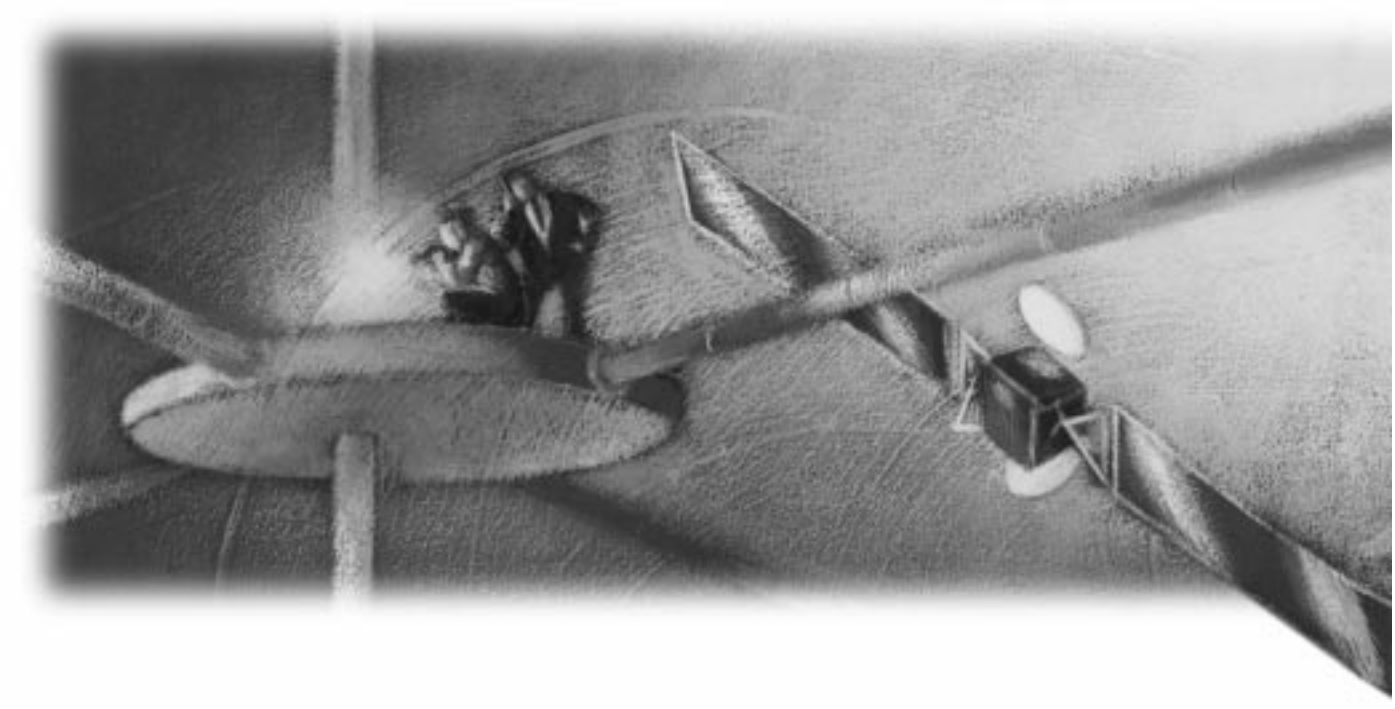


Objectivity/C++
Data Definition Language

Release 6.0

Objectivity/C++ Data Definition Language

Part Number: 60-DDL-0

Release 6.0, October 5, 2000

Contents

About This Book

This book describes how to use the Objectivity / C++ Data Definition Language (DDL) and the DDL processing tool to develop a schema within an Objectivity / DB federated database. A schema defines the types of persistent objects you can store in a database.

Audience

This book assumes that you are familiar with the C++ programming language and with the Objectivity / C++ programming interface.

Organization

- Chapter 1 provides basic terminology, a simple example, and the steps for creating a schema from the class definitions in one or more DDL files.
- Chapters 2 and 3 describe how to use the DDL to define persistence-capable classes and associations between them; Chapter 4 provides design guidelines when using multiple inheritance.
- Chapters 5 and 6 describe two alternative ways of modifying an existing schema: schema evolution and versioning.
- Chapters 7 and 8 describe mechanisms for organizing the definitions in a schema and techniques for tuning class definitions to promote efficient storage or faster access.
- Appendixes A and B contain reference information about the DDL processor and the pragma directives it accepts.
- Appendix C is an overview of Objectivity / C++ header files and their contents.
- Appendix D describes the Objectivity / DB primitive types that are stored in the schema for the corresponding Objectivity / C++ primitive types.
- Appendix E is a quick reference of schema-evolution operations.

Conventions and Abbreviations

Navigation

Table of contents entries, index entries, cross-references, and underlined text are hypertext links.

Typographical Conventions

`oobackup`	Command, literal parameter, code sample, filename, pathname, output on your screen, or Objectivity-defined identifier
`installDir`	Variable element (such as a filename or a parameter) for which you must substitute a value
Browse FD	Graphical user-interface label for a menu item or button
lock server	New term, book title, or emphasized word

Abbreviations

(administration)	Feature intended for database administration tasks
(*FTO*)	Feature of the Objectivity/DB Fault Tolerant Option product
(*DRO*)	Feature of the Objectivity/DB Data Replication Option product
(*IPLS*)	Feature of the Objectivity/DB In-Process Lock Server Option product
(*ODMG*)	Feature conforming to the Object Database Management Group interface

Command Syntax Symbols

[...]	Optional item. You may either enter or omit the enclosed item.
{…}	Item that can be repeated.
...|...	Alternative items. You should enter only one of the items separated by this symbol.
(…)	Logical group of items. The parentheses themselves are not part of the command syntax; do not type them.

Command and Code Conventions

In code examples or commands, the continuation of a long line is indented. Omitted code is indicated with the ellipsis (…) symbol. “Enter” refers to the standard key (labelled either Enter or Return) for terminating a line of input.

Getting Help

We have done our best to make sure all the information you need to install and operate Objectivity products is provided in the product documentation. However, we also realize problems requiring special attention sometimes occur.

Technical Support Web Site

You can find answers to frequently asked questions, supported platforms, known bugs, and bug fixes on the Objectivity Technical Support web site. Send electronic mail or call Objectivity Customer Support to gain access to the site.

How to Reach Objectivity Customer Support

You can contact Objectivity Customer Support by:

- **Telephone:** Call 1.650.254.7100 *or* 1.800.SOS.OBJY (1.800.767.6259) Monday through Friday between 6:00 A.M. and 6:00 P.M. Pacific Time, and ask for Customer Support.

  The toll-free 800 number can be dialed *only* within the 48 contiguous states of the United States and Canada.
- **Fax:** Send a fax to Objectivity at 1.650.254.7171.
- **Electronic Mail:** Send electronic mail to *help@objectivity.com*.

Before You Call

If you need help from Customer Support, please have the following information ready before you contact Objectivity:

- Your name, company name, address, telephone number, fax number, and email address
- Description of your workstation environment, including the type of workstation, its operating system version, compiler or interpreter, and windowing environment
- Information about the Objectivity product you are using, including the version of the Objectivity / DB libraries
- Detailed description of the problem you have encountered

1

Getting Started

This chapter provides an introduction to Objectivity/C++ Data Definition Language (Objectivity/DDL). Whereas Objectivity/C++ comprises the entire C++ programming interface to the Objectivity/DB object-oriented database management system, Objectivity/DDL is the portion that enables you to develop a *schema* of *persistence-capable class definitions*.

This chapter describes:

- General information about schema development
- Creating DDL files that contain the definitions of persistence-capable classes
- Processing DDL files to create a schema
- Contents and usage of C++ source files generated by the DDL processor
- Considerations when dividing definitions among multiple files
- General information about modifying a schema

For Objectivity/C++ basic concepts, see the Objectivity/C++ programmer's guide.

About Schema Development

The first step in developing an object-oriented application is to define the classes that capture the structure and behavior of the fundamental entities in the application. Such definitions usually arise naturally out of the logical modeling phase of application development.

In the logical model for a database application, you identify the classes whose instances are to be *persistent*—that is, must continue to exist even after the applications that define or manipulate them have finished. With classes whose instances will be persistent, two additional activities must be added to the class definition process:

- Each class definition must be made *persistence-capable* so that instances of the class can be persistent.

- Persistence-capable class definitions must then be added to a federated database *schema* so that instances of those classes can be stored in the database.

The process of defining persistence-capable classes and adding them to a schema is called *schema development*.

Persistence-Capable Classes

In a standard C++ program, instances of all classes are *transient*—they exist only for the life of the application that defines them.

In Objectivity / C++, you add persistence behavior to an application-defined class by deriving it directly or indirectly from the Objectivity / C++ class `ooObj`. Such classes are said to be *persistence-capable*. When a class is persistence-capable, an application can create both persistent and transient instances of it.

Classes that do *not* inherit persistence behavior are *non-persistence-capable classes*. All instances of such classes are transient. However, instances of non-persistence-capable classes are in fact stored in the federated database when you incorporate them in a persistent object—for example, when you use a non-persistence-capable class as an attribute type or base class of a persistence-capable class. A non-persistence-capable class must obey certain restrictions to be incorporated in a persistence-capable class.

Instances of persistence-capable classes are true persistent objects, in that each can be referenced by an address in the database; this address serves as the *object identifier* (OID) for the object. In contrast, instances of non-persistence-capable classes are not independently addressible, although they may be accessed indirectly through the persistent objects that embed them.

Federated Database Schemas

Each Objectivity / DB federated database contains a *schema* that describes the types of data that it can store persistently. A schema contains a type number for each persistence-capable class, along with *shape* information, which describes how persistent instances of the class are to be laid out in storage.

Note that a schema also contains type information for any non-persistence-capable class that is directly or indirectly incorporated in a persistence-capable class. However, a schema does *not* contain information about member functions or static data members; such information is maintained in the compiled applications that access the database.

Most federated databases contain a single schema. However, for project management purposes, you can partition class definitions among multiple schemas in the same database; see Chapter 7, “Partitioning a Data Model”.

Schema Development with Objectivity/DDL

You define a schema of persistence-capable classes using the *Data Definition Language* (DDL). The DDL is standard C++ with extensions that support persistence and language interoperability; DDL extensions also support data modeling features such as *associations* (relationships) between persistent objects.

Persistence-capable class definitions are placed in one or more text files called *DDL files*. You create a schema by preprocessing these DDL files with a tool called the *DDL processor*. This tool:

- Extracts type information from the DDL files and constructs the schema within a particular federated database.
- Generates C++ source files for you to compile into your application. These files contain:
  - Your persistence-capable class definitions augmented with additional members.
  - Definitions for system-defined classes that support referencing and iterating over instances of each application-defined persistence-capable class.

As you refine your application's logical model, you can incrementally add new persistence-capable class definitions and (re)process the DDL files containing them. You can also modify existing persistence-capable class definitions; however, if you want to preserve existing objects in the federated database, you must use the DDL processor to either *version* or *evolve* the changed classes.

Creating DDL Files

You define persistence-capable classes in one or more DDL files. These files resemble C++ header (`.h`) files, although they cannot be directly compiled with a C++ compiler; you must preprocess them with the DDL processor instead. As with C++ header files, you can combine multiple definitions in a single DDL file, or you can place your persistence-capable class definitions in separate DDL files. Each DDL file can have any base name, but the extension *must* be `.ddl`.

Basic DDL File Contents

In general, a DDL file contains:

- One or more persistence-capable class definitions written in the DDL. You separate or group these definitions among DDL files as you normally would among C++ header files.

- Any other definitions referred to or used within the persistence-capable classes, such as:
  - Non-persistence-capable classes (typically, base or embedded classes)
  - `typedef` statements
  - `extern` declarations
- Preprocessing directives such as `#include` and `#ifdef`. Note that:
  - A DDL file can include either a C++ header file or another DDL file, although these are handled differently (see "Treatment of Preprocessing Directives" on page 24).
  - Certain `#include` and `#pragma` directives are required when you place interdependent class definitions in different DDL files; see "Dividing Definitions Among Multiple Files" on page 25.
- C++ function declarations and definitions. Although function declarations are checked for syntax, function declarations and bodies are ignored for purposes of creating the schema.

Adapting Existing Header Files

If the classes to be made persistence-capable are already defined in C++ header files, the simplest way to create DDL files for them is to adapt these header files.

Changing the Filename Extension

You adapt the filenames of header files by changing their `.h` extensions to `.ddl`. For example, you would change `Book.h` to `Book.ddl`, `Patron.h` to `Patron.ddl`, `Library.h` to `Library.ddl` and so on.

Adapting Existing Class Definitions

Within each C++ header file you are adapting, you modify the inheritance and data types of each class definition to be made persistence-capable. For a complete discussion, see Chapter 2, "Defining Persistence-Capable Classes". In brief, you:

- Derive the class from an appropriate Objectivity / C++ base class, such as the basic object class `ooObj` or the container class `ooContObj`. You only need to modify your application's root classes; derived classes that inherit from these root classes automatically become persistence-capable.
- Adjust each data-member type to take advantage of platform-independent Objectivity / C++ types and to avoid the types of data (such as C++ pointer types) whose values cannot be stored persistently. Typically, you:
  - Replace each C++ numeric type, such as `int`, with an appropriate Objectivity / C++ primitive type, such as `int16`. This ensures that your

data will be accessed consistently by applications compiled on different machine architectures.

- Replace each character pointer type with an embedded C++ fixed-length string or an Objectivity/C++ string type, such as `ooVString`.
- Replace each class pointer type with an Objectivity/C++ *object reference* or an *association* to a persistence-capable class. If the referenced class is not to be persistence-capable, you must find some other way to replace the class pointer type, such as embedding a non-persistence-capable class type.
- Replace each pointer to an array with an embedded fixed-size array or with an Objectivity/C++ variable-size array type, such as `ooVArrayT<`*`element_type`*`>`.

For a complete list of valid data types, see "Defining Data Members" on page 40; see also "Prohibited Data Types" on page 51.

In C++, it is common to use pointers to implement directional links between related objects; a data member of the source object contains a pointer to the destination object. In Objectivity/C++, you link a persistent source object to a related persistent destination object through an object reference attribute (a Objectivity/C++ "smart pointer") or through an association (a richer relationship defined in the schema to support referential integrity, a specified cardinality, and so on).

In either case, the linking data member's type is a parameterized *object-reference class* whose definition is generated by the DDL processor. For example, a link to a persistence-capable class `Book` would use the DDL-generated object-reference class `ooRef(Book)`. For a list of generated parameterized classes, see "References Header File" on page 21. For more information about referencing persistence-capable classes, see "Object-Reference Types" on page 46, and Chapter 3, "Defining Associations".

NOTE You normally do not need to include any Objectivity/C++ system header files in a DDL file; the DDL processor knows where to find declarations for the Objectivity/C++ types used in persistence-capable class definitions. However, if a DDL file contains definitions that use special Objectivity/C++ features such as persistent collections or indexes, you must include the corresponding Objectivity/C++ header file; see Appendix C, "Objectivity/C++ Include Files".

Example: Adapting an Existing C++ Header File

This example creates persistence-capable class definitions for the `Vehicle` and `Car` classes by creating a DDL file from the C++ header file `vehicle.h` and adapting the original C++ definitions.

```
// Original C++ header file: vehicle.h
class Fleet;                    // Forward declaration

class Vehicle {
   public:
      char *license;
      char *type;
      int doors;
      int transmission;
      bool available;
      Fleet *inFleet;
   …
};

class Car : public Vehicle … {
   …
};
```

The class definitions in `vehicle.ddl` inherit persistence, use Objectivity/C++ primitive and string types, and define an association for linking `Vehicle` instances to a `Fleet`:

```
// DDL file: vehicle.ddl
class Fleet;                    // Forward declaration

class Vehicle: public ooObj { // Inherit persistence from ooObj
   public:
      ooVString license;
      ooVString type;
      int16 doors;
      int8 transmission;
      ooBoolean available;
      ooRef(Fleet) inFleet <-> hasVehicles[]; // association
   …
};
class Car : public Vehicle … {    // Inherits persistence
   …                              // from Vehicle
};
```

Processing DDL Files

After you have created DDL files containing persistence-capable class definitions, you process them using the DDL processor. The DDL processor extracts type information from the DDL files and either creates or modifies a schema in a federated database. The DDL processor also generates C++ header files and implementation files that add persistence behavior to your class definitions.

Before you run the DDL processor, you must:

- Ensure that a federated database exists in which to create or modify a schema. If necessary, use the `oonewfd` tool to create a federated database. This tool creates the boot file that the DDL processor will use to find and open the system-database file. The `oonewfd` tool is described in the Objectivity / DB administration book.
- Verify that the `bin` subdirectory of the Objectivity / DB installation directory is in your search path. This subdirectory contains the DDL processor.
- Start a lock server on the lock server host for the federated database. The lock server is described in the Objectivity / DB administration book.

You then invoke the DDL processor from a command window, script, or makefile:

- `ooddlx.exe` on Windows platforms
- `ooddlx` on UNIX platforms

When you invoke the DDL processor, you specify the file to be processed, the federated database boot file, and various options as appropriate. The options allow you to specify certain compiler flags, customize the names of the output files, specify how to handle changed DDL files, and so on. For a list of options, see `ooddlx` (page 170).

If you are using multiple DDL files, you invoke the DDL processor once for each DDL file, and you may have to consider the order in which they are processed; see “Dividing Definitions Among Multiple Files” on page 25.

EXAMPLE This UNIX example uses the `oonewfd` tool to create a federated database with a boot file named `myExample`, and then runs the DDL processor to load the specified DDL file into the federated-database schema.

```
> oonewfd -fdfilepath myExample.FDB
          -lockserverhost myMachine myExample

> ooddlx mySchema.ddl myExample
```

During schema development, you typically need to add or change persistence-capable class definitions; see “Modifying the Schema” on page 33.

Files Generated by the DDL Processor

Besides adding type information to a federated database schema, the DDL processor generates various class and function definitions that support the persistence behavior of your persistence-capable classes. For each DDL file *classDefFile*.ddl that you process, the DDL processor produces three standard C++ source files, listed here with default filename suffixes:

- A *primary header file*:
  classDefFile.h
- A *references header file* (sometimes called a *secondary header file*):
  *classDefFile*_ref.h
- A *method implementation file*:
  *classDefFile*_ddl.cpp for Windows
  *classDefFile*_ddl.C for UNIX

EXAMPLE This UNIX example shows the files created at each point in schema development.

```
> ls
Book.ddl
___
> oonewfd -fdfilepath myExample.FDB
          -lockserverhost machine95 myExample
> ls
Book.ddl
myExample
myExample.FDB
___
> ooddlx Book.ddl myExample
> ls
Book.ddl
Book_ddl.C
Book.h
Book_ref.h
myExample
myExample.FDB
```

You can change the default filename extension for any of the generated files by specifying the `-header_suffix`, `-ref_suffix` or `-C++_suffix` option of the DDL processor. This book uses the filename extension `.cxx` as a general, platform-independent extension for C++ implementation files.

Primary Header File

The *primary header file* (*classDefFile*.h) that is generated from a DDL file contains all the declarations and definitions from the original DDL file, along with any preserved preprocessing directives (see page 24).

The DDL processor adds new members to each persistence-capable class definition you define. These include:

- Members for obtaining type information and for creating and deleting instances.
- Member functions for creating, deleting, and accessing each association you defined in the class.

You include a primary header file in any application source file that uses a persistence-capable class defined in it. You may also need to include a primary header file in other DDL files (see "Dividing Definitions Among Multiple Files" on page 25).

The primary header file contains a generated `#include` directive to the Objectivity/C++ system header file `oo.h`. This means that including a primary header file automatically provides the system definitions of the Objectivity/C++ programming interface as well.

See the Objectivity/C++ programmer's reference for information about how to use the generated member functions in an application.

References Header File

The *references header file* (*classDefFile*_ref.h) contains generated definitions of parameterized classes that support referencing and iterating over persistent objects. For each persistence-capable class *className* defined in *classDefFile*.ddl, the secondary header file contains generated definitions for:

- Class `ooRef(`*className*`)` for creating *standard object references* to *className* objects.
- Class `ooShortRef(`*className*`)` for creating *short object references* to *className* objects; used for fine-tuning storage space usage.
- Class `ooHandle(`*className*`)` for creating *handles* to *className* objects. Handles are similar to object references but are optimized for referencing persistent objects while they are in memory.
- Class `ooItr(`*className*`)` for creating iterators on *className* objects.

The references header file is automatically included in the primary header file. This makes the generated definitions available to code in the primary header file and in any application source file that includes the primary header file. Making a persistence-capable class available to an application source file also makes the corresponding object reference, handle, and iterator classes available as well.

If the code in an application source file simply references a persistence-capable class without actually using it, you can include just the references header file explicitly. The references header file can be used by itself because, like the primary header file, it provides the Objectivity / C++ system definitions through an `#include <oo.h>` directive.

If the code in a DDL file references a persistence-capable class that is defined in another DDL file, you can use a DDL-specific `#pragma` directive to make the generated definitions available (see “Obtaining Generated Class Definitions” on page 28).

See the Objectivity / C++ programmer’s reference for information about using the generated parameterized classes in an application.

Method Implementation File

The *method implementation file* (*`classDefFile`*`_ddl.cxx`) contains:

- Definitions for the non-inline member functions declared in the generated header files.
- Registration code that binds the class definitions in an executing application to their counterparts in the schema. Among other things, this code associates each class name with the type number assigned to it in the schema.
- Registration code that enables Objectivity / DB to associate an application’s virtual-function tables with appropriate persistent objects (see page 24).

You compile the method implementation file with your application code files.

Example: Including Generated Header Files

This example shows a `main.cxx` program that includes the primary header file `a.h` to obtain the definition of class `A`. Because `a.h` includes the references header file `a_ref.h`, the program also obtains the class definition of `ooHandle(A)`.

```
// DDL file: a.ddl
class A : public ooObj {
   …
};
```

___

```
// Application code: main.cxx
#include <a.h>     // Provides definition of A; includes a_ref.h
{  …
   ooHandle(A) aH;       // Requires definition of ooHandle(A)
   aH = new() A;         // Requires definition of A
   …
};
```

Figure 1-1 shows the `#include` relationships among these files.

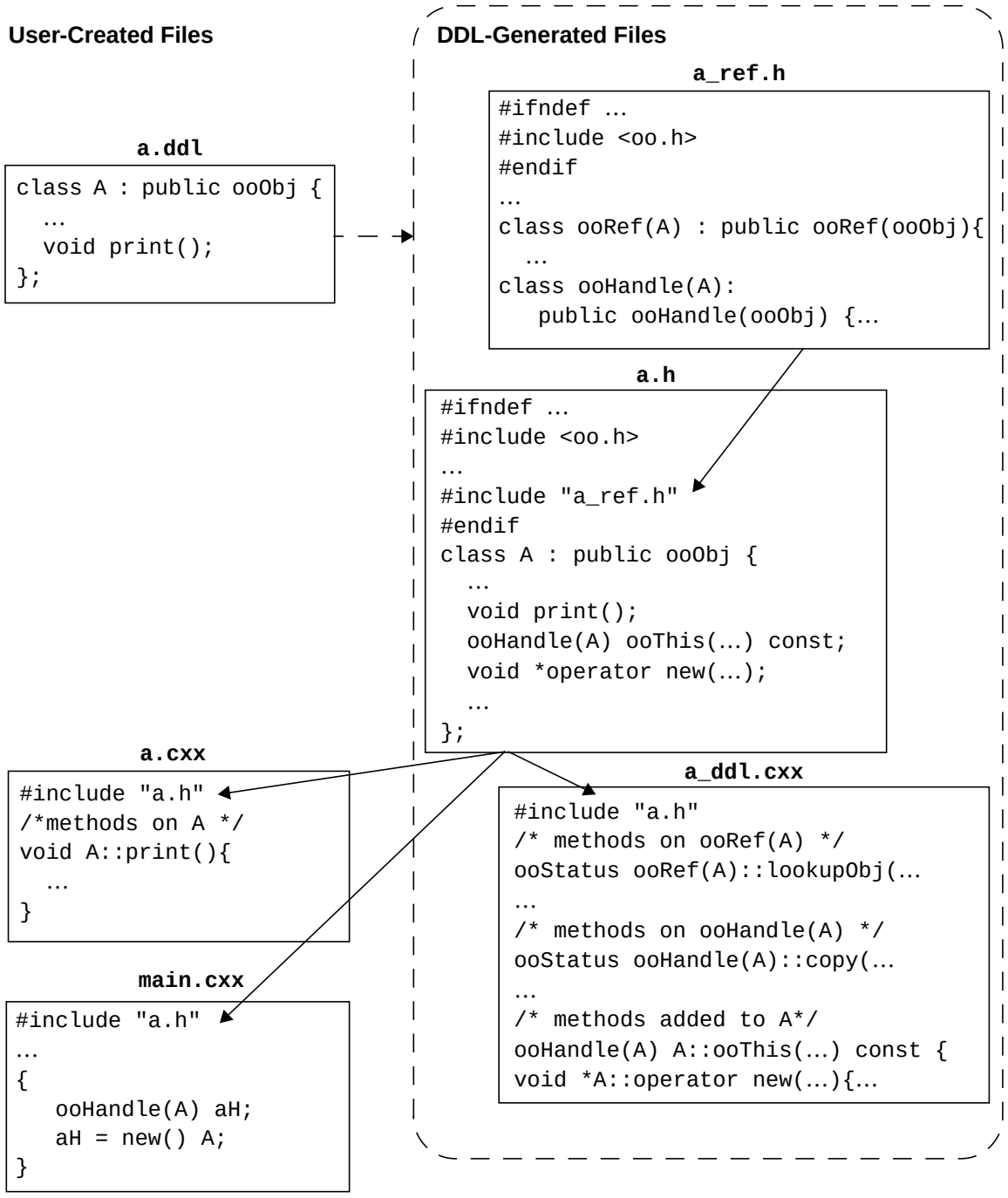


Figure 1-1 Include Relationships Among User-Created and DDL-Generated Files

Treatment of Preprocessing Directives

The DDL processor performs standard C++ preprocessing on DDL files. Therefore, if you put preprocessing directives in the DDL file, some are preserved in the primary header file, while others are interpreted and then removed.

The following directives are preserved in the primary header file and are therefore seen when the primary header file is compiled:

- `#include` directives that specify C++ header files
- `#define` and `#undef`
- `#pragma` directives (except for pragmas specific to the DDL processor)

In contrast, the DDL processor interprets and removes conditional compilation directives such as `#if`, `#else`, `#endif`. If you want conditional compilation directives to appear in the primary header file, you must place the directives in a separate header file, and include that header file in your DDL file.

The DDL processor also interprets and removes any `#include` directive that specifies another DDL file—for example:

`#include <`*`classDefFile`*`.ddl>`

The DDL processor responds to such directives by treating the included DDL file as if it were merged into the including DDL file—the result is to generate a single set of header and implementation files that contains the processed definitions from both DDL files. Note that any conditional compilation directives in the included DDL file are interpreted and removed.

Dependencies on DDL-Generated Code

For each persistence-capable class that implements a virtual member function, the DDL processor generates code for registering a virtual-function table with Objectivity / DB at runtime. This table is used by instances of the class to dispatch virtual member-function calls to the correct implementation. The DDL processor places the registration code in the generated method implementation file, which must be compiled and linked with your application.

Dependencies may exist on the registration code that is generated for a class, even if the application does not reference the class directly. For example, assume that:

- A persistence-capable class `C` implements a virtual function defined in a base class `B`.
- A federated database contains persistent objects of classes `B` and `C`.
- An application finds objects of class `C` indirectly (for example, by iterating over all `B` objects).

If the application then invokes the virtual function on a `C` object obtained this way, the virtual-function table must be present in order to correctly dispatch the call;

otherwise an error is signalled. Thus, even if the application does not reference any `C` objects directly, it must be linked with the object file produced from the method implementation file containing the registration code for class `C`.

You should take this dependency into account when considering whether to omit object files from your link rules. In general, you should link all object files that result from compiling DDL-generated method implementation files, even if your application does not directly reference all persistence-capable classes.

Dividing Definitions Among Multiple Files

A common practice in C++ programming is to divide type definitions among multiple header files—for example, to facilitate parallel development by separating the types used by one module from those used by other modules. In most programs, such modules are somewhat interdependent, as when one module uses types from another. In such cases, the using module obtains the required type definitions by including the header file that contains those definitions.

You can divide your persistence-capable class definitions among multiple DDL files so that the generated header files will conform to your module design. As with C++ header files, when you place interdependent definitions in separate DDL files, you must arrange for each DDL file *and* its generated header files to access any required definitions that reside in other files. You accomplish this by inserting an appropriate `#include` or `#pragma` directive in the DDL file.

Obtaining Application-Specific Definitions

In a C++ program, when a type definition in one header file requires a type definition in another, the first header file specifies the second header file in an `#include` directive. Similarly, a DDL file can include a C++ header file to obtain type definitions from it.

For example, assume a persistence-capable class `A` (defined in a DDL file `a.ddl`) embeds a non-persistence-capable class `Helper` (defined in a C++ header file `helper.h`); the DDL file must include the header file, as shown in Figure 1-2. The `#include` directive is preserved in the files generated from `a.ddl`.

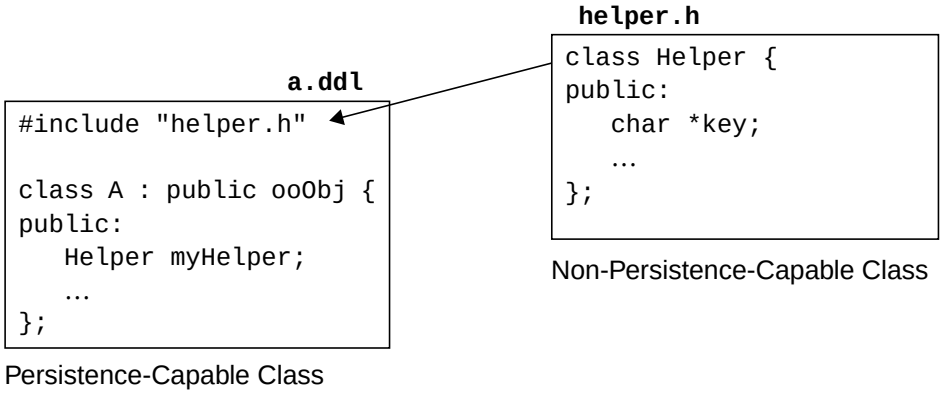


Figure 1-2 Including a C++ Header File in a DDL File

However, an extra condition holds when the code in one DDL file requires a persistence-capable class definition from another—as, for example, when a persistence-capable class is derived from another persistence-capable class that is defined in a separate DDL file. In such cases:

- The dependent DDL file must include the header file that is generated for the DDL file containing the required definition.
- The DDL file containing the required definition must be processed *before* the dependent DDL file is processed so that the generated header file exists before it is included.

  Thus, in the case of inheritance across DDL files, a DDL file containing a base class must be processed before any DDL files containing derived classes.

EXAMPLE In this example, class `A` (defined in the file `a.ddl`), is the base class for class `B` (defined in the file `b.ddl`), so class `B` requires the definition of class `A`. Therefore, the file `b.ddl` includes the generated primary header file `a.h`. Note that `a.h` does not exist until `a.ddl` is processed with the DDL processor, so `a.ddl` must be processed before `b.ddl` is processed.

```
// DDL file: a.ddl
class A : public ooObj {
   …
};
```

```
// DDL file: b.ddl
#include <a.h>        // Provides definition of A
class B : public A {  // Requires definition of A
   …
};
```

Figure 1-3 extends this example by adding class `C` (defined in the file `c.ddl`), which is derived from class `B`. To support the `#include` directives in the DDL files, the DDL processor must process the files in the order: `a.ddl`, `b.ddl`, `c.ddl`. Note that the generated primary header file for each class is included in the source file that implements member functions for that class.

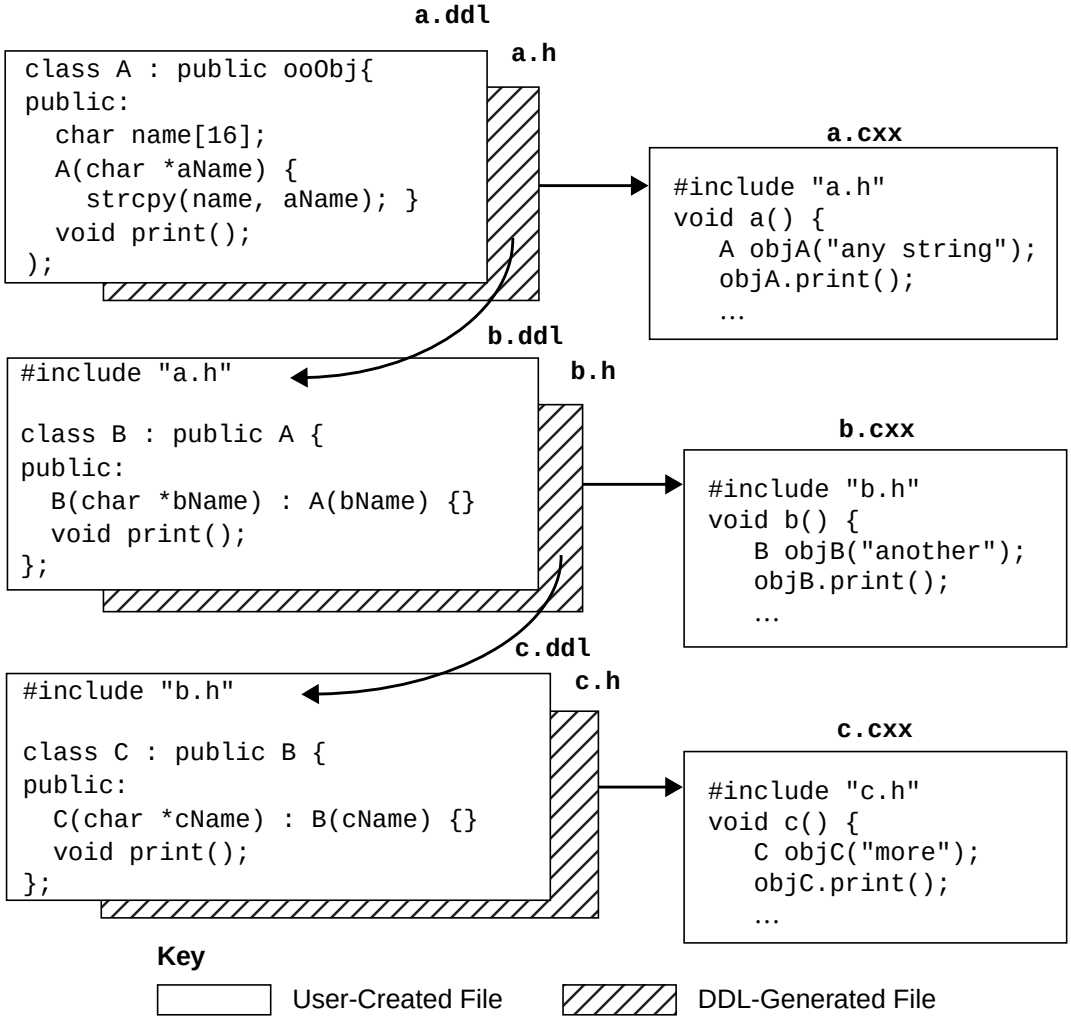


Figure 1-3 Inheritance Across Multiple DDL Files

Obtaining Generated Class Definitions

Persistence-capable definitions commonly require the definitions of one or more generated parameterized classes—for example:

- When one persistence-capable class (`Library`) is the source class for links to another destination class (`Book`) through an object reference or association, the source class requires the definition of the parameterized object-reference class that is generated for the destination class—in this case, `ooRef(Book)`.
- When a member function of one persistence-capable class (`Library`) accepts an object reference, handle, or iterator to instances of another persistence-capable class (`Book`), the function declaration requires the definition of the corresponding generated parameterized class—namely, `ooRef(Book)`, `ooHandle(Book)` or `ooItr(Book)`.

#pragma ooclassref Directive

If you define interdependent persistence-capable classes in the same DDL file, they can access each others' generated parameterized classes without extra steps. This is because the generated definitions are automatically placed in a single references header file that is included before all other definitions in the primary header file containing the persistence-capable class definitions.

If, on the other hand, the code in one DDL file references a persistence-capable class *`className`* defined in a different DDL file, you must arrange for *`className`*'s generated parameterized classes to be available too. You do this by inserting a forward declaration to *`className`* followed by a `#pragma ooclassref` directive in the referencing DDL file. You use the directive to identify the references header file that contains (or will contain) the generated parameterized classes for *`className`*. Thus, for a *`className`* defined in *`classDefFile`*`.ddl`, the directive specifies both *`className`* and the filename *`classDefFile`*`_ref.h`, with the filename in angle brackets or double quotes.

EXAMPLE Members of class `Library` (defined in `library.ddl`) reference the class `Book` (defined in `book.ddl`). Therefore, the `library.ddl` file contains a `#pragma ooclassref` directive as shown, to obtain the definitions of the generated classes `ooRef(Book)` and `ooHandle(Book)`.

```
// DDL file: library.ddl
class Book;
#pragma ooclassref Book <book_ref.h>

class Library : public ooObj {
   …
```

```
   ooRef(Book) book;
   ...
   ooBoolean ownsBook (ooHandle(Book) &aBook);
};
```

```
// DDL file: book.ddl
class Book : public ooObj {
   ...
};
```

When the DDL processor encounters the #pragma ooclassref directive in a DDL file, it:

- Allows the DDL file to reference the generated parameterized classes for *className* even if the references header file *classDefFile*_ref.h does not yet exist.
- Causes the generated primary header file to include the required references header file *classDefFile*_ref.h. The generated #include directive preserves the delimiters (angle brackets <> or double quotes "") used in the pragma.

  In the example above, processing library.ddl generates a header file library.h that contains an #include <book_ref.h> directive before any other definitions.

This behavior allows the DDL files to be processed in any order. However, before you compile the generated primary header file, you must process the DDL file that defines *className*. In the example above, you can process library.ddl and book.ddl in any order, but you must process book.ddl before you can compile library.h, because book_ref.h must exist before it is included.

One-Way Dependencies

A one-way dependency exists between two DDL files when the code in one DDL file references a persistence-capable class in a second DDL file, but the second file contains no references to any persistence-capable class in the first. The example in the preceding subsection shows a one-way dependency from class Library to class Book.

When a one-way dependency exists to a *className* defined in *classDefFile*.ddl, you have several alternatives—you can use the #pragma ooclassref directive as described in the preceding subsection, or you can explicitly include either *classDefFile*.h or *classDefFile*_ref.h. Using the #pragma ooclassref directive is recommended because it allows you to process DDL files in any order; if you use the #include directive, you must process *classDefFile*.ddl first.

Two-Way Dependencies

A two-way dependency exists between two DDL files when each file contains code that references or uses a definition from the other file. The most common case is when a bidirectional association exists between two persistence-capable classes defined in separate DDL files. When associations (or other references) span two DDL files, each file must use a `#pragma ooclassref` directive to obtain the definitions generated from the other, as the following example shows.

EXAMPLE Class `Library` (defined in the file `library.ddl`) has a one-to-many bidirectional association to class `Book` (defined in the file `book.ddl`). Therefore, the definition of `Library` requires the definition of `ooRef(Book)` and the definition of `Book` requires the definition of `ooRef(Library)`. The use of `#pragma ooclassref` in both DDL files correctly obtains the required definitions.

```
// DDL file: library.ddl
class Book;
#pragma ooclassref Book <book_ref.h>

class Library : public ooObj {
   ooRef(Book) books[] <-> theLibrary;
   …
};
```

___

```
// DDL file: book.ddl
class Library;
#pragma ooclassref Library <library_ref.h>

class Book : public ooObj {
   ooRef(Library) theLibrary <-> books[];
   …
};
```

___

Note that simply including `library_ref.h` and `book_ref.h` in `book.ddl` and `library.ddl`, respectively, generates an error, because no processing order exists that allows the DDL processor to generate these header files in time to satisfy the `#include` directives.

```
// DDL file: library.ddl
#include <book_ref.h>       // Generates an error
class Library : public ooObj {
   ooRef(Book) books[] <-> theLibrary;
   …
};
```

___

```
// DDL file: book.ddl
#include <library_ref.h>     // Generates an error
class Book : public ooObj {
   ooRef(Library) theLibrary <-> books [];
   ...
};
```

A second kind of two-way dependency exists when one DDL file contains code that *uses* a definition in a second DDL file, while the second DDL file contains code that *references* a definition in the first DDL file. For example, if class `C` (defined in the file `c.ddl`) inherits from class `B` (defined in the file `b.ddl`), and class `B` has a unidirectional association to class `C`, then:

- The using DDL file (`c.ddl`) obtains the definition of class `B` by including the primary header file generated from `b.ddl`—for example:

  ```
  #include <b.h>
  ```

- The referencing DDL file (`b.ddl`) contains an appropriate `#pragma ooclassref` directive to obtain the definition of `ooRef(C)`—for example:

  ```
  class C;
  #pragma ooclassref C <c_ref.h>
  ```

- The referencing DDL file (`b.ddl`) must be processed before the using DDL file (`c.ddl`) so that `b.h` will exist when `c.ddl` includes it.

If You Omit a #pragma ooclassref Directive

When the DDL processor encounters code that requires a definition for a generated parameterized class such as `ooRef(`*`className`*`)`, it determines whether the required definition can be found. In particular, the DDL processor is satisfied that such a definition exists if the DDL file being processed either:

- Contains the definition for *`className`*. If so, the references header file to be generated for this DDL file will contain the required definition.
- Includes an existing reference header file that contains the required definition.
- Contains a `#pragma ooclassref` directive specifying the references header file in which to find the required definition. The specified file may, but need not, exist yet.

If none of these conditions is true, the DDL processor reports any forward references it encounters, but otherwise assumes you will provide the required definitions by including any necessary generated header files in the correct order in your application source files.

EXAMPLE Assume that class `Library` (defined in the file `library.ddl`) has an association to class `Book` (defined in the file `book.ddl`), and neither file contains a `#pragma ooclassref` directive. When each file is processed, the DDL processor reports the forward references but does not otherwise issue an error.

```
// DDL file: library.ddl
class Book;
class Library : public ooObj {
   ooRef(Book) books[] <-> theLibrary;
   ...
};
```

```
// DDL file: book.ddl
class Library;
class Book : public ooObj {
   ooRef(Library) theLibrary <-> books[];
   ...
};
```

The definitions for `ooRef(Book)` and `ooRef(Library)` must then be made available through explicit `#include` directives in application source files that use `Library` and `Book`.

```
// Application code: main.cxx
#include <book_ref.h>
#include <library_ref.h>
#include <book.h>        // Includes book_ref.h
#include <library.h>     // Includes library_ref.h

main {
   ...     // Code that uses Library and Book

}
```

If, in the example, the references header files were not explicitly included in `main.C`, an incorrect inclusion order would result, because `library_ref.h` would not be seen before `book.h`. This produces a compiler error (such as the Microsoft Visual C++ class redefinition error message `C2011`), and an Objectivity/C++ error message such as the following:

```
"Ignore_the_compilers_error_message_The_real_error_is_
   Missing_definition_of_ooRef<className>"
```

The recommended way to correct such compilation errors is to adjust the relevant DDL files:

1. Verify that a definition for *`className`* exists in some DDL file *`classDefFile`*`.ddl`, and that *`className`* is persistence-capable. If *`className`* is a template class, verify that it is instantiated in a DDL file.
2. Insert a `#pragma ooclassref` directive in each DDL file that references (but does not define) *`className`*:

```
class className;
#pragma ooclassref className <classDefFile_ref.h>
```

Modifying the Schema

As you refine your application's logical model, you will probably need to add new persistence-capable class definitions or change existing ones. In general, the DDL processor allows you to add new definitions to a schema, but prevents you from changing definitions that are already in the schema; see "Adding to an Existing Schema" below.

If you need to modify existing definitions in a schema—for example, by renaming data members, altering the inheritance hierarchy, or changing data-member types—you must choose one of the following alternatives:

- Re-create the entire federated database and schema. This is a common approach during the early stages of development when the logical model is still volatile and the federated database contains test data you can discard. See "Replacing a Schema in Early Development" on page 34.
- Add the modifications by *evolving* existing definitions. This allows you to preserve existing persistent objects by subsequently converting them to the new schema representations. See Chapter 5, "Schema Evolution".
- Add the modifications by creating a new *version* of each affected definition. This allows you to preserve and access existing persistent objects (instances of old versions) while creating new objects from the new versions. See Chapter 6, "Class Versioning".

Adding to an Existing Schema

You can add new persistence-capable class definitions to an existing schema at any time, without affecting existing objects in the database. To add a definition to a schema, you:

1. Modify an existing DDL file or create a new one to contain the new definition.
2. Use the DDL processor to process the new or modified DDL file.
3. Incorporate the new header and source files in your application.

4. Recompile and link any applications that use DDL-generated header and implementation files. **Warning:** Omitting this step may cause runtime errors.

The DDL processor accepts several other kinds of modifications to DDL files (and any included C++ header files). Specifically, you can:

- Add, delete, or change any member function, operator, or constructor defined on an existing persistence-capable or non-persistence-capable class. This is because member function information is not included in the schema.
- Add new non-persistence-capable types (class, `typedef` statement, enumerated type, and so on).
- Change a non-persistence-capable type, but only if the changed type is not incorporated in a persistence-capable class (for example, through inheritance or embedding).

Replacing a Schema in Early Development

During the early stages of schema development, you typically need to make many schema changes, without needing to keep existing test data in your federated database. The simplest way to incorporate changes into a volatile schema is to:

1. Modify existing DDL files or create new ones as desired to add, delete, or modify definitions.
2. Delete the existing federated database with the obsolete schema. **Warning**: This deletes any existing data as well.
3. Create a new federated database.
4. Use the DDL processor to process each of the necessary DDL files.
5. Recompile and link any applications that use DDL-generated header and implementation files. **Warning:** Omitting this step may cause runtime errors.

A common technique is to include rules in your makefile for deleting and creating the federated database and for running the DDL processor.

Summary

Figure 1-4 shows the development flow for a typical C++ application. To simplify this example, only one DDL file is shown.

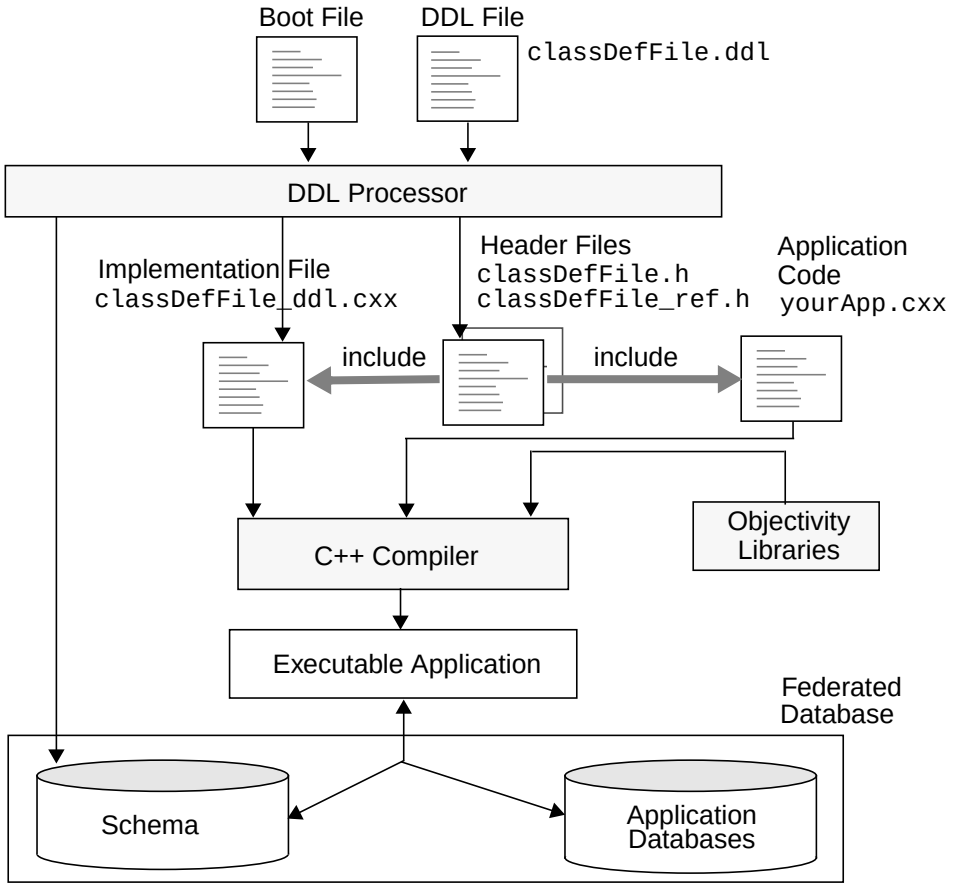


Figure 1-4 Application Development Flow

2

Defining Persistence-Capable Classes

A *persistence-capable class* is one whose instances can be made persistent and saved in a database. An application that needs to save objects in a database must define a persistence-capable class for each kind of object to be saved. Applications typically define persistence-capable classes for basic objects. An application that needs to save application-specific data with a container can also define persistence-capable container classes.

Defining a class to be persistence-capable affects the class's position in the inheritance hierarchy and the choice of types for its data members.

This chapter describes:

- The Data Definition Language (DDL) for defining persistence-capable classes
- How to make a class persistence-capable
- The data members you can define on a persistence-capable class
- Considerations for defining member functions in persistence-capable classes

Data Definition Language

You use the Data Definition Language (DDL) to describe the schema you want the DDL processor to create. The DDL therefore provides syntax for defining persistence-capable classes in Objectivity/C++. Except for a few extensions, the DDL is identical to C++ class declaration syntax. This syntax supports the definition of classes and class templates, the use of single and multiple inheritance, the declaration and definition of functions, the use of standard preprocessing directives, and so on.

The DDL extends the C++ language by:

- Implicitly including the Objectivity/C++ header file `oo.h`. This enables you to use types and classes from the Objectivity/C++ programming interface.
- Implicitly defining parameterized classes for referencing and iterating over instances of your persistence-capable classes. This enables you to use

parameterized classes such as `ooRef(`*`className`*`)` in a DDL file, even before the DDL processor generates explicit definitions for them.

- Providing syntax for defining *associations* on persistence-capable classes. Defining an association makes it possible for an application to link instances of the defining class (the source class of the association) with instances of a persistence-capable destination class.
- Providing `#pragma` preprocessing directives for controlling schema evolution, versioning, and the generation of `#include` directives in DDL-generated files.

Making a Class Persistence-Capable

Objectivity/C++ defines object persistence through inheritance. This approach follows the Object Database Management Group (ODMG) standard. Accordingly, you make a basic-object class persistence-capable by deriving it from the Objectivity/C++ class `ooObj`, either directly or through some other application-defined persistence-capable class. If you want to define a container class, you derive it from `ooContObj`.

EXAMPLE In this DDL file, `Vehicle` is a persistence-capable class because it derives from `ooObj`; `Truck` is a persistence-capable class because it derives from `Vehicle`.

```
// DDL file
class Vehicle: public ooObj {
   …
};

class Truck: public Vehicle {
   …
};
```

Multiple Inheritance

When adding persistence to a class that inherits from multiple base classes, you must ensure that:

- *Only one* base class is persistence-capable; this class must be specified first in the list of base classes that are part of the derived class definition.
- No base class is specified as virtual.
- Every non-persistence-capable base class meets the requirements for embedded non-persistence-capable classes (see page 47).

Chapter 4, “Multiple Inheritance,” discusses several strategies for adding persistence to classes with multiple ancestors.

WARNING If your application is to interoperate with Objectivity for Java or Objectivity / Smalltalk applications, your persistence-capable classes should not use multiple inheritance. Objectivity for Java and Objectivity / Smalltalk applications cannot access persistent objects of a class with more than one direct base class.

Class Templates

You make a class template persistence-capable as you would a simple class—by deriving it from `ooObj`, `ooContObj`, or an application-defined persistence-capable class. In cases of multiple inheritance, the persistence-capable base class must be first in the list of base classes.

For every instance of the persistence-capable class template you plan to use, you must provide an explicit instantiation directive in a DDL file. This enables the DDL processor to generate the usual parameterized object-reference, handle, and iterator classes for the resulting template class. That is, for every *`instantiationType`* to be used with the persistence-capable class template *`templateName`*, you must provide a directive of the following form in a DDL file:

`template class` *`templateName`*`<`*`instantiationType`*`>;`

In this directive, `template` and `class` are required C++ keywords. The class template definition must be visible to the DDL processor when the instantiation directive is processed.

The DDL processor suppresses explicit instantiation directives from the generated primary header files. This allows your C++ compiler to generate the template class definition automatically wherever it encounters a use of the template class name. If you want to override such automatic instantiation, you must repeat the explicit instantiation directive in the appropriate C++ source file of your program.

EXAMPLE The following DDL file defines a class template called `Segment`, which has associations to two point objects. The points can be described in either Cartesian or polar coordinates, so two instances of the template will be used: `Segment<CartesianPoint>` and `Segment<PolarPoint>`.

```
// DDL file
// Declare the class template Segment
template <class T>
class Segment : public ooObj {
public:
```

```
   ooRef(T) toPointA : copy(delete);
   ooRef(T) toPointB : copy(delete);
};

// Instantiate the individual template classes
template class Segment<CartesianPoint>;
template class Segment<PolarPoint>;
```

On some platforms, defining persistence-capable class templates entails extra steps when running the DDL processor; see "Platform-Specific Issues" on page 175.

Limit on Class-Name Length

The maximum length of a class name is 487 characters after all macros and `typedef` declarations have been expanded recursively. This is especially relevant when `typedef`s are used as parameters in template class names. You may be able to work around this limit by using the `-keep_typedefs` option of the DDL processor to suppress the expansion of `typedef` names; see `ooddlx` (page 170).

Defining Data Members

Every non-static data member on a persistence-capable class represents either an *attribute* or an *association*:

- The attributes of a class constitute its component persistent data; they enable an application to set the values that define the *state* of each instance of the class.
- The associations of a class enable an application to *associate* (form relationships between) persistent instances of the class and persistent instances of a related class.

For every non-static data member defined on a persistence-capable class, Objectivity / DB allocates appropriate storage space within each persistent instance of the class. The amount of storage is determined by the data-member type (for an attribute) or other syntax (for an association). In addition, persistent instances of a derived class have storage allocated for each data member inherited from the base class(es). From a data storage perspective, each base class is treated as an attribute whose type is an embedded class.

NOTE Although the DDL processor accepts static data members in persistence-capable class definitions, such members are created and destroyed entirely within the process lifetime; no storage for them is allocated as part of any persistent object.

The following subsections discuss data members that represent attributes, data members that represent associations, and data members that are neither (and are therefore prohibited in a persistence-capable class definition).

Data Members that Represent Attributes

You use ordinary C++ syntax to define the data members that represent attributes on a persistence-capable class. Each such data member:

- Must have a unique name within the scope of the defining class. In addition, the data member cannot have the same name as any direct base class. This is because base classes are treated internally as embedded attributes.
- Must be of a valid data type listed in Table 2-1. These data types ensure portability across architectures and prevent persistent objects from containing invalid data. (“Prohibited Data Types” on page 51 summarizes the data types you cannot use for members of a persistence-capable class.)
- May be scalar or a fixed-size array. Fixed-size arrays must use standard array-indexing notation—for example, `int[10]` but not `int *`.

  Warning: If your application is to interoperate with Objectivity for Java or Objectivity / Smalltalk applications, you should not use fixed-size arrays. Objectivity for Java and Objectivity / Smalltalk applications cannot access persistent objects of a class that contains a fixed-size array.

Table 2-1 lists the data types you can use in a persistence-capable class definition. The subsections that follow provide details about each data type.

Table 2-1: Valid Data Types for a Data Member Representing an Attribute

Valid Data Type	Data for the Attribute
Objectivity/C++ primitive types	Values of a particular numeric type (character, integer, floating-point, Boolean)
Accepted C++ types (enumeration, numeric, and pointer types)	Values of a particular enumeration type Architecture-specific values of a particular numeric type Transient pointers of a particular type (*accepted but not recommended*)
Objectivity/C++ object-reference types	Object references to instances of a particular persistence-capable class or structure

Table 2-1: Valid Data Types for a Data Member Representing an Attribute (Continued)

Valid Data Type	Data for the Attribute
Embedded-class types	Instances of a particular non-persistence-capable class embedded within the data of the containing instance
Objectivity/C++ variable-size array types (VArray types); the element type can be one of the following: ■ A primitive type ■ An object-reference type ■ An embedded-class type	Variable-size arrays (VArrays) of elements of a particular type

EXAMPLE Class `Vehicle` has data members of Objectivity / C++-defined data types.

```
// DDL file
class Vehicle: public ooObj {
// Persistent data
   public:
      ooVString license;    // Variable-length string class
      ooVString type;
      int16 doors;          // 16-bit integer type
      int8 transmission;    // 8-bit integer type
      ooBoolean available;  // Boolean type
   …
};
```

Objectivity/C++ Primitive Types

Objectivity / C++ primitive types represent basic numeric data (characters, integers, and floating-point numbers). These types ensure the portability of data across different machine architectures. For example, floating-point numbers of type `float32` and `float64` are stored in the native format of the architecture on which they are instantiated or modified; Objectivity / DB automatically converts floating-point formats among architectures in heterogeneous environments.

Objectivity / C++ defines several enumeration types. These types are typically used as parameters to Objectivity / C++ functions, not as attribute types. However, they are acceptable primitive types.

Figure 2-2 summarizes the primitive types and their names. Note that enumeration types are mapped to the Objectivity / C++ primitive type `int32` (see “Enumerations” on page 44 for details).

Table 2-2: Objectivity/C++ Primitive Type Names

Category	Type Name	Alternate Name	ODMG Name	Description
Integer	`int8`	`ooInt8`	(None)	8-bit signed integer type
	`uint8`	`ooUInt8`	`d_Octet`	8-bit unsigned integer type
	`int16`	`ooInt16`	`d_Short`	16-bit signed integer type
	`uint16`	`ooUInt16`	`d_UShort`	16-bit unsigned integer type
	`int32`	`ooInt32`	`d_Long`	32-bit signed integer type
	`uint32`	`ooUInt32`	`d_ULong`	32-bit unsigned integer type
	`int64`	`ooInt64`	(None)	64-bit signed integer type
	`uint64`	`ooUInt64`	(None)	64-bit unsigned integer type
Floating point [a]	`float32`	`ooFloat32`	`d_Float`	32-bit floating-point type
	`float64`	`ooFloat64`	`d_Double`	64-bit floating-point type
Character	`char`	`ooChar`	`d_Char`	8-bit integer type, equivalent to the native C++ `char`[b]
Boolean	`ooBoolean`	(None)	`d_Boolean`	8-bit unsigned integer type
Enumeration	*Any Objectivity/C++-defined enumeration type*		(None)	32-bit signed integer type

a. Objectivity/DB does not support floating point numbers larger than 64 bits. Consequently, Objectivity/C++ has no type corresponding to the 128-bit long double type provided by certain C++ compilers.

b. Objectivity/C++ character types are signed on architectures where the native C++ type is signed, and unsigned on architectures where the C++ type is unsigned. If you need to specify sign explicitly, you can use int8 or uint8, or you can use the C++ signed char or unsigned char type.

NOTE Objectivity / C++ string classes such as `ooVString` are embedded-class types, not primitive types.

When a data member in a DDL file is declared as an Objectivity / C++ primitive type, the DDL processor substitutes a language-independent type in the class description in the federated database schema. See “Mapping Objectivity / C++ Primitive Types” on page 189.

Accepted C++ Types

Certain C++ types are accepted by the DDL processor as valid data types for data members in a persistence-capable class. The accepted C++ types are enumerations, numeric types, and pointer types (see the following subsections). Non-persistence-capable C++ classes and structures are also valid data types; they are discussed separately in “Embedded-Class Types” on page 47.

The remaining derived C++ data types—unions, bit fields, and member pointers—are *not* accepted by the DDL processor. See “Prohibited Data Types” on page 51 for suggested workarounds.

Enumerations

An enumeration is a list of named integer constants. When a data member is of an enumerated type, Objectivity / DB stores the data member’s value in the federated database as a 32-bit signed integer (`int32`). However, Objectivity / DB does not check whether the stored value is within the range defined by the enumeration, because enumeration ranges are not represented in the schema. All applications that access an enumerated-type data member are responsible for enforcing the enumeration’s range; furthermore, interoperating Objectivity / C++, Objectivity for Java and Objectivity / Smalltalk applications must use the *same* enumerated type for the data member.

C++ Numeric Types

For compatibility with existing declarations, you can use basic C++ numeric types instead of Objectivity / C++ primitive types; however, the use of C++ numeric types reduces the portability of your application. Most C++ numeric types are equivalent to Objectivity / C++ primitives as shown in Table 2-3.

WARNING When using C++ primitive types, you should make sure that the primitive-type mappings allow portability across *all* of your target computing environments. Wherever possible, you should use Objectivity / C++ primitive types instead.

Table 2-3: C++ Primitive Types and equivalent Objectivity/C++ Primitive Types

Category	C++ Type	Equivalent Objectivity/C++ Primitive Type
Integer	`short` types	16-bit integer types
	`int` types	32-bit integer types
	`long` types	32-bit integer types on all platforms except DEC Alpha 64-bit integer types on DEC Alpha
	`long long` types (not available on Windows)	64-bit integer types
	`__int64` types (Windows only)	64-bit integer types
Floating point	`float`	`float32`
	`double`	`float64`
	`long double`	Not equivalent to any Objectivity/C++ type; `long double` may not be used for persistent data
Character	`char`	`char`
	`unsigned char`	`uint8`
	`signed char`	`int8`
	`wchar_t`	`uint16`
Boolean	`bool`	`int32` on Solaris 2.6 unsupported on IBM Risc/System 6000 `int8` on all other platforms
Enumeration	`enum` *`type`*	`int32`

When a data member in a DDL file is declared as a C++ primitive type, the DDL processor substitutes a language-independent type in the class description in the federated database schema. See “Mapping C++ Primitive Types” on page 190.

C++ Pointer Types

The DDL processor accepts data members whose types are C++ pointers, but generates warning messages because the use of such types is not recommended. This is because pointers (addresses in virtual memory) are meaningless as persistent data; when a pointer is stored in the database by one process, that pointer is useless (and possibly dangerous) when accessed by another process.

It is possible to define a pointer-typed data member if you want a persistent object to hold a pointer as *transient data*. When you define such a data member, space is allocated in the containing persistent object, so that your application can assign an appropriate pointer value during execution. However, the application should set the data member to 0 before committing the transaction to ensure that the pointer is not written to the database.

C++ pointers may *not* be used:

- As a link to an instance of a persistence-capable class; you must use an object-reference type instead.
- As a link to an instance of a non-persistence-capable class; consider using an embedded class type instead.
- As a link to the first element of an array; consider using either a VArray or a fixed-size array in standard array-indexing notation—for example, `int[10]` rather than `int *`.

Object-Reference Types

An object reference is a "smart pointer" to a persistent object. Whereas a C++ pointer refers to an object using its address in virtual memory, an Objectivity / C++ object reference refers to a persistent object using the object's storage location in the database (that is, its object identifier). Object references also provide member functions that allow you to manipulate persistent objects.

You can use an object reference to link a persistent source object to a persistent destination object, similar to using a C++ pointer to link a transient source object to a transient destination object. To use a standard object reference, you define a data member whose type is `ooRef(`*`className`*`)` in the source class; to use a short object reference, you define a data member whose type is `ooShortRef(`*`className`*`)`. In either case, *`className`* is the persistence-capable destination class. A parameterized class `ooRef(`*`className`*`)` is generated by the DDL processor for each application-defined persistence-capable class *`className`*; see "References Header File" on page 21 and "Obtaining Generated Class Definitions" on page 28.

As an alternative to object-reference types, you can consider using associations to link related persistent objects.

Although handles are similar to object references, you cannot use handles to link persistent objects together. The DDL processor signals an error if it encounters a data member of type `ooHandle(`*`className`*`)` in a persistence-capable class.

EXAMPLE The class `Vehicle` has a data member `fleet` to link a vehicle to its rental fleet. The class `Fleet` has a data member `vehicles` containing a fixed-size array of one thousand object references to vehicles; this data member links a rental fleet to the vehicles in it.

```
// DDL file
class Vehicle : public ooObj {
   ...
   ooRef(Fleet) fleet;      // Object reference to a fleet
};

class Fleet: public ooObj {
   ...
   ooRef(Vehicle) vehicles[1000];  // Fixed array of object refs
};
```

Objectivity/C++-defined persistent-collection classes are commonly linked through object-reference types—for example, a data member of type `ooRef(ooMap)` links an object to a persistent name map (a persistent instance of `ooMap`).

Object References and Template Classes

When *`className`* is a template class with multiple parameters, the name of the generated object-reference class contains the symbol `OO_COMMA` to separate the template parameters. For example, for a persistence-capable template class `Example<Float, Node>`, the generated object-reference class is `ooRef(Example<Float OO_COMMA Node>)`. This is because the macro syntax of the `ooRef` class name interprets embedded commas as separators between the macro parameters instead of as separators between the template parameters.

Embedded-Class Types

An application-defined class or structure is a valid data type if all of the following are true:

- The class is non-persistence-capable—that is, it does not inherit from `ooObj`.
- The class has no virtual base classes.
- All of the class's data members represent attributes—that is, the class contains no associations.
- Every data member is of a valid data type (see Table 2-1), and no data member is of a prohibited data type (see Table 2-4).

When a data member's type is a valid non-persistence-capable class, you can assign an instance of that class to the data member. The data of the assigned instance is then embedded within the data of the containing persistent object.

Certain Objectivity / C++-defined classes are commonly used as embedded-class types:

- String classes such as `ooVString`, `ooString(N)`, and `ooUtf8String`
- VArray classes

Although an object-reference type is actually an embedded object-reference class, object-reference data members are considered a separate category. Discussions of embedded-class types in this document do not include object-reference types.

Some Objectivity / C++-defined classes are prohibited as embedded-class types. Specifically, the DDL processor signals an error if it encounters a data member whose type is a handle class, an iterator class, or a temporary VArray class. See "Prohibited Data Types" on page 51.

EXAMPLE The class `Line` has a data member `points` containing a fixed-size array of points. Each element of the array is an embedded instance of the non-persistence-capable structure `Point`.

```
// DDL file
struct Point{
   int32 xCoord;
   int32 yCoord;
};

class Line : public ooObj {
   ...
   Point points[2];
   ...
};
```

WARNING If your application is to interoperate with Objectivity for Java or Objectivity / Smalltalk applications, your persistence-capable classes should avoid embedded-class types other than `ooVString` and `ooUtf8String`. Objectivity for Java and Objectivity / Smalltalk applications cannot access persistent objects with an embedded application-defined class, including `ooString(N)`.

Variable-Size Arrays (VArrays)

Objectivity / C++ variable-size arrays (VArrays) are similar to C++ arrays, except that they can change in size at runtime. To use a VArray as a data member in a persistence-capable class, you define the data member to be of type `ooVArrayT<`*`element_type`*`>`, where *`element_type`* is the type of each element in the VArray.

The *`element_type`* of a VArray can be a primitive type, an object-reference type, or a valid embedded-class type. However, a VArray cannot have elements that are themselves VArrays. This imposes an extra restriction on embedded-class types for elements—you can use an otherwise valid non-persistence-capable class as an element type only if none of its data members is of a VArray type.

Like the elements of fixed-size C++ arrays, the *`element_type`* of a VArray must have a default constructor (a constructor that can take no parameters). Note that if *`element_type`* is a class with application-defined constructors, one of these must be an explicitly-defined default constructor.

EXAMPLE The class `Polygon` has a data member `vertices` containing a VArray of points. Each element of the VArray is an embedded instance of the non-persistence-capable structure `Point`.

```
// DDL file
struct Point{
   int32 xCoord;
   int32 yCoord;
};

class Polygon : public ooObj {
   ...
   ooVArrayT<Point> vertices;
   ...
};
```

Class `ooVArrayT<`*`element_type`*`>` is a valid embedded-class type; instances of this class (sometimes called *standard VArrays*) can be saved persistently if they are embedded in (or inherited by) a persistent object. In contrast, the related class `ooTVArrayT<`*`element_type`*`>` is prohibited as an embedded-class type, because all of its instances (called *temporary VArrays*) must remain transient. The elements of a temporary VArray may be objects such as handles, which contain transient data.

Class `ooVArrayT<`*`element_type`*`>` is a template class. For backward compatibility, you can use the equivalent macro-expanded class `ooVArray(`*`element_type`*`)` instead.

Data Members that Represent Associations

The persistence-capable class that defines an association is called the *source class* for that association. The association indicates how an instance of the source class can be related to one or more instances of a *destination class*. The destination class can be any persistence-capable class, including the source class itself.

At runtime, associations can be formed, each one linking a particular instance of the source class, called the *source object*, to an instance of the destination class, called the *destination object*. An application can then:

- Traverse a link from a source object to find the destination object.
- Treat a group of associated objects as a single composite object for purposes of deleting or locking.
- Rely on the database to maintain referential integrity between objects related by a bidirectional association.

You define an association using C++ data-member syntax with DDL extensions. The data type is an object-reference class `ooRef(`*`className`*`)`, where *`className`* is the destination class. You use the extensions to specify the directionality and cardinality of the association, whether operations on objects are to propagate along links to destination objects, and how the links are handled when you create a copy or a version of a source object.

For each association you define, the DDL processor generates member functions for dynamically creating, navigating, and deleting actual associations from an object of the source class to an object of the destination class. You can think of an association as a data member whose values are accessed only through the generated interface. See:

- Chapter 3, “Defining Associations,” in this book for a complete discussion of the characteristics of associations and the DDL syntax for defining them.
- The associations chapter in the Objectivity/C++ programmer’s guide for information about using the associations you define.

EXAMPLE A pair of bidirectional associations link a fleet and its vehicles. The class `Vehicle` has a many-to-one association `fleet` that relates a vehicle to its fleet. The class `Fleet` has a one-to-many association `vehicles` that relates a rental fleet to the vehicles in it. The two associations are inverses of each other.

```
// DDL file
class Vehicle : public ooObj {
   ...
   ooRef(Fleet) fleet <-> vehicles[];
};
```

```
class Fleet: public ooObj {
   ...
   ooRef(Vehicle) vehicles[] <-> fleet;
};
```

Prohibited Data Types

Certain data types cannot be used within a persistence-capable class because they either compromise portability or allow persistent objects to contain invalid data. When the DDL processor encounters one of these types, it signals an error and leaves the schema unchanged. Table 2-4 lists the data types you cannot use in persistence-capable classes and suggests possible workarounds for them:

Table 2-4: Prohibited Data Types for Data Members of a Persistence-Capable Class

Prohibited Data Type	Workaround
Unions	See page 51
Bit fields	See page 52
Member pointers	See page 53
Embedded persistence-capable classes or structures	Use an object-reference type
Handle class `ooHandle(`*`className`*`)` Iterator class `ooItr(`*`className`*`)`	Use an object-reference type
Temporary VArray class `ooTVArrayT<`*`element_type`*`>`	Use a valid VArray type

Workarounds for Unions

Unions are not portable across architectures because objects containing unions do not carry enough information to enable Objectivity / DB to convert union branch types. If storage is not an issue, you can substitute `struct` for `union` in your expression. If storage is an issue, and you want an expression that is more closely related to your logical conceptualizing, you can define a base class corresponding to the union, along with derived classes corresponding to each union branch type.

EXAMPLE The non-persistence-capable class `Property1` has a `union` data member. An equivalent result is achieved by defining a persistence-capable class `Property2` whose derived classes `propertyInteger` and `propertyReal` correspond to the union branch types.

```
// Non-persistence-capable class; uses a union
class Property1 {
public:
   char name[32];
   int8 propType;
   union {
      int16 integer;
      float32 real;
   } value;
};
```

```
// Persistence-capable class and its derived classes
class Property2 : ooObj {   // Now persistence-capable
public:
   char name [32];
   int8 propType;
};

class propertyInteger : Property2 {
public:
   int16 integer;
};

class propertyReal : Property2 {
public:
   float32 real;
};
```

Workarounds for Bit Fields

Bit fields are not portable across architectures because different C++ language definitions do not guarantee a particular order of bits within an integer. You can achieve the functionality of a bit field either by separating the bit field into integer components or by specifying the packing of a common integer variable.

EXAMPLE The non-persistence-capable class `Picture1` defines a bit field. The persistence-capable class `Picture2` separates the bit field into integer components. The persistence-capable class `Picture3` uses member functions to specify the packing of a common integer variable.

```
// Non-persistence-capable class; uses bit fields
class Picture1 {
public:
   int32 image1: 4;      // Bit-field member
   int32 image2: 28;     // Bit-field member
};
```

```
// Persistence-capable class; uses integer data members
class Picture2 : public ooObj {
public:
   int32 image1;
   int32 image2;
};
```

```
// Persistence-capable class; specifies packing of variable
class Picture3 : public ooObj {
public:
   int32 composite;
   int32 get_image1() {
      return (composite&0XF0000000) >> 28;
   }
   int32 get_image2() {
      return (composite&0X0FFFFFFF);
   }
};
```

Workaround for Member Pointers

Member pointers (offsets into classes and structs) are not portable across architectures because they are represented differently by different compilers. You can use `int` variables as field offsets (see the `ooGetMemberOffset` global function in the Objectivity/C++ programmer's reference) or you can define a portable class wrapper for member pointers.

Member Function Considerations

In general, you can declare and define member functions in persistence-capable classes as you normally do in non-persistence-capable classes. Other than checking the syntax of function signatures, the DDL processor ignores member-function declarations and definitions for purposes of generating the schema.

Avoiding Multiple Declarations

The DDL processor adds new members to each persistence-capable class you define. These include:

- Member functions for obtaining type information.
- Operators `new` and `delete` for creating and deleting persistent instances.
- Member functions for creating, deleting, and traversing each association defined on the class.

To avoid compiler errors due to multiple declarations, you should not declare or define any member function that has the same name and parameters as a generated member function. An exception to this is the special-purpose constructor, which you can define in a persistence-capable class without producing multiple declarations.

Redefining Inherited new Operators

Because of the way C++ treats `operator new`, the `operator new` generated for a persistence-capable class will hide all `operator new` definitions that would otherwise be inherited from any non-persistence-capable ancestor classes. Consequently, you must redefine each such `operator new` in your persistence-capable class (for example, by providing an inline definition that calls the desired `operator new` on the ancestor class).

Special-Purpose Constructor

By default, the DDL processor generates a special-purpose constructor on each persistence-capable class. This constructor is called by the Objectivity / DB runtime code to create a temporary instance of the class during initialization. The generated constructor takes a single parameter of type `ooInternalObj` and is defined to invoke a suitable constructor on each base class or embedded type. Each invoked constructor is normally either:

- A similar generated constructor on a persistence-capable base class
- A default constructor (a constructor with no parameters) on a non-persistence-capable base class or embedded type

If a persistence-capable class incorporates (embeds or derives from) a non-persistence-capable type that has no default constructor, the special-purpose constructor cannot be generated, and the DDL processor fails with an error. There are several approaches to correcting this problem:

- You can define your own special-purpose constructor on the persistence-capable class, so the DDL processor does not need to generate one. The defined constructor must take a single parameter of type `ooInternalObj` and invoke an appropriate generated, default, or nondefault constructor on each incorporated type.
- You can define an explicit default constructor for each incorporated non-persistence-capable type that needs one, enabling the DDL processor to generate the special-purpose constructor on the persistence-capable class.

  If you do not want to define a default constructor on a type (or you do not want an existing default constructor to be invoked for this purpose), you can define a special-purpose `ooInternalObj` constructor on the type instead.

EXAMPLE The non-persistence-capable class `Embedded` is embedded in the persistence-capable class `PersCap`. Because class `Embedded` hides its implicit default constructor, the DDL processor is unable to generate an `ooInternalObj` constructor for class `PersCap`.

```
class Embedded {
public:
   ...                  // Data members
   Embedded(int);       // Explicit non-default constructor hides
};                      // implicit default constructor

class PersCap : public ooObj {
public:
   Embedded a;
};
```

Solution 1. Define an `ooInternalObj` constructor on class `PersCap` that calls the nondefault constructor on class `Embedded`; the DDL processor does not need to generate an `ooInternalObj` constructor. An advantage of this solution is that you do not need to modify class `Embedded`.

```
class Embedded {
public:
   ...
   Embedded(int);
};
```

```
class PersCap : public ooObj {
public:
   Embedded a;
   PersCap(ooInternalObj) : a(0) {}
};
```

Solution 2. Define an explicit default constructor for class `Embedded`. The DDL processor generates an `ooInternalObj` constructor on class `PersCap`; the generated constructor invokes the `Embedded` class's default constructor.

```
class Embedded {
public:
   ...
   Embedded(int);
   Embedded() {}
};

class PersCap : public ooObj {
public:
   Embedded a;
};
```

Solution 3. Define an explicit `ooInternalObj` constructor for class `Embedded`. The DDL processor generates an `ooInternalObj` constructor on class `PersCap`; the generated constructor invokes the `ooInternalObj` constructor you defined on the class `Embedded`. You can also use this solution to avoid calling a default constructor that performs some action that would be inappropriate during initialization, such as printing messages to the screen.

```
class Embedded {
public:
   ...
   Embedded(int);
   Embedded(ooInternalObj) {}
};

class PersCap : public ooObj {
public:
   Embedded a;
};
```

3

Defining Associations

An association is a property of a persistence-capable class, called the association's *source class*, that enables applications to link persistent instances of that class to instances of some *destination class*. When a source object is linked by an association to a destination object, an application can traverse the association from the source object to find the destination object. When the association between two objects is bidirectional, an application can rely on the database to maintain referential integrity. This chapter describes:

- The general characteristics and behavior of associations: directionality, cardinality, behavior under object-copying or versioning, delete and lock propagation, and storage layout
- How to define associations using the DDL
- A summary of the DDL syntax for defining associations

See the associations chapter in the Objectivity/C++ programmer's guide for information about using the associations you add to the schema.

About Associations

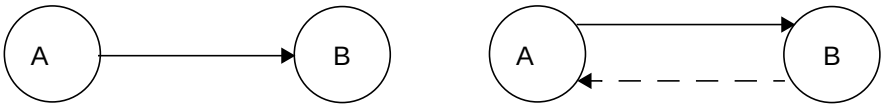
A standard practice in object modeling is to capture the links or relationships between the objects in a system. One way to implement such links is to define a persistence-capable class that contains data members of an object-reference type (for example, `ooRef(Fleet)`) and then set each object reference when objects are created. You are responsible for managing the link each time the linked objects are modified.

Objectivity/DB provides a capability for implementing links, called *associations*, that provides a higher level of functionality than simply using object references. Associations are maintained in the database by Objectivity/DB. Operations on a group of associated objects, known as a *composite object*, are handled by the database, thus reducing the amount of work you have to do to accomplish such tasks.

You specify the directionality and cardinality of an association, whether operations on objects are to propagate along an association, and how an association is handled when you create a copy or a version of an object.

Association Directionality

Association *directionality* is defined by the declaration of *traversal paths* that enable applications to locate related objects. When a single traversal path from class A to class B is declared, the association is *unidirectional*. When two traversal paths, one from class A to class B and an inverse path from class B to class A, are declared, the association is *bidirectional*.



Single traversal path
Unidirectional association

Inverse traversal paths
Bidirectional association

A source object that maintains a unidirectional association can locate its destination object, but the destination object cannot locate the source object. Unidirectional associations correspond closely to data members that contain object references, or, in a standard C++ data model, to data members that use pointers to link objects.

Bidirectional associations allow two related objects to locate each other. These associations can be connected and disconnected with a *single* method invocation; adding or removing an association in one direction simultaneously adds or removes the inverse association. In addition, bidirectional associations provide Objectivity / DB with enough information to maintain referential integrity; when a destination object is deleted, all bidirectional associations referencing that object are also deleted, reducing the likelihood of dangling object identifiers.

In contrast, it is not possible to ensure that a unidirectional association references a valid destination object. Unidirectional associations do, however, require somewhat less overhead and offer better performance than bidirectional associations.

If you are modeling a salesperson and the purchasing contacts they maintain, then you must choose whether to model this as a unidirectional association, giving the salesperson access to the contacts, or as a bidirectional association, giving the salesperson and contact objects access to each other. If it is necessary to be able to find the salesperson responsible for a given contact, then you should use a bidirectional association.

Syntax for defining an association's directionality is given in "Basic Association Syntax" on page 68.

Association Cardinality

An association's *cardinality* indicates the number of destination objects that can potentially be linked to a given source object. Objectivity / DB associations support four categories of cardinality:

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

NOTE Many-to-one and many-to-many associations must be *bidirectional*.

In the example of the salesperson with many purchasing contacts, the salesperson has a one-to-many association with the contacts. For a bidirectional association, the contacts have a many-to-one association with the salespersons. If the salespersons share contact information, then a many-to-many association would be appropriate.

Syntax for defining an association's cardinality is given in "Basic Association Syntax" on page 68.

Object Copying and Versioning

When an application creates a copy or new version of a source object that has an association, Objectivity / DB handles the association according to one of the following policies:

- *Delete* the association from the copy or new version of the source object, leaving the association in the original source object only. This is the default behavior.
- *Move* the association from the original source object to the copy or new version of the source object.
- *Copy* the association so that it exists in both the original source object and the copy or new version of the source object.

Syntax for specifying an association's object-copying or versioning behavior is given in "Specifying Object Copying and Versioning Behavior" on page 72.

The behavior specified for one path of a bidirectional association affects the inverse path of that bidirectional association. Furthermore, a different behavior can be specified for each traversal path of the association.

For example, assume a bidirectional association is defined between a salesperson `S1` and a contact `C1`, as shown in Figure 3-1.

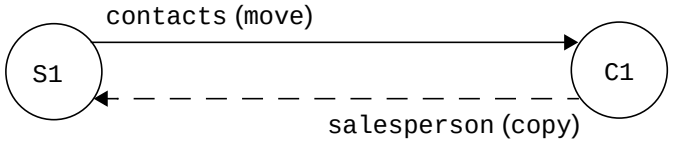


Figure 3-1 Bidirectional Association Between a Salesperson and a Contact.

When a salesperson leaves the company, all associated contacts are normally transferred to a different salesperson. The application implements this by copying the old salesperson, updating the copy with the new salesperson's individual data, and then deleting the old salesperson. Accordingly, the traversal path from `Salesperson` to `Contact` specifies the copy behavior as `move`, causing Objectivity / DB to automatically transfer all of the old salesperson's `contacts` associations to the new salesperson when the old one is copied. Because this is a bidirectional association, moving each `contacts` association automatically moves the corresponding `salesperson` association, too. Thus, when salesperson `S1` is copied to `S2` as shown in Figure 3-2, the entire bidirectional association between `C1` and `S1` is deleted, and a new one is set between `C1` and `S2`.

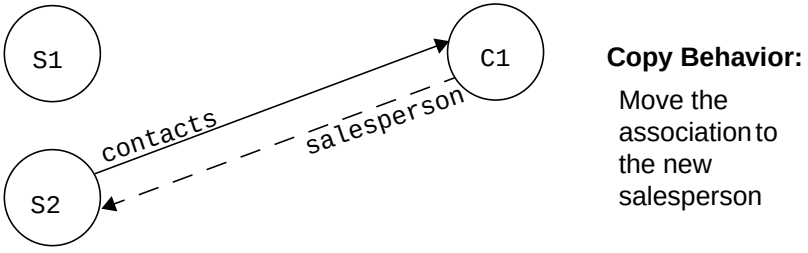
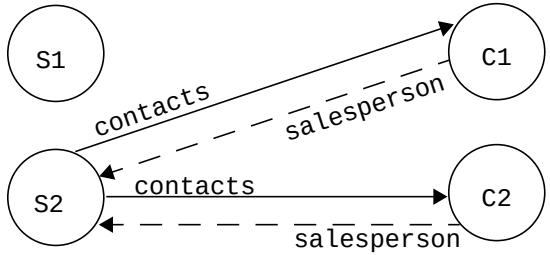


Figure 3-2 Behavior of Association When Salesperson S1 is Copied.

A given salesperson should, however, be able to make a copy or new version of a contact and maintain an association with both the original and new objects. Accordingly, the traversal path from `Contact` to `Salesperson` specifies the copy behavior as `copy`, so that when a contact is copied, its `salesperson` association is copied, too. Thus, when contact `C1` is copied to `C2` as shown in Figure 3-3, the original bidirectional association between `C1` and `S2` is kept and a new bidirectional association between `C2` and `S2` is created.



Copy Behavior:

Copy the association when a new contact is created

Figure 3-3 Behavior of Association When Contact C1 is Copied.

Propagating Operations

You can define associations so that a delete operation or an explicit lock operation will *propagate* from one object to the next along the association. Propagation is a very useful property when you wish to treat associated objects as a group, known as a *composite object*. You specify which operations should propagate, and the direction of propagation, when you define the associations in your classes. Propagation along an association is optional, and the default behavior for both delete and lock is non-propagation.

When a propagating operation is applied to an object, Objectivity / DB first identifies all objects that are affected (by identifying associations that are declared to have propagation). It then applies the operation to all affected objects in a single atomic operation. This guarantees that a propagating operation will eventually terminate, even though the propagation graph may contain cycles.

In the example from the previous section, suppose that salespersons take their contacts with them when they leave, so deleting a salesperson should delete any associated contacts as well. To support this, you enable delete propagation on the traversal path from `Salesperson` to `Contact`. (More realistically, of course, salesperson and contact objects are fairly loosely coupled, so you would probably leave propagation operations disabled in either direction of the association.)

Assume that a bidirectional association exists between a specific salesperson `S1` and contact `C1`. Because the association is bidirectional, `S1` has an object reference `R1` to `C1`, and `C1` has an object reference `~R1` to `S1`, as shown in Figure 3-4.

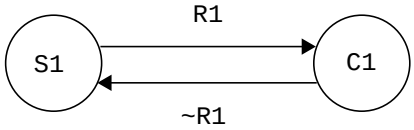


Figure 3-4 Salesperson S1 and Contact C1 Have Object References to Each Other.

Because delete propagation is enabled from `Salesperson` (but not from `Contact`):

- Deleting `S1` automatically deletes `C1`, `R1`, and `~R1`.
- Deleting `C1` does not delete `S1`. However, because a bidirectional association provides referential integrity, deleting `C1` automatically deletes `R1`.

Syntax for enabling propagation for an association is given in "Requesting Propagation Operations" on page 71.

Association Storage

Associations can be stored either *non-inline* (the default) or *inline*.

Non-Inline Associations

A *non-inline* association is stored in a *system default association array*. Each persistent object with associations has a system default association array in which all non-inline associations are stored. In the array, each association is identified by the association name (an identifier, not a string) and the object identifier (OID) of the associated object. To trace a particular association on a source object, Objectivity / DB finds the element(s) of that source object's association array that have the correct association name; it then gets the object identifier(s) for the destination object(s) from those array elements.

Inline Associations

You can also define *inline* associations. To-one inline associations are embedded as data members of an object, while to-many inline associations are placed in their own array instead of the system default association array.

There are two types of inline associations. A *standard* inline association uses a standard object identifier to refer to the destination object; a *short* inline association uses a short object identifier to refer to the destination object. A short inline association uses less storage space to maintain the association, resulting in better runtime performance. However, you can use a short inline association only if every destination object is in the same container as its source object.

NOTE For bidirectional associations, both traversal paths must have the same storage properties. If one path is inline, the other path must also be inline. If one path is short inline, the other path must also be short inline.

Syntax for specifying inline associations is given in "Inline Association Syntax" on page 70.

WARNING If your application is to interoperate with Objectivity/Smalltalk applications, you should not use inline associations. Objectivity/Smalltalk applications cannot access persistent objects of a class that contains an inline association.

Storage Requirements for Associations

The standard storage overhead for a basic object is 14 bytes. This overhead is constant and is independent of an application's use of associations. The following storage requirements are for unidirectional associations. Each bidirectional association requires storage equivalent to two unidirectional associations.

A *non-inline* association requires the following additional space:

- 4 bytes for the object reference to the system default association array, whether there are associated destination objects in the array or not.
- 14 bytes for the system default association array, if there are any associated destination objects.
- 12 bytes per associated destination object.

An *inline to-one* association requires the following additional space:

- 8 bytes for a standard object reference, whether there is an associated destination object or not.
- 4 bytes for a short object reference, whether there is an associated destination object or not.

An *inline to-many* association requires the following additional space:

- 4 bytes per association for the object reference to the association array, whether there are associated destination objects or not.
- 14 bytes per association for the association array, if there are any associated destination objects.
- 8 bytes per associated destination object for a standard object reference.
- 4 bytes per associated destination object for a short object reference.

Since objects are stored on eight-byte boundaries, you should round up your size calculations to the nearest eight bytes.

EXAMPLE The class `A` has various bidirectional associations:

- A non-inline one-to-one association `toB` with destination class `B`.
- A non-inline one-to-many association `toC` with destination class `C`.
- An inline one-to-one association `toD` with destination class `D`.
- An inline one-to-many association `toE` with destination class `E`.

- A short inline one-to-one association `toF` with destination class `F`.
- A short inline one-to-many association `toG` with destination class `G`.

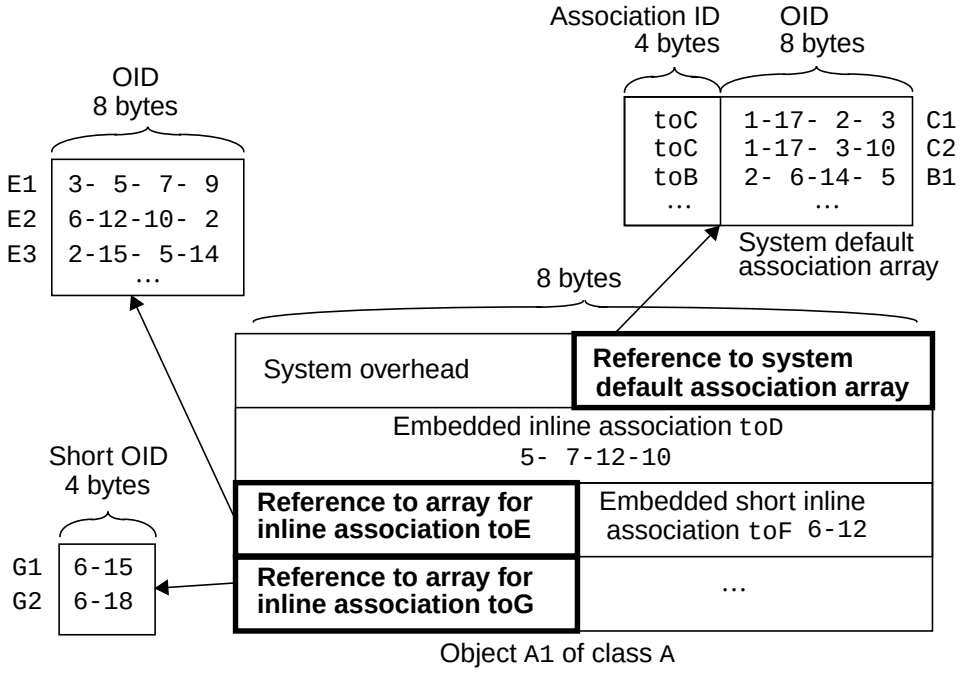
```
// DDL file
class A: public ooObj {
public:
   // Non-inline one-to-one association toB
   ooRef(B) toB <-> toA;

   // Non-inline one-to-many association toC
   ooRef(C) toC[] <-> toA;

   // Inline one-to-one association toD
   inline ooRef(D) toD <-> toA;

   // Inline one-to-many association toE
   inline ooRef(E) toE[] <-> toA;

   // Short inline one-to-one association toF
   inline ooShortRef(F) toF <-> toA;

   // Short inline one-to-many association toG
   inline ooShortRef(G) toG[] <-> toA;
};

class B: public ooObj {
public:
   ooRef(A) toA <-> toB;
};

class C: public ooObj {
public:
   ooRef(A) toA <-> toC[];
};

class D: public ooObj {
public:
   inline ooRef(A) toA <-> toD;
};

class E: public ooObj {
public:
   inline ooRef(A) toA <-> toE[];
};
```

```
class F: public ooObj {
public:
   inline ooShortRef(A) toA <-> toF;
};

class G: public ooObj {
public:
   inline ooShortRef(A) toA <-> toG[];
};
```

At runtime an instance of A, called A1, is created; A1 is linked:

- By a toB association to destination object B1.
- By toC associations to destination objects C1 and C2.
- By a toD association to destination object D1 (whose object identifier is 5-7-12-10).
- By toE associations to destination objects E1, E2 and E3.
- By a toF association to destination object F1 (located in slot 12 of page 6 of A1's container).
- By toG associations to destination G1 and G2.

The following figure illustrates the storage for the source object A1.



Choosing Between Non-Inline and Inline Storage

Choosing between non-inline and inline associations depends on how many associated destination objects your application expects a given source object to have. Non-inline associations use very little space for small numbers of destination objects, because the overhead of only one extra array is required. However, there is an implied limit on the total number of destination objects for a single source object, because the entire system default association array must fit into available swap space when the array is opened. Also, traversing a non-inline association, particularly as the number of destination objects gets large, is not very efficient, because Objectivity/DB must traverse the association names in the association array until the desired association is located.

Inline associations have a higher space overhead. To-one inline associations are embedded within objects, so they take up space even when they are not used. However, traversing inline associations is very efficient. The destination object of a one-to-one inline association can be found quickly because its object identifier is embedded in the source object. A destination object in a one-to-many inline association can also be found quickly because the application needs to traverse only the array for that particular association instead of the system default array for all non-inline associations of the source object.

Schema evolution is a second factor that may influence the choice between non-inline and inline storage. Adding a non-inline association to a class has no effect on existing instances of the class, whereas adding an inline association requires conversion of existing objects to the new representation (see “Adding an Association” on page 106).

Changing How an Association is Stored

You can change how an association is stored through schema evolution. This means modifying the class definition containing the association, processing the DDL files to generate a new schema description of the class, and modifying and rebuilding your applications with the newly-generated header and implementation files (see “Changing Association Properties” on page 121).

Objectivity/C++ supports all permutations of conversions between the different ways of storing associations. You should note, however, the following behaviors that accompany certain types of conversions:

- In any schema evolution involving a bidirectional association, you must evolve both classes at the same time to the same type of storage mode.
- Whenever associations are converted to *short inline* from any other format, references contained by the converted objects are set to null if the referenced objects are *not* in the same container.

Defining an Association

Before you can use associations in Objectivity/C++, you must first define them in the source classes whose objects are to be linked to destination objects. Associations may be defined only in application-specific persistence-capable classes; the destination class may be any persistence-capable class.

Defining an association on a source class merely makes it *possible* to associate instances of that class to objects of the destination class. As new instances are created dynamically, actual associations between those instances must also be dynamically created and deleted. Associations are created and deleted either explicitly by your application or implicitly by Objectivity/DB when objects are copied or versioned.

You use the DDL to declare the traversal paths for associations in class definitions. When you run the DDL processor, the association definitions are added to the schema as part of the class definitions. For every association you define on a class, the DDL processor generates a set of member functions on that class. The generated member functions are your sole interface to the association; you use them to dynamically create and delete associations between objects, and to navigate between associated objects. See the associations chapter in the

Objectivity/C++ programmer's guide for information about using the generated member functions.

Basic Association Syntax

You define the basic characteristics (including directionality and cardinality) of an association by declaring one or two traversal paths. A traversal path resembles a data-member definition in which:

- The data type (`ooRef(`*`className`*`)`) is an object-reference type for a persistence-capable destination class *`className`*.
- The data-member name (*`linkName`*) is the name of the association (and is the basis for the names of the generated member functions).

  `linkName` must be a unique data-member name within the defining class. Furthermore, *`linkName`* must be different from the name of any direct base class of the source class.
- Additional syntax specifies whether the association is unidirectional or bidirectional, and whether its cardinality is to-one or to-many.

Unidirectional Associations

You define a unidirectional association by declaring a single traversal path that specifies an association *`linkName`* to the destination class *`className`*:

- One-to-one

  `ooRef(`*`className`*`)` *`linkName`* `:` *`bSpec`* `{,` *`bSpec`*`};`
- One-to-many

  `ooRef(`*`className`*`)` *`linkName`*`[] :` *`bSpec`* `{,` *`bSpec`*`};`

In these descriptions, *`bSpec`* stands for a copy behavior specifier, a versioning behavior specifier, or a propagation specifier. You must specify one or more behavior specifiers for a non-inline unidirectional association so that the DDL processor can distinguish it from an attribute data member whose type is an object reference. The DDL processor will generate member functions only for associations, not for attribute data members. You must include a behavior specifier explicitly, even if you want the default behavior. The specifier for default behavior is `copy(delete)`.

EXAMPLE Assume you are modeling a salesperson with associated purchasing contacts. The following code fragment models this as the unidirectional one-to-many association `contacts` from the source class `Salesperson` to the destination class `Contact`. This association allows you to navigate from a salesperson to his or her contacts, but not from a contact to the responsible salesperson.

Although `copy(delete)` is the default behavior, this copy behavior specifier is used explicitly to distinguish the unidirectional association from an attribute data member of type `ooRef(Contact)`, such as the one shown as a code comment.

```
// DDL file
class Salesperson: public ooObj {
public:
...
   ooRef(Contact) contacts[] : copy(delete);   // Association

// Attribute data member of type ooRef(Contact)
// ooRef(Contact) contacts[];
};

class Contact: public ooObj {
public:
...
};
```

Bidirectional Associations

You define a bidirectional association between two classes by declaring a pair of traversal paths, one in each class. Each traversal path in the pair specifies an association *linkName* to the other class *className*, and identifies the inverse association *inverseLinkName* that is specified in the corresponding traversal path. You choose the exact syntax of each traversal path depending on the desired cardinality:

- One-to-one

  ooRef(*className*) *linkName* <-> *inverseLinkName*
  [: *bSpec* {, *bSpec*}];
- One-to-many

  ooRef(*className*) *linkName*[] <-> *inverseLinkName*
  [: *bSpec* {, *bSpec*}];
- Many-to-one

  ooRef(*className*) *linkName* <-> *inverseLinkName*[]
  [: *bSpec* {, *bSpec*}];
- Many-to-many

  ooRef(*className*) *linkName*[] <-> *inverseLinkName*[]
  [: *bSpec* {, *bSpec*}];

You can optionally include one or more copy behavior specifier, versioning behavior specifier, or propagation specifier, as indicated by *bspec*.

EXAMPLE Assume you are modeling a salesperson with associated purchasing contacts. The following code fragment models this as a bidirectional one-to-many association, which allows you to navigate from a salesperson to his or her contacts, and from a contact to the responsible salesperson.

Note that the two traversal paths are inverses of each other. The `contacts` association, defined in the class `Salesperson`, is one-to-many (one salesperson to many contacts), whereas its inverse association `salesRep`, defined in the class `Contact`, is many-to-one (many contacts to one salesperson).

```
// DDL file
class Salesperson: public ooObj {
public:
…
   ooRef(Contact) contacts[] <-> salesRep; // One-to-many
};

class Contact: public ooObj {
public:
…
   ooRef(SalesPerson) salesRep <-> contacts[]; // Many-to-one
};
```

Inline Association Syntax

You define an inline association by inserting the `inline` keyword at the beginning of a traversal path declaration. You specify a short inline association by using `ooShortRef(`*`className`*`)` instead of `ooRef(`*`className`*`)`.

NOTE For bidirectional associations, both traversal paths must have the same storage properties. If one path is inline, the other path must also be inline. If one path is short inline, the other path must also be short inline.

EXAMPLE Assume that you plan to create every contact in the same container as the associated salesperson. Then you can define short inline associations between them.

```
// DDL file
class Salesperson: public ooObj {
public:
…
```

```
   inline ooShortRef(Contact) contacts[] <-> salesRep;
};

class Contact: public ooObj {
public:
...
   inline ooShortRef(SalesPerson) salesRep <-> contacts[];
};
```

Requesting Propagation Operations

You request propagation operations (delete propagation, lock propagation, or both) by adding the appropriate behavior specifier(s) to the end of a traversal path declaration. The traversal path may define an association of any directionality or cardinality.

Delete Propagation

You request delete propagation for an association as follows:

traversalPath `: delete(propagate);`

If you omit this behavior specifier, delete operations on a source object are not propagated to the associated destination object(s).

Lock Propagation

You request lock propagation behavior for an association as follows:

traversalPath `: lock(propagate);`

This behavior specifier applies only to *explicit* locks (locks obtained through the `lock` member function on a handle). Locks acquired implicitly (for example, through operations such as opening an object) are never propagated.

If you omit this behavior specifier, explicit lock operations on a source object are not propagated to the associated destination object(s).

EXAMPLE Assume a bidirectional association between a salesperson and his or her contacts. Assume further that explicitly locking a salesperson object should lock the associated contacts, but deleting a salesperson should not delete the contacts. To achieve this, you add just the behavior specifier for lock propagation as shown.

Notice that no propagation behavior is specified on the association defined in class `Contact`. Consequently, deleting or locking a contact does not delete or lock the associated salesperson.

```
// DDL file
class Salesperson: public ooObj {
public:
…
   ooRef(Contact) contacts[] <-> salesRep : lock(propagate);
};

class Contact: public ooObj {
public:
…
   ooRef(Salesperson) salesRep <-> contacts[];
};
```

Specifying Object Copying and Versioning Behavior

You specify object copying and versioning behavior by adding one or more behavior specifiers to the end of a traversal path declaration. These behavior specifiers determine what will happen to an existing association when a new copy or version of its source object is created.

Object Copying

You specify a copy behavior specifier as follows:

- To copy the association from the original source object to the copy of the source object:

  traversalPath : copy(copy);

- To move the association from the original source object to the copy of the source object:

  traversalPath : copy(move);

- To delete the association in the copy of the source object and leave it in the original source object (the default behavior):

  traversalPath : copy(delete);

The copy behavior specifier causes *shallow copy* only; that is, it does not cause propagation of the copy along the association links.

EXAMPLE Assume a bidirectional association between a salesperson and his or her contacts. Assume further that:

- Copying a salesperson object should move the associations to contacts from the original salesperson to the new salesperson copy.
- Copying a contact should copy its association to a salesperson, so that both the original contact and the copied contact are associated to the same salesperson.

To achieve this, you add behavior specifiers as shown:

```
// DDL file
class Salesperson: public ooObj {
public:
…
   ooRef(Contact) contacts[] <-> salesRep : copy(move);
};

class Contact: public ooObj {
public:
…
   ooRef(Salesperson) salesRep <-> contacts[] : copy(copy);
};
```

Object Versioning

You specify a versioning behavior specifier as follows:

- To copy the association from the original source object to the new version of the source object:

  traversalPath `: version(copy);`
- To move the association from the original source object to the new version of the source object:

  traversalPath `: version(move);`
- To delete the relationship in the new version of the source object and leave it in the original source object (the default behavior):

  traversalPath `: version(delete);`

Combining Behavior Specifiers

You can combine the lock, delete, copy, and versioning behavior specifiers for an association by separating them with a comma.

EXAMPLE Assume a bidirectional association between a salesperson and his or her contacts. The code fragment below supports the following requirements:

- Locking a salesperson object should lock the associated contacts, but deleting a salesperson should not delete these contacts.
- When a salesperson is copied, its associations should be moved to the new copy.
- When salesperson is versioned, its associations should be moved to the new version.

```
// DDL file
class Salesperson: public ooObj {
public:
…
   ooRef(Contact) contacts[] <-> salesRep : lock(propagate),
                                            copy (move),
                                            version(move);
};

class Contact: public ooObj {
public:
…
   ooRef(Salesperson) salesRep <-> contacts[] : copy(copy);
};
```

Association Syntax Summary

Unidirectional Associations

- One-to-one

  `[inline] ooRef(`*`className`*`) `*`linkName`*` : `*`bSpec`*` {, `*`bSpec`*`};`
- One-to-many

  `[inline] ooRef(`*`className`*`) `*`linkName`*`[] : `*`bSpec`*` {, `*`bSpec`*`};`

where

className	Name of the destination class
linkName	Name of the association
bSpec	Behavior specifier (at least one required in non-inline unidirectional associations)

Bidirectional Associations

- One-to-one

```
[inline] ooRef(className) linkName <-> inverseLinkName
   [: bSpec {, bSpec}];
```

- One-to-many

```
[inline] ooRef(className) linkName[] <-> inverseLinkName
   [: bSpec {, bSpec}];
```

- Many-to-one

```
[inline] ooRef(className) linkName <-> inverseLinkName[]
   [: bSpec {, bSpec}];
```

- Many-to-many

```
[inline] ooRef(className) linkName[] <-> inverseLinkName[]
   [: bSpec {, bSpec}];
```

where

className	Name of the destination class
linkName	Name of the association in the source class
inverseLinkName	Name of the inverse association in the destination class
bSpec	Behavior specifier

Behavior Specifiers

bspec can be any of the following behavior specifiers:

`delete(propagate)`	Propagate deletion to all destination objects.
`lock(propagate)`	Propagate locking to all destination objects.
`copy(copy)`	When copying a source object, copy this association.
`copy(move)`	When copying a source object, move this association to the new copy of the source object.
`copy(delete)`	(Default) When copying a source object, delete this association from the new copy of the source object and leave it in the original source object.
`version(copy)`	When versioning a source object, copy this association.
`version(move)`	When versioning a source object, move this association to the new version of the source object.
`version(delete)`	(Default) When versioning a source object, delete this association from the new version of the source object and leave it in the original source object.

4

Multiple Inheritance

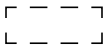
Multiple inheritance is a feature of the C++ language that allows classes to inherit attributes from more than one class. Multiple inheritance enables you to consider several data modeling alternatives for introducing object persistence.

This chapter describes:

- An extended example for illustrating the alternative data models
- Data models that use root persistence, leaf persistence, or a mixture of root and leaf persistence

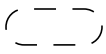
The following notational conventions are used in the figures in this chapter:

Non-persistence-capable class

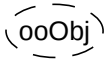


Abstract non-persistence-capable class (non-persistence-capable class from which objects will not be created)

Persistence-capable class (made persistence-capable by either direct or indirect inheritance from `ooObj`)



Abstract persistence-capable class (persistence-capable class from which objects *will not* be created)



Objectivity/C++-defined abstract persistence-capable class `ooObj`

Vehicle Data Model

Assume you have defined a vehicle data model in which classes of vehicles have either two wheels or four wheels and may also have a top speed or a cargo capacity. Instances of class `Vehicle` are not created in this data model; only particular kinds of vehicles such as vans, motorcycles, trailers, and cargo vans are useful. Furthermore, objects of the following classes are never created: `topSpeed`, `cargoCapacity`, `v4wheel`, or `v2wheel`. These are part attributes that help define particular kinds of vehicles.

Figure 4-1 and the following example show the basic vehicle data model before persistence-capability is introduced.

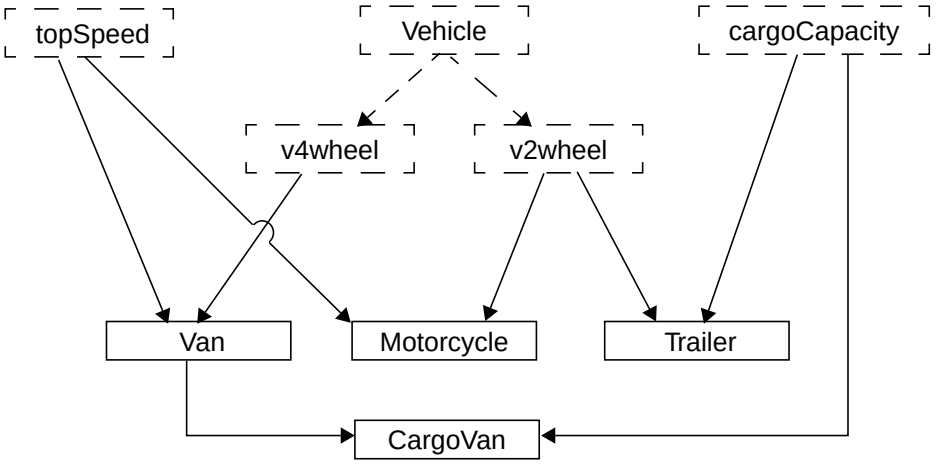


Figure 4-1 Multiple Inheritance in the Vehicle Data Model

EXAMPLE The class definitions in the vehicle model use multiple inheritance as follows:

- `Van` inherits from class `topSpeed` and class `v4wheel`.
- `Motorcycle` inherits from class `topSpeed` and class `v2wheel`.
- `Trailer` inherits from class `v2wheel` and class `cargoCapacity`.
- `CargoVan` inherits from class `Van` and class `cargoCapacity`.

```
//DDL file
class Van : public topSpeed, public v4wheel {
   ...
};
```

```
class Motorcycle : public topSpeed, public v2wheel {
   …
};

class Trailer : public v2wheel, public cargoCapacity {
   …
};

class CargoVan : public Van, public cargoCapacity {
   …
};
```

Persistence through Inheritance

Following the Object Database Management Group (ODMG) standard, Objectivity/C++ introduces object persistence through inheritance. Specifically, you make an application-defined class persistence-capable by deriving it from one of the Objectivity/C++ persistence-capable classes, typically `ooObj` or `ooContObj`. Depending on the needs of your application, persistence can be inherited either indirectly (*root persistence*) or directly (*leaf persistence*). The remainder of this chapter shows how to use each of these techniques to make the derived vehicle classes (`Van`, `Motorcycle`, `Trailer`, and `CargoVan`) persistence-capable.

Note that a derived class can inherit from *only one* persistence-capable base class, which must be specified first in the list of base classes in the derived class's definition. Furthermore, persistence-capable classes can inherit from multiple non-persistence-capable base classes of any kind *except* virtual base classes.

Data Modeling Using Root Persistence

You can use root persistence to make classes `Van`, `Motorcycle`, `Trailer`, and `CargoVan` persistence-capable. To do this, you make the common base (root) class `Vehicle` persistence-capable. When a root class is persistence-capable, all of its derived classes inherit persistence.

Adding persistence at the root level of a class hierarchy is the simplest way of making multiple derived classes persistence-capable. You can cast and convert among classes in the hierarchy just as you could before persistence was added. However, root persistence is less flexible than leaf persistence for data modeling.

EXAMPLE Root persistence is used as follows to make the classes in the model persistence-capable (see Figure 4-2):

- `Vehicle` inherits persistence from class `ooObj`.
- `v4wheel` and `v2wheel` inherit persistence from class `Vehicle`.
- `Van` inherits persistence from class `v4wheel`.
- `Motorcycle` and `Trailer` inherit persistence from class `v2wheel`.
- `CargoVan` inherits persistence from class `Van`.

```
//DDL file

class Vehicle : public ooObj {
   …
};

class v4wheel : public Vehicle {
   …
};

class v2wheel : public Vehicle {
   …
};

class Van : public topSpeed, public v4wheel {
   …
};

class Motorcycle : public topSpeed, public v2wheel {
   …
};

class Trailer : public v2wheel, public cargoCapacity {
   …
};

class CargoVan : public Van, public cargoCapacity {
   …
};
```

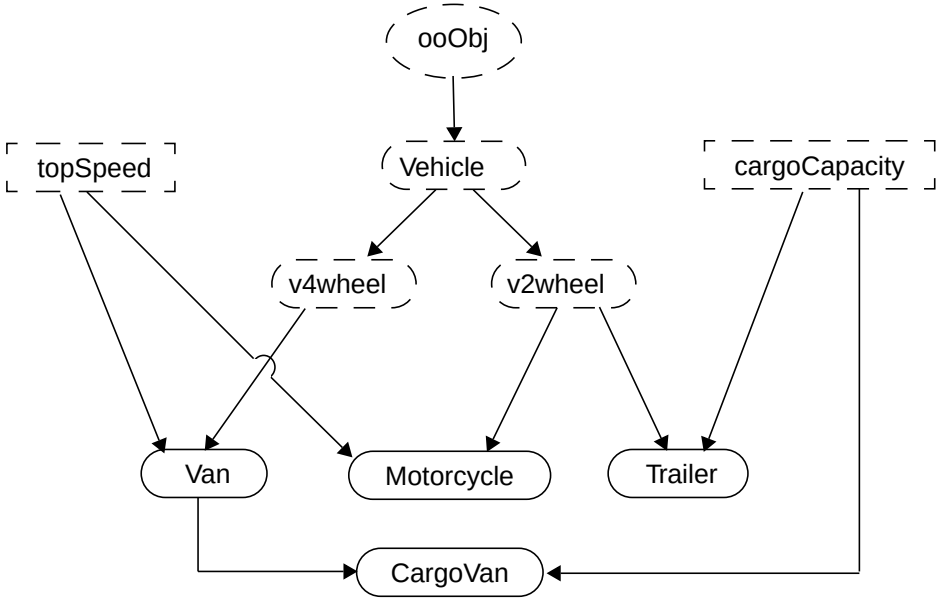


Figure 4-2 Root Persistence

Composite Objects and Root Persistence

Assume that you want to introduce a class `Motor` in the vehicle data model so that:

- `Motor` objects can be stored independently in the federated database.
- Objects of any derived vehicle class (`Van`, `Motorcycle`, `Trailer`, and `CargoVan`) have a motor.

You can make the class `Motor` persistence-capable through root persistence and then use associations to link together the persistence-capable classes `Motor` and `Vehicle`. This enables your application to create persistent objects of class `Motor` and associate them individually with instances of the derived vehicle classes. When the application sets such an association (for example, between a particular `Motor` object and a particular `Van` object), a *composite object* is created in which each component object obtains persistence through its own root.

EXAMPLE You can define associations to link motors to vehicles as follows (see Figure 4-3):

- `Vehicle` inherits persistence from class `ooObj`.
- `Vehicle` has a bidirectional association link to class `Motor`.
- `Motor` inherits persistence from class `ooObj`.
- `Motor` has a bidirectional association link to class `Vehicle`.

```
//DDL file
class Vehicle : public ooObj {
public:
   ooRef(Motor) toMotor <-> toVehicle;
   …
};

…

class Motor : public ooObj {
public:
   ooRef(Vehicle) toVehicle <-> toMotor;
   …
};
```

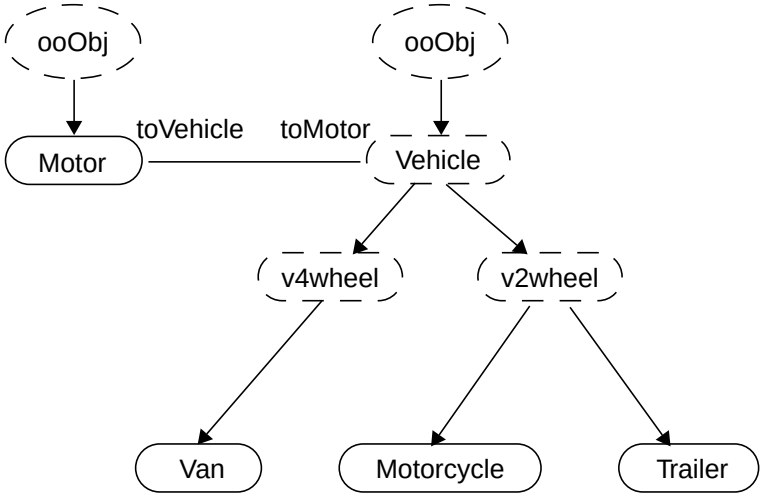


Figure 4-3 Using Composite Objects

Data Modeling Using Leaf Persistence

Leaf persistence is a more flexible way to model your data than root persistence. As shown in Figure 4-2, root persistence unnecessarily makes several abstract classes (`Vehicle`, `v4wheel`, and `v2wheel`) persistence-capable. These classes do not need persistence because no instances, persistent or otherwise, will ever be created from them. You can correct this situation with leaf persistence—by deriving persistence-capable variants of `Van`, `Motorcycle`, `Trailer`, and `CargoVan` directly from `ooObj`.

Although the data model shown in Figure 4-4 looks more complicated than the root persistence version in Figure 4-2, leaf persistence is the simplest way to make *selected* classes persistence-capable in an existing class hierarchy. The original non-persistence-capable classes are preserved and can be used with existing class libraries. Using leaf persistence, it is easy to make other classes persistence-capable as you extend the hierarchy.

EXAMPLE Leaf persistence is used as follows to make persistence-capable classes `pVan`, `pMotorcycle`, `pTrailer`, and `pCargoVan` (see Figure 4-4):

- `pVan` inherits persistence-capability from class `ooObj`, and top speed and four wheels from class `Van`.
- `pMotorcycle` inherits persistence-capability from class `ooObj`, and top speed and two wheels from class `Motorcycle`.
- `pTrailer` inherits persistence-capability from class `ooObj`, and two wheels and cargo capacity from class `Trailer`.
- `pCargoVan` inherits persistence-capability from class `ooObj`, and top speed, four wheels, and cargo capacity from class `CargoVan`.
- Classes `Vehicle`, `v4wheel`, `v2wheel`, `Van`, `Motorcycle`, `Trailer`, and `CargoVan` remain non-persistence-capable classes.

```
//DDL file
class Vehicle {
   ...
};

class pVan : public ooObj, public Van {
   ...
};

class pMotorcycle : public ooObj, public Motorcycle {
   ...
};
```

```
class pTrailer : public ooObj, public Trailer {
   …
};

class pCargoVan : public ooObj, public CargoVan {
   …
};
```

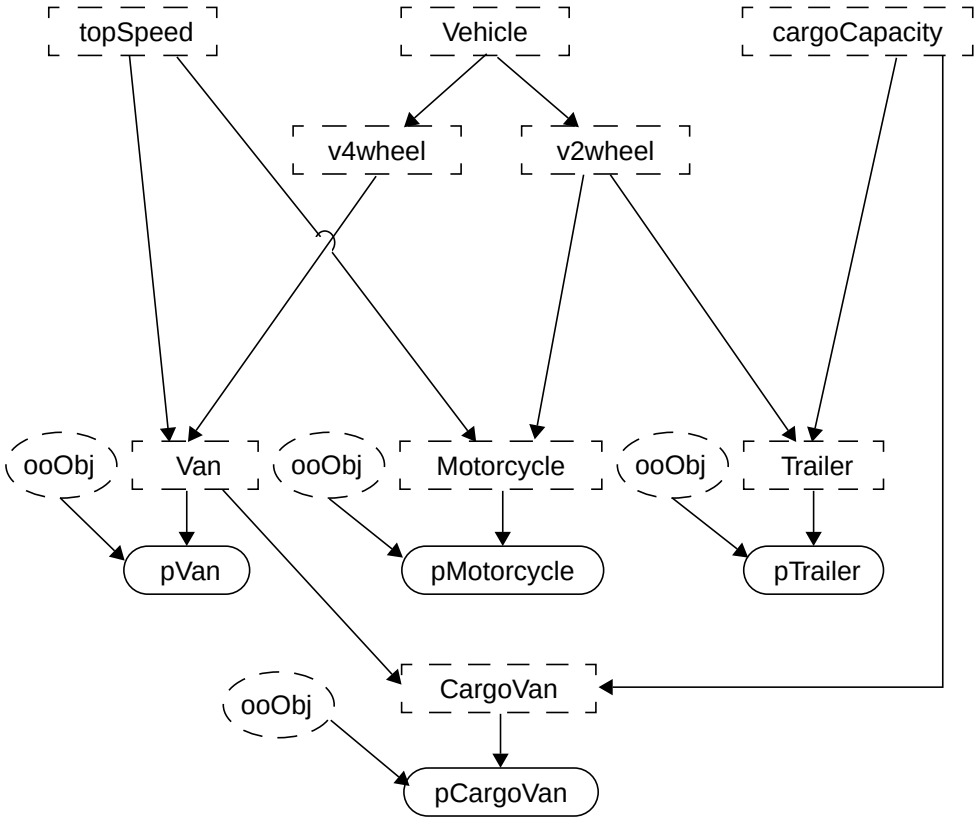


Figure 4-4 Leaf Persistence

Enhanced Leaf Persistence

Assume that you want to share a common set of associations between objects of several persistence-capable classes. You can do this by creating a persistence-capable base class for a related group of leaf classes and defining the associations common to the group in the base class. The persistence-capable leaf classes can then inherit both their persistence and their association links from that class.

EXAMPLE Leaf persistence is enhanced as follows to allow classes `pVan` and `pCargoVan` to inherit a common set of associations (see Figure 4-5):

- `paVan` inherits persistence-capability from class `ooObj` and defines associations to classes `Motor` and `airbag`.
- `pVan` inherits persistence-capability and the common associations from class `paVan`, while inheriting top speed and four wheels from class `Van`.
- `pCargoVan` inherits persistence-capability and the common associations from class `paVan`, while inheriting top speed, four wheels, and cargo capacity from class `CargoVan`.

```
//DDL file
class paVan : public ooObj {
public:
   ooRef(Motor) paVanToMotor <-> motorToPaVan [];
   ooRef(airbag) paVanToAirbag [] <-> airbagToPaVan [];
   …
};

…

class pVan : public paVan, public Van {
   …
};

…

class pCargoVan : public paVan, public CargoVan {
   …
};
```

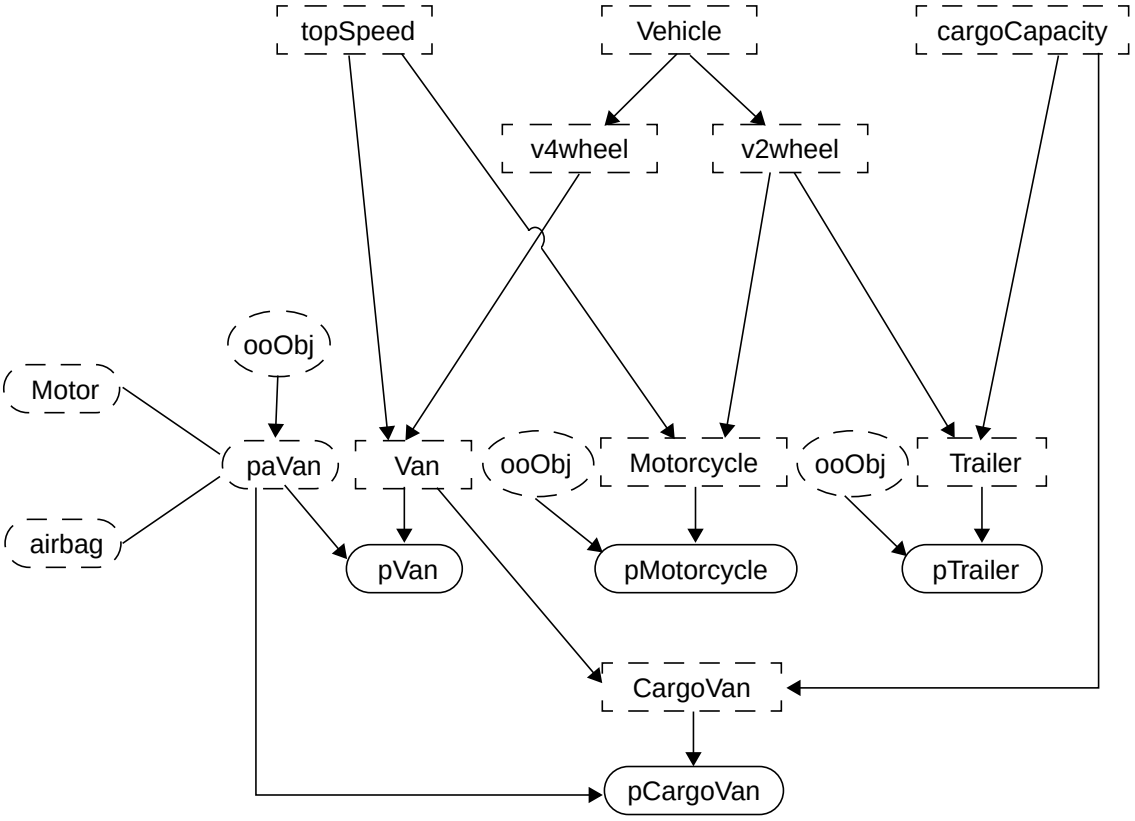


Figure 4-5 Enhanced Leaf Persistence

Mixing Root and Leaf Persistence

It is often advantageous to make some classes in a hierarchy persistence-capable through root inheritance and other classes persistence-capable through leaf inheritance. When most of the classes used by your applications are in one branch of the hierarchy, you can use root persistence for that branch and use leaf persistence for selected classes in other branches of the hierarchy.

Recall from Figure 4-3 that you used a bidirectional association to incorporate a persistence-capable class `Motor` into the vehicle data model. That solution essentially used root-persistence in two separate class hierarchies (one for `Motor` and one for `Vehicle`).

Alternatively, you can mix root and leaf persistence in a single class hierarchy that contains:

- *Persistence-capable* classes `Van`, `Motorcycle`, `Trailer`, and `CargoVan` that inherit from class `Vehicle`
- A *non-persistence-capable* motor class (`Motor`) so that objects of the persistence-capable classes can have motors through inheritance
- A *persistence-capable* motor class (`pMotor`) so that persistent motor objects can be stored in the database

Using root persistence is the easiest way to make the classes derived from `Vehicle` persistence-capable. Using leaf persistence to create class `pMotor` leaves open other possibilities for the `Motor` branch of the class hierarchy. For example, you could now create other non-persistence-capable motor classes for particular kinds of motors (four cylinder, six cylinder, turbo, and so on) through inheritance from class `Motor`, and use leaf persistence if you want to make any of these motor classes persistence-capable.

EXAMPLE Root and leaf persistence are mixed in the same class hierarchy as follows (see Figure 4-6):

- `pMotor` inherits persistence-capability from class `ooObj` and motor attributes from the non-persistence-capable class `Motor` (leaf persistence).
- `Vehicle` inherits persistence-capability from class `ooObj` (root persistence).
- `Van` inherits persistence-capability from class `v4wheel` (root persistence), and a motor from class `Motor`.
- Objects of class `pMotor` can be persistent.

```
class Motor {
   …
};

class pMotor : public ooObj, public Motor {
   …
};

class Vehicle : public ooObj {
   …
};

class Van : public v4wheel, public topSpeed, public Motor {
   …
};
```

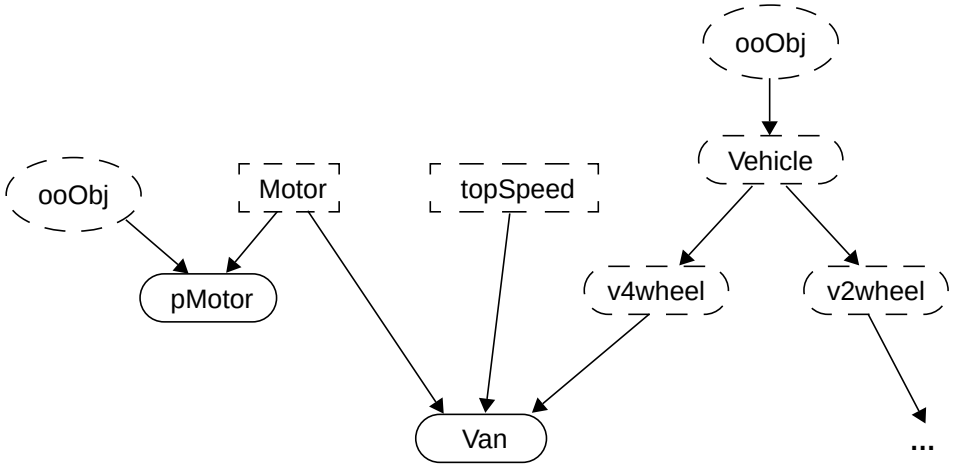


Figure 4-6 Mixing Root and Leaf Persistence

5

Schema Evolution

During basic schema development, the DDL processor allows you to add new definitions to a schema, but prevents you from changing definitions that are already in the schema. If you need to modify an existing definition—for example, by renaming data members, altering the inheritance hierarchy, or changing data-member types—you can choose to *evolve* the class definition in the schema.

When you evolve an existing class definition, you change its representation in the schema. This means that all existing instances of the class will be converted to the new representation the next time they are opened. If, however, your goal is to preserve and access existing instances of the original definition while creating new objects from the new definition, you should use class versioning instead of schema evolution (Chapter 6, “Class Versioning”).

This chapter describes:

- General information about schema evolution
- General information about performing schema-evolution operations, including a list of supported operations
- Operations for evolving class members
- Operations for evolving whole classes
- Distributing an evolved schema to deployed federated databases

About Schema Evolution

During Objectivity / C++ application development, you normally modify the data model many times as a consequence of the iterative, object-oriented design process. At various points in the design process, you will want to test your schema by building and populating a new federated database that uses the evolved schema. Each time you do this, you will likely delete the old federated database and simply recreate it with the new schema. You can repeat this process over the entire course of development until you are satisfied that the schema meets the design needs for your database product.

The effect of changing the schema gets more complicated, however, once you deploy a federated database, its schema, and database applications to your end users. For deployed products, you must consider how schema changes will affect data created by your end users in federated databases that are based on the old schema. Since it is not practical for your end users to delete their federated databases and recreate them, you must provide them with the ability to:

- Change, or *evolve*, the existing definitions in a schema.
- Convert existing data in a federated database to conform to the new definitions.

This chapter focuses primarily on schema evolution itself—how to change the schema of a federated database so that the modified schema can be distributed to end users and integrated with their existing federated databases.

A separate chapter about object conversion in the Objectivity/C++ programmer's guide describes the various ways to convert existing objects in a federated database to match a modified schema. Although object conversion is described briefly in the following subsections, you should also read the object conversion chapter *before* performing any schema-evolution operations, because:

- Deciding how to convert objects is an essential part of an overall schema-evolution plan.
- Performing object conversion is an intermediate step in certain schema-evolution operations.

Schema-Evolution Operations

As you add or update applications in your database product, you may need to change some of the C++ classes that model your persistent data. Such changes must be reflected in the schema of any existing federated database that these applications access. A *schema-evolution operation* is a set of steps for making a particular change to the representation of a class in a federated-database schema.

Each schema-evolution operation described in this chapter consists of one or more basic building blocks called *cycles*. In a schema-evolution cycle, you:

- Modify the appropriate class definitions in one or more DDL files.
- Process the affected DDL files by running the DDL processor with evolution-specific options.

The result of a cycle is to update the schema in a federated database and generate an updated set of header and implementation files. You use these files to create or rebuild any application that will access the federated database.

In some operations, cycles are followed by steps that convert existing data to match the changed schema, usually in preparation for subsequent cycles. Other operations allow you to choose when and how conversion will occur.

What You Can Change

You can evolve any application-defined class that has a representation in a federated-database schema. Thus, you can change the definitions of:

- Application-defined persistence-capable classes
- Application-defined non-persistence-capable classes that are incorporated in persistence-capable classes—for example, as data-member types or as base classes

Objectivity / DB supports schema-evolution operations for changing:

- The members of a class—for example, deleting, adding, reordering, or changing data members and certain virtual member functions.
- Whole classes—for example, deleting or renaming a class, changing the derivation of a class, or splitting or merging two classes.

The following changes do *not* involve schema-evolution operations:

- Adding a new persistence-capable class (achieved by simple DDL processing).
- Adding or changing a non-persistence-capable class that is used only for creating transient objects (such classes have no representation in the schema).
- Adding or changing a member function of a class in the schema. However, because the schema represents the presence or absence of a virtual-function table (`vtbl`) for the class, you must evolve the schema to add the first virtual member function or delete the last such function.

Impact on Objects

The definitions in a schema describe the *shapes* of objects in the database (how these objects are laid out in storage). The shape of an object is a “blueprint” that specifies the size of each data member, the order of the data members, the amount of space for a `vtbl` pointer, and so on. Objectivity / DB uses this blueprint when allocating storage space for new objects and for reading and writing the persistent data stored in existing objects.

Certain schema-evolution operations affect these shape descriptions, rendering the shapes of existing objects out-of-date. When this happens, the existing objects must be converted to their new shapes. Schema-evolution operations that require the conversion of existing objects are called *conversion operations.* For example, adding a data member to a class is a conversion operation, because additional space must be allocated for the new member in each existing object of that class.

The existing objects that are affected by a conversion operation are called *affected objects.* At a minimum, the affected objects for a given conversion operation include all objects of the class whose definition was changed. In a typical database, other objects are affected, too—namely, objects of classes derived from a changed class, objects that embed objects of a changed class, and so on. Thus,

when a data member is added to a base class, additional space must be inserted into the objects of every derived class, too.

Not all schema changes affect the shape descriptions of objects—for example, changing the name of a data member changes information in the schema, but does not change the storage layout of an object. Operations that do not change object shapes are called *non-conversion operations*.

Conversion of Existing Objects

Objectivity / DB preserves consistency between a changed schema and existing objects by automatically:

- Identifying the objects that are affected by conversion operations.
- Converting each affected object the first time it is opened by an application after evolution.

Thus, the most basic scenario is to perform a schema-evolution operation and then simply run deployed applications, letting objects be converted in the normal course of events. This is called *deferred* object conversion, because the conversion of each object is deferred until the object is first accessed. Objectivity / C++ also provides functions for converting groups of affected objects *on demand*. You can invoke such functions from deployed applications or from special-purpose *conversion applications* that you create and run before restarting deployed applications.

In most cases, you decide when and how to trigger object conversion based on performance requirements and database availability. However, certain schema-evolution operations incorporate object conversion as a required step:

- In some operations, you create and run a conversion application as an intermediate step between schema-evolution cycles. Such applications typically augment standard conversion by setting the values of added or changed data members.
- In other operations, you create and run a specific kind of conversion application, called an *upgrade application*, as a final step. An upgrade application ensures proper conversion in certain internally complex cases and must be run before any other application can access the affected objects.

Impact on Persistent Data

After you perform a schema-evolution operation that adds or deletes a data member or changes a data-member type, the conversion of each affected object changes the persistent data stored in the database. In general:

- After you add a new data member, add elements to a fixed-array data-member type, or insert a new base class into an inheritance graph, the additional space allocated for each affected object contains nulls (0). If the affected data is

primitive (numerical or character), you can generally specify a default value other than 0.

- After you delete a data member, remove elements from a fixed-array data-member type, or remove a base class from an inheritance graph, the corresponding space in each affected member is deallocated, and the existing values are lost when the space is reused.
- After you change the type of a primitive data member to another primitive type, the existing value in each affected object is automatically converted to the new type according to the C++ type-conversion semantics for your architecture (compiler and platform).
- After you change the type of a non-primitive data member, the existing value in each affected object is lost unless you write a special-purpose conversion application to set the new value based on the original value.
- After you change the storage properties of an association, any existing reference is preserved, except when changing from standard to short inline.

Impact on Existing Applications

An application that accesses a federated database must be compiled with the same definitions that exist in the federated database's schema. This ensures that:

- The correct amount of space is allocated for each persistent object, both in memory and in the database.
- The application's requests for data are correctly interpreted by Objectivity / DB.
- The responses to such requests are correctly interpreted by the application.

The DDL processor generates the required definitions whenever it creates or evolves a schema. Consequently, after evolving one or more definitions in a schema, you must rebuild all existing deployed applications to provide them with the changed definitions. Depending on the nature of the change, you may need to modify such applications first.

WARNING You risk data corruption if you start an existing application after schema evolution without rebuilding the application first. When such an application accesses an affected object, the object is automatically converted to its evolved shape, which may be very different from the shape expected by the application. In the best case, data read from the object may be misinterpreted by the application; in the worst case, misinterpreted values may be written to the database and committed, with no error signaled.

Impact on Performance

When a conversion operation increases the size of an affected object, the object may need to be stored on a different page in the database. A stub is placed in the object's original location, with an object reference to the new page. *Redirecting* the object in this way allows for contiguous storage while preserving the object's original OID. However, an additional I/O operation may be required when a user application accesses a redirected object through an object reference or association.

You can reduce the need for redirection when you design your classes. If you anticipate adding new data members to a class in a future schema evolution, you can pad the class with placeholder data members.

Schema-Evolution History

Schema evolution is cumulative, so if you perform a series of conversion operations on the same class, a history of the previous shapes is retained for that class in the schema. This history is preserved until you purge it programmatically through a conversion or upgrade application.

The retained history is used during object conversion. When an affected object is first accessed after a schema-evolution operation, its shape is found in the history recorded for the class. By comparing the found shape with the class's current shape, Objectivity/DB is able to construct a program that converts the object to the current shape. This behavior enables deferred conversion to work no matter when an affected object is first accessed by an application.

WARNING Do not rely on deferred conversion when using a conversion mechanism to set values in converted objects. Although shapes are stored in a class's schema-evolution history, the code you write for setting values is *not*. You should therefore use on-demand conversion to guarantee that values are set in all affected objects before you perform any subsequent schema changes.

Schema Distribution

You normally perform schema-evolution operations on a federated database at a development site, where you test the results along with any new or updated applications. When these applications are ready for deployment, you distribute the schema changes to the existing federated databases at your end-user sites. Although it is possible to perform schema evolution directly on a deployed federated database, doing so would make your DDL files available to your end users.

You distribute schema changes as a series of optionally encoded text files, each representing the state of the schema after a schema-evolution cycle. As described

in “Distributing Schema Changes” on page 146, you use these text files, along with any conversion or upgrade applications you created, to reproduce your schema-evolution operations in the end-user database.

The distributed schema changes include the schema-evolution history for each class, which enables object conversion to take place in deployed federated databases. This means you should not consider purging schema-evolution history from the development federated database until all schema changes have been distributed and all affected objects have been converted.

Performing Schema-Evolution Operations

This section provides:

- A list of the supported schema-evolution operations. (See also Appendix E, “Schema-Evolution Quick Reference” for a summary of details.)
- Information about the steps common to schema-evolution operations:
  - “Setting Up a Development and Test Environment” on page 97
  - “Planning Schema Changes” on page 97
  - “Modifying Class Definitions” on page 98
  - “Processing Class Definitions” on page 99
  - “Capturing the Modified Schema for Distribution” on page 101
  - “Converting Objects” on page 101
  - “Modifying and Rebuilding Applications” on page 103

Supported Schema-Evolution Operations

Operation	See Page
Adding a class	125
Adding a data member	104
Adding an association	106
Adding an attribute	104
Adding a virtual member function	124
Changing class inheritance	130
Adding a non-persistence-capable base class	132
Adding persistence	141

Operation	See Page
Restructuring classes	142
Merging two associated classes	144
Merging a class with a derived class	145
Splitting a class into two associated classes	142
Splitting a class into a pair of base and derived classes	143

Setting Up a Development and Test Environment

You prepare your development site by setting up several federated databases on which to perform and test schema-evolution operations. The schemas of the development federated databases must be identical to those of the deployed federated databases to which you will distribute the changes. You can obtain a suitable development federated database through either of the following methods:

- Creating a new federated database, running the DDL processor with the original DDL files, and then populating the database with test data.
- Copying a deployed federated database, either by making a backup using the `oobackup` tool or by using the `oocopyfd` tool (see the Objectivity / DB administration book).

You should consider making one copy of the development federated database for performing schema-evolution operations and a second copy for testing the deployment and object-conversion processes.

You should also make backup copies of any DDL files and application source-code files before modifying them.

Planning Schema Changes

Before performing schema evolution, you should plan the changes you need to make, and determine how these changes will affect existing deployed applications and federated databases. To help with this analysis, you can look up the proposed changes in the “Supported Schema-Evolution Operations” on page 95 and familiarize yourself with the required steps.

The plan you develop should answer the following questions:

- Does any schema-evolution operation depend on the completion of other operations?
- How many cycles are required for each operation? (That is, how many times and in what order do you need to modify and process DDL files?)

- For each cycle, which DDL files will you modify and which will you process?
- Will you perform any conversion operations? If so, when and how do you want to trigger object conversion? In particular:
  - If you have a choice, should you use deferred conversion, on-demand conversion, or a combination of these?
  - Does any operation require you to create and run a conversion application between cycles?
  - Does any operation require you to create an upgrade application to convert all objects before restarting deployed applications?
- Does any operation affect a previously evolved class that may still have unconverted instances? Can you run a conversion application to bring these objects to the same stage of evolution before you perform the planned operations?
- Does any operation require you to set values in converted objects? If so, what mechanism will you use?
- If you are adding large data members to many objects, do you have enough disk space to accommodate the resulting database growth?
- How will your end users obtain the evolved schema? If your end users are to reconstruct the operation on their deployed federated databases, what deliverables and instructions must you prepare?
- What deployed applications will be affected by the schema changes? Will any require modification before they are rebuilt? When can the rebuilt applications be restarted?

Your plan should help you understand the state of the schema and the state of affected objects in the federated database at each point during your schema-evolution operations. Wherever possible, you should plan to convert all affected objects before proceeding with subsequent schema-evolution operations; this is essential if the conversion process is to set values in the affected objects.

Modifying Class Definitions

You begin a schema-evolution cycle by modifying one or more class definitions in your DDL files. The number of definitions you modify depends on the nature of the change. For example, deleting a primitive data member affects a single class definition, while deleting a bidirectional association affects two definitions. Depending on the application, these definitions may be in the same or in different DDL files.

DDL Pragma Directives

Some changes require you to insert a DDL pragma directive into a DDL file. In most cases, the directive helps the DDL processor interpret a modified definition correctly. For example, the `#pragma oorename` directive enables the DDL processor to distinguish a renamed data member (whose value is to be preserved in each converted object) from a deleted data member whose place has been taken by an unrelated added data member.

One DDL directive, namely `#pragma oodefault`, affects object conversion. This directive allows you to specify a default value to be set in every affected object for a new primitive data member (see Table 2-2 on page 43 for a list of primitive types).

For descriptions of the DDL pragma directives, see Appendix B, “DDL Pragmas”.

Processing Class Definitions

You complete a schema-evolution cycle by applying the changed class definitions to the schema. To do so, you run the DDL processor with the `-evolve` option on the relevant DDL file(s). In a few operations, you must specify both the `-evolve` and `-upgrade` options. The DDL processor updates the schema of the specified federated database and generates a new set of DDL output files.

Definitions You Must Process

At a minimum, you must process each modified definition, along with any related (but unmodified) definitions. More specifically:

- If you modified the definition of a persistence-capable class, you must process the definitions of that class and all its derived classes.

  Note: As for normal (non-evolution) DDL processing, the definition of a class must be processed along with the definitions of its base classes.
- If you modified a persistence-capable class template, you must process the template, all of its instantiations, and their specializations.
- If you modified the definition of a non-persistence-capable class, you must process the definitions of that class and any persistence-capable classes that incorporate it—for example, as a base class or an embedded class.
- If a change involves a bidirectional association, you must process both of the associated classes.

WARNING All of the relevant definitions must be processed before you build and run any applications; otherwise data corruption may result from the inconsistent schema.

Processing Definitions in Separate DDL Files

When the definitions to be processed reside in separate DDL files, you can generally process these DDL files sequentially in the original (non-evolution) processing order. For example, if you perform schema operations on two classes residing in separate DDL files (`one.ddl` and `two.ddl`), you would invoke the DDL processor twice:

```
ooddlx -evolve one.ddl myFD
ooddlx -evolve two.ddl myFD
```

Some changes require that multiple classes be available during a single invocation of the DDL processor while it updates the schema. For example, when a change involves a bidirectional association, both associated classes must be available. Similarly, when you change a base class, the changed class and any derived classes must be available. If these classes are defined in separate DDL files, you can use the following technique to make them available:

1. Create a dummy DDL file that contains an `#include` directive specifying each required DDL file.

   For example, assume you are changing a bidirectional association between classes `A` and `B`, defined in files `a.ddl` and `b.ddl`, respectively. You create the following `dummy.ddl` file:

   ```
   // DDL file dummy.ddl
   #include "a.ddl"
   #include "b.ddl"
   ```

2. Evolve the schema by processing the dummy DDL file without generating header or implementation files:

   ```
   ooddlx -evolve -nooutput dummy.ddl myFD
   ```

3. If you inserted any DDL `#pragma` directives in the DDL files, delete these directives from the DDL files.

4. Generate the required header and implementation files by processing the individual DDL files without evolving (or otherwise modifying) the schema:

   ```
   ooddlx -nochange a.ddl myFD
   ooddlx -nochange b.ddl myFD
   ```

DDL Processor Messages

The DDL processor informs you that the schema has been changed by printing a warning message describing each change. The DDL processor signals an error if you attempt an invalid schema-evolution operation, such as deleting a class that has derived classes. When an error is signaled, the schema is left unchanged, even if the file you were processing also contained valid changes.

Processing DDL Pragma Directives

DDL `#pragma` directives should be processed only once per definition. If you added a DDL `#pragma` directive to a DDL file as part of a schema-evolution operation, you should delete the directive after the file has been processed.

Capturing the Modified Schema for Distribution

After a complete schema-evolution cycle, you use the `ooschemadump` tool to write the evolved schema to an output file for later distribution to your end-user sites (see "Distributing Schema Changes" on page 146). You can specify the `-encode` option to encode the schema representation.

When a single operation has multiple cycles, you must capture the results of each cycle in an ordered series of output files. Therefore, you should establish a way to indicate the output order of the files—for example, through filename conventions.

WARNING Do not modify any output file produced by `ooschemadump`. Doing so will result in serious unpredictable errors when the file is used during distribution.

Converting Objects

Most schema-evolution operations allow you to choose when and how the affected objects are to be converted. For these operations, you can use any combination of:

- Deferred conversion, which allows deployed applications to trigger the conversion of individual objects as they are accessed.
- On-demand conversion, in which one or more *conversion transactions* invoke special functions to access, and therefore convert, all affected objects in particular containers, databases, or the entire federated database.

On-demand conversion can take place in deployed applications or in special-purpose conversion applications that you create and run before restarting deployed applications.

A few schema-evolution operations require that you create and use these and other object-conversion mechanisms to either set data member's values or release classes from upgrade protection.

NOTE For details about creating and using conversion transactions, conversion applications, and the object-conversion mechanisms described below (namely, conversion functions and upgrade applications), see the object conversion chapter in the Objectivity/C++ programmer's guide.

Setting Values

Many schema-evolution operations require that you set data-member values in each affected object as it is converted, usually to preserve existing data in some form. For example, when you replace one data member with another, you can use the value of the original member to calculate a value for the new member. In some cases, an original value is simply transferred to a new member; in other cases, the original value must be converted to a different type or combined with other values.

NOTE If the *same* default value is to be set for a primitive data member in *every* affected object, you can specify the value using `#pragma oodefault` in the DDL file. Otherwise, you set values with conversion functions or conversion applications.

Conversion Function

When the values to be set are primitive, and possibly different for each affected object, you can write a *conversion function* that uses a special interface to get one or more existing values from an object's pre-conversion representation, and then set the new value(s) in the object's post-conversion representation. You register a conversion function with any application that is to trigger either deferred or on-demand conversion; the function is called automatically during the conversion of each affected object.

You can safely use a conversion function with deferred conversion for an isolated schema-evolution operation. However, this may allow values to be set in some affected objects but not others (specifically, the unaccessed objects that remain unconverted). You must use on-demand conversion to convert all affected objects when you are setting values as an intermediate step in a schema-evolution operation or when you are preparing to perform a subsequent operation.

If an application is to convert the affected objects of multiple changed classes, you can register a separate conversion function for each class. However, in a given application, you can register at most one conversion function per class. Consequently, all existing objects of a class must be at the same stage of evolution to allow a registered conversion function to apply consistently; conversion functions from prior operations on the class cannot be registered in the same application.

Conversion Application

When the values to be set are non-primitive (VArrays, associations, objects of embedded classes such as Objectivity/C++ strings, and so on), you must build and run a special-purpose conversion application as an intermediate step between two cycles. Typically, the first cycle adds a new data member, to which the conversion application transfers an existing value; the second cycle then deletes an obsolete member.

For such operations, the conversion application must iterate over *every* instance of the modified class, access the obsolete member, possibly perform some computation or type conversion, and finally set the new member. In so doing, this application triggers the conversion of every accessed object.

For purposes of setting non-primitive values, the conversion application uses just the standard Objectivity/C++ iteration interface, without calling any special functions. If, however, you evolved other classes in the same cycle, you can perform general on-demand conversion in the same application by including a conversion transaction.

Releasing Classes from Upgrade Protection

Certain schema-evolution operations result in internally complex conversion processes. To ensure proper conversion, each such operation requires that you run the DDL processor with both the `-evolve` and `-upgrade` options.

The `-upgrade` option marks the changed classes (and certain related classes) as *protected* in the schema. When classes are under upgrade protection, their instances are essentially locked until you create and run a special kind of conversion application called an *upgrade application*. An upgrade application invokes a specific function that automatically converts all affected objects in the federated database and then releases the marked classes (and their instances) from upgrade protection. At this point, the affected objects can be accessed by other applications.

Modifying and Rebuilding Applications

After every schema-evolution operation, you must rebuild deployed applications before restarting them. You rebuild the applications using the new DDL-generated header and implementation files produced by the final cycle in the operation.

Before you rebuild a deployed application, you may need to modify it to make it consistent with the changed schema or to take advantage of new schema features. For example, you must modify an existing application if the evolved schema:

- Adds a data member to a class and the application is to access the new data member in any affected objects.

- Renames a class or data member that is referenced by the application (for example, in a query).
- Deletes an association that the application sets or traverses.

You do not need to modify any application that accesses affected objects without referring to any changed aspects. However, such applications must still be rebuilt with up-to-date header and implementation files.

If you are using deployed applications to trigger object conversion, you may choose to modify such applications by adding a conversion transaction that converts all affected objects within a particular storage object or by registering a conversion function for a particular class.

Evolving Class Members

This section describes operations for evolving individual members of a class. These operations include adding, deleting, renaming, replacing, or changing a data member; changing association properties; and adding or removing virtual member functions.

See also “Evolving Classes” on page 125 for operations that affect entire classes.

Adding a Data Member

You can add a data member representing an attribute to a persistence-capable or a non-persistence-capable class; you can add a data member representing an association to a persistence-capable class. (For information about attributes and associations, see “Defining Data Members” on page 40.)

Adding an Attribute

You perform a *conversion* operation when you add a data member that represents an attribute. The data member may be scalar or a fixed-size array of any valid data-member type that is permitted for attributes of persistence-capable classes. Valid data types for attributes include the primitive types listed in Table 2-1 on page 41, object-reference types, embedded-class types, or VArray types, subject to the usual limits.

During object conversion, each new primitive-typed data member is set to a default value of 0. You can request a value other than 0 by evolving the schema with a `#pragma oodefault` directive. If the new primitive-typed data member is to have a different value in each affected object (for example, computed from the object’s other members), you can set the value through either a conversion function or a conversion application.

No default values are set for new non-primitive data members during object conversion, although space is allocated for new members in the affected objects. It is your responsibility to set any required values for the added data members.

NOTE Objectivity/C++ strings (for example, of class `ooVString`) are embedded-class types, not primitive types, so you set values for a string data member as you would any other non-primitive data member.

To perform the operation:

1. Add the data member to the class definition.
2. If the new data member is of a primitive type and you want to specify a default value other than 0, insert a `#pragma oodefault` directive with the desired value immediately before the added data member—for example:

   ```
   #pragma oodefault 100
   int16 anInt16; // New data member
   ```
3. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.
4. Write the modified schema to a file by running the `ooschemadump` tool.
5. If you used a `#pragma oodefault` directive, remove it.
6. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.
7. *Optional.* Create and run a conversion application that iterates over each affected object and sets a value for the new data member (typically, a value copied or computed from another member).

   Alternatively, you can set a non-default value for a primitive-typed member by creating a conversion function and registering it with the application that is to trigger object conversion.

EXAMPLE This example adds the data member `anInt16` with an initial value of 100 to the definition of class `A`.

```
// Original DDL file
class A : public ooObj {     // Persistence-capable class
public:
   uint8 aUint8;
   float64 aFloat64;
   char aChar;
};
```

```
// Modified DDL file
class A : public ooObj {     // Persistence-capable class
public:
   uint8 aUint8;
   float64 aFloat64;
   char aChar;
   #pragma oodefault 100     // Remove after processing the file
   int16 anInt16;            // New data member
};
```

Adding an Association

Because of the way associations are stored:

- Adding an inline association is a *conversion* operation like adding any other data member.
- Adding a non-inline association is a *non-conversion* operation. No conversion is required because the new association will be stored in the pre-existing system default association array.

When objects are converted after the addition of an inline association, space for the association is allocated in each affected object. It is your responsibility to set the actual associations using a conversion (or other) application.

To perform the operation:

1. Add the association to the class definition(s):
   - If the association is unidirectional, add a single traversal path in the associating class definition.
   - If the association is bidirectional, add an appropriate traversal path in each of the two associated class definitions.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** If you added a bidirectional association, you must process *both* of the modified classes through a single DDL file.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.
5. *Optional.* Create and run a conversion application that iterates over each affected object and sets the desired association.

EXAMPLE This example adds an *inline* bidirectional association between class A and class C and a *non-inline* unidirectional association from class A to class B.

```
// Original DDL file
class A : public ooObj {     // Persistence-capable class
public:
   …
};

class B: public ooObj {      // Persistence-capable class
   …
};

class C : public ooObj {     // Persistence-capable class
public:
   …
};
```

———

```
// Modified DDL file
class B;
class C;
class A : public ooObj {     // Persistence-capable class
public:
   …
   inline ooRef(C) assocToC <-> assocToA;   // New association
   ooRef(B) assocToB : copy(delete);        // New association
};

class B: public ooObj {      // Persistence-capable class
   …
};

class C : public ooObj {     // Persistence-capable class
public:
   …
   inline ooRef(A) assocToA <-> assocToC;   // New association
};
```

Deleting a Data Member

You can delete a data member that represents an attribute or an association.

Deleting an Attribute

You perform a *conversion* operation when you delete a data member that represents an attribute. The steps below apply to data members that are scalar or fixed-size arrays of primitive, object-reference, embedded-class, VArray types.

When an affected object is converted, the existing value of the deleted data member is lost when space is reallocated.

To perform the operation:

1. Remove the data member from the class definition.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example deletes the data member `aFloat64` from the definition class `A`.

```
// Modified DDL file
class A : public ooObj {     // Persistence-capable class
public:
   uint8 aUint8;
   float64 aFloat64;
   char aChar;
};
```

Deleting an Association

Deleting an inline or non-inline association is a *conversion* operation.

To perform the operation:

1. Delete the association from the class definition(s).
   - If the association is unidirectional, delete the single traversal path from the relevant class definition.
   - If the association is bidirectional, delete *both* of the traversal paths from the relevant class definitions.

2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** If you deleted a bidirectional association, you must process *both* of the modified classes through a single DDL file.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example deletes the unidirectional association `assocToA` from the definition of class `D`.

```
// Modified DDL file
class D : public ooObj {     // Persistence-capable class
public:
   char *aName;
   float32 anArrayOfFloats[20];
   ooRef(A) assocToA : copy(delete); // Unidirectional
   inline ooShortRef(C) assocToC : copy(delete);
};
```

Renaming a Data Member

Renaming a data member is a *non-conversion* operation. The steps below apply to data members of any type, including associations.

To perform the operation:

1. Change the data-member name in the class definition. The new name must be unique within the changed class.

   Note: If you are renaming one traversal path of a bidirectional association, *do not* adjust the name in the inverse path in the associated class's definition at this time; the DDL processor will make the necessary adjustments in the generated files.
2. Specify the data member's original name by inserting a `#pragma oorename` directive immediately before the renamed member—for example:

   ```
   #pragma oorename aFloat64       // Original name: aFloat64
   float64 myFloat64;              // New name: myFloat64
   ```
3. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** If you renamed a bidirectional association, you must process *both* of the modified classes through a single DDL file. Any other related (but unmodified) classes need not be processed.
4. Write the modified schema to a file by running the `ooschemadump` tool.

5. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files. **Note:** Be sure to change any indexing or query operations that reference the data member by name.
6. Clean up your DDL files before processing them again for any reason:
   - Remove the `#pragma oorename` directive.
   - If you renamed one traversal path of a bidirectional association, adjust the name of the inverse path in the associated class's definition.

EXAMPLE This example renames the data member `aFloat65` in class `A` and the unidirectional association `assocToA` in class `D`.

```
// Original DDL file
class A : public ooObj {        // Persistence-capable class
public:
   uint8 aUint8;
   float64 aFloat64;
   char aChar;
};

class D : public ooObj {        // Persistence-capable class
public:
   char *aName;
   ooRef(A) assocToA : copy(delete);
};
```

```
// Modified DDL file
class A : public ooObj {        // Persistence-capable class
public:
   uint8 aUint8;
   #pragma oorename aFloat64    // Remove after processing
   float64 myFloat64;           // Renamed data member
   char aChar;
};

class D : public ooObj {        // Persistence-capable class
public:
   char *aName;
   #pragma oorename assocToA    // Remove after processing
   ooRef(A) assocToClassA : copy(delete); // Renamed association
};
```

Replacing a Data Member

You can replace obsolete data members by *adding* new data members and *deleting* the obsolete ones. The steps you use depend on whether the data members are of primitive types or non-primitive types. In either case, you can compute the values of the new members from the values of the obsolete members during object conversion.

Replacing one or more data members is useful for:

- Changing *both* the name and type of a data member within a class (instead of changing just the name or just the type).
- Moving a data member into a different class—typically, a base or derived class.
- Splitting a single attribute into several data members.
- Consolidating several attributes into a single data member.

Replacing Primitive Data Members Within a Class

Replacing primitive data members within a class combines two *conversion* operations in a single cycle. (Primitive data members are of any type listed in Table 2-2 on page 43.) The steps below apply when both the new *and* the obsolete data members are of primitive types (although not necessarily the same type), and belong to the same class.

To perform the operation:

1. In the same class definition:
   a. Remove the obsolete primitive data members.
   b. Add the new primitive data members.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Create a conversion function that sets the values of the new members based on the existing values of the replaced members; register this function with the application that is to trigger object conversion.
5. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example replaces the data members `firstUint8` and `secondUint8` with a single `totalUint8` data member. Because these primitive data members are all in the same class, you can register a conversion function that sets the value of `totalUint8` in each affected object to be the sum of that object's `firstUint8` and `secondUint8` values.

```
// Modified DDL file
class A : public ooObj {     // Persistence-capable class
public:
   uint8 firstUint8;         // Obsolete member
   uint8 secondUint8;        // Obsolete member
   uint8 totalUint8;         // New member
   float64 aFloat64;
   char aChar;
};
```

Replacing Non-Primitive Data Members

Replacing non-primitive data members combines two *conversion* operations, each in a separate cycle. The steps below apply when either the new or the obsolete data members are of non-primitive data types (including Objectivity/C++ string classes), are associations, or are defined in different classes.

This technique evolves the schema and converts existing objects twice. When objects are converted after the first cycle (steps 1 and 2), the affected objects contain both the obsolete and the new data members, so that a conversion application can use the former to compute values for the latter. The second cycle (steps 5 and 6) then prunes the obsolete data member.

To perform the operation:

1. Add the new data members to the appropriate class definitions. **Note:** If you are adding a bidirectional association, add the appropriate traversal paths to the two associated class definitions.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** If you added a bidirectional association, you must process *both* of the modified classes through a single DDL file.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Using the header and implementation files generated in step 2, create and run a conversion application that iterates over every object of the modified classes and sets the values of the new data members based on the values of the members you intend to replace.

   N**ote:** Conversion may increase the size of the database significantly, particularly when the converted objects contain both new and obsolete data

members. You should check disk space availability before adding large amounts of data to many affected objects.

5. Delete the obsolete data members from the appropriate class definitions.
6. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** If you deleted a bidirectional association, you must process *both* of the modified classes through a single DDL file.
7. Write the modified schema to a second file by running the `ooschemadump` tool.
8. Modify existing applications as necessary and rebuild them with the header and implementation files generated in step 6.

EXAMPLE This example replaces the data member `refToA` with an association called `assocToA`. The new association is added in the first cycle.

```
// Original DDL file
class A;
class D: public ooObj {      // Persistence-capable class
public:
   float32 aFloat32;
   ooRef(A) refToA;                  // Member to be replaced
};
```

___

```
// Modified DDL file: First cycle
class A;
class D: public ooObj {      // Persistence-capable class
public:
   float32 aFloat32;
   ooRef(A) refToA;                  // Member to be replaced
   ooRef(A) assocToA : copy(delete);  // New association
};
```

After the first cycle, a conversion application converts each affected object, setting the new association to the object referenced by `refToA`.

___

The obsolete member `refToA` is deleted in the second cycle, leaving only the association.

```
// Modified DDL file: Second cycle
class A;
class D: public ooObj {      // Persistence-capable class
public:
   float32 aFloat32;
```

```
   ooRef(A) refToA;                    // Member to be replaced
   ooRef(A) assocToA : copy(delete);  // New association
};
```

Changing a Data Member

You can change a data member's type (primitive or non-primitive), size (for a fixed-size array), storage properties (standard / short), position in the class definition, and access control. For changes that apply only to association data members, see "Changing Association Properties" on page 121.

Changing Between Primitive Types

Changing a data member from one primitive type to another is a single *conversion* operation. The data member can be scalar or a fixed-size array of any of the types listed in Table 2-2 on page 43.

During object conversion, the value of the changed data member is converted according to the C++ type-conversion semantics for the architecture (compiler and platform) of the application triggering the conversion. Precision may be lost, depending on the architecture and the types being converted. In particular, precision may be lost for:

- Integer-to-integer conversions or float-to-float conversions, where bits are lost from truncation
- Integer-to-integer conversions, where signedness is changed
- Float-to-integer conversions, where float is a non-integral value
- Integer-to-float conversions, where the integer value cannot be exactly represented in the float

WARNING If multiple applications are to trigger object conversion for a single federated database, these applications should be created on the same architecture, to ensure that primitive data-member types are converted consistently. Objectivity / C++ does not resolve any differences in conversion semantics across architectures.

For explicit control over the conversion of values, you can use a conversion function.

To perform the operation:

1. Change the data-member type in the class definition.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.

3. Write the modified schema to a file by running the `ooschemadump` tool.
4. *Optional.* Create a conversion function that sets a new value for the changed member based on the existing value; register this function with the application that is to trigger object conversion.
5. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example changes the type of data member `aUint8` from `uint8` to `int16`.

```
// Original DDL file
class A : public ooObj {     // Persistence-capable class
public:
   uint8 aUint8;
   ...
};
```

```
// Modified DDL file
class A : public ooObj {     // Persistence-capable class
public:
   int16 aUint8;             // Change type to int16
   ...
};
```

Changing the Size of a Fixed-Size Array

Changing the size (number of elements) of a fixed-size array is a single *conversion* operation. The fixed-size array may be of any data type, including object references to persistence-capable classes. You can:

- Increase the size of the fixed-size array. This includes changing a scalar (non-array) data member to be a fixed-size array of the same data type.
- Decrease the size of the fixed-size array. This includes changing an array data member to be a scalar (non-array) data member of the same data type.

When an affected object is converted, the changed data member is initialized with a copy of the original array. If the new array is larger, additional elements are set to 0. If the new array is smaller, data is lost from the end of the array when space for the excess elements is reallocated.

To perform the operation:

1. Change the array size as desired in the class definition.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.

3. Write the modified schema to a file by running the ooschemadump tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example increases the size of a fixed-size array of integers and decreases the size of a fixed-size array of object references.

```
// Original DDL file
class D;
class C : public ooObj {          // Persistence-capable class
public:
   int32 anArrayOfInts[20];
   ooRef(D) anArrayOfRefs[100];
};
```

```
// Modified DDL file
class D;
class C : public ooObj {          // Persistence-capable class
public:
   int32 anArrayOfInts[100];      // Changed from 20 to 100
   ooRef(D) anArrayOfRefs[20];    // Changed from 100 to 20
};
```

Changing Between Standard and Short Storage

You perform a *conversion* operation when you:

- Change an object reference from standard to short or the reverse.
- Change an inline association from standard to short or the reverse.

When an affected object is converted, space is allocated or deallocated according to the storage requirements for the particular kind of association or object reference. If storage was changed:

- From *short* to *standard*, additional space is allocated for the standard representation; existing references are preserved.
- From *standard* to *short*, space is deallocated for the short representation. Existing references are preserved only if the referenced objects are in the same container as the affected object. Otherwise existing references are set to null, because the short representation does not preserve database and container information.

Both traversal paths of a bidirectional association must have the same storage properties. For example, if one path is standard inline, the other must also be standard inline; if one path is short inline, the other must also be short inline.

To perform the operation:

1. Change the association or object reference to the desired representation in the class definition(s). **Note:** If the association is bidirectional, change *both* traversal paths in the relevant class definitions.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** If you changed a bidirectional association, you must process *both* of the modified classes through a single DDL file.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example changes a short inline association `assocToC` to standard inline.

```
// Original DDL file
class C;
class D : public ooObj {     // Persistence-capable class
public:
   char *aName;
   inline ooShortRef(C) assocToC : copy(delete);
};
```

```
// Modified DDL file
class C;
class D : public ooObj {     // Persistence-capable class
public:
   char *aName;
   inline ooRef(C) assocToC : copy(delete);
};
```

Changing the Data Type of a Non-Primitive Member

Changing the data type of a non-primitive data member or association requires a combination of *conversion* and *non-conversion* operations. Together, these operations *replace* the original data member with a dummy member of the desired type, and then *rename* the dummy member with the original member name. With this technique, you can:

- Change a fixed-size array to a variable-length array (VArray) or to a fixed-size array with a different number of dimensions. (But changing the size of a fixed-size array is a simple conversion operation.)
- Change an object reference to an association, or, conversely, change an association to an object reference. (But changing between standard and short storage is a simple conversion operation.)

- Embed a different non-persistence-capable class—for example, by changing a data-member type from `X` to `Y`, where `X` and `Y` are non-persistence-capable classes.
- Specify a different referenced or associated class—for example, by changing a data member or association type from `ooRef(A)` to `ooRef(B)`, where `A` and `B` are persistence-capable classes.

This technique evolves the schema (and converts existing objects) three times. The first two cycles (steps 1, 2, 5 and 6) enable a conversion application to use the original values of the changed data member to compute the new values in the affected objects. The third cycle (steps 8 and 9) restores the data member's original name.

NOTE If you are changing a data member to or from a bidirectional association, be sure to edit and process the definitions of both classes in the association.

To perform the operation:

1. Add a dummy data member with the desired type to the class definition.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Using the header and implementation files generated in step 2, create and run a conversion application that iterates over every object of the modified class, setting the dummy member's value based on the original member's value.

   N**ote:** Conversion may increase the size of the database significantly, particularly when the converted objects contain both the dummy and original data members. You should check disk space availability before adding large amounts of data to many affected objects.
5. Delete the original data member from the class definition in the DDL file.
6. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.
7. Write the modified schema to a second file by running the `ooschemadump` tool.
8. Rename the dummy data member in the DDL file, giving it the name of the original data member; specify the dummy name in a `#pragma oorename` directive immediately preceding the renamed member.
9. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.

10. Write the modified schema to a third file by running the ooschemadump tool.
11. Remove the #pragma oorename directive.
12. Modify existing applications as necessary and rebuild them with the header and implementation files generated in step 9.

EXAMPLE This example changes the referenced class of the someRef data member by changing the data-member type from ooRef(A) to ooRef(B). Assume that class B is derived from class A and that every existing association is set to the A part of an existing B object.

The dummy data member (of type ooRef(B)) is added in the first cycle.

```
// Original DDL file
class A;
class D: public ooObj {            // Persistence-capable class
public:
   float32 aFloat32;
   ooRef(A) someRef;               // Original data member
};
```

```
// Modified DDL file: First cycle
class A;
class B;
class D: public ooObj {            // Persistence-capable class
public:
   float32 aFloat32;
   ooRef(A) someRef;               // Original data member
   ooRef(B) dummy;                 // Dummy data member
};
```

After the first cycle, a conversion application converts each affected object, setting the dummy data member to the B object whose A part is referenced by someRef.

The original member someRef is deleted in the second cycle.

```
// Modified DDL file: Second cycle
~~class A;~~
class B;
class D: public ooObj {            // Persistence-capable class
public:
   float32 aFloat32;
   ~~ooRef(A) someRef;~~               ~~// Original data member~~
   ooRef(B) dummy;                 // Dummy data member
};
```

The dummy data member is renamed to someRef in the third cycle.

```
// Modified DDL file: third cycle
class B;
class D: public ooObj {            // Persistence-capable class
public:
   float32 aFloat32;
   #pragma oorename dummy          // Remove after processing
   ooRef(B) someRef;               // Renamed data member
};
```

Changing the Position of a Data Member

Changing the order (position) of a data member within a class is:

- A *conversion* operation for the following types of data members: primitive, fixed-size array, VArray, embedded class, object-reference, and inline association.
- A *non-conversion* operation for a non-inline association (because a non-inline association is stored in a separate system default association array).

The steps below apply to data members of any type, including associations. Note that changing the order of data members normally accompanies other changes, and is not usually worth the overhead of object conversion by itself.

To perform the operation:

1. Reorder the data members in the class definition.
2. Process the appropriate DDL file(s) using the DDL processor with the -evolve option. **Note:** A warning message occurs if this is the only conversion change on the class.
3. Write the modified schema to a file by running the ooschemadump tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

Changing the Access Control of a Data Member

Changing the access control (public, private, protected) of a data member is a *non-conversion* operation. The steps below apply to data members of any type, including associations.

To perform the operation:

1. Change the access control of the data member.
2. Process the appropriate DDL file(s) using the DDL processor with the -evolve option. **Note:** You can process just the DDL file containing the

modified class; related but unmodified classes need not be processed for this operation.

3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example changes the access control of the data member `aChar`.

```
// Original DDL file
class A : public ooObj {     // Persistence-capable class
public:
   uint8 aUint8;
   float64 aFloat64;
   char aChar;
};
```

```
// Modified DDL file
class A : public ooObj {     // Persistence-capable class
public:
   uint8 aUint8;
   float64 aFloat64;
private:
   char aChar;               // Changed access control
};
```

Changing Association Properties

You can change an association's storage properties (inline / non-inline, standard / short), behavior specifiers, and cardinality. For general changes that apply to any data member, see "Changing a Data Member" on page 114.

Changing Between Inline and Non-Inline Storage

You perform a *conversion* operation when you:

- Change an association from inline (standard or short) to non-inline.
- Change an association from non-inline to inline (standard or short).

When an affected object is converted, space is allocated or deallocated according to the storage requirements for the particular kind of association. If the affected association was changed:

- From *inline* to *non-inline*, existing references are preserved in the system default association array.

- From *non-inline* to *standard inline*, existing references are preserved in the space or array allocated for the association.
- From *non-inline* to *short inline*, existing references may be set to null; see "Changing Between Standard and Short Storage" on page 116.

Both traversal paths of a bidirectional association must have the same storage properties. For example, if one path is non-inline, the other must also be non-inline; if one path is short inline, the other must also be short inline.

To perform the operation:

1. Change the association representation in the class definition(s). **Note:** If the association is bidirectional, change *both* the traversal paths in the relevant class definitions.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** If you changed a bidirectional association, you must process *both* of the modified classes through a single DDL file.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example changes the non-inline association `assocToA` to standard inline.

```
// Original DDL file
class A;
class D : public ooObj {     // Persistence-capable class
public:
   char *aName;
   ooRef(A) assocToA : copy(delete);
};
```

```
// Modified DDL file
class A;
class D : public ooObj {     // Persistence-capable class
public:
   char *aName;
   inline ooRef(A) assocToA : copy(delete);
};
```

Changing Association Behavior Specifiers

Changing an association's behavior specifiers is a *conversion* operation. You can change the behavior specifiers for lock and delete propagation and for versioning and copy operations.

During object conversion, encoded information about the association is changed within each affected associating object, possibly increasing or decreasing the object's size.

To perform the operation:

1. Change the behavior specifiers of the association in the class definition.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example adds delete propagation to the unidirectional association `assocToA`.

```
// Original DDL file
class A;
class D : public ooObj {     // Persistence-capable class
public:
   char *aName;
   ooRef(A) assocToA : copy(move);       // Unidirectional
};
```

```
// Modified DDL file
class A;
class D : public ooObj {     // Persistence-capable class
public:
   char *aName;
   ooRef(A) assocToA :   copy(move),
                         delete(propagate); // Added specifier
};
```

Changing Association Cardinality

Changing the cardinality of an association requires a combination of *conversion* operations. Follow the steps for changing a data-member type, replacing the original association with a dummy association of the desired cardinality, and then renaming the dummy association with the original name.

Adding or Removing a Virtual Member Function

You perform a *conversion* operation when you make one of the following changes to a non-persistence-capable class that is incorporated in a persistence-capable class (for example, as a base class or embedded class):

- Introduce the *first* virtual member function (by adding the function or by changing an existing non-virtual member function to be virtual).
- Remove the *last* virtual member function (by deleting the last or only virtual member function or by changing it to be non-virtual).

These operations change whether storage is to be allocated in the affected objects for a pointer to a virtual-function table (`vtbl`). Making either change to a persistence-capable class is a *non-conversion* operation because persistent objects *always* have `vtbl`-pointer storage (inherited from `ooObj`), whether or not any virtual member functions are defined on the class.

After introducing the first virtual member function to a non-persistence-capable class, object conversion allocates additional space to every affected object to accommodate the added `vtbl` pointer. After removing the last virtual member function, object conversion reduces the size of each affected object, which loses the `vtbl` pointer.

EXAMPLE This example adds the first virtual member function to class `X`, which is embedded in a persistence-capable class (not shown).

```
// Original DDL file
class X {                    // Non-persistence-capable class
public:
   uint8 aUint8;
};
```

```
// Modified DDL file
class X {
public:
   uint8 aUint8;
   virtual void aVirtual(); // First virtual member function
};
```

Evolving Classes

This section describes schema-evolution operations that affect an entire class. These operations include renaming or deleting a class, changing the inheritance of a class, adding persistence, removing persistence, and restructuring classes by merging or splitting them. A related non-evolution operation, adding a class, is also described.

See also “Evolving Class Members” on page 104.

Adding a Class

Adding a persistence-capable class to a schema does not require schema evolution. You can add a persistence-capable class (a class that derives directly or indirectly from `ooObj`) simply by adding its definition to a DDL file and processing that file. You must, however, add a class to the schema before you can make it a base class of another existing class.

Adding a non-persistence-capable class does not require DDL processing of any kind unless the class is being added as an embedded data type or a base class of a persistence-capable class.

Renaming a Class

Renaming a class is a *non-conversion* operation. Renaming a class substitutes a new class name for the original one in the schema, while preserving the original type number, shape information, and so on. You can reuse the original name only after you have completed the steps below.

To perform the operation:

1. Change the class name in the class definition. The new name must be unique within the schema.
2. Specify the class’s original name by inserting a `#pragma oorename` directive immediately before the class definition—for example:

```
#pragma oorename A                  // Original name: A
class newClassA : public ooObj {... // New name: newClassA
```

3. Change the class name in any:
   - Data members that embed the class (if it is non-persistence-capable)
   - Associations or object references to the class (if it is persistence-capable)
   - Definitions of derived classes that list the class as a base class
4. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.
5. Write the modified schema to a file by running the `ooschemadump` tool.

6. Remove the #pragma oorename directive.
7. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files. **Note:** Be sure to change any indexing or query operations that reference the class by name.

EXAMPLE This example renames class A to newClassA, and makes the corresponding adjustment to the unidirectional association assocToA in class D.

```
// Original DDL file
class A : public ooObj {           // Persistence-capable class
public:
   ...
};

class D : public ooObj {           // Persistence-capable class
public:
   ooRef(A) assocToA : copy(delete); // Unidirectional
};
```

```
// Modified DDL file
#pragma oorename A                 // Remove after processing
class newClassA : public ooObj {   // Persistence-capable class
public:
   ...
};

class D : public ooObj {           // Persistence-capable class
public:
   ooRef(newClassA) assocToA : copy(delete); // Unidirectional
};
```

Deleting a Class

Deleting a class marks it as deleted in the schema. Existing instances of the class remain in the database files but cannot be accessed by applications. Deleting a class is normally accompanied by other *conversion* operations that remove all usage of the deleted class from other classes.

You can delete one or more leaf (non-base) classes or you can delete a base class along with all of its derived classes. If you want to delete a base class but preserve its derived classes, you must first evolve the schema to remove the obsolete base class from the inheritance graph.

A non-persistence-capable class can be deleted only if no persistence-capable class uses it as an embedded data-member type. Similarly, a persistence-capable

class can be deleted only if no other persistence-capable class has an association or object reference to it. You must decide whether to delete or to change the type of each data member that embeds, associates, or references an obsolete class.

You can reuse the name of a deleted class only after you have completed the steps below.

To perform the operation:

1. Prepare each obsolete class for deletion:
   - If the class has derived classes, make it a leaf class—for example, by moving the derived classes up the inheritance graph or, if the obsolete class is non-persistence-capable, by removing it from the base list of each derived class.
   - If the class is an embedded data-member type in any persistence-capable classes, and you want to preserve these data members in some way, evolve the schema to change their types.
2. Create and run an application that deletes every object of the obsolete class(es) in the federated database.
3. For each obsolete class:
   a. Delete the definition of the class from the DDL file.
   b. Specify the deleted class in a forward class declaration immediately followed by a `#pragma oodelete` directive—for example:

      ```
      class A;     // Forward reference to a deleted class A
      #pragma oodelete A
      ```

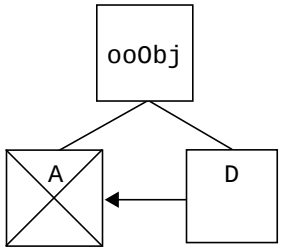
      In most cases, the declaration and pragma may be located anywhere in the DDL file from which you deleted the definition. However, when deleting a base class and its derived classes in the same operation, you must put *all* of the forward declarations and pragma directives into a single DDL file.
4. In the definitions of related classes:
   - Delete any data members that embed an obsolete non-persistence-capable class.
   - Delete any associations or object references to an obsolete persistence-capable class.
5. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.

   Note: If the deleted class has a base class to which associations or object references are defined, you must use *both* the `-evolve` and `-upgrade` options; see “When Links Exist to a Base Class” on page 129.
6. Write the modified schema to a file by running the `ooschemadump` tool.

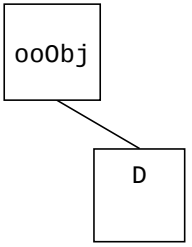
7. If you specified the `-upgrade` option in step 5, create and run an upgrade application against the federated database. No other application can access the affected objects until the upgrade application has completed.
8. Remove the forward declaration(s) and `#pragma oodelete` directive(s).
9. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example deletes the persistence-capable class `A` and the association to it. Because class `A` has no application-defined base class to which other classes are linked, the modified DDL file is processed using just the `-evolve` option, and no upgrade application is required.

Before



After




```
// Original DDL file
class A : public ooObj { // Obsolete persistence-capable class
public:
   uint8 aUint8;
   char aChar;
};

class D : public ooObj { // Persistence-capable class
public:
   char *aName;
   ooRef(A) assocToA : copy(delete); // Obsolete association
};
```

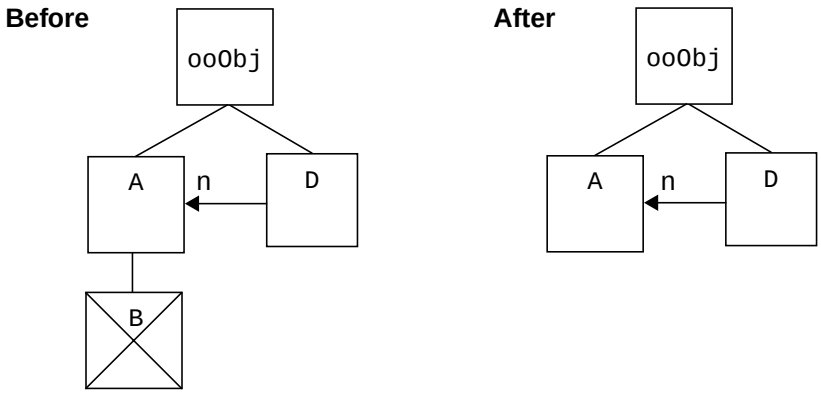
```
// Modified DDL file
class A;                 // Remove after processing
#pragma oodelete A       // Remove after processing

class D : public ooObj { // Persistence-capable class
public:
   char *aName;
   ooRef(A) assocToA : copy(delete); // Obsolete association
};
```

When Links Exist to a Base Class

When the class to be deleted is persistence-capable and has a base class to which associations or object references are defined, Objectivity/DB requires extra safeguards to ensure the proper conversion of the affected objects. Therefore, the DDL file(s) must be processed with the `-upgrade` option (step 5 on page 127), which puts the base class and the classes that reference it under upgrade protection. Applications cannot access instances of these classes until you perform object conversion using an upgrade application (step 7 on page 128).

EXAMPLE This example deletes class `B`, which derives from the base class `A`. Because class `D` has a to-many association to class `A`, the modified DDL file must be processed using *both* the `-evolve` and `-upgrade` options. (The definition of class `D` is shown in the same DDL file for convenience; this definition can, but need not be, among the processed definitions.) Because the `-upgrade` option places class `D` under upgrade protection, object conversion is performed by an upgrade application.

The upgrade steps ensure the proper conversion of each `D` object that has associations set to both `A` objects and `B` objects through `assocToA`; the associations to `A` objects are preserved while any associations to `B` objects are deleted. **Note:** The upgrade steps are still required, even if you deleted all `B` objects prior to schema evolution, as is recommended in step 2 on page 127. An error is signaled if an application attempts to access a `D` object before the upgrade application runs.



```
// Original DDL file
class A : public ooObj {     // Persistence-capable class
public:
   …
};

class B: public A {          // Class derived from class A
public:
   …
};

class D : public ooObj {     // Class associated to class A
public:
   char *aName;
   ooRef(A) assocToA[] : copy(delete); // To-many association
};
```

```
// Modified DDL file
class A : public ooObj {     // Persistence-capable class
public:
   …
};

class B;
#pragma oodelete B           // Remove after processing

class D : public ooObj {     // Class associated to class A
public:
   char *aName;
   ooRef(A) assocToA[] : copy(delete); // To-many association
};
```

Changing the Inheritance of a Class

A persistence-capable class may inherit data members from one or more base classes in addition to defining data members of its own. You can add or remove inherited data members by modifying the *inheritance graph* of a persistence-capable class. By definition, such an inheritance graph contains:

- Exactly one *persistence-capable branch* whose root is the Objectivity / C++ class `ooObj`. Persistence is inherited through this branch; every direct or indirect base class in this branch is itself a persistence-capable class.
- Zero or more *non-persistence-capable branches*; every direct or indirect base class in this branch is a non-persistence-capable class.

Figure 5-1 shows the inheritance graph of a persistence-capable class B, whose persistence-capable branch contains a direct base class A and an indirect base class ooObj. In its non-persistence-capable branch, B has a direct base class X and an indirect base class Y. Each base class (ooObj, A, Y, and X) has a corresponding part in every persistent instance of B, as shown in the layout sketch in Figure 5-1. In essence, a B object contains an embedded instance of each of its base classes.

Note that an object's layout depends on the order in which the base classes are listed. Because the persistence-capable base class must always be listed first, class B has the base list `public : A, X`. The data members inherited through the persistence-capable branch therefore precede any other data members.

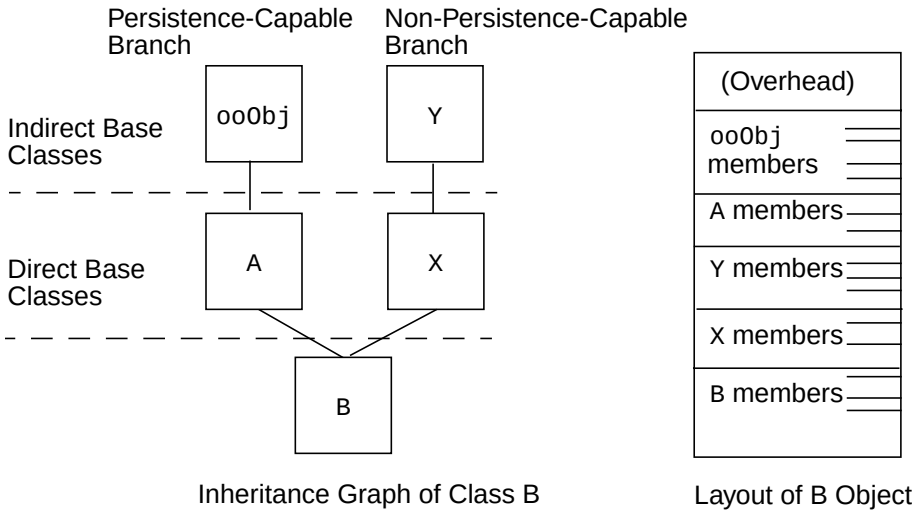


Figure 5-1 Inheritance Graph and Object Layout for the Persistence-Capable Class B

You can change the composition of a persistence-capable class by adding or removing direct or indirect base classes from any of its branches—that is, you can:

- Add a non-persistence-capable base class to any application-defined class in the inheritance graph. For example, you could add a non-persistence-capable class Z as a base class of X, Y, A, or B. This would insert the members of class Z (and its base classes, if any) into the appropriate place within each B object.
- Move the persistence-capable class lower in the inheritance graph by replacing a parent class with a sibling class. For example, if class A had a sibling class C through ooObj, you could derive class A directly from class C, making ooObj an indirect base class of A. This would insert the members of class C before the members of class A in each A object and in each B object.
- Remove a non-persistence-capable class and its base classes (if any) from the inheritance graph. For example, you could remove class X from the base list of class B to remove the X and Y parts from each B object.

- Move the persistence-capable class higher in the inheritance graph by replacing a parent class with one or more ancestor classes. For example, you could derive class `B` directly from class `Y` instead of class `X`. This would remove the `X` part (but preserve the `Y` part) of each `B` object.

You can also change the order and change the access control of a base class.

Adding a Non-Persistence-Capable Base Class

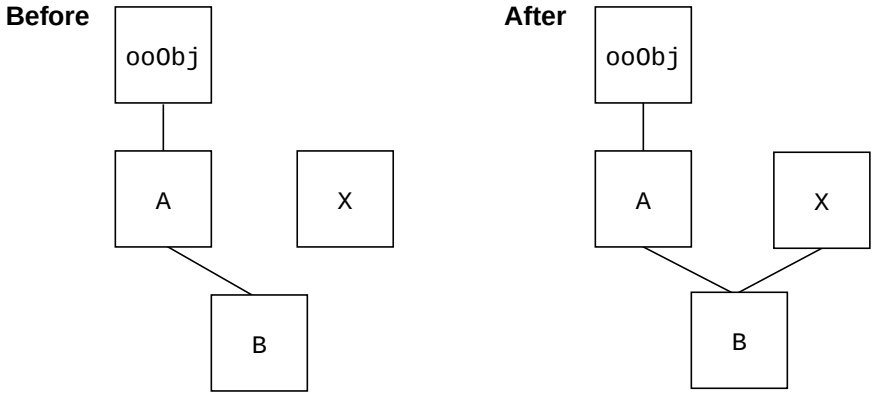
You perform a *conversion* operation when you add a non-persistence-capable class as a direct or indirect base class of a persistence-capable class.

During object conversion, each affected object is resized to add the members of the new base class. No default values are set for the new data members; you must use a conversion application to set values for such data members.

To perform the operation:

1. Add the name of the new base class to the base list of the appropriate class in the inheritance graph. You can add a non-persistence-capable base class to any application-defined class in the inheritance graph of a persistence-capable class.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** You must process the definitions of the new base class, the affected persistence-capable class, any existing base classes, and any existing derived classes.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. *Optional.* Create and run a conversion application that iterates over each affected object and sets values for the data members introduced by the new base class.
5. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example adds the non-persistence-capable class X to the base list of the persistence-capable class B, and processes the definitions of classes A, B, and X with the `-evolve` option.




```
// Original DDL file
class A : public ooObj {        // Persistence-capable class
public:
   …
};

class B : public A {            // Persistence-capable class
public:
   …
};
```

```
// Modified DDL file
#include x.h           // Defines non-persistence-capable class X
…                               // No changes to class A
class B : public A, X {         // Persistence-capable class
public:
   …
};
```

Moving a Class to a Lower Inheritance Level

You perform a *conversion* operation when you move a persistence-capable class to a lower level in its inheritance graph. You accomplish this by changing the derivation of a class in the graph so that it inherits from a *descendent* of a former direct base class. Typically, this means deriving a class from a (former) sibling instead of the parent it had shared with that sibling; the (former) parent remains in the graph as an indirect base class.

You can use this technique to insert a new base class in the middle of any branch in an inheritance graph. If the class to be inserted is persistence-capable, it must already exist in the schema and be derived from the appropriate parent class; you cannot both add a new persistence-capable class and insert it into an inheritance graph in the same evolution operation. See "Adding a Class" on page 125.

During object conversion, each affected object is resized to add the members of the new base class. These members are inserted in the appropriate place among the members of the existing base class(es), whose values are preserved. No default values are set for the new members; you must use a conversion application to set values for them.

An upgrade application is required to convert affected objects and release them from upgrade protection.

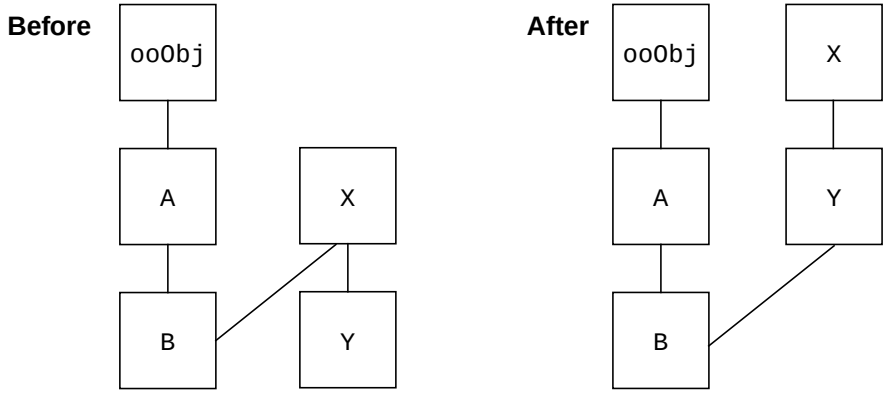
To perform the operation:

1. In the base list of the class to be lowered in the inheritance graph, replace a listed base class with an existing class that is derived from that base class.
2. Specify the replaced and the new base classes in a `#pragma oochangebase` directive located immediately before the modified class definition—for example:

   ```
   #pragma oochangebase X -> Y// Base class changed from X to Y
   class B : public A, Y { ...// Y is derived from X
   ```
3. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` and `-upgrade` options. **Note:** You must process the definitions of the affected persistence-capable class, its base classes (including the new one), and any derived classes.
4. Write the modified schema to a file by running the `ooschemadump` tool.
5. Create and run an upgrade application against the federated database. No other application can access the affected objects until the upgrade application has completed.
6. Remove the `#pragma oochangebase` directive.
7. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example moves class B down in the inheritance graph by replacing its parent class X with the sibling class Y that is derived from X. This operation inserts class Y into the inheritance graph of B, making class X an indirect base class of B.

After the base list of class B is changed, the definitions of classes A, B, X, and Y are processed with the `-evolve` and `-upgrade` options. An upgrade application is required to convert affected objects and release them from upgrade protection.




```
// Original DDL file
class A : public ooObj {        // Persistence-capable class
public:
   ...
};

class B : public A, X {         // Persistence-capable class
public:
   ...
};

class X {                       // Non-persistence-capable class
public:
   ...
};

class Y : public X {            // Non-persistence-capable class
public:
   ...
};
```

```
// Modified DDL file
...                                 // No changes to class A
#pragma oochangebase X -> Y         // Remove after processing
class B : public A, Y {             // Persistence-capable class
public:
   ...
};
...                                 // No changes to classes X and Y
```

Removing a Non-Persistence-Capable Base Class

You perform a *conversion* operation when you remove a non-persistence-capable class (and all its base classes) from the inheritance graph of a persistence-capable class. This prunes part or all of a non-persistence-capable branch from the graph.

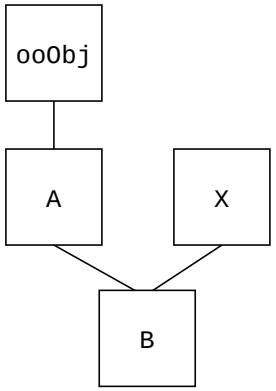
During object conversion, each affected object is resized to remove the formerly inherited data members. The data is lost when the space is reallocated.

To perform the operation:

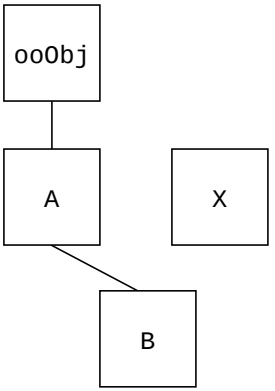
1. Remove the name of the obsolete non-persistence-capable base class from the base list of the appropriate class in the inheritance graph.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** You must process the definitions of the affected persistence-capable class, any remaining base classes, and any derived classes.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example removes the non-persistence-capable class `X` from the base list of the persistence-capable class `B` and processes the definitions of classes `A` and `B` with the `-evolve` option.

Before



After




```
// Original DDL file
#include x.h          // Defines non-persistence-capable class X
class A : public ooObj {        // Persistence-capable class
public:
   …
};

class B : public A, X {         // Persistence-capable class
public:
   …
};
```

```
// Modified DDL file
…                               // No changes to class A
class B : public A {            // X removed from base list
public:
   uint16 aUint16;
   float32 aFloat32;
};
```

Moving a Class to a Higher Inheritance Level

You perform a *conversion* operation when you move a persistence-capable class to a higher level in its inheritance graph. You accomplish this by changing the derivation of a class in the graph, replacing a direct base class with one or more ancestor classes in the same branch of the graph. You use this technique to remove an obsolete base class from the middle of an inheritance graph.

During object conversion, each affected object is resized to remove the members of the obsolete base class. The values of these members are lost when the space is reallocated; the values of members inherited from the remaining base classes are preserved.

An upgrade application is required to convert affected objects and release them from upgrade protection.

To perform the operation:

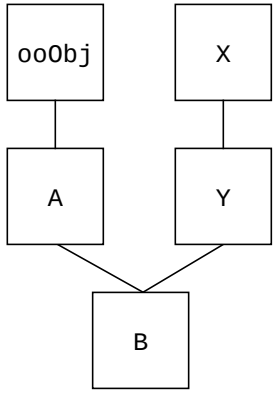
1. In the base list of the class to be raised in the inheritance graph, replace the obsolete base class with one or more of its own ancestor classes.
2. Specify the replaced and the new base classes in a `#pragma oochangebase` directive located immediately before the modified class definition—for example:

   ```
   #pragma oochangebase Y -> X// Base class changed from Y to X
   class B : public A, X { ...// Y is derived from X
   ```
3. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` and `-upgrade` options. **Note:** You must process the definitions of the affected persistence-capable class, its base classes, and any derived classes.
4. Write the modified schema to a file by running the `ooschemadump` tool.
5. Create and run an upgrade application against the federated database. No other application can access the affected objects until the upgrade application has completed.
6. Remove the `#pragma oochangebase` directive.
7. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

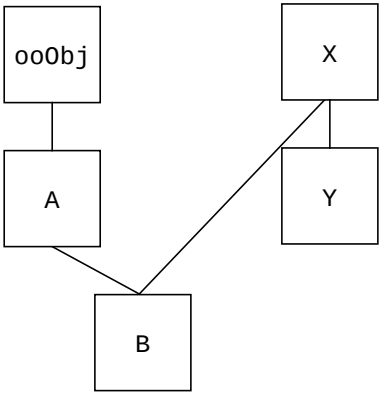
EXAMPLE This example moves class `B` up in the inheritance graph by replacing its base class `Y` with `Y`'s base class `X`. This operation removes class `Y` from the inheritance graph of `B`, making class `X` a direct base class of `B`.

After the base list of class `B` is changed, the definitions of classes `A`, `B`, and `X` are processed with the `-evolve` and `-upgrade` options. An upgrade application is required to convert affected objects and release them from upgrade protection.

Before



After




```
// Original DDL file
class A : public ooObj {        // Persistence-capable class
public:
   …
};

class B : public A, Y {         // Persistence-capable class
public:
   …
};

class X {                       // Non-persistence-capable class
public:
   …
};

class Y : public X {            // Non-persistence-capable class
public:
   …
};
```

```
// Modified DDL file
…                               // No changes to class A
#pragma oochangebase Y -> X     // Remove after processing
class B : public A, X {         // Persistence-capable class
public:
   …
};
…                               // No changes to classes X and Y
```

Changing the Order (Position) of a Base Class

Changing the order (position) of a base class is a *conversion* operation. Within the base list of a persistence-capable class, the persistence-capable base class must appear first (the leftmost class in the base list).

When an affected object is converted, the inherited members are reordered to match the reordered base list.

To perform the operation:

1. Reorder the base classes in the base list of the desired class definition.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option. **Note:** A warning message occurs if this is the only conversion change on the class.
3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example changes the order of the base class `Z` in class `C`.

```
// Original DDL file
#include xyz.h  // Defines non-persistence-capable classes X, Z
class C : public ooObj, X, Z{      // Persistence-capable class
public:
   …
};
```

```
// Modified DDL file
#include xyz.h  // Defines non-persistence-capable classes X, Z
class C : public ooObj, Z, X{      // Persistence-capable class
public:
   …
};
```

Changing the Access Control of a Base Class

Changing the access control of a base class is a *non-conversion* operation.

To perform the operation:

1. Change the access control of the base class in the desired class definition.
2. Process the appropriate DDL file(s) using the DDL processor with the `-evolve` option.

3. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

EXAMPLE This example changes the access control of a base class `Z` in class `C`.

```
// Original DDL file
#include xyz.h  // Defines non-persistence-capable classes X, Z
class C : public ooObj, X, Z{      // Persistence-capable class
public:
   ...
};
```

```
// Modified DDL file
#include xyz.h  // Defines non-persistence-capable classes X, Z
class C : public ooObj, X, private Z{    // Changed access
public:
   ...
};
```

Adding Persistence

You perform at least one *conversion* operation when you add persistence to a non-persistence-capable class that is already in the schema (for example, because it is inherited by or embedded in a persistence-capable class). This operation involves deleting the non-persistence-capable class from the schema and then reintroducing it as a persistence-capable class.

To perform the operation:

1. Follow the steps of “Deleting a Class” on page 126 to delete the non-persistence-capable class from the schema.
2. Add the class definition to a DDL file as a persistence-capable class (that is, with `ooObj` as a direct or indirect base class).
3. Process the DDL file (you do not need to specify the `-evolve` option).
4. Write the modified schema to a file by running the `ooschemadump` tool.
5. Modify existing applications as necessary and rebuild them with the newly generated header and implementation files.

Removing Persistence

You perform at least one *conversion* operation when you remove persistence from a persistence-capable class—for example, before using that class as a base class or embedded data type for another persistence-capable class.

This operation involves deleting the class from the schema and then reintroducing it as a non-persistence-capable class.

To perform the operation:

1. Follow the steps of "Deleting a Class" on page 126 to delete the persistence-capable class from the schema.
2. Add the class definition to a DDL file or C++ header file as a non-persistence-capable class (that is, without `ooObj` as a direct or indirect base class).
3. If you are adding the non-persistence-capable class as a base class or embedded data type of a persistence-capable class:
   a. Process the appropriate DDL file(s). **Note:** If the persistence-capable class already exists in the schema, specify the `-evolve` option to the DDL processor.
   b. Write the modified schema to a file by running the `ooschemadump` tool.
4. Modify existing applications as necessary and rebuild them with the most recently generated header and implementation files.

Restructuring Classes

Splitting a Class into Two Associated Classes

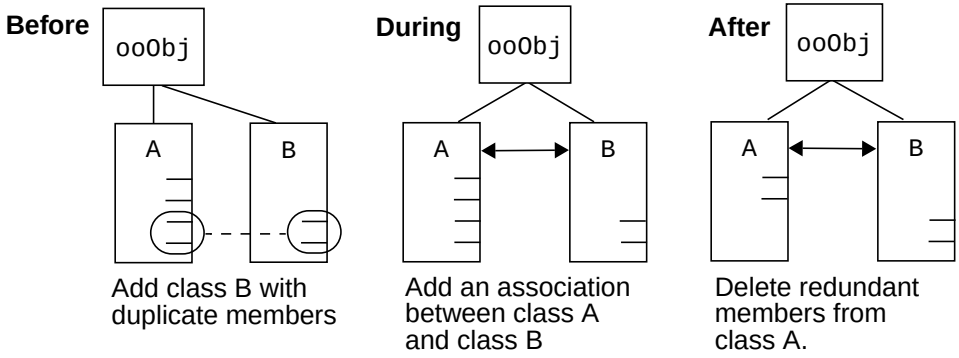
Splitting a persistence-capable class into two associated classes requires a combination of *conversion* operations. The split class keeps a subset of its original data members, while the other data members are, in effect, transferred to a new, associated class.

This technique adds a class to the schema and then evolves the schema twice. After the first cycle, a conversion application sets the necessary associations and transfers values from each object of the split class to the newly associated object.

To perform the operation:

1. Add a new persistence-capable class that contains a duplicate of each data member to be transferred from the original class (the class to be split).
2. Process the DDL file (you do not need to specify the `-evolve` option).
3. Follow steps 1 through 3 in "Adding an Association" on page 106 to add the desired association between the original class and the new class.

4. Using the header and implementation files generated in step 3, create and run a conversion application that iterates over each object of the original class, and, for each such object:
   a. Creates an object of the new class (or finds an object created earlier in the iteration).
   b. Sets each data member of the new object with the corresponding value from the existing object.
   c. Sets the association between the new object and the existing object.
5. Follow the steps in "Deleting a Data Member" on page 108 to delete the redundant data members from the original class.



Splitting a Class to Form a New Base Class

Splitting a persistence-capable class to form a new base class requires a combination of *conversion* operations. The split class keeps a subset of its original data members, while the other data members are, in effect, transferred to (and therefore inherited from) the new base class.

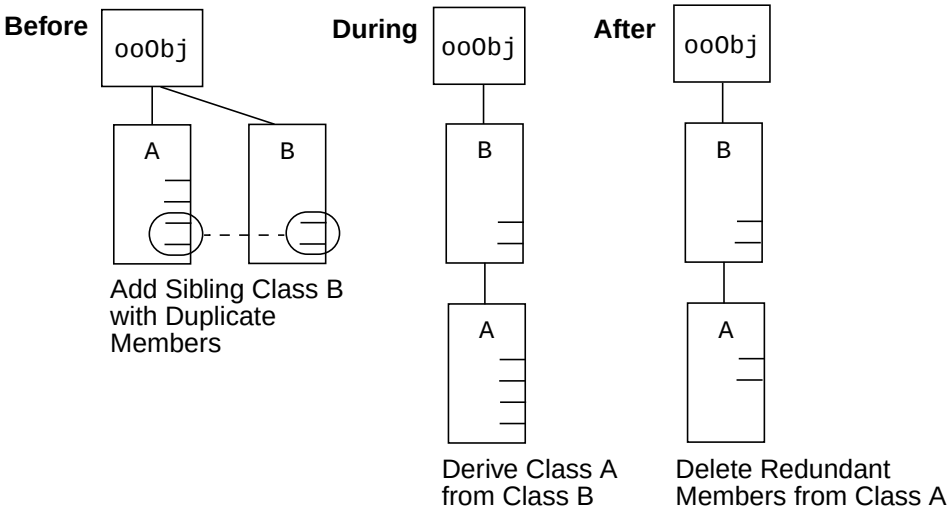
This technique adds a class to the schema and then evolves the schema twice. After the first cycle, a conversion application transfers values within the split class from the existing (direct) data members to the new (inherited) data members.

To perform the operation:

1. Add a new persistence-capable class that contains a duplicate of each data member to be transferred from the original class (the class to be split). Add the new class as a sibling of the original class.
2. Process the DDL file (you do not need to specify the `-evolve` option).
3. Follow steps 1 through 6 in "Moving a Class to a Lower Inheritance Level" on page 134 to derive the original class from the new sibling class.
4. Using the header and implementation files generated in step 3, create and run a conversion application that iterates over each object of the original class and

sets each new inherited data member with the value of the corresponding original member.

5. Follow the steps in “Deleting a Data Member” on page 108 to delete the redundant data members from the original class.



Merging Two Associated Classes

Merging two associated persistence-capable classes requires a combination of *conversion* operations. The resulting class contains its original data members plus the data members transferred from a formerly associated class.

This technique evolves the schema twice. After the first cycle, a conversion application transfers values into each affected object from the associated object of the class being merged.

To perform the operation:

1. Decide which of the two associated classes is to acquire the members of the other.
2. Change the class you chose in step 1 by adding duplicates of the data members of the associated class. For steps, see “Adding a Data Member” on page 104.
3. Using the header and implementation files generated in step 2, create and run a conversion application that iterates over each object of the changed class and sets each new data member with the corresponding value from the associated object.
4. Follow the steps in “Deleting a Class” on page 126 to delete the associated class along with the remaining class’s association to it.

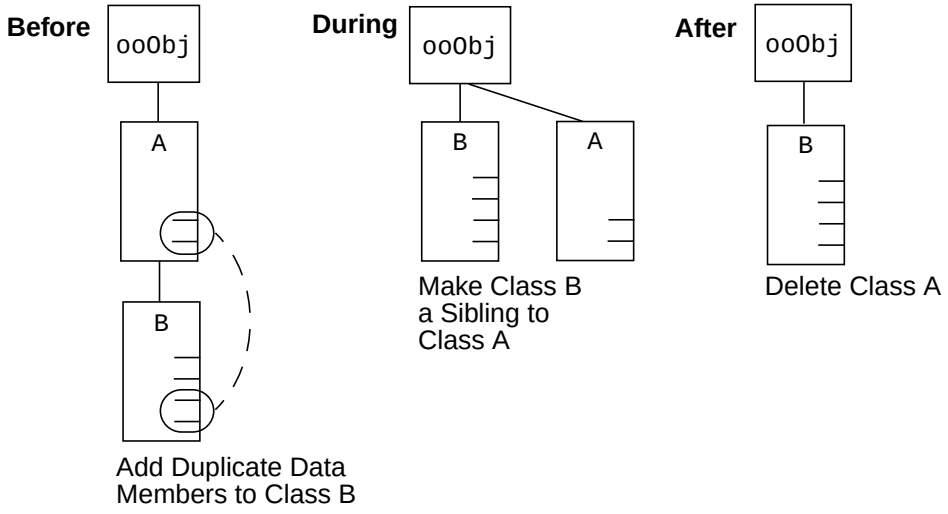
Merging a Base Class Into a Derived Class

Merging a persistence-capable base class into a derived class requires a combination of *conversion* operations. The resulting class contains its original data members plus the data members transferred from the former base class.

This technique evolves the schema three times. After the first cycle, a conversion application transfers values within the derived class from the existing inherited data members to the new direct data members.

To perform the operation:

1. Change the derived class by adding duplicates of the data members of the base class to be merged. For steps, see "Adding a Data Member" on page 104.
2. Using the header and implementation files generated in step 1, create and run a conversion application that iterates over each object of the changed class and sets each new data member with the value of the corresponding inherited data member.
3. Follow steps 1 through 6 in "Moving a Class to a Higher Inheritance Level" on page 138 to move the derived class up in the inheritance hierarchy, making the base class a leaf class.
4. Follow the steps in "Deleting a Class" on page 126 to delete the base class.



Distributing Schema Changes

You normally perform schema-evolution operations and test any required object conversion mechanisms on federated databases at your development site. When you are ready to release the evolved schema, you distribute the changes to your end-user sites, where you or your end users reproduce the schema-evolution operation in the deployed federated databases and deploy the new or updated applications.

You transfer the changed schema to your end users by using the `ooschemadump` and `ooschemaupgrade` tools (see the Objectivity / DB administration book). These tools enable you to apply schema changes to deployed federated databases without disclosing the schema to end users and without requiring your end users to run the DDL processor at their sites.

WARNING You can transfer schema changes only if the schema of the deployed federated database is identical to the schema that existed in your development federated database before you performed schema evolution.

Preparing for Distribution

You prepare for the eventual distribution of schema changes during every schema-evolution operation you perform on a development federation database—that is, whenever you:

- Run the `ooschemadump` tool after a schema-evolution cycle to write the modified schema to an output file.
- Build and run any conversion application or upgrade application required by the operation.

Simple operations generally produce just a single output file; operations with multiple cycles should result in a series of output files with corresponding conversion or upgrade applications. Because each output file in a series captures a particular intermediate state of the schema, you must adopt a convention for indicating the order in which they were generated.

When schema-evolution operations are complete, you prepare for distribution by:

1. Working with your end users to determine whether the deployed federated database still contains affected objects left over from any prior schema evolution. If necessary, you should build a preprocessing conversion application to bring these objects up-to-date.

2. Assembling a distribution package consisting of the files to be used for reproducing schema operations in each deployed federated database:
   - The output file(s) you generated with `ooschemadump`. If you distribute multiple files, you must indicate their order (the order in which they were generated).

     Warning: Advise your end users *not* to modify any output file. To do so will result in serious unpredictable errors when the file is used.
   - The `ooschemaupgrade` tool.
   - Any preprocessing, conversion, or upgrade applications you developed.
   - The applications that comprise your product, modified and rebuilt to match the evolved schema.
   - *Optional.* A shell script or batch file that runs the various tools and applications in the order indicated in "Reproducing a Schema Operation" on page 147.
3. Working with your end users to develop a plan for deploying the new or rebuilt applications.

Reproducing a Schema Operation

When your end users receive the distribution package you prepared, they must reproduce each schema-evolution operation on their deployed federated database by performing the following steps:

1. Shut down existing applications according to the plan for deploying updated applications. If any existing application is to continue running, arrange to minimize its impact on any conversion or upgrade application that will run concurrently—for example, by performing the remaining steps during off-peak hours.
2. Back up the deployed federated database—for example, using the `oobackup` tool (see the Objectivity / DB administration book).
3. Run any preprocessing application provided in the distribution package.
4. For each `ooschemadump` output file in the specified order:
   a. Run the `ooschemaupgrade` tool to load that file into the deployed federated database.
   b. Run any corresponding conversion or upgrade application.

      Warning: If any affected object is locked by an existing application running concurrently, an upgrade application will fail without taking action.
5. Start the new or rebuilt applications according to the plan for deploying updated applications.

Deploying Updated Applications

The steps for reproducing schema-evolution operations on a deployed federated database must take into account a plan for deploying the updated applications (the new or modified applications that have been compiled with the changed definitions). From a schema-evolution perspective, the safest plan is to simply shut down all existing deployed applications, reproduce the schema-evolution operations on the federated database, and then start the updated applications. However, such a plan does not support typical availability requirements.

You can increase availability by allowing any number of existing applications to continue while schema evolution operations are being reproduced. When these operations are complete, you can retire these applications and switch to their updated counterparts as convenient. New applications can be started at any time after the schema is changed.

WARNING *Do not* restart an existing (non-updated) application after the schema is changed; see “Impact on Existing Applications” on page 93.

When an existing application is to continue during schema-evolution operations, you should be aware that the continuing application:

- Keeps using the pre-evolution schema with which it was initialized, and therefore does not trigger the conversion of any affected object it accesses.
- Can run concurrently with any conversion application that sets values in affected objects. However, depending on the conversion application’s behavior, values may be lost if an affected object is locked by the continuing application.
- Can run concurrently with an upgrade application. However, the upgrade application will fail without releasing upgrade protection if an affected object is locked by the continuing application.
- Cannot find objects that have been converted (for example, by a concurrent conversion application). While this prevents data corruption resulting from mismatched schemas, it also limits the effectiveness of the continuing application.

You should arrange to minimize the impact of a continuing application on a concurrent conversion or upgrade application—for example, by reproducing the schema-evolution operation during off-peak hours or advising end users to avoid accessing the affected objects.

6

Class Versioning

By default, the DDL processor allows you to add new definitions to a schema, but prevents you from changing definitions that are already in the schema. If you need to modify an existing definition—for example, by renaming data members, altering the inheritance hierarchy, or changing data-member types—you can choose to create a new *version* of the class in the schema.

Creating a new class version allows you to preserve and access existing persistent objects (instances of the old version) while creating new objects from the new version. If, however, your goal is to convert all existing instances of the changed class to match the changed definition, you should use schema evolution instead of class versioning (see Chapter 5, “Schema Evolution”).

This chapter describes:

- General information about class versioning
- Creating a new version of a class
- Creating new versions of multiple interrelated classes

About Class Versions

A schema can contain multiple versions of a class, where each version stores a different definition for the class. For example, a class named `Widget` might have an initial version with three attributes, a second version with four attributes, a third version with different attribute types, and so on.

Because the versions of a class may have different definitions, they are essentially different types within the schema. Consequently, each version has a unique type number, and the instances of different versions of a class are as distinct as the instances of different classes. This allows you to maintain objects of multiple versions in the same federated database, and to access those objects using the member functions defined for the appropriate class version.

Within the schema, all versions of a given class share the same class name, to which an appropriate *version number* is appended. Thus, when you first add a class named `Widget` to a schema, the schema's name for the class is `Widget%1`; subsequent versions are `Widget%2`, `Widget%3`, and so on.

You can create multiple versions of any application-defined class in the schema. Thus, besides versioning persistence-capable classes, you can also version any non-persistence-capable class that is incorporated in a persistence-capable class—for example, a non-persistence-capable class used as an attribute type or base class of a persistence-capable class.

Class Versioning and Schema Evolution

Class versioning is fundamentally different from schema evolution:

- Class versioning stores changed definitions using separate but related types. Different versions of a class, along with their instances, can coexist indefinitely within a federated database.
- Schema evolution stores changed definitions using separate shapes (storage representations) within a single type. All old instances of a class are converted to the current shape.

Because class versioning creates separate versions of classes, you can use schema evolution on a versioned class, just like any other class.

Creating a New Version of a Class

In general, you create a new version of a class by:

1. Providing a "nickname" for the original version of the class and adding it to the schema. Once the new version is created, all applications will use this nickname to create or access instances of the original version.
2. Modifying the class definition as desired and adding the modified version to the schema. All applications will obtain the modified version when they use the original class name.
3. Building (or rebuilding) applications with the header and implementation files generated for *both* the original and new versions.

These steps are described in the following subsections. A few additional considerations apply when versioning a class that is interrelated with other classes—for example, through inheritance or associations; see "Versioning Interrelated Classes" on page 154.

Providing a Nickname for the Original Class

Multiple versions of a class are distinguished in the schema by a version number appended to the class name—for example, the versions of class `Widget` are `Widget%1`, `Widget%2`, and so on. However, C++ applications have no knowledge of the version numbers assigned within the schema and must rely instead on class names. Therefore, before you create a new version for a class, you must give the original version a nickname so that new and rebuilt C++ applications can distinguish the two versions.

To create a nickname for the original version of a class:

1. Choose the desired nickname.
2. Make a copy of the DDL file containing the original class definition. Choose a filename to indicate that this copy will contain the nicknamed version.
3. In the DDL file you created in step 2, insert a `#pragma ooclassname` directive such as the following to specify the nickname:

   `#pragma ooclassname` *`oldClassName nickName`*

   You place a `#pragma ooclassname` directive after the definition of the class to which it applies. Do *not* make any other changes to the class definition in the DDL file—not even to the class name.
4. Process the DDL file to generate a new set of header and implementation files. Use the DDL processor options you normally would; do not specify any special option for versioning.

The DDL processor generates header and implementation files containing the usual application-defined and generated definitions, with one important difference: *`nickName`* is substituted wherever the original class name would normally appear. That is:

- The primary header file contains the renamed class definition.
- The references header file contains definitions for parameterized classes like `ooRef(`*`nickName`*`)`, `ooHandle(`*`nickName`*`)`, and so on.
- The implementation file contains registration code that binds *`nickName`* to the schema-assigned type number for the original version.

EXAMPLE Assume that you plan to version a class named `Widget`, which exists in the schema of the `partsData` federated database, and that class `Widget` was originally defined in the DDL file `widget.ddl`.

```
// Original DDL file: widget.ddl
class Widget : public ooObj {
public:
   float partDiameter;
   float threadSpacing;
   void partCount();
};
```

In a copied DDL file called `oldWidget.ddl`, you insert a `#pragma ooclassname` directive specifying the nickname `OldWidget`. You make no other modifications to the class definition.

```
// DDL file copy: oldWidget.ddl
class Widget : public ooObj {
public:
   float partDiameter;
   float threadSpacing;
   void partCount();
};
#pragma ooclassname Widget OldWidget
```

You then process `oldWidget.ddl` as you normally would, which generates the files `oldWidget.h`, `oldWidget_ref.h`, and `oldWidget_ddl.cxx`:

```
ooddlx oldWidget.ddl partsData.boot
```

The generated files contain definitions and implementations for classes `OldWidget`, `ooRef(OldWidget)`, and so on. When you compile and link these files in an application, it must use the name `OldWidget` to obtain the original version of `Widget` (that is, the class internally named `Widget%1`).

Creating the New Version

After you have nicknamed the original version of a class, you create the new version and add it to the schema:

1. Make a new copy of the DDL file containing the original class definition. Choose a filename to indicate that this copy will contain the new version.
2. In the DDL file you created in step 1, modify the class definition as desired. Do not change the class name.

3. Validate the modified class definition by running the DDL processor with the `-nochange` option. If at least one comparison error is reported on the class you changed, the next step will produce a new version of the class.
4. Create the new version in the schema by running the DDL processor with the `-version` option. A new set of header and implementation files is generated.

The generated files preserve the original class name, which now refers to the modified definition. The implementation file contains registration code that binds the original class name to the new type number assigned to the new definition.

EXAMPLE Continuing the example from the previous section, you copy the original DDL file `widget.ddl` to a new file `newWidget.ddl`, in which you modify the class definition of `Widget` by adding a new attribute:

```
// DDL file copy: newWidget.ddl
class Widget : public ooObj {
public:
   float partDiameter;
   float threadSpacing;
   void partCount();
   float partLength;     // Added attribute
};
```

You process `newWidget.ddl` with the `-nochange` option, which compares the definition in the DDL file to the definition in the schema, and verifies that changes have been made:

```
ooddlx -nochange newWidget.ddl partsData.boot
```

You then process `newWidget.ddl` with the `-version` option, which creates a new version of `Widget` and generates the files `newWidget.h`, `newWidget_ref.h`, and `newWidget_ddl.cxx`:

```
ooddlx -version newWidget.ddl partsData.boot
```

The generated files contain new definitions and implementations for the classes `Widget`, `ooRef(Widget)`, and so on. When you compile and link these files in an application, all occurrences of the class name `Widget` obtain the new version (that is, the class internally named `Widget%2`).

Using the Old and New Versions

As a result of nicknaming the old version of a class and creating a new version, you now have two new sets of generated header and implementation files. You must compile and link *both* sets of generated files in any application that needs to access instances of both versions. If errors report duplicate class definitions, you should make sure that the old version was nicknamed properly and that you are using the files generated from the nicknaming process.

Note that you may continue to use old applications (applications compiled and linked with the files that were generated from the original DDL file prior to versioning). Of course, such applications can access only instances of the original class.

If you want to use old application code to access the new versions, you should rebuild the application, using the new generated files in place of the original ones.

Versioning Interrelated Classes

When you are planning to version a class, you must also plan to version any other classes that incorporate (depend on) the class being versioned. This includes any class that:

- Is derived from the class being versioned.
- Has an association to the class being versioned.
- Contains an object reference to the class being versioned.

You must create a new version of each dependent class, even if no changes will be made to the dependent class's definition.

You version a class and its dependents by:

1. Putting the class definitions in the same DDL file, if necessary.
2. Providing nicknames for each of the classes.
3. Making the desired modifications and creating the new versions.

These steps are described in the following subsections.

Preparing a Suitable DDL File

For purposes of versioning, a class and its dependents must be processed in the same DDL file. Thus, the same DDL file must contain the definitions of the class to be modified, plus any additional classes that derive from, have associations to, or have object references to that class.

If your original DDL file already contains all the necessary class definitions, you can simply use copies of the file for nicknaming and versioning. If, however, the

necessary class definitions were originally split into individual DDL files, you should combine these definitions into one DDL file that you can then copy for nicknaming and versioning.

If, on the other hand, your original DDL file contains not only the class definitions necessary for versioning, but also definitions for additional unrelated classes, you should consider splitting out the unrelated definitions to a separate DDL file. (Unrelated non-persistence-capable classes can be moved to standard header (`.h`) files that can then be included in the appropriate DDL files.) You then process the DDL file of unrelated classes as you normally would to obtain a new set of generated header and implementation files, and use copies of the remaining DDL file for nicknaming and versioning. Isolating the classes to be versioned reduces the number of nicknamed classes that will result.

Nicknaming Multiple Classes in a DDL File

When you have a suitable DDL file containing just the classes to be versioned and their dependents, you:

1. Make a copy of the DDL file. Choose a filename to indicate that this copy will define the nicknames.
2. Insert a `#pragma ooclassname` directive after *every* class defined in the DDL file. You must define a nickname for every class in the file, even those dependent classes whose definitions will not be changed.
3. Process the DDL file to generate a set of header and implementation files that use the nicknames.

EXAMPLE Assume the schema of the `Example` federated database contains classes `one`, `two`, and `three`, and that you plan to modify the definitions of classes `one` and `two`. Because class `three` has associations to classes `one` and `two`, you must version all of the classes.

Accordingly, you copy their definitions from their original separate DDL files into a single new file called `older.ddl`. You add three `#pragma ooclassname` directives to provide a nickname for each class in `older.ddl`:

```
// DDL file: older.ddl
class one : public ooObj {
   public:
      int i1;
      ooRef(two) toTwo <-> toOne;
};
#pragma ooclassname one old_one
```

```
class two : public ooObj {
   public:
      int i2;
      ooRef(one) toOne <-> toTwo;
};
#pragma ooclassname two old_two
class three : public ooObj {
   public:
      int i3;
      ooRef(two) toTwo : copy(delete); // Unidirectional
      ooRef(one) toOne : copy(delete); // Unidirectional
};
#pragma ooclassname three old_three
```

You then process `older.ddl` as you normally would to generate the files `older.h`, `older_ref.h`, and `older_ddl.cxx`:

```
ooddlx older.ddl Example.boot
```

The generated files contain definitions and implementations for classes `old_one`, `old_two`, `old_three`, `ooRef(old_one)`, `ooRef(old_two)`, and so on. Applications that compile and link with these files must use these names to access the original versions of classes `one`, `two`, and `three`.

Note that the member functions generated for associations among these classes use nicknames and therefore associate instances of the original versions. For example, the `toTwo` member function generated in class `old_three` returns a handle for the original version of class `two`—that is, a `ooHandle(old_two)` instead of a `ooHandle(two)`, which would reference the new version.

Versioning Multiple Classes in a DDL File

After you provide nicknames for the classes to be versioned and their dependents, you:

1. Make a new copy of the DDL file containing the classes to be versioned and their dependents. Choose a filename to indicate that this copy will contain the new versions.
2. In the DDL file, modify one or more class definitions, as desired. Do not change any class names.
3. Validate the modified class definitions by running the DDL processor with the `-nochange` option. If at least one comparison error is reported on each class you changed, the next step will produce a new version of the changed classes.

4. Create the new versions in the schema by running the DDL processor with the `-version` option. A new set of header and implementation files is generated.

EXAMPLE Continuing the example from the previous section, you copy the definitions of classes `one`, `two`, and `three` from their original separate DDL files into a single new file called `newer.ddl`, where you make the following definition changes:

- Add a new data member to class `one`.
- Change a data-member type in class `two`.
- Change the cardinality and behavior specifier of the association between classes `one` and `two`.

No changes are made to the definition of class `three`; however, a new version will be created for it automatically because of its associations to classes `one` and `two`.

```
// DDL file newer.ddl
class one : public ooObj {
   public:
      int i1;
      int j1;               // New data member
      ooRef(two) toTwo[] <-> toOne : delete(propagate);
};

class two : public ooObj {
   public:
      double d2;            // Changed data-member type
      // Changed association cardinality and behavior
      ooRef(one) toOne <-> toTwo[] : delete(propagate);
};

class three : public ooObj { // **Definition not changed**
   public:
      int i3;
      ooRef(two) toTwo : copy(delete);
      ooRef(one) toOne : copy(delete);
};
```

You process `newer.ddl` with the `-nochange` option, which compares the definitions in the DDL file to the corresponding definitions in the schema, and verifies that changes have been made:

```
ooddlx -nochange newer.ddl Example.boot
```

You then process `newer.ddl` with the `-version` option, which creates a new version of classes `one`, `two`, and `three` in the schema, and generates the files `newer.h`, `newer_ref.h`, and `newer_ddl.cxx`:

```
ooddlx -version newer.ddl Example.boot
```

7

Partitioning a Data Model

For project management purposes, you may choose to partition a complex data model into multiple distinct yet interrelated domains. For example, a company that manufactures engines might require different applications to track engineering specifications, costs, and manufacturing processes. Such applications could be developed separately but must share interrelated data.

You can represent a partitioned data model by creating multiple schemas in a single federated database. For example, the engine company's applications might access a single federated database that has one schema of engineering data types, another schema of accounting data types, and a third schema of manufacturing data types.

This chapter describes:

- General information about multiple schemas
- How to add all the definitions in a DDL file to a single named schema
- How to switch among multiple schemas when processing a single DDL file

About Multiple Schemas

Every federated database is created with a single initial schema, called the *default schema*. At creation, the default schema contains type information for the persistence-capable classes and types defined by Objectivity/C++. When you process DDL files containing application-specific class definitions, these definitions are normally added to the default schema.

You can create additional schemas by specifying an appropriate DDL processor option or by specifying an appropriate `#pragma` directive in a DDL file. In either case, you must specify a unique name for the schema (the implicit name for the default schema is *). Schemas are distinguished only by their names; there is no implicit or explicit hierarchy of schemas.

When you process a DDL file containing a new class definition, you specify the schema to which to add that class definition. Each schema therefore describes a subset of the class definitions that have been processed into the federated database. Taken together, multiple schemas can express the organization of your classes. Multiple schemas do not, however, affect how classes are used by applications:

- Associations can exist between instances of two persistence-capable classes that are processed into different schemas.
- Any application can `#include` the generated header file for any persistence-capable class, regardless of the schema into which the class was processed; consequently, any including application can create or access instances of a class belonging to any schema.
- Instances of any persistence-capable class can be created in any container or database, regardless of the schema into which the class was processed.

WARNING If your application is to interoperate with Objectivity for Java, Objectivity/Smalltalk, or Objectivity/SQL++ applications, you should not use multiple schemas. Only Objectivity/C++ applications can access classes that reside in any schema other than the default schema `*`.

To avoid confusion, you should process each class definition into one and only one schema, so that every class name is unique within a particular federated database. Although the DDL processor allows multiple definitions of the same name to be processed into different schemas (where they are distinguished by unique type numbers), only one of these definitions can be compiled into any given application. Multiple schemas do *not* correspond to C++ name spaces, which enable applications to define and access duplicate type names through qualification.

For example, if a persistence-capable class `Agency` has been processed into each of three named schemas, only one of the resulting sets of generated header and implementation files can be compiled into an application (otherwise a C++ compiler error would result from a multiply defined name). Therefore, all of the application's references to class `Agency` are bound to the unique compiled definition; the application has no knowledge of either of the other two definitions.

Each schema in a federated database has a unique schema number, which is the basis for calculating the type numbers of the definitions in the schema. Consequently, different schemas define different ranges of type numbers. This enables you to use the `ooschemadump` and `ooschemaupgrade` tools to propagate

schemas between federated databases without overwriting definitions in other schemas. Note, however, that:

- All such schemas must originate in the same federated database to guarantee unique schema numbers.
- Schemas do not provide separate ranges for the numbers assigned to associations.

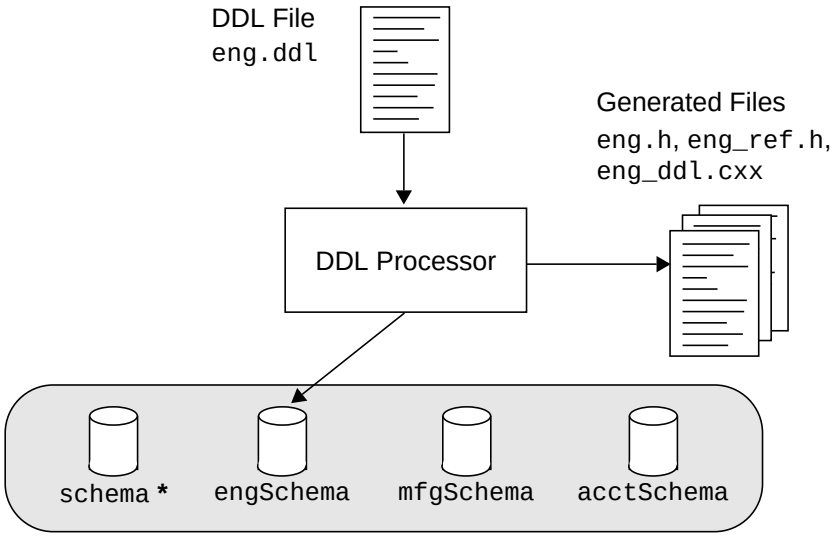
Adding Definitions to a Named Schema

You add class definitions to a named schema by invoking the DDL processor with the `-schema` option to specify the schema's name. When you use the `-schema` option, all of the classes that are defined or referenced in the DDL file must belong to the same specified schema. Omitting the `-schema` option processes definitions into the default schema `*`.

If a schema with the specified name does not yet exist in the federated database, it is created.

EXAMPLE The following command adds the classes defined in `eng.ddl` to the schema named `engSchema` in the federated database specified by `DBInc.boot`:

```
ooddlx -schema engSchema eng.ddl DBInc.boot
```



Federated Database `DBInc`

Switching Between Multiple Schemas

A single DDL file can define or reference classes that belong to different schemas. This situation typically arises when a class being added to one schema references a second class that already belongs to a different schema.

When processing such a file, the DDL processor must switch among the relevant schemas. To direct the DDL processor to switch to a schema called *schemaName* while processing a DDL file, you insert the following `#pragma ooschema` directive in the file:

`#pragma ooschema` *schemaName*

where

schemaName	Name of the schema that is to contain (or that already contains) the next persistence-capable class to be processed. Specify * to indicate the default schema. Omit this parameter to return to the schema that you specified when you invoked the DDL processor.

When a DDL file defines or references multiple classes belonging to different schemas, you prepare that file for processing as follows:

1. Identify the schema that contains (or will contain) the majority of the classes defined or referenced in the file. You will set this schema with the `-schema` option when you invoke the DDL processor.
2. Edit the DDL file to find each forward reference or definition for a class belonging to a different schema:
   a. Before each such forward reference or class definition, insert a `#pragma ooschema` directive with a parameter that specifies the desired schema.
   b. After each such forward reference or class definition, insert a `#pragma ooschema` directive with no parameter to switch back to the schema set by the DDL processor.

EXAMPLE The DDL file `eng.ddl` contains a definition for class `Rod`, which is to be processed into a schema named `engSchema`. Class `Rod` contains an association to class `Gear`, whose definition has already been processed (through the DDL file `mfg.ddl`) into a schema named `mfgSchema`.

The DDL file contains `#pragma ooschema` directives around the forward reference to class `Gear`. Assuming that the DDL processor specifies `engSchema` through the `-schema` option, the `#pragma` directives switch the DDL processor from `engSchema` to `mfgSchema` and back.

```
// DDL file eng.ddl
#pragma ooschema mfgSchema
class Gear;
#pragma ooclassref Gear <mfg_ref.h>
#pragma ooschema

class Rod : public ooObj {
public:
   float diameter;
   float length;
   ooRef(Gear) toGear <-> toRod;
};
```

The DDL processor is invoked as follows:

```
ooddlx -schema engSchema eng.ddl DBInc.boot
```

Figure 7-1 shows files `eng.ddl` and `mfg.ddl` processed into in their respective schemas, where `eng.ddl` contains class `Rod` and `mfg.ddl` contains class `Gear`.

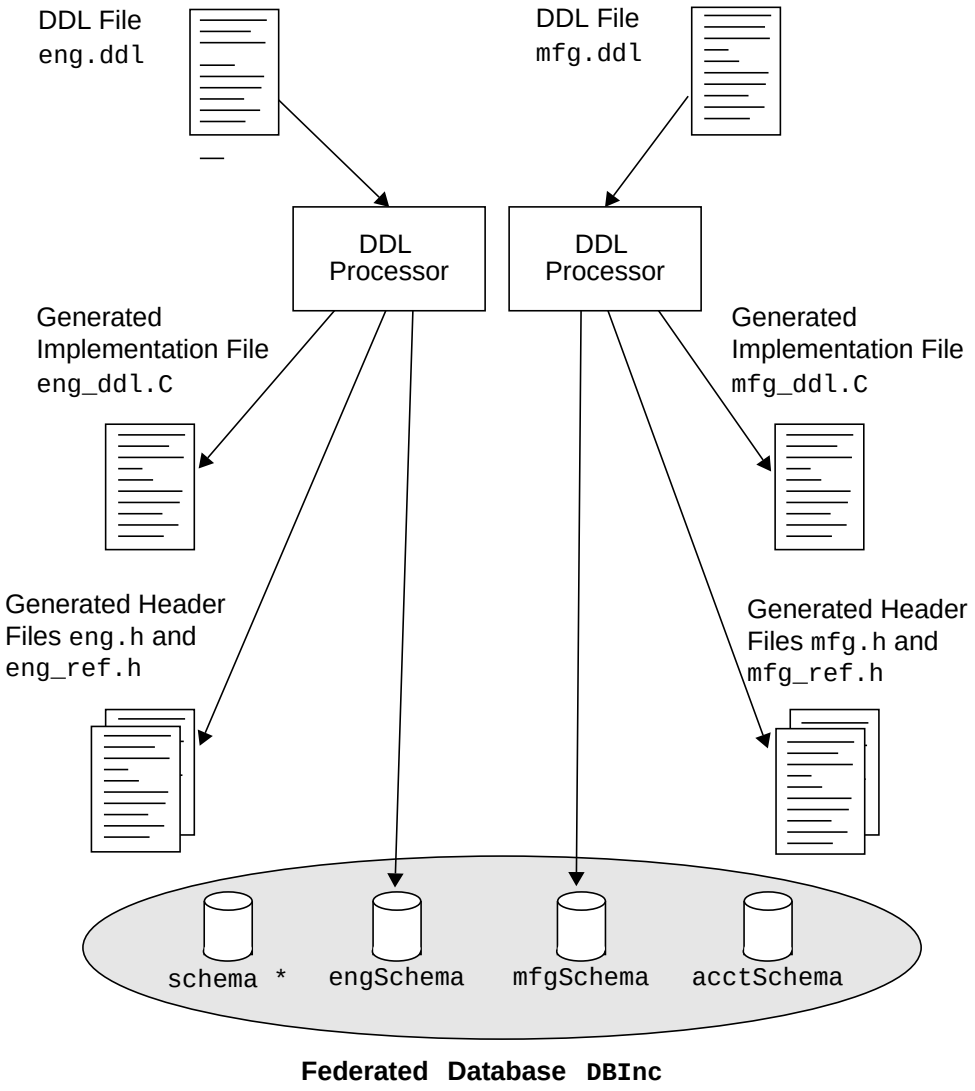


Figure 7-1 Class Definitions Processed into Two Different Schemas

8

Data Model Tuning

This chapter presents data modeling and design suggestions to help tune the performance of Objectivity/DB federated databases. Following these guidelines, you may be able to improve your federated database in terms of:

- Federated database size
- Speed of database applications

For additional suggestions, see the chapter on monitoring and tuning performance in the Objectivity/C++ programmer's guide.

Tuning for Federated Database Size

The following actions may help reduce the size of an Objectivity/DB federated database.

Use Inline Associations

You can reduce the storage required for associations by using inline associations (page 62) instead of non-inline associations. Note, however, that inline to-one associations are embedded within objects, so they take up space even when they are not used.

When all associated objects reside in the same container as the associating object, you can further reduce federated database size by using *short* inline associations instead of *long* inline association. Whereas long inline associations use standard OIDs to refer to associated objects, short inline associations use short OIDs, which do not contain database and container information for the associated objects.

Store Data Efficiently

When you use Objectivity/C++ primitive data types (page 42), be sure to choose the data type that requires the least amount of storage necessary for each element

of your data. For example, use type `uint8` instead of type `uint16` for a small integer value.

You can also optimize storage through the order in which you define your data members. Declaring larger data types first can optimize the packing of your data, which can make a significant difference when you have many objects.

EXAMPLE For example, because most compilers require data of type `float64` to start on an 8-byte boundary, and data of type `uint16` to start on a 2-byte boundary, the following declaration will produce the packing shown in Figure 8-1, occupying 32 bytes and wasting 12 bytes of storage in the shaded areas:

```
class myclass {
   uint16 a;
   float64 b;
   uint16 c;
   float64 d;
};
```

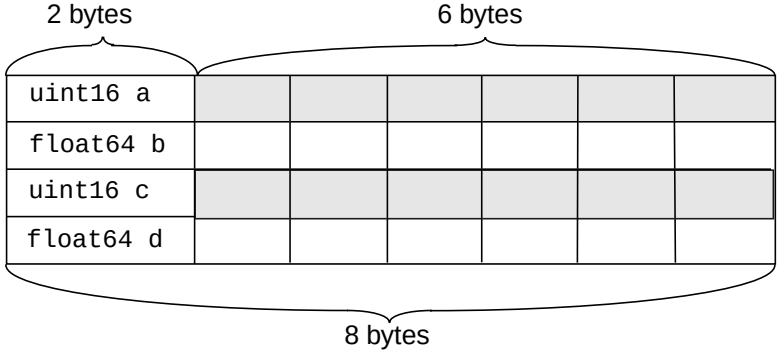


Figure 8-1 Data Packing When Alternating Long and Short Types

---

The following declaration will produce the packing shown in Figure 8-2, occupying 24 bytes and wasting only the 4 bytes of storage shown in the shaded area:

```
class myclass {
   float64 b;
   float64 d;
   uint16 a;
   uint16 c;
};
```

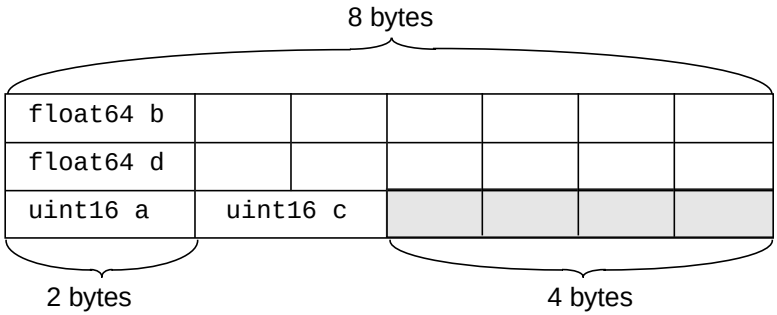


Figure 8-2 Data Packing When Declaring Long Types First

Tuning for Application Speed

To achieve optimal speed for an application, you must minimize I/O, networking overhead, and CPU time. The following actions may significantly improve the speed of your application by increasing the efficiency with which it can access and manipulate objects in your federated database.

Use Inline Associations

You can speed up association traversal by using inline associations (page 62) instead of non-inline associations. A one-to-one inline association can be traversed very quickly because it is embedded in the object. A one-to-many inline association can also be traced quickly because the application needs to traverse only the associated objects for that association, instead of all the associations on the object.

However, inline associations are less efficient when you need to add more associations—whereas adding a non-inline association to a class has no effect on existing instances of the class, adding an inline association requires the conversion of existing objects to the new representation.

A

Tools

This appendix describes the Objectivity/C++ DDL processor.

Tool Names

The names of tools are the same on Windows and UNIX, with the exception that, on Windows, the filenames of the executables have an extension (`.exe`) that is not required on UNIX.

Tool Options and Arguments

The command-line syntax for most tools includes either or both of the following:

- *Options*, which modify the way the tool works. Syntactically, options are characters prefixed with a hyphen and set off with spaces—for example, `-help`. Some options are followed by *values*—for example, `-host` *`hostName`*.
- *Arguments*, which specify values directly to the tool. For example, many tools accept a *`bootFilePath`* argument.

When specifying options for an Objectivity/DB tool, you need to type only as many letters of the option as are necessary to identify it uniquely. This is also true for the fixed values sometimes associated with a command name or option.

Most options and arguments to Objectivity/DB tools are case sensitive, and in most cases, options and arguments are lower case. Be sure to type options and arguments using the correct case.

ooddlx

Processes definitions in the specified DDL file and adds them to the schema in the specified federated database.

```
ooddlx | oocddl
     [-D…] [-E] [-I…] [-U…]
     [-c [-noc++] [-cheader_suffix suffix]
                [-ifdef_cheader variable]]
     [-c++_suffix suffix]
     [-header_suffix suffix]
     [-ifdef_header variable]
     [-ifdef_ref variable]
     [-include_header pathName]
     [-include_ref pathName]
     [-keep_typedefs]
     [-noanachronism]
     [-noincludeoo]
     [-noline]
     [-nooutput]
     [-noref]
     [-notitle]
     [-notouch]
     [-nowarn]
     [-ref_suffix suffix]
     [-schema name]
     [-standalone]
     [-storage_specifier specifier]
     [-validate]
     [-version | -evolve [-upgrade] | -nochange]
     [-help]
     classDefFile.ddl
     [bootFilePath]
```

Options

`-D`*varNameAndValue*

Same as the UNIX C compiler `-D` flag. Do not put any whitespace between `-D` and *varNameAndValue*.

`-E`

Same as the UNIX C compiler `-E` flag. You cannot combine this option with the `-nooutput` options nor with any option that is incompatible with `-nooutput`.

`-I`*pathName*

Same as the UNIX C compiler `-I` flag. Do not put any whitespace between `-I` and *pathName*. Included files are searched for in the `-I` options' directory arguments in the order in which the options appear on the command line.

`-U`*`varName`*

Same as the UNIX C compiler `-U` flag. Do not put any whitespace between `-U` and *`varName`*. All `-D` options are processed before all `-U` options.

`-c`

Produces C as well as C++ output files.

`-noc++`

Suppresses C++ output. This option requires the `-c` option. This option cannot be combined with the `-c++_suffix`, `-header_suffix`, `-ifdef_header`, `-ifdef_ref`, `-include_header`, `-include_ref`, or `-ref_suffix` options.

`-cheader_suffix` *`suffix`*

Uses *`suffix`* as the suffix of the C output header file instead of the default `_c.h`. This option requires the `-c` option.

`-ifdef_cheader` *`variable`*

Uses *`variable`* as the preprocessor variable in the `#ifndef` directive wrapped around the contents of the C output header file. If you omit this option, the variable is constructed from the file's name (*`module`*`_C_H`). This option requires the `-c` option.

`-c++_suffix` *`suffix`*

Uses *`suffix`* as the suffix of the C++ method implementation file. If you omit this option, the default suffix is `_ddl.cpp` for Windows platforms and `_ddl.C` for UNIX platforms.

`-header_suffix` *`suffix`*

Uses *`suffix`* as the suffix of the C++ generated header file. If you omit this option, the default suffix is `.h`.

`-ifdef_header` *`variable`*

Uses *`variable`* as the preprocessor variable in the `#ifndef` directive wrapped around the contents of the C++ generated header file. If you omit this option, the variable is constructed from the file's name (*`module`*`_H`).

`-ifdef_ref` *`variable`*

Uses *`variable`* as the preprocessor variable in the `#ifndef` directive wrapped around the contents of the C++ references header file. If you omit this option, the variable is constructed from the file's name (*`module`*`_REF_H`).

`-include_header` *`pathName`*

Uses *`pathName`* as the name of the C++ generated header file to `#include` in the C++ method implementation file. If you omit this option, the default is to use the name of the actual C++ generated header file.

`-include_ref` *`pathName`*

Uses *`pathName`* as the name of the C++ references header file to `#include` in the C++ generated header file. If you omit this option, the default is to use the name of the actual C++ references header file.

`-keep_typedefs`

Suppresses expansion of typedef names when recording class names in a schema. This option is useful for working around the limit on class-name length, particularly when the name of a template class instantiation exceeds the limit after typedef declarations have been expanded.

`-noanachronism`

Signals an error for each use of a feature provided only for backward compatibility. When you omit this option, such features are silently permitted.

`-noincludeoo`

Suppresses the `#include <oo.h>` directive in output files. You use this option to avoid multiple inclusion of the system file `oo.h` when using the output produced by the `ooddlx` command with the `-E` option as input to another invocation of `ooddlx`.

`-noline`

Suppresses `#line` directives in the C++ output.

`-nooutput`

Creates or updates the specified schema without generating any C++ or C output files. This is useful for creating a schema in a new federated database for an existing application. You cannot combine this option with the following options: `-c`, `-cheader_suffix`, `-c++_suffix`, `-header_suffix`, `-ifdef_cheader`, `-ifdef_header`, `-ifdef_ref`, `-include_header`, `-include_ref`, `-noline`, `-notouch`, or `-ref_suffix`.

`-noref`

Suppresses the C++ references header file. You cannot combine this option with the following options: `-ifdef_ref`, `-include_ref`, `-noc++`, `-nooutput`, or `-ref_suffix`. This option is for backward compatibility only.

`-notitle`

Suppresses the copyright notice and program title banner. Useful when invoking the tool from another tool or product.

`-notouch`

Prevents the DDL processor from rewriting any output file that already contains what would otherwise be placed in it.

`-nowarn`

Signals errors but not warnings. This option does not apply when the `-evolve` option is used.

`-ref_suffix` *`suffix`*

Uses *`suffix`* as the suffix of the C++ references header file. If you omit this option, the default suffix is `_ref.h`.

`-schema` *`name`*

Adds class definitions into the schema called *`name`*. If the named schema does not yet exist, it is created. If you omit this option, definitions are added into the default schema, which is called `*`.

`-standalone`

Nonconcurrent mode. Use this option if no lock server is running or to bypass a running lock server.

Warning: Corruption can occur if concurrent access to the federated database is attempted while any process is using this mode.

`-storage_specifier` *`specifier`*

Interprets *`specifier`* in the input as `__declspec(ddlexport)` and preserves it in the output unchanged. This option is incompatible with the `-nooutput` and `-noc++` options.

`-validate`

Validates but does not commit class definition changes in the schema. You cannot combine this option with the `-nochange` option.

`-version`

Versions the schema representation for each class that has a changed definition in the input DDL file. You cannot combine this option with the `-evolve` or `-nochange` options.

`-evolve`

Evolves the schema representation for each class that has a changed definition in the input DDL file. You cannot combine this option with the `-version` or `-nochange` options. Before you reuse any DDL files that you processed with this option, you must manually remove any pragma statements used for schema evolution.

`-upgrade`

Indicates that certain object conversion operations will be performed by an upgrade application. This option requires the `-evolve` option.

`-nochange`

Processes the specified DDL file without changing the schema:

- If the class definitions in the DDL file match those in the schema, the DDL processor simply regenerates the output header and implementation files.
- If the DDL file contains a new or changed class definition, the DDL processor signals an error and quits without regenerating the output files.

You cannot combine this option with the `-version` or `-evolve` option.

`-help`

Prints the tool syntax and definition to the screen. No other action is taken.

Arguments

`classDefFile``.ddl`

Name of the DDL file that contains the class definitions to be added to the schema.

`bootFilePath`

Path to the boot file of the federated database whose schema is to receive the class definitions. You can omit this argument if you set the `OO_FD_BOOT` environment variable to the correct path. See the Objectivity / DB administration book for a description of the `OO_FD_BOOT` environment variable.

Discussion

If a schema exists in the specified federated database, the DDL processor modifies it; otherwise a new schema is created. You must have write permission to the system-database file.

The DDL processor signals an error if it encounters an invalid definition. Whenever an error is signaled, the schema is left unchanged, even if the file you were processing also contained valid definitions.

Adding or Changing Definitions

By default, the DDL processor allows new class definitions to be added to an existing schema and prohibits changes to any class already in the schema; an error is signaled if the specified DDL file contains a changed definition. Alternatively, you can specify one of the following:

- Use the `-nochange` option to prohibit all schema changes or additions. This is useful for checking whether a DDL file contains changes or additions without actually modifying the schema.
- Use the `-version` option to permit schema changes by versioning the schema representation of the changed classes.
- Use the `-evolve` option to permit schema changes through schema evolution, which may entail the conversion of existing objects to a new storage layout.

Limit on Class-Name Length

The DDL processor signals an error if any class name is longer than 487 characters after C++ preprocessing (that is, after `typedef` declarations have been expanded recursively). This limit applies to the names of persistence-capable classes and non-persistence-capable classes that are incorporated in persistence-capable classes (for example, as base or embedded classes).

In some cases, you can work around this limit by using the `-keep_typedefs` option to suppress the expansion of `typedef` names—for example, in names of template class instantiations. If, however, the affected definitions are later reprocessed with `typedef` names expanded, the DDL processor will report these definitions as schema changes.

Compiler Flags

When using `ooddlx`:

- You can use any number of `-D`, `-I`, and `-U` options. The `ooddlx` command issues a warning for each repetition of any other option, except for an option repeated with different arguments, for which it signals an error.
- The `ooddlx` command signals an error when two `-D` options set the same preprocessor variable to different values. It issues a warning whenever two `-D`, two `-I`, or two `-U` arguments are the same.

DDL-Generated Files

By default, the DDL processor generates two C++ header files and a method implementation file from the specified DDL file. When included in and compiled with your application, these C++ files provide the persistence-capable classes defined in the DDL file, along with generated definitions for classes and member functions that support persistence. You can use the `-c` option to request the generation of a single C header file that provides equivalent definitions for accessing the data members of the classes.

Platform-Specific Issues

Due to a bug in the DEC C++ Version 5.5 compiler, you must perform the following steps when running `ooddlx` to properly instantiate templates on DEC systems:

1. Always run `ooddlx` with the `-noline` flag.
2. When running `ooddlx` on file `foo.ddl` to create files `foo.h`, `foo_ref.h`, and `foo_ddl.C`, create a dummy file `foo_ref.C` and a source file `foo.C`.
3. Whenever declaring an `ooVArray` or `ooString` (but not `ooVString`) in file `bar.h`, also create the source file `bar.C`.

B

DDL Pragmas

This appendix describes the `#pragma` directives that you can use in a DDL file to guide the behavior of the DDL processor.

Reference Summary

Schema Evolution	`oochangebase`
	`oodefault`
	`oodelete`
	`oorename`

Class Versioning	`ooclassname`
Generating #include Directives	`ooclassref`
Specifying a Schema	`ooschema`

Reference Index

`oochangebase`	Indicates that a base class is being replaced by another class (or set of classes) during schema evolution.
`ooclassname`	Provides a nickname for a particular version of a class.
`ooclassref`	Enables the code in one DDL file to reference a persistence-capable class that is defined in a different DDL file.
`oodefault`	Specifies a default value for an added primitive data member during schema evolution.

oodelete	Indicates that the specified class is being deleted from the schema during schema evolution.
oorename	Indicates that the specified data member or class is being renamed during schema evolution.
ooschema	Directs the DDL processor to interact with the specified schema while processing a DDL file.

Pragmas

oochangebase

Indicates that a base class is being replaced by another class (or set of classes) during schema evolution.

```
#pragma oochangebase existingBase -> newBase {,newBase}
```

Arguments

existingBase

Name of the base class to be replaced.

newBase

Name(s) of the replacing class(es). You can specify one or more base classes of *existingBase*, or you can specify a single class derived from *existingBase*:

- Replacing *existingBase* with one or more of its own base classes has the effect of removing *existingBase* from the inheritance graph of the class to which the `#pragma oochangebase` directive applies.
- Replacing *existingBase* with a derived class has the effect of inserting the derived class into the inheritance graph of the class to which the `#pragma oochangebase` directive applies.

Discussion

You should remove this pragma directive after you process the DDL file containing it. This directive must be removed before the DDL file is processed again.

ooclassname

Provides a nickname for a particular version of a class.

```
#pragma ooclassname oldClassName nickName [public]
```

Arguments

oldClassName

Original name for the class to be nicknamed.

nickName

Name to be substituted for the original class name in DDL-generated header and implementation files.

`public`

Makes all members of the renamed class public in the generated primary header file. This is useful if you are writing a program to convert instances of the old version into instances of the new version.

Discussion

When a schema contains multiple versions of a class, Objectivity / DB distinguishes those versions by a version number appended to the class name, which remains the same across all versions. However, C++ applications have no knowledge of the version numbers assigned within the schema and must rely instead on class names. Therefore, before you create a new version for a class, you normally give the original version a nickname so that new and rebuilt C++ applications can distinguish the two versions.

You place a `#pragma ooclassname` directive in a DDL file after the definition of the class to which it applies. Do *not* change the class name in the definition itself. When the DDL processor encounters a `#pragma ooclassname` directive for a definition, it generates header and implementation files in which the specified *nickname* is substituted for the original class name. That is:

- The primary header file contains a definition for class *nickname*.
- The references header file contains definitions for parameterized classes like `ooRef(`*nickName*`)`, `ooHandle(`*nickName*`)`, and so on.
- The implementation file contains registration code that binds *nickName* to the schema-assigned type number for the version matching the processed definition.

Applications compiled with the generated header and implementation files must use *nickName* to access instances of the relevant version.

If you nickname one class defined in a DDL file, you must nickname *every* class defined in that file, even classes that are not being changed. This is because the DDL processor generates registration code with global scope in the method implementation file. You can reduce the number of nicknamed classes by preparing DDL files that contain only the class definitions to be versioned, plus the definitions of any dependent classes—that is, classes derived from, associated with, or containing object references to the versioned classes.

You normally use this pragma to provide a nickname for an older version of a class when you create the new version. This allows applications to use the original class name to refer to the new version. However, you can choose to nickname the new version of a class if you have legacy code that should continue to use the original class name to work with the old version.

ooclassref

Enables the code in one DDL file to reference a persistence-capable class that is defined in a different DDL file.

```
class className;
#pragma ooclassref className <classDefFile_ref.h>
```

Arguments

`className`

Name of the persistence-capable class to be referenced.

`classDefFile``_ref.h`

Name of the references header file to be generated from the DDL file that defines *`className`*. This file need not exist at the time the DDL processor scans the `#pragma` directive.

You can use either angle brackets <> or double quotes " " around this name.

Discussion

This `#pragma` directive is always preceded by a forward declaration.

The DDL processor issues an error if you insert the `#pragma ooclassref` directive into a DDL file that contains a definition for *`className`*.

When the DDL processor encounters the `#pragma ooclassref` directive in a DDL file, it:

- Allows the DDL file to reference the generated parameterized classes for *`className`* even if the references header file *`classDefFile`*`_ref.h` does not yet exist. The generated classes are:
  - `ooRef(`*`className`*`)`
  - `ooHandle(`*`className`*`)`
  - `ooShortRef(`*`className`*`)`
  - `ooItr(`*`className`*`)`
- Causes the generated primary header file to `#include` the required references header file *`classDefFile`*`_ref.h`. The generated `#include` directive preserves the delimiters (angle brackets <> or double quotes "") you used in the pragma.

If a DDL file contains opportunities for multiple `#pragma ooclassref` directives, and the file contains at least one such directive, then similar directives must be used for all other cases, or else the DDL processor issues an error such as the following:

`"missing definition of ooRef(`*`className`*`)"`

oodefault

Specifies a default value for an added primitive data member during schema evolution.

```
#pragma oodefault value
```

Arguments

value

A value whose type and range match those of the new data member. The value must be of an Objectivity / C++ primitive type (see “Objectivity / C++ Primitive Types” on page 42).

Discussion

You place a `#pragma oodefault` directive immediately before the added data member. Omitting this directive causes the new member to be set to 0. You can use this directive only for a newly added data member whose data type is a primitive type listed in Table 2-2 on page 43. The data member can be scalar or a fixed-length array of primitive-typed elements.

NOTE Objectivity / C++ strings (for example, of class `ooVString`) are not primitive types, but non-primitive (embedded-class) types. You use a conversion application to set new values of non-primitive data members.

The specified default value is preserved in the schema evolution history of a definition. Consequently, the value will be remembered and set, even in objects whose conversion is deferred for several schema evolution cycles.

You should remove this pragma directive after you process the DDL file containing it. This directive must be removed before the DDL file is processed again.

oodelete

Indicates that the specified class is being deleted from the schema during schema evolution.

```
class className;
#pragma oodelete className
```

Arguments

className

Name of the class whose definition has been removed from the DDL file.

Discussion

This directive must follow a forward declaration to the deleted class. You can place the forward declaration and directive anywhere in the DDL file.

You should remove this pragma directive after you process the DDL file containing it. This directive must be removed before the DDL file is processed again.

oorename

Indicates that the specified data member or class is being renamed during schema evolution.

```
#pragma oorename existingName
```

Arguments

existingName

Original name of the data member or class that is being renamed.

Discussion

When renaming a class, you must put this directive immediately before the class definition in the DDL file. When renaming a data member, you must put this directive immediately before the renamed data member in the class definition.

You should remove this pragma directive after you process the DDL file containing it. This directive must be removed before the DDL file is processed again.

ooschema

Directs the DDL processor to interact with the specified schema while processing a DDL file.

```
#pragma ooschema schemaName
```

Arguments

schemaName

Name of the schema in which to add (or find) the next persistence-capable class to be processed. You can:

- Specify a string name. If necessary a schema with this name is created.
- Specify * to indicate the default schema.
- Omit *schemaName* to indicate the schema that was set by `-schema` option of the DDL processor.

Discussion

You use this `#pragma` directive in a DDL file that defines or references classes that belong to different schemas. This situation typically arises when a class being processed into one schema references a second class that belongs to a different schema.

As the DDL processor works through a DDL file, it looks for existing definitions in, and adds new definitions to, the schema specified by the `-schema` option at

invocation. When the DDL processor encounters a `#pragma ooschema` directive, it switches to the schema specified in the directive.

C

Objectivity/C++ Include Files

The following table contains an overview of the Objectivity/C++ include files and what they provide.

To Use Any of:	Include:
General Objectivity/C++ classes, global functions, macros, types and constants (but no special-purpose classes or application-defined classes).	`oo.h`[a]
Application-defined class *`appClass`* (defined in the DDL file *`myClasses`*`.ddl`). Handle, object-reference, and iterator classes for *`appClass.`*	Generated primary header file *`myClasses`*`.h`
Name-map class `ooMap`. Handle, object-reference, and iterator classes for `ooMap`. Name-map element class `ooMapElem`. Handle and object-reference classes for `ooMapElem`. Name-map iiterator class `ooMapItr`.	`ooMap.h`
Scalable-collection classes (`ooCollection` and derived classes). Handle, object-reference, and iterator classes for scalable-collection classes. Scalable-collection iterator classes (`ooCollectionIterator` and derived classes). Administrator and comparator classes. Handle, object-reference, and iterator classes for administrator and comparator classes.	`ooCollections.h`
ODMG date and time classes.	`ooTime.h`
Java-compatibility classes. Handle, object-reference, and iterator classes for Java compatibility classes.	`javaBuiltins.h`

To Use Any of:	Include:
Key-description class `ooKeyDesc`. Handle, object-reference, and iterator classes for `ooKeyDesc`. Key-field class `ooKeyField`. Handle, object-reference, and iterator classes for `ooKeyField`. Lookup-key class `ooLookupKey`.	`ooIndex.h`
Administration functions `ooCleanup`, `ooGetActiveTrans` or `ooGetResourceOwners`.	`ooRecover.h`

a. A DDL file never needs to include oo.h explicitly. A source file does not need to include oo.h explicitly if it includes a generated primary header file *myClasses*.h, because generated files include oo.h.

D

Schema Class Descriptions

The schema of an Objectivity / DB federated database describes every class whose objects are saved in the federated database. The schema is shared by all applications that access the federated database. The schema description for a class includes the name of the class and data type of each attribute. The schema uses class names and attribute data types that are independent of the application source language. This language-independent representation allows applications written in C++, Java, and Smalltalk to read and write persistent objects in the same federated database. Each application maps data for an object between the Objectivity / DB data types specified in the schema description for its class and data types native to the application.

Content of a Schema Class Description

A class description in the schema contains a class name, type number, the name(s) of the class's immediate base classes, an ordered collection of attribute descriptions, and an ordered collection of association descriptions.

Each attribute description specifies the attribute's name, data type, and access control.

Each association description specifies:

- The association name
- The name of the destination class
- The association's directionality and cardinality, its delete and lock propagation behavior, its copying and versioning behavior, and its storage properties
- If the relationship is bidirectional, the name of its inverse association

By specifying the order and size of a class's data members, a class description determines the shape of each object of the class.

Schema Class Names

The schema identifies each class with a unique class name that is set when the class description is added to the schema. If a class description is added to the schema by the DDL processor, the name of the C++ class is used as the schema class name. If the class description is added by Objectivity for Java or Objectivity / Smalltalk, the schema class name need not be the same as the Java or Smalltalk class name. If necessary, Java and Smalltalk applications must explicitly map a schema class name to the Java or Smalltalk class name.

NOTE A class name in the schema can contain a maximum of 487 characters.

Objectivity/DB Primitive Types

Within the schema of an Objectivity / DB federated database, all attributes that contain numeric, character, or Boolean data are stored as of one of the language-dependent Objectivity / DB primitive types:

Category	Objectivity/DB Primitive Type	Description
Integer	`int8`	8-bit signed integer type
	`uint8`	8-bit unsigned integer type
	`int16`	16-bit signed integer type
	`uint16`	16-bit unsigned integer type
	`int32`	32-bit signed integer type
	`uint32`	32-bit unsigned integer type
	`int64`	64-bit signed integer type
	`uint64`	64-bit unsigned integer type
Floating point	`float32`	32-bit floating-point type
	`float64`	64-bit floating-point type

Each programming interface to Objectivity / DB is able to convert these language-independent types into a native type in the corresponding programming language. For example, Objectivity / C++ defines numeric types of

the same names; Objectivity for Java converts these types into Java primitive types such as `byte`, `short`, and `boolean`.

Mapping Objectivity/C++ Primitive Types

When a data member in a DDL file is declared as an Objectivity / C++ primitive type, the DDL processor substitutes the corresponding Objectivity / DB primitive type in the class description in the federated database schema:

Category	Objectivity/C++ Primitive Type in DDL File			Objectivity/DB Type in Schema
	Type Name	Alternative Name	ODMG Name	
Integer	`int8`	`ooInt8`	(None)	`int8`
	`uint8`	`ooUInt8`	`d_Octet`	`uint8`
	`int16`	`ooInt16`	`d_Short`	`int16`
	`uint16`	`ooUInt16`	`d_UShort`	`uint16`
	`int32`	`ooInt32`	`d_Long`	`int32`
	`uint32`	`ooUInt32`	`d_ULong`	`uint32`
	`int64`	`ooInt64`	(None)	`int64`
	`uint64`	`ooUInt64`	(None)	`uint64`
Floating point	`float32`	`ooFloat32`	`d_Float`	`float32`
	`float64`	`ooFloat64`	`d_Double`	`float64`
Character	`char`	`ooChar`	`d_Char`	`int8` on architectures where C++ `char` is signed `uint8` on architectures where C++ `char` is unsigned
Boolean	`ooBoolean`	(None)	`d_Boolean`	`uint8`
Enumeration	*Any Objectivity/C++-defined enumeration type*		(None)	`int32`

Mapping C++ Primitive Types

When a data member in a DDL file is declared as a C++ numeric, character, boolean, or enumeration type, the DDL processor substitutes the corresponding Objectivity / DB primitive type in the class description in the federated database schema. These mappings, shown in the following table, are compatible with most C++ compilers on 32-bit CPU architectures.

Category	C++ Type in DDL File	Objectivity/DB Type in Schema

Integer	`short`
	`short int`
	`signed short`
	`signed short int`
		`int16`

	`unsigned short` `unsigned short int`	`uint16`
	`int` `signed int`	`int32`
	`unsigned int`	`uint32`

	`long`
	`long int`
	`signed long`
	`signed long int`
		`int32` (on DEC Alpha, `int64`)

	`unsigned long` `unsigned long int`	`uint32` (on DEC Alpha, `uint64`)

	`long long`
	`signed long long`
	(not available on Windows)
		`int64`

	`unsigned long long` (not available on Windows)	`uint64`

	`__int64`
	`signed __int64`
	(Windows only)
		`int64`

	`unsigned __int64` (Windows only)	`uint64`

Category	C++ Type in DDL File	Objectivity/DB Type in Schema
Floating point	`float`	`float32`
	`double`	`float64`
	`long double`[a]	Not mapped to any Objectivity/C++ type. **Note:** `long double` may not be used for a data member in a persistence-capable class or in a non-persistence-capable class that is embedded in a persistence-capable class.
Character	`char`	`int8` on architectures where `char` is signed `uint8` on architectures where `char` is unsigned
	`unsigned char`	`uint8`
	`signed char`	`int8`
	`wchar_t`	`uint16`
Boolean	`bool`	`int32` on Solaris 2.6 unsupported on IBM Risc/System 6000 `int8` on all other platforms
Enumeration	`enum` *`type`*	`int32`

a. Objectivity/DB does not support floating point numbers larger than 64 bits.

Note that:

- The mappings for `long`, `unsigned long`, `char` and `bool` are platform-dependent.
- The C++ Boolean type `bool` maps to `int8` or `int32`, unlike the platform-independent Objectivity / C++ `ooBoolean` type, which maps to `uint8`.

WARNING When using C++ primitive types, you should make sure that the primitive-type mappings allow portability across *all* of your target computing environments. Wherever possible, you should use Objectivity / C++ primitive types instead of C++ primitive types.

E

Schema-Evolution Quick Reference

This table summarizes various details of the supported schema-evolution operations. For a comprehensive discussion and complete steps, see Chapter 5, “Schema Evolution”.

Operation	Conversion Operation?	DDL Options	DDL Pragma Used in a Cycle	No. of Cycles	See pg.
Adding a class (non-evolution)	—	—	—	—	125
Adding a data member					104
Adding an association	Yes	`-evolve`	—	One	106
Adding an attribute	Yes	`-evolve`	`oodefault`[a] (Optional)	One	104
Adding a virtual member function	Yes	`-evolve`	—	One	124
Changing class inheritance					130
Adding a non-persistence-capable base class	Yes	`-evolve`	—	One	132
Adding persistence to a class (deleting and reintroducing)	Yes	`-evolve`	`oodelete`	One or more[b]	141
Changing the access control of a base class	No	`-evolve`	—	One	140
Changing the order of a base class	Yes	`-evolve`	—	One	140
Moving a class higher in the inheritance graph	Yes	`-evolve` `-upgrade`	`oochangebase`	One[c]	138
Moving a class lower in the inheritance graph	Yes	`-evolve` `-upgrade`	`oochangebase`	One[c]	134

Operation	Conversion Operation?	DDL Options	DDL Pragma Used in a Cycle	No. of Cycles	See pg.
Removing a non-persistence-capable base class	Yes	`-evolve`	—	One	136
Removing persistence from a class (deleting and reintroducing)	Yes	`-evolve`	`oodelete`	One or more[b]	142
Changing a data member					114
Access control	No	`-evolve`	—	One	120
Association behavior specifiers	Yes	`-evolve`	—	One	123
Association cardinality	Yes	`-evolve`	—	Three[d]	123
Association storage (inline, non-inline)	Yes	`-evolve`	—	One	121
From one non-primitive type to another	Yes	`-evolve`	`oorename`	Three[d]	117
From one primitive type to another	Yes	`-evolve`	—	One	114
Position (order)	Yes	`-evolve`	—	One	120
Size of a fixed-size array	Yes	`-evolve`	—	One	115
Storage (long, short)	Yes	`-evolve`	—	One	116
Deleting a class	Usually[b]	`-evolve`	`oodelete`	One or more[b]	126
When links exist to a base class	Yes	`-evolve` `-upgrade`	`oodelete`	One or more[b, c]	129
Deleting a data member					108
Deleting an association	Yes	`-evolve`	—	One	108
Deleting an attribute	Yes	`-evolve`	—	One	108
Deleting a virtual member function	Yes	`-evolve`	—	One	124
Renaming a class	No	`-evolve`	`oorename`	One	125
Renaming a data member	No	`-evolve`	`oorename`	One	109

Operation	Conversion Operation?	DDL Options	DDL Pragma Used in a Cycle	No. of Cycles	See pg.
Replacing a data member					111
Replacing non-primitive data members	Yes	`-evolve`	—	Two[d]	112
Replacing primitive data members	Yes	`-evolve`	—	One[e]	111
Restructuring classes					142
Merging two associated classes	Yes	`-evolve`	`oodelete`	Two[d]	144
Merging a base class into a derived class	Yes	`-evolve` `-upgrade`	`oochangebase` `oodelete`	Three[b,d]	145
Splitting a class into two associated classes	Yes	`-evolve`	—	Two[d]	142
Splitting a class into a pair of base and derived classes	Yes	`-evolve` `-upgrade`	`oochangebase`	Two[b, d]	143

a. The #pragma oodefault directive applies only when adding primitive-typed attributes.
b. Deleting a class is usually accompanied by conversion operations on other classes; several schema-evolution cycles may be required to prepare a class for deletion.
c. An upgrade application must be run after the cycle that uses the DDL processor option -upgrade.
d. A conversion application must be run between cycles.
e. A conversion function must be registered with the application that is to trigger object conversion.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W