access to a persistent object must be performed within a transaction. The ODA enforces this rule by automatically starting a transaction before each IDL-defined operation on a persistent object begins execution (if there is no transaction already active), and by committing (or aborting) the auto-started transaction at the end of the operation. In the ODA's default transaction mode, each IDL operation corresponds to a database transaction.

Sometimes multiple IDL operations must be performed within a single transaction. The ODA addresses these situations by providing functions that allow grouping a sequence of operations in a single transaction.

1.2.3 The ODA Architecture

The ODA architecture is shown in the figure below. Rather than a replacement of the Basic Object Adapter (BOA) specified by CORBA, the ODA is an add-on to the BOA, implemented as a library that uses and extends the BOA services. No ORB-specific information is kept in the database. Hence, a change of ORB requires no schema evolution. It is even possible for a a single database to be simultaneously

accessed by CORBA servers based upon different ORBs.

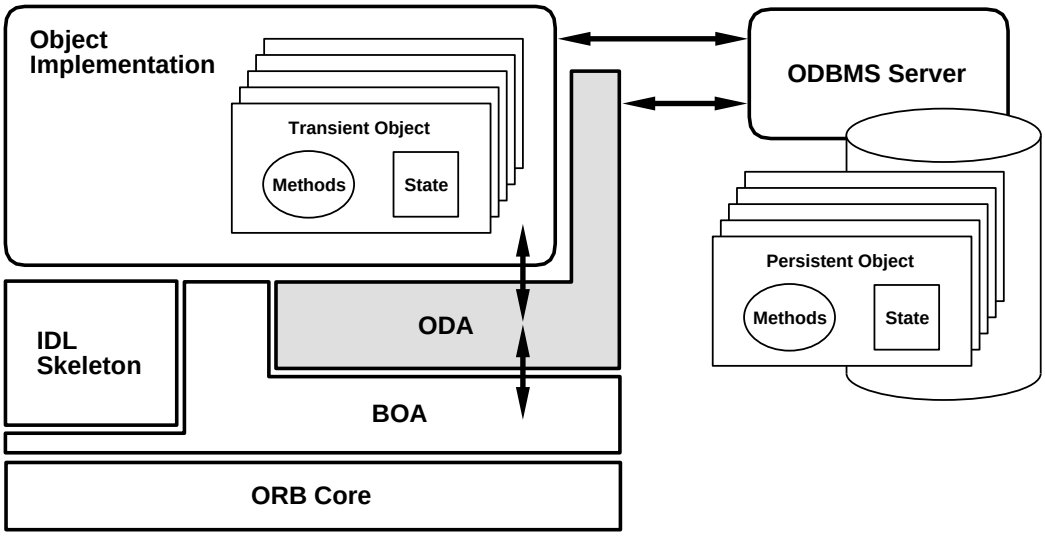

Figure 1.1: The ODA architecture.

1.2.4 Dependency Upon ORB and ODBMS Features

Among the Orbix and VisiBroker features not generally available in all ORBs, the following ones are used by the ODA:

- full support to object activation, through user-defined loader (Orbix) or activator (VisiBroker) classes[5];
- support to the delegation approach to interface implementation;
- incoming request pre-marshal and outgoing reply post-marshal handlers.

The ODBMS requirements are minimal: pseudopersistence uses only object identity (ODBMS object references), plus the ability to convert ODBMS references to strings and vice-versa. Because these are fairly common ODBMS features, other ODBMSs can be employed instead of ObjectStore. Porting the ODA to other ODBMS involves only simple changes in the ODA code. With the use of an object/relational

[5]Although object activation is in the CORBA standard, this feature is listed here because it is neither fully defined by CORBA nor supported by all ORB implementations.

mediator, even a relational DBMS can be employed. In this case, primary keys play the role of ODBMS references. The mSQL release of ODA demonstrates the applicability of the pseudopersistence scheme to relational DBMSs.

2

Programmer's Guide

This chapter shows how a server writer implements persistent CORBA objects using the ODA release for Orbix 2.x and ObjectStore 4.x. It assumes that the reader is familiar with the C++ programming language, and has previous knowledge of Orbix and ObjectStore.

2.1 Getting Started

Consider a simple application in which a CORBA server manages information on the collection of books in a library. In this bare-bones example, a `Book` object only has the attributes `author` and `title`. A `Library` object has a single attribute, the library's `name`, and two operations: `add_Book`, which adds a new book to the library's collection, `get_Book_list`, which returns a list of all the books in the library. This "library server" example will be fully developed in the subsections that follow.

2.1.1 Defining IDL Interfaces

Because the ODA realizes the single-level store model, object persistence is transparent to CORBA clients, which do not have to invoke any "store/restore" operations. ODA usage has no effect on the IDL interfaces available to clients: interfaces of persistent objects are defined exactly like interfaces of transient objects. Figure 2.1 shows the file `library.idl`, which contains the definitions of the interfaces `Book` and `Library`.

```
interface Book {
    readonly attribute string author;
    readonly attribute string title;
};

typedef sequence<Book> BookList;

interface Library {
    readonly attribute string name;
    void add_Book(in string name, in string author);
    BookList get_Book_list();
};
```

Figure 2.1: File `library.idl`.

2.1.2 Writing Implementation Classes

Interfaces defined in IDL are given as input to the Orbix IDL compiler. For each interface, this compiler generates an pair of C++ classes: an *interface class* and an *skeleton class*. Interface classes have the same names as their corresponding IDL interfaces, and contain the stub member functions called by clients.

Skeleton classes are employed at the server side only. As the term "skeleton" suggests, these classes do not implement interfaces. They merely act as a path from the object adapter to the actual *implementation classes* provided by the object implementor. These concepts are defined by CORBA and apply to any ORB implementation, regardless of whether the ODA is used or not.

The single requirement introduced by the ODA is on how skeleton and implementation classes relate to each other: the delegation (tie) approach must be used. In this approach, the IDL compiler generates a special form of skeleton class, called a *tie*. For each such class, the object implementor provides a corresponding implementation class. Tie objects (instances of tie classes), which "tie" implementation objects (instances of implementation classes) to IDL interfaces, are then created at runtime.

In the case of the interfaces in Figure 2.1, the Orbix IDL compiler generates the interface classes `Book` and `Library`. The corresponding implementation classes, say

`Book_impl` and `Library_impl`[1], are provided by the server writer. Instead of using the Orbix TIE macros to "tie together" interface and implementation classes, the server writer uses special ODA macros, as shown in Figure 2.2. By using an ODA macro instead of an Orbix TIE macro, the server writer both ties together an interface and an implementation class and specifies that the corresponding CORBA objects (tie objects) should be made persistent by the ODA. Orbix TIE macros are used for transient objects only.

```
#include "library.hh" // IDL-generated interface and skeleton classes
#include <oda.h> // ODA header file

...  // the definitions of classes Book_impl and Library_impl go here

ODA_def_persistent(Book,Book_impl)
ODA_def_server(Library,Library_impl)
```

Figure 2.2: Specifying that `Book_impl` and `Library_impl` implement persistent CORBA objects.

Two ODA macros — `ODA_def_persistent` and `ODA_def_server` — appear in Figure 2.2. The macro

`ODA_def_persistent(`*`interf`*`,`*`impl`*`)`

is used for interface classes of objects made persistent through ObjectStore. An interface class that corresponds to the "top-level" object of a server, however, is an exception to this rule. For such a class, the macro

`ODA_def_server(`*`interf`*`,`*`impl`*`)`

is used instead. Although a CORBA server may implement a large number of objects, it typically has one top-level object, which represents the server itself. Within a server, `ODA_def_server` is used for the interface class of the server's top-level object, and `ODA_def_persistent` is used for all other interface classes whose implementation objects live in ObjectStore databases.

[1]Names of implementation classes are completely arbitrary.

Referring to Figure 2.2, the library server uses `ODA_def_server` for class `Library`, because this is the interface class of the server's top-level object. This example uses `ODA_def_persistent` just for class `Book`.

Persistent relationships between CORBA objects are realized in terms of the underlying implementation objects. In the library server example, a `Library` has a collection of `Book`s. Figure 2.3 shows how this fact is expressed at the level of the implementation classes `Library_impl` and `Book_impl`, using the ObjectStore template class `os_List`.

```
// implementation class for Book
class Book_impl {
    ...
};

// implementation class for Library
class Library_impl {
public:
    ...
private:
    ...
    os_List<Book_impl*>& _book_list; // n.b.:  Book_impl, not Book
};
```

Figure 2.3: The relationship between a library and its books is expressed as a relationship between `Library_impl` and `Book_impl` objects.

2.1.3 Obtaining CORBA References to Persistent Objects

By following object relationships expressed at the level of implementation classes, the server code reaches implementation objects. Whenever the corresponding CORBA objects are needed, they can be obtained via calls to the ODA.

Suppose that the library server has a pointer to an implementation object of class `Book_impl`, and wants the corresponding CORBA object, of class `Book`. The function

```
Book_ptr ODA_persistent_Book(Book_impl* p);
```

returns a `Book_ptr` (an object reference to a `Book`) given a pointer to a `Book_impl`. The C++ code for this function is generated by the ODA, as a result of the expansion of the macro call

```
ODA_def_persistent(Book,Book_impl)
```

The server writer does not need to provide a prototype for `ODA_persistent_Book`; the prototype is generated by the macro call `ODA_def_persistent(Book,Book_impl)`. Normally such a macro call will be placed in a header file, which will be included by any source files that use the function `ODA_persistent_Book`. In general, a macro call

```
ODA_def_persistent(interf,impl)
```

declares and defines the ODA-generated function

```
interf_ptr ODA_persistent_interf(impl* p);
```

Figure 2.4 shows how the operation `get_Book_list` uses `ODA_persistent_Book` to get CORBA references to persistent `Book` objects. In each call, the argument to the function `ODA_persistent_Book` is the corresponding implementation object, of class `Book_impl`, retrieved from the `_book_list`. Note that the server code does not call `_duplicate` on the `Book` references obtained from `ODA_persistent_Book`, because this function already returns `_duplicate`d references.

```
BookList* Library_impl::get_Book_list(CORBA::Environment&) const {
    CORBA::ULong len = _book_list.cardinality();
    Book_ptr* buf = BookList::allocbuf(len);
    for (CORBA::ULong i = 0; i < len; i++)
        buf[i] = ODA_persistent_Book(_book_list.retrieve(i));
    return new BookList(len, len, buf, 1);
}
```

Figure 2.4: Using `ODA_persistent_Book` to translate pointers to `Book_impl` objects into references to `Book` objects.

A call to the ODA is the only way the library server can translate a pointer to a `Book_impl` object into a reference to a persistent `Book` object. If this `Book` object is

not currently active, the ODA creates a tie to the implementation (`Book_impl`) object passed to `ODA_persistent_Book`, and embeds a stringfied ObjectStore reference to this implementation object into the `id` field of the `Book` object reference. Object references returned by calls to the ODA are persistent, in the sense specified by CORBA.

2.1.4 Generating the Database Schema

Database schema generation is performed with ObjectStore's schema generator, `ossg`, by following the procedure described in the ObjectStore documentation. Unfortunately, `ossg` does not yet accept some C++ constructs generally accepted by C++ compilers. Because these constructs appear in Orbix header files and in header files generated by the Orbix IDL compiler, such files cannot be exposed to `ossg`. Conditional compilation must be used to circumvent this problem, as illustrated by Figures 2.5, 2.6, and 2.7.

Figure 2.5 shows file `schema.cc`, the schema definition file for the library example.

```
// This file is _not_ compiled with CC.
// Instead, it is processed by ossg (ObjectStore's schema generator),
// which creates the database schema.

#include <ostore/ostore.hh>
#include <ostore/manschem.hh>

#ifndef _SCHEMA_
#define _SCHEMA_
#endif

#include "library.h" // contains "#ifndef _SCHEMA_" directives

void dummy() {
    OS_MARK_SCHEMA_TYPE(Library_impl);
    OS_MARK_SCHEMA_TYPE(Book_impl);
}
```

Figure 2.5: File `schema.cc`, the schema definition file of the library example.

The symbol _SCHEMA_ will be #defined during schema generation only. Header files included by the schema definition file should use #ifndefs to hide from ossg all Orbix header files, IDL-generated header files, and CORBA-defined types.

Figures 2.6 and 2.7 show the file library.h, which contains the complete declarations of the implementation classes in the library server. Since this file is included by schema.cc, it uses directives "#ifndef _SCHEMA_" to hide from ossg the IDL-

```
#ifndef LIBRARY_H
#define LIBRARY_H

#ifndef _SCHEMA_
#include "library.hh" // IDL-generated interface and skeleton classes
#endif

#include <oda.h> // ODA header file
#include <aux/pstring.h> // auxiliary persistent string class, String

// implementation class for Book
class Book_impl {
public:
    Book_impl(const char* author, const char* title);
    virtual ~Book_impl();
    static os_typespec* get_os_typespec(); // for ObjectStore
#ifndef _SCHEMA_
    char* author(CORBA::Environment& env) const {
        return _author.corba_string();
    }
    char* title(CORBA::Environment& env) const {
        return _title.corba_string();
    }
#endif
private:
    String _author;
    String _title;
};
```

Figure 2.6: File library.h, which declares the implementation classes of the library example.

```
// implementation class for Library
class Library_impl {
public:
    Library_impl(const char* name = "");
    virtual ~Library_impl();
    static os_typespec* get_os_typespec(); // for ObjectStore
#ifndef _SCHEMA_
    char* name(CORBA::Environment& env) const {
        return _name.corba_string();
    }
    void add_Book(const char* name, const char* author,
                  CORBA::Environment& env);
    BookList* get_Book_list(CORBA::Environment& env) const;
#endif
private:
    String _name;
    os_List<Book_impl*>& _book_list; // n.b.:  Book_impl, not Book
};

// ODA directives
#ifndef _SCHEMA_
ODA_def_server(Library,Library_impl); // top-level obj in server is a Library
ODA_def_persistent(Book,Book_impl);   // Books are made persistent by the ODA
#endif // _SCHEMA_

#endif // LIBRARY_H
```

Figure 2.7: File `library.h`, which declares the implementation classes of the library example (continued).

generated header file `library.hh`, the CORBA-defined type `CORBA::Environment`, and the IDL-defined types `Library`, `Book`, and `BookList`. When inserting these `#ifndef`s, care must be taken to ensure that the declarations handled to `ossg` (the ones in effect when `_SCHEMA_` is `#define`d) are equivalent to the declarations handled to the C++ compiler (the ones in effect when `_SCHEMA_` is not `#define`d) with respect to the memory layout of objects. For schema equivalence rules, see the chapter on schema evolution in the ObjectStore User's Guide.

2.1.5 Writing the Server Mainline

Figures 2.8 and 2.9 show the file `server.cc`, which implements the `main` function of the library server. This Orbix server is also an ObjectStore client; it gives CORBA clients access to objects persistently stored in an ObjectStore database. As such, its mainline has initialization calls to both Orbix and ObjectStore. It also calls `ODA::initialize`, the initialization function of the ODA.

```
#include <iostream.h>
#include "dbname.h"

// one and only one source file must #define ODA_DEFS before including
// oda.h (since library.h includes oda.h, we are doing it here).
#define ODA_DEFS
#include "library.h"

CORBA::ORB_ptr orb;

int main(int argc, char **argv)
{
    // ObjectStore setup
    objectstore::initialize();
    os_collection::initialize();
    OS_ESTABLISH_FAULT_HANDLER;
    os_database* db;

    try {
        // initialize ORB
        orb = CORBA::ORB_init(argc, argv, "Orbix");

        // open the database
        db = os_database::open(app_db_name, 0, 0664);

        // initialize ODA
        ODA::initialize();

        // register update operations with the ODA
        ODA::register_update_op("Library", "add_Book");
```

Figure 2.8: File `server.cc`, the CORBA server mainline

```
        // start a database transaction and look for the Library_impl object
        os_transaction::begin(os_transaction::update);
        os_database_root* a_root = db->find_root("Library_impl");
        if (!a_root) { // then create it
            a_root = db->create_root("Library_impl");
            a_root->set_value(
                new(db, Library_impl::get_os_typespec()) Library_impl(),
                Library_impl::get_os_typespec()
            );
        }
        // get the Library_impl object
        Library_impl* lib_impl = (Library_impl*)
            a_root->get_value(Library_impl::get_os_typespec());

        // create the top-level server object, a tie to this Library_impl
        ODA_server(Library,Library_impl) lib(lib_impl);

        // server is ready to handle requests
        CORBA::Orbix.impl_is_ready("LibraryDB", 0);

        // end of transaction, changes are committed to the database
        os_transaction::commit();

        // start the server event loop
        CORBA::Orbix.processEvents(CORBA::Orbix.INFINITE_TIMEOUT);
    }
    catch (const CORBA::SystemException& excep) {
        cout << "Server exception:  " << endl;
        cout << (CORBA::SystemException*) &excep << endl;
    return 1;
    }
    catch (...)  {
        cerr << "Unexpected server exception" << endl;
    }
    db->close();
    OS_END_FAULT_HANDLER;
    return 0;
}
```

Figure 2.9: File `server.cc`, the CORBA server mainline (continued)

The ODA ensures that every IDL operation is executed within a database transaction. Its default transaction mode, used by the library server, is *transaction per operation*. In this mode, the ODA starts and ends transactions so that each operation is encompassed by an individual transaction. Read-only transactions are used by default. The call to `ODA::register_update_op` in Figure 2.8 tells the ODA to use an update transaction in the case of the operation `add_Book` of the `Library` interface.

Before calling `impl_is_ready` on `CORBA::Orbix`, the server mainline creates the top-level server object, an instance of the class `ODA_server(Library,Library_impl)`. This is a "tie class" that ties together the top-level interface class `Library` and its implementation class `Library_impl`. Its declaration is generated by the expansion of the macro call

```
ODA_def_server(Library,Library_impl)
```

in file `library.h` (see Figure 2.6).

The top-level server object is created by the line

```
ODA_server(Library,Library_impl) lib(lib_impl)
```

of `server.cc` (see Figure 2.9). This line is a variable declaration. It declares and defines the variable `lib`, of type `ODA_server(Library,Library_impl)`, which is initialized to a server object that corresponds to the implementation object `lib_impl`.

Finally, note the definition of the symbol `ODA_DEFS` in Figure 2.8. The ODA macros `ODA_def_server` and `ODA_def_persistent` expand into declarations and definitions of ODA-generated classes and functions. While the declarations may appear in many source files, the definitions should appear just once in all source files. If the symbol `ODA_DEFS` is `#defined` by the time `oda.h` is included, both the declarations and the definitions are generated. Otherwise, only declarations are generated. One and only one source file must `#define ODA_DEFS` before including the server header files that call the macros `ODA_def_server` and `ODA_def_persistent`. These header files must include `oda.h`. In the library example, `library.h` is the only such header file.

2.1.6 Compiling and Linking the Library Server

Most of the C++ code for the library example was already presented. The file `library.cc`, which contains member function definitions for classes `Book_impl` and

Library_impl, is shown in Figure 2.10. (One such member function was already presented in Figure 2.4. For completeness, it appears again in Figure 2.10.)

```
#include "library.h"

Book_impl::Book_impl(const char* author, const char* title)
    : _author(author), _title(title) {}

Book_impl::~Book_impl() {}

Library_impl::Library_impl(const char* name)
    : _name(name),
      _book_list(os_List<Book_impl*>::create(os_segment::of(this))) { }

Library_impl::~Library_impl() {}

void Library_impl::add_Book(const char* name, const char* author,
                            CORBA::Environment&) {
    _book_list.insert(new(os_segment::of(this),
                      Book_impl::get_os_typespec()) Book_impl(name, author));
}

BookList* Library_impl::get_Book_list(CORBA::Environment&) const {
    CORBA::ULong len = _book_list.cardinality();
    Book_ptr* buf = BookList::allocbuf(len);
    for (CORBA::ULong i = 0; i < len; i++)
        buf[i] = ODA_persistent_Book(_book_list.retrieve(i));
    return new BookList(len, len, buf, 1);
}
```

Figure 2.10: File library.cc, containing member function definitions for classes Book_impl and Library_impl

.

The set of source files for the library server consists of:

- library.idl, shown in Figure 2.1;
- library.h, shown in Figures 2.6 and 2.7;
- library.cc, shown in Figure 2.10;

- `server.cc`, shown in Figures 2.8 and 2.9;
- `schema.cc`, shown in Figure 2.5.

File `library.idl` is given as input to the Orbix IDL compiler. File `schema.cc` is processed by the ObjectStore schema generator, `ossg`. The remaining files, as well as the C++ source files generated the IDL compiler (`library.hh`, `libraryC.cc`, and `libraryS.cc`) and the one generated by `ossg` (`osschema.cc`), are given as input to the C++ compiler. Figures 2.11, 2.12, and 2.13 present a `Makefile` that builds the library server in a Solaris 2.x (SunOS 5.x) environment.

```
#
# Database files will be placed in LIBRARY_DB_DIR
#

# Compiler flags
CCC             = CC # Using Sun C++ compiler
CCFLAGS         = -pta -vdelx -mt
CPPFLAGS        = -I. -I$(ODA_ROOT)/include \
                  -I$(ORBIX2_HOME)/include -I$(OS_ROOTDIR)/include

# Link flags and libraries
LDFLAGS         = -L$(ODA_ROOT)/lib -L$(ORBIX2_HOME)/lib -L$(OS_ROOTDIR)/lib
SRVLIBS         = -loda -loscol -los -losthr -mt \
                  -lorbix -lITini -Bdynamic -lnsl -lsocket
CLTLIBS         = -lorbix -lITini -Bdynamic -lnsl -lsocket

# Schema flags and macros
SCHEMA_SRC      = schema.cc
APP_SCHEMA_SRC  = osschema.cc
APP_SCHEMA_OBJ  = osschema.o
APP_SCHEMA_HDRS = library.h # just one in this case (there could be more)
APP_SCHEMA_DB   = $(SCHEMA_DB_DIR)library.adb
LIB_SCHEMA_DBS  = $(OS_ROOTDIR)/lib/liboscol.ldb
SCHEMA_DB_DIR   = $(LIBRARY_DB_DIR)/
APP_DB_DIR      = $(LIBRARY_DB_DIR)/
```

Figure 2.11: Makefile for the library server

.

```
# Other flags and macros
EXECUTABLES     = server
APP_DB          = $(APP_DB_DIR)library.db
IDL             = $(ORBIX2_HOME)/bin/idl
IDLFLAGS        = -c C.cc -s S.cc

### Targets ###

all:  ${EXECUTABLES}

### IDL targets ###
# Generate library.hh, libraryC.cc, and libraryS.cc from library.idl
idl library.hh libraryC.cc libraryS.cc:  library.idl

        $(IDL) $(IDLFLAGS) library.idl

### Schema targets ###

# Generate $(APP_SCHEMA_SRC) and $(APP_SCHEMA_DB) from
# $(SCHEMA_SRC) and $(LIB_SCHEMA_DBS)
$(APP_SCHEMA_SRC): $(SCHEMA_SRC) $(APP_SCHEMA_HDRS)
        ossg -assf $(APP_SCHEMA_SRC) -asdb $(APP_SCHEMA_DB) \
                $(CPPFLAGS) $(SCHEMA_SRC) $(LIB_SCHEMA_DBS)

# Build $(APP_SCHEMA_OBJ) from $(APP_SCHEMA_SRC)
$(APP_SCHEMA_OBJ): $(APP_SCHEMA_SRC)
        $(CCC) $(CPPFLAGS) $(TFLAGS) -c $(APP_SCHEMA_SRC)

### Other targets ###

server :  server.o library.o libraryS.o $(APP_SCHEMA_OBJ)
        $(CCC) -o server server.o library.o libraryS.o \
                $(APP_SCHEMA_OBJ) $(LDFLAGS) $(SRVLIBS)
        os_postlink server

server.o :  dbname.h server.cc library.h libraryS.cc
```

Figure 2.12: Makefile for the library server (continued)

.

```
library.o :  library.cc library.h libraryS.cc

dbname.h:  Makefile
        echo 'static char *app_db_name= "$(APP_DB)";' > dbname.h

clean:
        -osrm -f $(APP_SCHEMA_DB) $(APP_DB)
        -rm -f $(EXECUTABLES) $(APP_SCHEMA_SRC) $(APP_SCHEMA_OBJ) *.o *.hh
        -rm -f dbname.h core *.~*~ *C.cc *S.cc
        ptclean

### Rules ###

.SUFFIXES: .o .cc

.cc.o:
        $(CCC) $(CPPFLAGS) $(CCFLAGS) -c -o $@ $<
```

Figure 2.13: Makefile for the library server (continued)

.

This `Makefile` is Solaris-specific. Changes will be probably needed in order to build the library server on other operating system. When modifying the `Makefile`, one should pay special attention to `ossg` usage and switches, which tend to vary among operating systems.

2.2 Additional Topics

For simplicity, the "library server" example avoided a number of issues. These issues are examined in the following subsections.

2.2.1 Server-Side Exception Handling

At the client side, exceptions work exactly as described in the Orbix documentation. Orbix clients handle exceptions in the same way, regardless of whether they use an ODA-based server or not.

At the server side, however, differences arise when the ODA is used. This is a consequence of the differences between the exception mechanisms employed by Orbix and by ObjectStore. Orbix relies on native C++ exceptions; ObjectStore uses its own exception handling facility (TIX exceptions).

To aid the server writer, the ODA provides a set of exception macros that unifies the exception handling mechanisms of Orbix and ObjectStore. These macros, defined in the header file `oda/except.h`, mimic the C++ style of exception handling. They are similar to the `TRY`/`CATCH` macros provided by Orbix for use with C++ compilers without exception handling. Four exception macros are defined in `oda/except.h`: `TRY`, `CATCH`, `CATCHANY`, and `ENDTRY`. Figure 2.14 illustrates their usage.

```
TRY {
    ...  // do something
}
CATCH(SomeException, some_except) {
    ...  // branch to this point if SomeException
}
CATCH(CORBA::SystemException, sys_except) {
    ...  // branch to this point if CORBA::SystemException
}
CATCHANY {
    ...  // branch to this point if any other exception
}
ENDTRY
...  // proceed here if no exception
```

Figure 2.14: `TRY`/`CATCH` macros for server-side exception handling.

Whenever a `TRY` clause is used, an `ENDTRY` and must appear at the end. The `CATCH` and `CATHCHANY` clauses are optional and have the same semantics as the corresponding C++ constructs. That is, the clause

```
CATCH(SomeException, some_except)
```

works like the C++ construct

```
catch(SomeException& some_except)
```

and a `CATCHANY` clause works like the C++ construct `catch(...)`.

Both Orbix and ObjectStore exceptions may be raised within a `TRY` block. The `TRY` macro converts ObjectStore TIX exceptions to native C++ exceptions. All ObjectStore exceptions are mapped to a specific system exception standardized by CORBA, `CORBA::PERSIST_STORE`. The "report" string of an ObjectStore exception is passed as a parameter to the constructor of the system exception `CORBA::PERSIST_STORE`. Occurrence of an ObjectStore exception within a `TRY` block has also the effect of aborting the current database transaction.

When an exception is raised inside a `TRY` block, control is *immediately* transferred to the appropriate `CATCH` or `CATCHANY` block. If the exception was not generated by ObjectStore, it works exactly like a native C++ exception: any objects declared inside the `TRY` block have their destructors called upon exit of the block. In the case of an ObjectStore exception just mapped to `CORBA::PERSIST_STORE`, however, these destructors may *not* be called. For this reason one should not declare, within a `TRY` block that may raise ObjectStore exceptions, any local objects that allocate extra memory via operator `new` and that rely upon their destructors to release this memory. This restriction applies only to ObjectStore exceptions just generated and mapped to `CORBA::PERSIST_STORE`, and not to the propagation of `CORBA::PERSIST_STORE` exceptions.

Note that there is no exception macro to raise an exception. Exceptions are raised using the C++ statement `throw`. Uncaught exceptions are propagated to the caller, up to the CORBA client. Consider, for instance, an IDL operation that does not handle any exception, but simply converts ObjectStore exceptions into C++ exceptions that propagate up to the CORBA client. The implementation of such an operation would look like

```
void some_operation_implementation(CORBA::Environment& env) {
    // the TRY block is here just to convert
    // ObjectStore exceptions into C++ exceptions
    TRY {
        ... // perform operation  \\
    }
    CATCHANY { throw; } ENDTRY
}
```

Revisiting the Library Example

The library server, as implemented in Section 2.1, does not handle any ObjectStore exceptions that may be raised during the execution of IDL operations. The occurrence of such an exception would therefore cause the server to exit. To fix this problem, exception handling code should be added to each of the functions that implement an IDL operation:

- `Book_impl::author`,
- `Book_impl::title`,
- `Library_impl::name`,
- `Library_impl::add_Book`, and
- `Library_impl::get_Book_list`.

The function `Library_impl::add_Book`, for example, should be rewritten as shown in Figure 2.15. This way an ObjectStore exception raised within the `TRY` block is converted into a `CORBA::PERSIST_STORE` exception. Rather than causing the server to exit, the converted exception is now passed on to the CORBA client.

```
void Library_impl::add_Book(const char* name, const char* author,
                            CORBA::Environment&) {
    TRY {
        _book_list.insert(new(os_segment::of(this),
                          Book_impl::get_os_typespec()) Book_impl(name,
                                                                  author));
    }
    CATCHANY { throw; } ENDTRY
}
```

Figure 2.15: Using a `TRY` block in an IDL operation

.

2.2.2 Deleting Persistent CORBA Objects

The function

```
void ODA::Delete(CORBA::Object_ptr objp);
```

must be called to delete a persistent CORBA object. This function deletes just a "tie object" managed by the ODA, not its implementation object. It should be called when the implementation object is removed from its database: the destructor of the implementation object should call `ODA::Delete`.

Revisiting the Library Example

The library server, as implemented in Section 2.1, does not provide any way to remove a book from the library's collection. There is no IDL operation that performs this task. If there were such an IDL operation, its implementation would simply call the C++ operator `delete` on a persistent `Book_impl` object. The destructor of this should be written as shown in Figure 2.16. In this destructor, the call to `ODA::Delete` has the purpose of ensuring that the corresponding "tie object" is also deleted.

```
Book_impl::~Book_impl() {
    ODA::Delete(ODA_persistent_Book(this));
};
```

Figure 2.16: Using `ODA::Delete`.

2.2.3 Referential Integrity of Persistent CORBA Objects

A problem arises if persistent CORBA objects may be deleted upon request of clients. Suppose that clients A and B both hold CORBA references to a given persistent object. If client B uses its reference to delete the object, the reference held by client A will be no longer valid. Client A, not being aware of this fact, might still use its reference to issue an operation on a persistent object that does not exist anymore. In the absence of referential integrity, such an attempt results in a server crash. *Referential integrity* means that the CORBA server checks the validity of incoming object references before

using them. With referential integrity, a client may still attempt to issue an operation on a deleted object. In this case, rather than causing a server crash, the client will get a system exception `CORBA::OBJECT_NOT_EXIST`.

The ODA supports referential integrity of persistent CORBA objects. Because referential integrity has a cost, it is provided as an optional feature: the ODA library comes in two versions, one without support to referential integrity, another one with support to referential integrity. Usage of referential integrity is highly recommended in any environment that admits deletion of persistent CORBA objects. To have referential integrity, the server writer must do three things:

1. `#define` the symbol `oda_REF_PROTECTED` before each inclusion of the ODA header `oda.h`.

2. Employ the appropriate ODA library. In an Unix environment, this means specifying the `-lodap` switch in the server link line. The server is therefore linked with the library `libodap.a`, which supports referential integrity.

   In the same Unix environment, a server that does not want referential integrity should be generated with the `-loda` switch in the link line. The server is therefore linked with the library `liboda.a`, which does not support referential integrity.

   The ODA library employed at link time must be consistent with the definition of `oda_REF_PROTECTED` at compile time: `-lodap` can be used if and only if a definition of `oda_REF_PROTECTED` was seen, at compile time, before each inclusion of `oda.h`.

3. Ensure that each persistent implementation object that may be deleted is registered as such with ObjectStore, at object creation time.

Item 3 is neccessary because the ODA support to referential integrity of persistent CORBA objects relies upon the corresponding ObjectStore feature for implementation objects. It is accomplished by creating, for each implementation object that may be deleted, an instance of the ObjectStore-defined class `os_Reference_protected` referring to the implementation object. This should be done within the constructor of the implementation object, as shown in the following example.

Revisiting the Library Example

To have referential integrity of `Book` objects, the `Book_impl` constructor should be rewritten as shown in Figure 2.17. Note that the variable `my_os_ref_protected` is local to the constructor block. The short lifetime of this variable is not a problem, because ObjectStore retains information on protected references even after their destruction.

```
Book_impl::Book_impl(const char* author, const char* title)
    : _author(author), _title(title) {
    os_Reference_protected<Book_impl> my_os_ref_protected = this;
};
```

Figure 2.17: Registering `Book_impl` objects with ObjectStore, for referential integrity.

2.2.4 Database Transactions

In ObjectStore, every access to persistent data must be performed within a database transaction. Even so, the library server presented in Section 2.1 does not contain explicit ObjectStore calls to start and commit transactions. The library example uses the ODA's default transaction mode, *transaction per operation*. In this mode, each IDL operation corresponds to a database transaction. Before an IDL operation begins executing, the ODA automatically starts a database transaction. At the end of the operation, the ODA commits (or aborts) the transaction.

In the transaction per operation mode, auto-started transactions are automatically committed by the ODA at the end of the current operation, except for the following cases:

1. If an ObjectStore-generated `CORBA::PERSIST_STORE` exception occurs, the ODA immediately aborts the auto-started transaction.

2. If the Orbix server calls `ODA::abort_transaction`, the ODA aborts the auto-started transaction at the end of the current operation.

3. If the Orbix server calls `ODA::multi_op_transaction_mode`, the ODA switches to *multioperation transaction mode*. The auto-started transaction is neither committed nor aborted by the ODA the at the end of the current operation.

Grouping IDL Operations in a Single Transaction

A call to `ODA::multi_op_transaction_mode` instructs the ODA to extend the duration of the current auto-started transaction. Instead of being committed at the end of the current operation, this transaction will encompass subsequent operations. In multioperation transaction mode, the ODA ceases auto-starting transactions at the beginning of each operation. A sequence of operations is performed within a single transaction (the last auto-started one), until one of the following events occur:

1. If an ObjectStore-generated `CORBA::PERSIST_STORE` exception is raised, the ODA immediately aborts the transaction.

2. If the Orbix server calls `ODA::commit_transaction`, the ODA commits the transaction at the end of the current operation.

3. If the Orbix server calls `ODA::abort_transaction`, the ODA aborts the transaction at the end of the current operation.

Each of these events reverts the transaction mode back to transaction per operation.

Defining the Type of Auto-Started Transactions

By default, auto-started transactions are read-only transactions. The ODA starts update transactions only in the case of IDL operations previously registered as update operations. A call to the function

```
void ODA::register_update_op(const char* interf, const char* op);
```

must be issued for each update operation.

2.2.5 Implementation Objects Not Managed By ObjectStore

Managing persistent CORBA objects whose implementation objects live in an ObjectStore database is the most commonly used ODA service. The is the case addressed by the macro `ODA_def_persistent`, which

- assumes that implementation objects are managed by ObjectStore, and therefore

- instructs the ODA to manage CORBA objects, but not their corresponding implementation objects.

In this setting, ObjectStore is responsible for activating and deactivating implementation objects, as it transfers these objects back and forth between main memory and magnetic storage.

In some situations[2], however, keeping entire implementation objects in an ObjectStore database is undesirable. Instead, just part of the implementation object's state should be kept in the database. The rest of the object's state should be dynamically retrieved from other data sources, such as flat files or other CORBA servers. The ODA addresses these situations with the macro

`ODA_def_activated(`*`interf`*`,`*`impl`*`)`

`ODA_def_activated` is used for persistent CORBA objects whose implementation objects are managed by the ODA, not by ObjectStore. It is similar to the macros `ODA_def_persistent` and `ODA_def_server`, in that it "ties together" an interface class and an implementation class. With `ODA_def_activated`, however, implementation objects are activated by the ODA, through calls to an activation function provided by the server writer. This scheme gives the server writer complete freedom for specifying the set of data sources from which the activation function will retrieve the object's state.

Writing ODA-Activated Implementation Classes

An implementation class passed as the *impl* parameter to `ODA_def_activated` must have `ODA::Activated` as a public base class:

```
class MyImplementationClass : public ODA::Activated {
    ...
};
```

[2]As an example of such a situation, consider the TeleMed system, a CORBA-based virtual patient record system developed at Los Alamos National Laboratory. A TeleMed server (a CORBA server) dynamically harvests patient data from multiple repositories. While part of the information on a given patient may be stored in an ObjectStore database local to the server, additional information may come from other TeleMed servers. Both local and remote data is merged into a "virtual patient" object, which encapsulates all the available information on the patient.

Moreover, the implementation class must redefine three functions inherited from class `ODA::Activated`: `_key`, `_id`, and `_activate`.

```
class MyImplementationClass : public ODA::Activated {
    // private section
    //   application-specific data members and/or member functions
    ...
public:
    // public section
    //   application-specific data members and/or member functions
    ...
    //   redefinition of functions inherited from class ODA::Activated
    virtual CORBA::ULong _key();
    virtual char* _id();
    static void* _activate(const char* id);
};
```

The purpose and the specification of the functions `_key`, `_id`, and `_activate` is:

`CORBA::ULong _key();`

This is a public member function declared as pure virtual in the base class `ODA::Activated`. The server writer should redefine it in the implementation class, so that it returns the object's key.

The key of an implementation object should be a 32-bit integer that uniquely identifies the object amongst all instances of its implementation class. The server writer is responsible for assigning keys to implementation objects. Distinct instances of a given implementation class should have different key values. Instances of different implementation classes, however, may have the same key value.

`char* _id();`

This is also a public member function declared as pure virtual in the base class `ODA::Activated`. The server writer should redefine it in the implementation class, so that it returns a buffer allocated via `CORBA::String_alloc`, containing the implementation object's ID.

The ID of an implementation object should be a null-terninated string that uniquely identifies the object amongst all instances of its implementation class. IDs and keys play the same role, but have different types: keys are integer values, IDs are strings. When assigning IDs to implementation objects, the server writer is free to choose object IDs that do not bear any relationship with object keys. The natural choice, however, is to pick object IDs that depend upon object keys (by converting keys to strings of hexadecimal digits, for instance).

```
static void* _activate(const char* id);
```

This public function is declared as static by the base class `ODA::Activated`. The server writer should redefine `_activate` in the implementation class, so that it activates the implementation object identified by the `id` parameter. The redefined function should create the implementation object in virtual memory (possibly after retrieving its state from various data sources) and return its address.

An ODA-activated implementation object may have part of its state in an ObjectStore database. In a typical setting, the "main part" of the implementation object, instantiated in transient memory, would contain an ObjectStore reference to the object's "persistent part", which lives in an ObjectStore database. The function `_activate` should construct the main part of the object and initialize its data members, including the one that refers to the persistent part of the object.

Obtaining CORBA References to ODA-Activated Objects

The macro call

`ODA_def_activated(`*interf*`,`*impl*`)`

declares and defines the ODA-generated functions

```
interf_ptr ODA_activated_interf(impl* p);
interf_ptr ODA_activated_interf(long keyval);
```

Both functions return an object reference to the CORBA object that corresponds to a given implementation object. The first one receives as input parameter a pointer to the

implementation object[3]. The second one receives as input parameter the implementation object's key. It returns a null pointer if no active instance of the implementation class has this key value.

2.2.6 IDL Interfaces Defined Within a Module

The macros

`ODA_def_server(`*interf*`,`*impl*`)`

`ODA_def_persistent(`*interf*`,`*impl*`)`

`ODA_def_activated(`*interf*`,`*impl*`)`

work only in the case of IDL interfaces not defined within a module. A similar set of macros is provided for use in the case of interfaces defined within an IDL module:

`ODA_Def_Server(`*module*`,`*interf*`,`*impl*`)`

`ODA_Def_Persistent(`*module*`,`*interf*`,`*impl*`)`

`ODA_Def_Activated(`*module*`,`*interf*`,`*impl*`)`

The first parameter taken by these macros is the name of the IDL module in which *interf* is defined.

2.2.7 Persistent Strings

The library server presented in Section 2.1 uses an auxilary class, `String`, whose instances are persistent strings. Class `String` is especially useful for CORBA servers that need to copy persistent strings into dynamically allocated CORBA strings (i.e., strings allocated via `CORBA::String_alloc`). Such task, performed whenever an IDL operation returns a string retrieved from an ObjectStore database, is encapsulated by the member function `String::corba_string`.

Class `String` is implemented by the header file `aux/pstring.h` provided with the ODA release. This class is not really part of the object database adapter, nor does it make use of the adapter. It is included in the ODA release just because it is likely to be useful to ODA applications.

---

[3]This function is similar to the function `ODA_persistent_`*interf*, generated by the macro call `ODA_def_persistent(`*interf*`,`*impl*`)`.

3

Reference Manual

This chapter describes the C++ API provided by the ODA release for Orbix 2.x and ObjectStore 4.x. The ODA C++ interface consists of the following elements:

- the class `ODA`;
- persistence directives, implemented as C++ macros;
- ODA-generated functions on persistent CORBA objects;
- exception handling directives, implemented as C++ macros.

The class `ODA` and the persistence directives are defined in the header file `oda.h`. The C++ code of the functions on persistent CORBA objects is generated by the macro expansion of persistence directives. The exception handling directives are defined in the header file `oda/except.h`.

3.1 The class ODA

Figure 3.1 shows the public members of this class.

```
class ODA {
public:
    static void initialize(unsigned object_cache_size = 1023,
                           unsigned update_ops_table = 1023,
                           unsigned interf_table_size = 101);
    static void multi_op_transaction_mode();
    static void commit_transaction();
    static void abort_transaction();
    static void register_update_op(const char* interf_name,
                                   const char* op_name);
    static void Delete(CORBA::Object_ptr objp);
private:
    ...
};
```

Figure 3.1: The C++ class ODA.

3.1.1 ODA::initialize

```
static void initialize(unsigned object_cache_size = 1023,
                       unsigned update_ops_table = 1023,
                       unsigned interf_table_size = 101);
```

A server must call this function before issuing any other call to the ODA. The parameters to `ODA::initialize` specify the number of pseudopersistent objects cached by the ODA, the size of the hash table in which the ODA keeps information on update operations (IDL operations that must be performed within update transactions), and the size of the hash table in which the ODA keeps information on IDL interfaces.

3.1.2 ODA::multi_op_transaction_mode

```
static void multi_op_transaction_mode();
```

Enter multioperation transaction mode. The current transaction will be extended to encompass a sequence of IDL operations. In the absence of ObjectStore-generated exceptions, this sequence ends when an operation calls `ODA::commit_transaction` or `ODA::abort_transaction`. Such operation is also performed within the current transaction, and is the last one encompassed by the transaction.

The sequence of IDL operations encompassed by the transaction may also end with an ObjectStore-generated `CORBA::PERSIST_STORE` exception, which immediately aborts the current transaction. At the end of a multioperation transaction, in any case (call to `ODA::commit_transaction`, call to `ODA::abort_transaction`, or ObjectStore exception), the transaction mode switches back to transaction per operation.

3.1.3 ODA::commit_transaction

```
static void commit_transaction();
```

Causes the current transaction to be committed at the end of the current operation. When called in multioperation transaction mode, it also reverts the transaction mode back to transaction per operation.

A server calls `ODA::commit_transaction` to leave multioperation transaction mode and commit the current transaction. Calling this function when the transaction mode is already transaction per operation is superfluous, as the transaction would be committed anyway, by default, at the end of the current operation.

3.1.4 ODA::abort_transaction

```
static void abort_transaction();
```

Causes the current transaction to be aborted at the end of the current operation. When called in multioperation transaction mode, it also reverts the transaction mode back to transaction per operation.

3.1.5 ODA::register_update_op

```
static void register_update_op(const char* interf_name,
                               const char* op_name);
```

Installs `interf_name::op_name` in the hash table of update operations maintained by the ODA. Subsequent invocations of this operation, in transaction per operation mode, will be encompassed by update transactions.

3.1.6 ODA::Delete

```
static void Delete(CORBA::Object_ptr objp);
```

Deletes a persistent CORBA object. Just the "tie object" managed by the ODA is deleted, not its implementation object. Should be called only by destructors of implementation classes passed to `ODA_def_persistent` or `ODA_Def_Persistent`.

3.2 Persistence Directives and ODA-Generated Functions

ODA persistence directives are implemented as C++ macros. A set of macros, shown in Figure 3.2, is used in the case in the case of IDL interfaces not encompassed by any module. Another set of macros, shown in Figure 3.3, is used in the case of interfaces defined within a module.

```
ODA_def_server(interf,impl)
ODA_def_persistent(interf,impl)
ODA_def_activated(interf,impl)
```

Figure 3.2: ODA macros used for interfaces not encompassed by any module.

```
ODA_Def_Server(module,interf,impl)
ODA_Def_Persistent(module,interf,impl)
ODA_Def_Activated(module,interf,impl)
```

Figure 3.3: ODA macros used for interfaces defined within a module.

Each of these directives declares/defines ODA-generated functions. In what follows, ODA-generated functions are examined together with the corresponding persistence directive.

ODA persistence directives expand into declarations/definitions of ODA-generated classes and functions. While the declarations may appear in many source files, the definitions should appear just once in all source files. If the symbol `ODA_DEFS` is `#defined` by the time `oda.h` is included, these directives expand into declarations and definitions. Otherwise they expand into declarations only. A persistence directive may — and usually will — be visible to several source files, typically through the inclusion of a header file that contains the directive. One and only one of these source files must `#define ODA_DEFS` before including `oda.h`.

3.2.1 ODA_def_server

```
ODA_def_server(interf,impl)
```

C++ macro that "ties together" the interface class and the implementation class of the top-level object in a server. Used for IDL interfaces not encompassed by any module. Its expansion generates the following "tie class":

```
class _oda_interf_impl_server {
public:
    _oda_interf_impl_server(impl* p);
    ...
    virtual void* _deref();
private:
    ...
};
```

Application code should avoid any dependence on the name of the tie class above, the ODA-generated identifier `_oda_`*interf*`_`*impl*`_server`. The macro call

```
ODA_server(interf,impl)
```

expands into this identifier, and should be used instead.

ODA_def_server-Generated Functions

`ODA_server(`*interf*`,`*impl*`)(`*impl*`* p);`

Constructor of the tie class `ODA_server(`*interf*`,`*impl*`)`. Note that this constructor is public. The server mainline should use it to create the top-level server object, an instance of this class, before it calls `impl_is_ready`.

`void* _deref();`

Member function of the ODA-generated tie class. Returns the address of the corresponding implementation object.

3.2.2 ODA_def_persistent

```
ODA_def_persistent(interf,impl)
```

C++ macro that "ties together" the interface and implementation classes of persistent CORBA objects. Used for IDL interfaces not encompassed by any module. Its expansion generates the following "tie class":

```
class _oda_persistent_interf {
public:
    ...
    virtual void* _deref();
private:
    ...
};
interf_ptr ODA_persistent_interf(impl* p);
```

Application code should avoid any dependence on the name of the tie class above.

ODA_def_persistent-Generated Functions

`void* _deref();`

> Member function of the ODA-generated tie class. Returns the address of the corresponding implementation object.

interf`_ptr ODA_persistent_`*interf*`(`*impl*`* p);`

> Returns a reference to a persistent CORBA object (whose interface class is *interf*), given a pointer to its implementation object (of class *impl*).

The constructor of the ODA-generated tie class is not public. Instances of this class are managed by the ODA and cannot be created by the application code.

3.2.3 ODA_def_activated

```
ODA_def_activated(interf,impl)
```

C++ macro that "ties together" the interface and implementation classes of ODA-activated CORBA objects. Used for IDL interfaces not encompassed by any module. Its expansion generates the following "tie class":

```
class _oda_activated_interf {
public:
    ...
    virtual void* _deref();
private:
    ...
};
interf_ptr ODA_activated_interf(impl* p);
interf_ptr ODA_activated_interf(long keyval);
```

Application code should avoid any dependence on the name of the tie class above.

ODA-Activated Implementation Classes

An implementation class passed as the *impl* parameter to `ODA_def_activated` must have `ODA::Activated` as a public base class, and must redefine the following functions inherited from `ODA::Activated`:

```
CORBA::ULong _key();
char* _id();
static void* _activate(const char* id);
```

See details in pages 32–34.

Functions Generated by ODA_def_activated

`void* _deref();`

Member function of the ODA-generated tie class. Returns the address of the corresponding implementation object.

interf`_ptr ODA_activated_`*interf*`(`*impl*`* p);`

Returns a reference to an ODA-activated CORBA object (whose interface class is *interf*), given a pointer to its implementation object (of class *impl*).

interf`_ptr ODA_activated_`*interf*`(long keyval);`

Returns a reference to an ODA-activated CORBA object (whose interface class is *interf*), given the key to its implementation object (of class *impl*). Returns a null pointer if no active instance of the implementation class has this key value.

The constructor of the ODA-generated tie class is not public. Instances of this class are managed by the ODA and cannot be created by the application code.

3.2.4 ODA_Def_Server

```
ODA_Def_Server(module,interf,impl)
```

C++ macro that "ties together" the interface class and the implementation class of the top-level object in a server. Used for IDL interfaces defined within a module. Its expansion generates the following "tie class":

```
class _oda_module_interf_impl_server {
public:
    _oda_module_interf_impl_server(impl* p);
    ...
    virtual void* _deref();
private:
    ...
};
```

Application code should avoid any dependence on the name of the tie class above, the ODA-generated identifier `_oda_`*module*`_`*interf*`_`*impl*`_server`. The macro call

```
ODA_Server(module,interf,impl)
```

expands into this identifier, and should be used instead.

ODA_Def_Server-Generated Functions

`ODA_Server(`*module*`,`*interf*`,`*impl*`)(`*impl*`* p);`

Constructor of the tie class `ODA_Server(`*module*`,`*interf*`,`*impl*`)`. Note that this constructor is public. The server mainline should use it to create the top-level server object, an instance of this class, before it calls `impl_is_ready`.

`void* _deref();`

Member function of the ODA-generated tie class. Returns the address of the corresponding implementation object.

3.2.5 ODA_Def_Persistent

```
ODA_Def_Persistent(module,interf,impl)
```

C++ macro that "ties together" the interface and implementation classes of persistent CORBA objects. Used for IDL interfaces defined within a module. Its expansion generates the following "tie class":

```
class _oda_persistent_module_interf {
public:
    ...
    virtual void* _deref();
private:
    ...
};
interf_ptr ODA_persistent_module_interf(impl* p);
```

Application code should avoid any dependence on the name of the tie class above.

ODA_Def_Persistent-Generated Functions

`void* _deref();`

> Member function of the ODA-generated tie class. Returns the address of the corresponding implementation object.

interf`_ptr ODA_persistent_`*module*`_`*interf*`(`*impl*`* p);`

> Returns a reference to a persistent CORBA object (whose interface class is *interf*), given a pointer to its implementation object (of class *impl*).

The constructor of the ODA-generated tie class is not public. Instances of this class are managed by the ODA and cannot be created by the application code.

3.2.6 ODA_Def_Activated

```
ODA_Def_Activated(module,interf,impl)
```

C++ macro that "ties together" the interface and implementation classes of ODA-activated CORBA objects. Used for IDL interfaces defined within a module. Its expansion generates the following "tie class":

```
class _oda_activated_module_interf {
public:
    ...
    virtual void* _deref();
private:
    ...
};
interf_ptr ODA_activated_module_interf(impl* p);
interf_ptr ODA_activated_module_interf(long keyval);
```

Application code should avoid any dependence on the name of the tie class above.

ODA-Activated Implementation Classes

An implementation class passed as the *impl* parameter to `ODA_Def_Activated` must have `ODA::Activated` as a public base class, and must redefine the following functions inherited from `ODA::Activated`:

```
CORBA::ULong _key();
char* _id();
static void* _activate(const char* id);
```

See details in pages 32–34.

Functions Generated by ODA_Def_Activated

`void* _deref();`

Member function of the ODA-generated tie class. Returns the address of the corresponding implementation object.

interf`_ptr ODA_activated_`*module*`_`*interf*`(`*impl*`* p);`

Returns a reference to an ODA-activated CORBA object (whose interface class is *interf*), given a pointer to its implementation object (of class *impl*).

interf`_ptr ODA_activated_`*module*`_`*interf*`(long keyval);`

Returns a reference to an ODA-activated CORBA object (whose interface class is *interf*), given the key to its implementation object (of class *impl*). Returns a null pointer if no active instance of the implementation class has this key value.

The constructor of the ODA-generated tie class is not public. Instances of this class are managed by the ODA and cannot be created by the application code.

3.3 Exception Handling Directives

The following macros support server-side exception handling:

`TRY`

Replaces the C++ construct `try`.

`CATCH(`*`class`*`,`*`var`*`)`

Replaces the C++ construct `catch(`*`class`*`& `*`var`*`)`.

`CATCHANY`

Replaces the C++ construct `catch(...)`.

`ENDTRY`

Must be placed immediately after the last `CATCH` or `CATCHANY` block of a `TRY`/`CATCH`/`CATCHANY` sequence.

These macros are defined in the header file `oda/except.h`. See pages 24–26 for details on their usage.