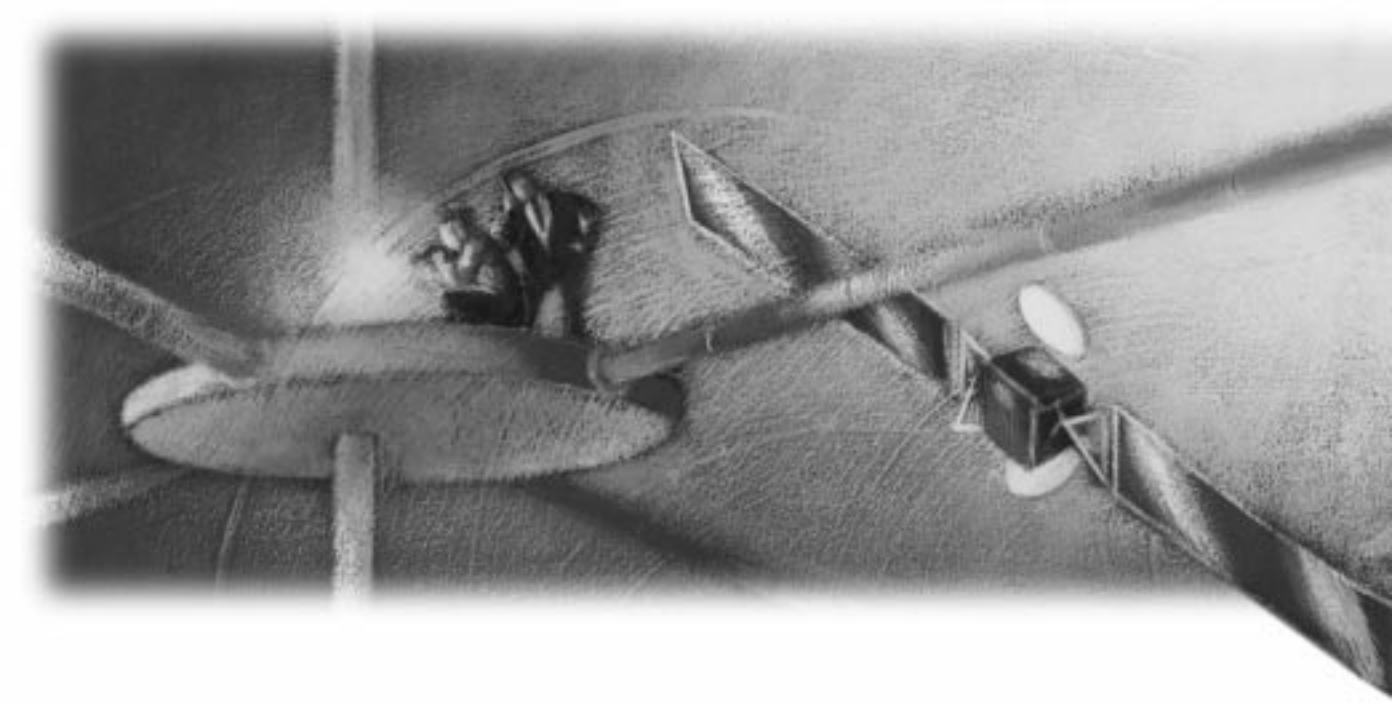

Objectivity/C++ Active Schema

Release 6.0

Objectivity/C++ Active Schema

Part Number: 60-AS-0

Release 6.0, October 5, 2000

Contents

About This Book

This book describes the capabilities of Objectivity/C++ Active Schema and provides reference documentation for the programming interface to Active Schema.

Audience

This book assumes that you are familiar with programming in C++, Objectivity/C++, and the Objectivity/C++ Data Definition Language.

Organization

- Part 1 introduces the concepts that are basic to Active Schema and describes how to use this product.
- Part 2 describes the programming interface through which an application (such as a database browser) interacts with Active Schema.
- Appendix A describes Objectivity/DB internal classes that may appear in the schema description of a class.
- Appendix B contains the source code for programming examples; excerpts from some of these examples appear in Part 1.
- The Glossary contains definitions of terms and concepts that are used in this book.
- Four indexes allow you to look up information:
  - The Topic Index lists topics that are discussed in this book.
  - The Classes Index lists the Active Schema classes, with member functions listed under each class.
  - The Functions Index lists all functions in the Active Schema programming interface, including member functions.

- The Types and Constants Index lists the non-class types and constants in the Active Schema programming interface.

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 (labeled 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

Part 1 USING ACTIVE SCHEMA

1

Getting Started

Objectivity/C++ Active Schema provides a programming interface for accessing the schema of an Objectivity/DB federated database.

In This Chapter

About Active Schema

A programmer can use Active Schema to develop database-development tools such as class and object browsers. An Active Schema application can:

- Obtain descriptions of the classes in the schema.
- Access and modify persistent objects using descriptions of their classes that are obtained dynamically from the schema (not compiled into the program).
- Modify the schema by adding new classes and changing the descriptions of existing classes.

- Add persistent objects to the federated database, including instances of newly defined classes.
- Convert persistent objects to be consistent with the evolved schema descriptions of their classes.

Federated Database Schema

Every Objectivity / DB federated database uses a *schema* to describe the data that it contains. The schema consists of descriptions of the data types that can be stored persistently. The data types available for use in a schema are classes and property types.

Objectivity / DB assigns a unique *type number* to each class in the schema. In addition, Active Schema assigns a unique type number to each property type in the schema.

Modules

The schema of an Objectivity / DB federated database may be divided into disjoint modules that group related classes. In the terminology of the Objectivity / C++ Data Definition Language (DDL), these modules are called *schemas*. Each federated database schema contains at least one module, called the *default schema*; that module contains descriptions of the property types and internal classes defined by Objectivity / DB. Additional modules, called *named schemas*, can be added to the default schema.

Each application-defined class belongs to exactly one module, either the default schema or a named schema.

NOTE To differentiate between the entire federated database schema and one of the disjoint portions within it, this document uses the term *module* for a disjoint portion instead of *schema*; it uses *top-level module* instead of *default schema* and *named module* instead of *named schema*. In the remainder of this document, the word *schema* always refers to the entire federated database schema.

Classes

A federated database schema can contain descriptions of:

- Persistence-capable classes, whose instances can be stored persistently and accessed independently.
- Non-persistence-capable classes, whose instances can be embedded in the data of instances of persistence-capable classes.

  Instances of non-persistence-capable classes can be stored persistently as part of the data of the containing instance. They cannot be accessed independently, but only through their containing instances.

The classes described in the schema can be either *internal classes*, which are defined by Objectivity / DB, or application-defined classes. Appendix A, "Internal Classes," lists internal Objectivity / DB that can appear in the schema.

Properties of a Class

The *properties* of a class consist of its attributes and its relationships. A persistence-capable class can have properties of both kinds. In contrast, a non-persistence-capable class can have attributes but cannot have relationships.

- The *attributes* of a class constitute its component data. The data for a class consists of the attributes it defines as well as the attributes it inherits from its base classes. Attributes correspond to standard data members of a C++ class, fields of a Java class, or instance variables of a Smalltalk class.
- An application-defined persistence-capable class can define *relationships*. A relationship associates an instance of the defining class (or source class) to one or more instances of a destination class. The destination class can be any persistence-capable class, including the source class itself.

NOTE Although a relationship is conceptually different from an attribute, Objectivity / DB implements relationships as a special kind of attribute. For this reason, many Active Schema operations that apply to the attributes of a class work for relationships of the class as well. For example, when you iterate over the attributes of a class, the iterator finds relationships as well as attributes.

Versions of a Class

For most classes, the schema contains a description of a single *version* of the class. However, if an application has used the Objectivity / C++ class-versioning feature, the schema may contain descriptions for multiple versions of particular classes. The different versions of a class are given sequential version numbers starting with 1 for the original version.

The different versions of a class are considered different types. Each version has a different type number. The federated database may contain instances of each of the different versions.

Shape of a Class

The physical layout of storage for an instance of a class includes space to store the data for each property that is defined in the class or inherited by the class. The base classes and properties of a class, therefore, determine the *shape* that instances of the class occupy in storage. As each class is added to the schema, it is assigned a unique *shape number*, which is identical to its type number.

Any class may undergo *evolution*, which means that its definition in the schema may change over time. Each time the definition is modified in a way that affects its storage layout, a new description is added to the schema with a new shape number for the class.

For any given class A, a new description of A with a new shape number is added to the schema whenever any of the following classes evolves in a way that affects storage layout:

- The class A itself
- Any non-persistent class embedded in A
- Any base class of A

The different shapes of a particular class are *not* considered different types. After evolution has occurred, instances of the evolved class are converted to the new shape. When an application accesses one of these objects, it is converted to the new storage layout.

If the schema includes multiple versions of a given class, any version of the class can evolve. That is, the definition of any version can be modified, getting a new shape number for that version of the class. When an application accesses instances of the evolved version, they are converted to the new shape for that version.

Attributes

An attribute is defined in some particular class and represents a particular piece of data that instances of that class can have. An attribute's name must be unique among the names of the immediate base classes, attributes, and relationships of the class.

If a class inherits properties from a base class, the data corresponding to those properties is embedded in the data for an instance of the subclass, just as if the base class were an embedded attribute of the subclass. For this reason, base classes and embedded classes are sometimes treated as identical in the Active Schema programming interface.

Attribute Types

Every attribute has one particular attribute type, which specifies the kind of data that can be stored in the attribute. The following table describes the available attribute types and the kind of data that each can store.

Attribute Type	Data for the Attribute
Basic numeric types	Values of a particular fundamental character, integer, floating-point, or pointer type Notes: ■ The C++ type `ooBoolean` is described in the schema as the unsigned 8-bit integer type; the schema does not record the legal range of values specified by the `ooBoolean` type. ■ Application-defined enumerations are described in the schema as the 32-bit integer type; the schema does not record the legal range of values specified by the enumeration. ■ The C++ pointer type is a basic numeric type. Typically, a C++ pointer attribute is used for transient data because a pointer value saved by one process will not be meaningful (or valid) in a different process that retrieves the value. Although pointer attributes contain transient data, the schema description of a class includes those attributes so that the shape of the class will be correct.
Object-reference types	Object references to instances of a particular persistence-capable class
Embedded-class types	Instances of a particular non-persistence-capable class embedded within the data of the containing instance
Variable-size array types (VArray types); the element type can be one of the following: ■ A basic numeric type ■ An object-reference type ■ An embedded-class type	Variable-size arrays (VArrays) of elements of the same type

Attribute Values

A given attribute can hold either a single value of its type, or a fixed-size array of values of its type.

Although an Objectivity / C++ class may have a multidimensional array of values in a persistent data member, the schema treats any fixed-size C++ array as a one-dimensional array with a fixed number of elements. For example, a data member declared in a DDL file as containing a 3-by-4 two-dimensional array of 16-bit signed integers is described in the schema as an attribute containing a 12-element array of 16-bit signed integers.

Relationships

A relationship is defined in some particular class, called its *source class*. A relationship represents a directional association that can exist between instances of the source class and instances of a *destination class* (which may be the same as the source class).

A relationship's name must be unique among the names of the immediate base classes, attributes, and relationships of the class.

Relationship Types

Each relationship type has two defining characteristics:

- The *directionality* of relationships of this type (unidirectional or bidirectional)
- The destination class for relationships of this type.

If a relationship `R` is bidirectional, its destination class defines an *inverse* relationship `S`; the destination class of relationship `S` is the source class of relationship `R`.

Relationship Characteristics

In addition to its type, a relationship has characteristics that specify:

- Whether the relationship is to-one or to-many. This characteristic is called the *cardinality* of the relationship.

  If a relationship is to-one, a particular instance of the source class, called the *source object*, can be associated with one particular instance of the destination class, called the *destination object*. If the relationship is to-many, a given source object can be associated with more than one destination object.
- Whether the relationship is inline.
- Whether source objects store references to their associated destination objects as standard object references or short object references.

- The copy mode, which specifies what happens to an association from a source object to a destination object when the source object is copied.
- The versioning mode, which specifies what happens to an association from a source object to a destination object when a new version of the source object is created.
- The propagation behavior, which specifies whether the locking and deletion operations are propagated from locked or deleted source objects to their associated destination objects.

Descriptors

As we have seen, the federated database schema contains information about entities of the following kinds:

- Modules
- Types, including classes and property types
- Properties of classes, that is, attributes and relationships

Active Schema allows a program to obtain schema information about any of these entities. The information is embodied in an object called a *descriptor*. A descriptor is a meta-object; that is, an object that provides information about a "real" object, namely, a module, a type, or a property. This document calls modules, types, and properties *entities* rather than *objects* to distinguish them from the persistent objects stored in the federated database.

Each descriptor provides information about a particular named entity in a federated database. For example, the descriptor for a module can provide the name of that module and can find all modules and types the module contains.

Scope

An entity in the federated database schema that organizes the other entities is called a *scope*. A scope can limit search when Active Schema looks up descriptions of entities.

Two kinds of entity serve as scopes: modules and classes.

- Any module (top-level or named) is the scope for the application-defined classes that have been assigned to that module.
- The top-level module is the scope for all modules in the schema, for all property types, and for all internal Objectivity / DB classes.

  The scope of the top-level module encompasses the scopes of all its contained modules. So, you can look up a class defined in a named module

either in the scope of its containing module or in the scope of the top-level module.

- A class is the scope for all its properties.

NOTE Module scope provides a grouping or organization of classes in the schema. Modules, however, do *not* define independent name scopes.

Active Schema Applications

The source code for an Active Schema application must include the header file `ooas.h`, which declares the Active Schema global types, constants, classes and their member functions. Any Active Schema application is an Objectivity/C++ application that uses ObjectSpace Standard Template Library (STL).

You must link your compiled files with the Active Schema library and the Objectivity/DB library. On some platforms you must specify the ObjectSpace STL header directory when you compile your files and you must link your files with the ObjectSpace STL library.

See the Installation and Platform Notes for the appropriate link libraries and link options for your platform.

Structure of an Active Schema Application

As with any Objectivity/C++ application:

- An Active Schema application must initialize Objectivity/DB before performing any Objectivity/DB and Active Schema operations. A single-threaded application calls the `ooInit` global function; a multithreaded application calls the `ooInitThread` global function in each thread.
- The application opens the federated database by calling the `open` member function of a federated database handle at the beginning of the first transaction. You must open the federated database at the beginning of each subsequent transaction to verify that your application is accessing the same federated database.
- If the application is multithreaded and is to run on Windows or support future portability to Windows, it should call `ooExitCleanup` after performing all Objectivity/DB and Active Schema operations.

Opening a federated database gives your application access to the federated database schema. All interactions with the federated database to access schema descriptions or persistent data must take place within transactions.

If your application uses Active Schema to view or modify persistent objects in the federated database, it is subject to the same rules about transactions and locking that apply to any Objectivity / C++ application. In particular, any access to a persistent object must occur within a transaction and the application must be able to obtain a read lock on that object's container. Any member function that creates or modifies a persistent object requires a write lock on the object's container; changes are written to the federated database only when the transaction is committed.

EXAMPLE This application checks whether the federated database schema contains a class named `Library`. Its `main` function initializes Objectivity / DB, calls `schemaOps`, and prepares Objectivity / DB for shutdown.

```
#include <ooas.h>
...
int main(const int argc, const char *const argv[]) {
   int retval = 0;
   // Initialize Objectivity/DB
   ooInit();
   // Perform Objectivity/DB and Active Schema operations
   retval = schemaOps();
   return retval;
}
```

The function `schemaOps` performs all interactions with Objectivity / DB and Active Schema. Inside a transaction, it opens the federated database, obtains a descriptor for the top-level module, and looks for the class `Library`.

```
int schemaOps() {
   ooTrans trans;
   ooHandle(ooFDObj) fdH;
   // Start a transaction
   trans.start();
   // Open the federated database
   if (fdH.open("myFederatedDatabase", oocRead) != oocSuccess) {
      cerr << "Failed to open federated database" << endl;
      trans.abort();
      return 1;
   }
   // Get a descriptor for the top-level module
   const d_Module &topMod = d_Module::top_level();
   // See whether the class Library exists
   const d_Class &lib = topMod.resolve_class("Library");
   if (lib) {
      cout << "Library class found" << endl;
   }
```

```
   else {
      cout << "Library class not found" << endl;
   }
   trans.commit();
   return 0;
}
```

Programming Guidelines

Member functions that find all entities with particular characteristics return iterators that get descriptors for the specified entities. Chapter 6, “Working With Iterators,” explains how to use the iterators returned by these member functions.

Member functions of many Active Schema classes may throw exceptions. Chapter 7, “Error Handling,” explains how your applications can catch exceptions or disable exceptions altogether.

It is possible, though uncommon, to use Active Schema in conjunction with Objectivity/C++ schema evolution and object conversion operations. If you do so, be aware that:

- Active Schema does not change the schema in ways that require you to run an upgrade application
- If necessary an Active Schema application could be used as an upgrade application that calls the `upgradeObjects` on a federated-database handle.

For a description of the Objectivity/C++ schema evolution operations, see Chapter 5, “Schema Evolution,” in the Objectivity/C++ Data Definition Language book. For a description of the Objectivity/C++ object conversion operations, including upgrade applications, see Chapter 19, “Object Conversion,” in the Objectivity/C++ programmer’s guide

2

Examining the Schema

Active Schema applications can examine the schema of any Objectivity/DB federated database, finding the names of all entities in the schema and getting detailed descriptions of those entities. You might use these capabilities to develop a class browser for Objectivity/DB federated databases.

In This Chapter

Schema Descriptors

Schema descriptors provide information about the entities in a schema. Each descriptor provides information about a particular named entity in a federated database, called its *described entity*. Different descriptor classes provide information about entities of different kinds.

Module descriptors and class descriptors act as scopes, providing hierarchical access to other descriptors. The top-level module is the scope for named modules, for classes, and for property types; a named module is the scope for classes; a class is the scope for its properties, both attributes and relationships.

Figure 2-1 shows the inheritance graph for the major schema-descriptor and scope classes.

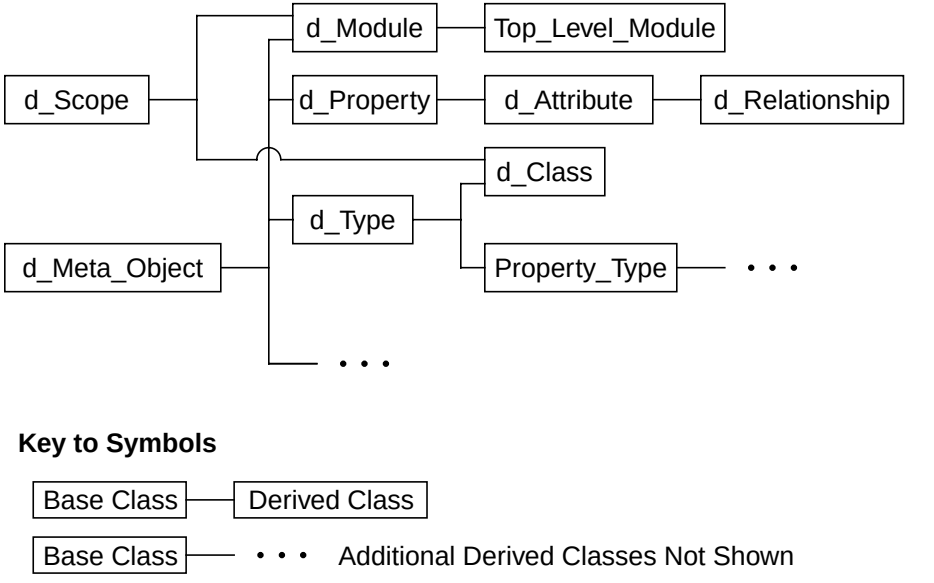

Figure 2-1 Schema-Descriptor and Scope Classes

“Classes that Describe the Schema” on page 148 lists all classes that are relevant to examining the schema of a federated database.

Traversing the Schema

The first step for any application that examines the schema of its connected federated database is to obtain a descriptor for the top-level module. From that descriptor, the program can traverse the entire schema. From the descriptor for the top-level module, you can get descriptors for all named modules, property types, and classes. From the descriptor for a named module you can get descriptors of the classes defined in that module. From the descriptor for any class, you can get descriptors of the attributes and relationships of that class. From the descriptor for any attribute or relationship, you can get a descriptor for the type of the described attribute or relationship. From the descriptor for any relationship you can get a descriptor for the destination class; if the relationship is bidirectional, you also can get a descriptor for the inverse relationship.

NOTE You never instantiate any descriptor class directly. Instead, you obtain descriptors by looking up an individual entity or by iterating over a group of entities.

Restrictions on Descriptors

Once your application has set a descriptor variable, it may not use that variable as the left operand to an assignment operator; if you attempt to do so, Active Schema throws an exception.

You may not copy descriptors; the behavior of a copied descriptor is undefined.

Null Descriptors

Any member function that looks up a descriptor returns a descriptor object; unsuccessful searches return a *null descriptor*. After obtaining a descriptor, you should check that it is non-null before trying to obtain information about its described entity. You can use a descriptor as an integer expression to test whether that descriptor is valid (not null). A valid descriptor is converted to nonzero, a null descriptor to zero.

EXAMPLE This example gets a class descriptor for the class named `Library` and verifies that the class descriptor is valid before using it.

```
ooTrans trans;
trans.start();
...
// Get a descriptor for the class Library
const d_Class &lib = topMod.resolve_class("Library");
```

```
if (lib) {
   … // Proceed to work with the descriptor
}
else {
   cerr << "Library class not found" << endl;
}
…
trans.commit();
```

Getting Information From Descriptors

Various member functions of the descriptor classes return information using the following data types:

- C++ numbers or strings

  For example, the name of an entity is returned as a string (`char *`); its ID is returned as an integer (`uint32`).

- Objectivity/C++ types and constants

  For example, the copy mode of a relationship is returned as one of the Objectivity/C++ constants `oocCopyDrop`, `oocCopyMove`, or `oocCopyCopy`.

- Active Schema non-class types and constants (see “Global Types and Constants” on page 153)

  For example, the kind of data for a numeric type descriptor is returned as a value of the Active Schema type `ooBaseType`.

- Descriptor classes

  For example, the class for an embedded-class type is returned as a class descriptor; the inverse relationship of a bidirectional relationship is returned as a relationship descriptor.

- Other Active Schema classes

  For example, the default value of an attribute is returned as a numeric value (`Numeric_Value`)

Examining Modules

To examine a module in the schema of a federated database, you obtain a descriptor for the module and call the descriptor’s member functions to get information about the module.

Obtaining a Module Descriptor

To get a module descriptor (`d_Module`) for the top-level module, call the static member function `d_Module::top_level`.

Once you have a descriptor for the top-level module, you can call its member functions to get descriptors for named modules, which are defined in the scope of the top-level module:

- To look up a module by name:
  - Call `resolve_module` to get a module descriptor for the module.
  - Call `resolve` to get a generic descriptor (`d_Meta_Object`) for the module.
- To get an iterator that finds all named modules:
  - Call `named_modules_begin` to get an iterator that returns module descriptors for all named modules.
  - Call `defines_begin` to get an iterator that returns generic descriptors for all modules, classes, and property types.

A generic descriptor can describe any kind of entity. If necessary, you can call the generic descriptor's `is_module` member function to test whether it describes a module. When you are sure that a generic descriptor describes a module, you can cast the generic descriptor to a module descriptor.

EXAMPLE This code iterates through the entities in the scope of the top level module. If an entity is a module, this example casts the generic descriptor to a module descriptor and proceeds to examine the described module.

```
trans.begin();
...
meta_object_iterator itr = topMod.defines_begin();
while (itr != topMod.defines_end()) {
   const d_Meta_Object &curEntity = *itr;
   if (curEntity.is_module()) {
      // Described entity is a module
      // Cast generic descriptor to module descriptor
      const d_Module &curMod = (const d_Module &)curEntity;
      ... // Examine the described module
   } // End if module
   ...
   ++itr;
} // End while more entities
...
trans.commit();
```

Getting Information About a Module

You can call member functions of a module descriptor to get information about its described module:

- Call the inherited `name` member function to get the module's name.
- Call `id` to get the unique ID that identifies the module.
- Call `schema_number` to get the module's schema number.
- Call `next_type_number` to get the next available type number for any new class or new version of a class that is added to the module.
- Call `next_assoc_number` to get the next available association number for any new relationship that is added to a class in the module.
- Call `is_top_level` to test whether the module is the top-level module of the federated database.

EXAMPLE This code iterates through the named modules, printing the name and schema number of each.

```
trans.begin();
...
module_iterator itr = topMod.named_modules_begin();
while (itr != topMod.named_modules_end()) {
   const d_Module &curMod = *itr;
   cout << curMod.name();
   cout << " (" << curMod.schema_number() << ")" << endl;
   ++itr;
} // End while more named modules
...
trans.commit();
```

Examining Classes

To examine a class, you obtain a descriptor for the class and call the descriptor's member functions to get information about the class.

Class Descriptions

A class descriptor allows an Active Schema application to access a particular class description in the schema. Each class description contains information about a particular shape of a particular version of a particular class. The class can be either an *internal class*, which is defined by Objectivity / DB, or an application-defined class.

A given class descriptor provides information about one of the following:

- An internal Objectivity / DB persistence-capable class (such as `ooContObj` or `ooMap`).

  Such an internal class appears in the schema if an application-defined class uses the internal class as a base class or as the referenced class of an object-reference attribute.
- An application-defined persistence-capable class.
- An internal Objectivity / DB non-persistence-capable class (such as `ooVString`).
- An application-defined non-persistence-capable class.

  The optimized string classes `ooString(`*`N`*`)` are application-defined non-persistence-capable classes. These classes are not predefined by Objectivity / DB and so are not internal. Instead, any C++ application that uses a string attribute optimized for some particular string length defines an optimized string class with a fixed-size array to accommodate strings of that length. The name `ooString(`*`N`*`)` is a macro that expands to a template class whose parameter is *`N`*.

  For example, an application that uses an attribute to store strings that are generally less than 20 characters long would define the corresponding data member of type `ooString(20)` in a DDL file. When the DDL file is processed, the description of an application-defined class named `ooString_20` is added to the schema.

Appendix A, "Internal Classes," describes the Objectivity / DB internal classes that can appear in the schema description of an application-specific class. Additional undocumented internal classes may appear within the schema description of the internal classes themselves.

Components of a Class

The *components* of a class are its immediate base classes, its attributes, and its relationships. A class descriptor contains a list of components in the described class; these are the entities in the scope of the described class. This list consists of:

- Every immediate base class of the described class, in the order in which they appear in the class definition
- Every attribute and relationship defined in the class, in the order in which they appear in the class definition

A base class is treated the same as an embedded-class attribute whose name is the name of the base class and whose type is the base class.

For example, consider the following class definitions:

```
class Base1 {
   int32 a[10];
   ooVString b; };
class Base2 : public Base1 {
   ooVArray(uint16) c;
   float32 d; };
class Info {
   char e;
   ooVArray(ooVString) f; };
class Test : public ooObj, Base2 {
   ooRef(Test) x : copy(delete);
   Info y;
   ooRef(Test) z; };
```

A class descriptor for `Test` has a list of five properties:

1. `ooObj` is an embedded-class attribute of type `ooObj`.
2. `Base2` is an embedded-class attribute of type `Base2`.
3. `x` is a unidirectional relationship to a persistent object of the `Test` class.
4. `y` is an embedded-class attribute of type `Info`.
5. `z` is an object-reference attribute to a persistent object of the `Test` class

Attribute Position

Each component of a class has an *attribute position* that indicates the position of the component's data within the data of an instance of the class. Within the layout of a class, the components of the class are given sequential, zero-based scalar attribute positions.

Figure 2-2 illustrates the physical layout of the data for an instance of the class `Test`. Below each attribute in the figure is its attribute position within the class that defines it.

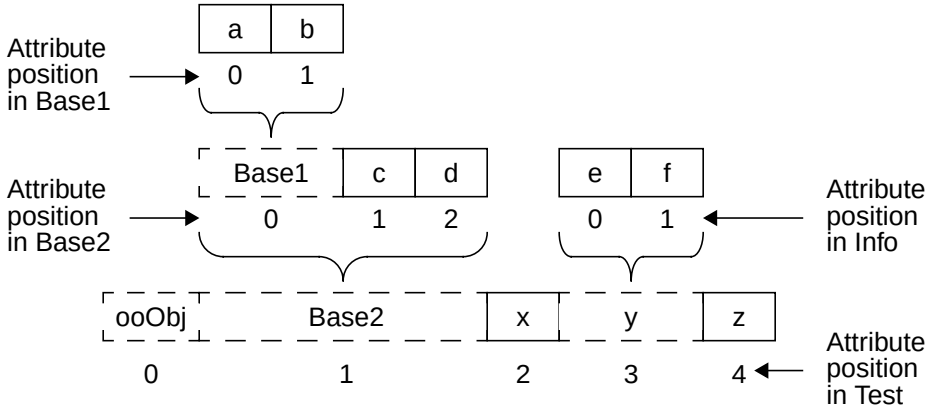

Figure 2-2 Attribute Positions of Defined Attributes

Attribute IDs

When a class definition is added to the schema, its components are given sequential, one-based indexes, called *attribute IDs*. A given attribute retains the same ID permanently, even if its position changes as the class definition evolves.

You should not need to work directly with attribute IDs; however, if an attribute's name has evolved unpredictably, the ID could conceivably be the only way to positively determine an attribute's identity.

If you need to work with attribute IDs, remember that they are independent of the attribute positions. For example, if a given class has three attributes, their positions will be 0, 1, and 2. If the class has undergone considerable evolution, however, their IDs might be 15, 3, and 28.

Class Position

A component has a *class position* within every class that defines or inherits that component. The class position for a given attribute is different in its defining class and in the classes that inherit the attribute.

Class position indicates nesting of attributes within the data inherited from each ancestor class. Conceptually, a class position is a sequence of attribute positions; the sequence defines a path through the class's inheritance graph. The first number in the sequence is an attribute position within the class itself. If that position corresponds to an embedded parent class, the second number in the sequence is an attribute position within the parent class. If that position

corresponds to an embedded grandparent class, the third number in the sequence is an attribute position within the grandparent class, and so on.

Because a class does not inherit attributes from an embedded class, class position is one characteristic in which parent classes differ from embedded-class attributes. Whereas attributes of a parent class have class positions within the child class, the attributes of an embedded class do not have class positions within the embedding class. For example, the attributes `e` and `f` of the embedded class `Info` do not have class positions in the embedding class `Test`.

Figure 2-3 illustrates the class position of each attribute defined in, or inherited by, the class `Test`. Below each attribute in the figure is its class position in the `Test` class, shown as a sequence of numbers enclosed in parentheses.

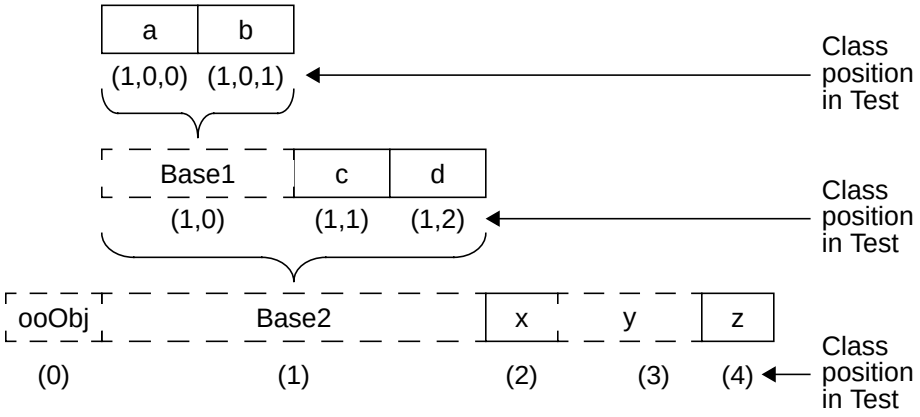

Figure 2-3 Class Positions of Defined and Inherited Attributes

The class position for a property defined in the class or for an immediate parent class can be converted to an integer attribute position; the class position for an inherited property of a parent class cannot.

Obtaining a Class Descriptor

You can obtain a descriptor for a class either from a module descriptor for the module that contains the class or from a module descriptor for the top-level module.

To look up a class by name or type number:

- Call `resolve_class` to get a class descriptor (`d_Class`) for the class.
- Call `resolve_type` to get a type descriptor (`d_Type`) for the class.

To look up a class by name:

- Call `resolve` to get a generic descriptor for the class.

To get an iterator for classes in the module's scope:

- Call `defines_types_begin` to get an iterator that returns type descriptors for all types (both classes and property types) in the module's scope.
- Call `defines_begin` to get an iterator that returns generic descriptors for all modules, classes, and property types in the module's scope.

All entities in the scope of a named module are classes, so if you get an iterator from a named module's descriptor, you can be sure that the returned iterator finds descriptors for classes only. However, if you get an iterator from the top-level module's descriptor, the returned iterator finds descriptors for all named modules, classes, and property types in the schema.

You can call the `is_class` member function of a generic descriptor or a type descriptor to test whether it describes a class. When you are sure that a generic descriptor or a type descriptor describes a class, you can cast it to a class descriptor.

Getting Information About a Class

You can call member functions of a class descriptor to get information about its described class.

Identifying Information

To get identifying information about the described class:

- Call the inherited `name` member function to get the name of the class.
- Call either of the following inherited member functions to get the module that defines the class:
  - Call `defined_in_module` to get a module descriptor for the module.
  - Call `defined_in` to get the containing scope (`d_Scope`); because the scope of every class is a module, you can cast the returned scope to a module descriptor.
- Call `id` to get the unique ID that identifies the class, namely its type number.
- Call `type_number` to get the unique type number for the class. Unlike `id`, this member function returns an `ooTypeNumber`.

Kind of Class

To find out what kind of class is described:

- Call `persistent_capable` to test whether the class is persistence-capable.
- Call `is_internal` to test whether the class is an internal Objectivity / DB class.

- Call `is_string_type` to test whether the class is a recognized string class: `ooVString`, `ooUtf8String`, `ooSTString`, or `ooString(N)`. String classes are treated specially when accessing persistent data; see Chapter 3.

EXAMPLE This example gets a descriptor for the top-level module, looks up a module named `Inventory`, then prints the names of all classes defined in the latter module, indicating the ones that are persistence-capable. To do so, it iterates over all types in the scope of the `Inventory` module. Every such type is guaranteed to be a class because named modules contain only classes, not property types.

```
ooTrans trans;
trans.start();
...
// Get a descriptor for the top-level module
const d_Module &topMod = d_Module::top_level();

// Get a descriptor for the module named "Inventory"
const d_Module &invMod = topMod.resolve_module("Inventory");

// Check for a valid descriptor
if (invMod) {
   // invMod is valid
   // Get an iterator for all classes in the Inventory module
   type_iterator itr = invMod.defines_types_begin();
   while (itr != invMod.defines_types_end()) {
      if ((*itr).is_class()) {
         // Cast current type descriptor to a class descriptor
         const d_Class &curClass = (const d_Class &)*itr;
         cout << curClass.name();
         if (curClass.persistent_capable()) {
            cout << " (persistence capable)";
         }
         cout << endl;
      } // End if current type is a class
      ++itr;
   } // End while more classes
} // End if valid
else {
   // invMod is a null descriptor
   cout << "Inventory module not found" << endl;
} // End else null descriptor
...
trans.commit();
```

Components

To get information about the components of the described class:

- Call `number_of_attributes` to get number of components in the class.
- Call `has_base_class` to test whether the described class is derived from the specified base class.
- Call `position_in_class` to get the class position (`Class_Position`) of the specified property within the described class.

Physical Layout

To get information about the physical layout of an instance of the described class:

- Call `has_extent` to test whether the class has a nonzero physical size.
- Call the inherited `dimension` member function to get the layout size for an instance of the class.
- Call `has_virtual_table` to test whether the physical layout of the class includes space for a virtual-table pointer.

Version and Shape

To get information about the version and shape of the described class:

- Call `version_number` to get the version number for the class.
- Call `latest_version` to get a class descriptor for the latest version of the class.
- Call `shape_number` to get the shape number for the class. If the class's shape number is the same as its type number, the class descriptor provides information about the original shape for the particular version of the particular class.
- Call `next_shape` to get a class descriptor for the next shape of the class.
- Call `previous_shape` to get a class descriptor for the previous shape of the class.

Inheritance Connections

An inheritance descriptor (`d_Inheritance`) provides information about the inheritance connection between a parent (or base) class in the schema and an immediate child (or derived) class.

You call member functions of a class descriptor to find immediate parent classes and child classes of the described class:

- Call `base_class_list_begin` to get an iterator that returns inheritance descriptors for all inheritance connections from an immediate parent class to

the described class. To get the parent class from one of these inheritance descriptors, call its `derives_from` member function.

- Call `sub_class_list_begin` to get an iterator that returns inheritance descriptors for all inheritance connections from the described class to an immediate child class. To get the child class from one of these inheritance descriptors, call its `inherits_to` member function.

You can call member functions of the inheritance descriptor to get additional information:

- Call `access_kind` to get access kind (public, protected, or private) of the parent class as specified in the declaration of the child class.
- Call `position` to get the layout position of data for the parent class within the storage of a persistent instance of the child class.

EXAMPLE The function `showInheritance` prints the parent classes and child classes of a particular class. If the class does not have public access to a parent class, the access kind is shown.

```
void showInheritance(const d_Class &aClass) {

   // Print parent classes, if any
   inheritance_iterator itr = aClass.base_class_list_begin();
   if (itr != aClass.base_class_list_end()) {
      cout << "Parent classes of " << aClass.name();
      cout << ":" << endl;
      while (itr != aClass.base_class_list_end()) {
         const d_Inheritance &curInh = *itr;
         const d_Class &curParent = curInh.derives_from();
         cout << curParent.name();
         d_Access_Kind access = curInh.access_kind();
         if (access == d_PROTECTED) {
            cout << " (protected)";
         }
         else if (access == d_PRIVATE) {
            cout << " (private)";
         }
         cout << endl;
         ++itr;
      } // End while more parents
      cout << endl;
   } // End if any parents
   else {
      cout << aClass.name() << " has no parent classes";
      cout << endl << endl;
   } // End else no parents
```

```
   // Print child classes, if any
   itr = aClass.sub_class_list_begin();
   if (itr != aClass.sub_class_list_end()) {
      cout << "Child classes of " << aClass.name();
      cout << ":" << endl;
      while (itr != aClass.sub_class_list_end()) {
         const d_Inheritance &curInh = *itr;
         const d_Class &curChild = curInh.inherits_to();
         cout << curChild.name() << endl;
         ++itr;
      } // End while more child classes
      cout << endl;
   } // End if any child classes
   else {
      cout << aClass.name() << " has no child classes";
      cout << endl << endl;
   } // End else no child classes
} // End showInheritance
```

Examining Properties

To examine a particular property of a class, you obtain a descriptor for the property and call the descriptor's member functions to get information about the property.

Obtaining a Property Descriptor

Once you have a class descriptor, you can call its member functions to get descriptors for its properties.

- To look up a property by name:
  - Call `resolve_attribute` to get an attribute descriptor (`d_Attribute`) for an attribute or a relationship.
  - Call `resolve_relationship` to get a relationship descriptor (`d_Relationship`) for a relationship.
  - Call `resolve` to get a generic descriptor for an attribute or a relationship.
- To look up a property by position, call `attribute_at_position` to get an attribute descriptor for an attribute or a relationship.
- To look up a property by attribute ID, call `attribute_with_id` to get an attribute descriptor for an attribute or a relationship.
- To get an iterator for properties in the class's scope:
  - Call `defines_attribute_begin` to get an iterator that returns attribute descriptors for all attributes and relationships.
  - Call `defines_relationship_begin` to get an iterator that returns relationship descriptors for all relationships.
  - Call `defines_begin` to get an iterator that returns generic descriptors for all attributes and relationships.
- To get an iterator for inherited attributes and relationships:
  - Call `attributes_plus_inherited_begin` to get an iterator that returns attribute descriptors for all attributes and relationships in the described class, whether they are defined in that class or inherited.
  - Call `base_classes_plus_inherited_begin` to get an iterator that returns attribute descriptors for all ancestor classes of the described class, whether they are immediate base classes or inherited. (Remember that the base classes of a class are described as if they were embedded-class attributes.)

  By default, the iterators returned by `attributes_plus_inherited_begin` and `base_classes_plus_inherited_begin` treat the Objectivity / C++ persistent-object base class `ooObj` and the storage-object classes `ooContObj`, `ooDBObj`, and `ooFDObj` as if they were root base classes, inheriting from no

other classes. You can override this behavior, allowing access to internal ancestor classes at all levels.

- To enable access to ancestors of Objectivity/C++ persistent-object and storage-object classes, call the static member function `d_Class::enable_root_descent`.
- To disable access to ancestors of Objectivity/C++ persistent-object and storage-object classes, call the static member function `d_Class::disable_root_descent`.
- To test whether access is enabled to ancestors of Objectivity/C++ persistent-object and storage-object classes, call the static member function `d_Class::root_descent_is_enabled`.

Testing the Kind of Property

Many member functions that get a descriptor for a property return either a generic descriptor or an attribute descriptor, which may describe either an attribute or a relationship. You can test the kind of described property and, if necessary, cast the descriptor to a more specific class.

- If you obtain an attribute descriptor, you can call its `is_relationship` member function to test whether the described entity is a relationship. If so, you can cast the descriptor to a relationship descriptor; if not, the described entity is an attribute.
- If you obtain a generic descriptor from a class scope, you know the described entity must be an attribute or a relationship. You can cast the generic descriptor to an attribute descriptor, then call the attribute descriptor's `is_relationship` member function to test whether you should cast it to a relationship descriptor.

Getting Information About any Property

You can call member functions of an attribute descriptor or a relationship descriptor to get information that is common to all properties.

Identifying Information

To get identifying information about the described property:

- Call the inherited `name` member function to get the name of the property.
- Call either of the following inherited member functions to get the class in which the property is defined:
  - Call `defined_in_class` to get a class descriptor for the class.
  - Call `defined_in` to get the containing scope; because the scope of every property is a class, you can cast the returned scope to a class descriptor.

- Call `id` to get the attribute ID that permanently identifies the property within its class.
- Call `position` to get the attribute position of the property within its defining class.

Type and Access Kind

To get the type and access kind of the described property:

- Call `type_of` to get a type descriptor for the type of value that the property contains. You can call the type descriptor's member function to get additional information about the type of the attribute; see "Examining Types" on page 47.
- Call `access_kind` to get the access kind (public, protected, or private) of the property.

Layout Size

To get information about the layout size of the described property:

- Call `array_size` to get the number of elements in the fixed-size array of values for the described property (or one if the property contains a single value instead of an array).
- Call `element_size` to get the physical layout size for a single value of the property's type.
- Call `dimension` to get the physical layout size of the property.

For all properties the following is true:
`dimension = elementSize * arraySize`

Relationships always have a single value, not an array of values, so for a relationship, the dimension is always the same as the element size.

EXAMPLE This example shows part of the function `showProperties`, which prints a brief description of every property of a class. This function iterates through the properties of the class. It prints information that is common to all properties, then tests the kind of property and casts the descriptor for any relationship to a relationship descriptor. Appendix B, "Programming Examples," contains the complete definition of `showProperties`.

```
void showProperties(const d_Class &aClass) {
   cout << aClass.name() << " Properties:" << endl;

   // Iterate through all properties (defined and inherited)
   attribute_plus_inherited_iterator itr =
      aClass.attributes_plus_inherited_begin();
```

```
   while (itr != aClass.attributes_plus_inherited_end()) {
      // Get descriptor for current property
      const d_Attribute &curAttr = *itr;
      // Print property name
      cout << endl << curAttr.name();

      // Test whether property is inherited
      const Class_Position pos =
         aClass.position_in_class(curAttr);
      if (! pos.is_convertible_to_uint()) {
         cout << " (inherited)";
      }

      // Describe the property
      if (curAttr.is_relationship()) {
         const d_Relationship &rel =
            (const d_Relationship &)curAttr;
         ... // Describe the relationship
      } // End if property is a relationship
      else {
         ... // Describe the attribute
      } // End else property is an attribute
      cout << endl;
      ++itr;
   } // End while more properties
   cout << endl;
} // End showProperties
```

Getting Information About an Attribute

You can call member functions of an attribute descriptor to get information that is specific to attributes (as opposed to relationships):

- Call `has_default_value` to test whether the described attribute has a default value. A default value is relevant only for numeric attributes. It is used if an earlier shape of the same class did not include this attribute. When persistent objects of the earlier shape are converted to the new shape, they are given the default value for the new numeric attribute.
- If the attribute has a default value, call `default_value` to get the default value.
- Call `is_base_class` to test whether the described attribute is a base class.
- If the attribute is an embedded-class attribute or a base class, call `class_type_of` to get a class descriptor for the embedded class or base class.

Getting Information About a Relationship

You can call member functions of a relationship descriptor to get information that is specific to relationships (as opposed to attributes).

- Call `other_class` to get the relationship's destination class.
- Call `is_bidirectional` to test whether the relationship is bidirectional.
- If the relationship is bidirectional, call `inverse` to get a relationship descriptor for the inverse relationship.
- Call `is_to_many` to test whether the relationship is to-many.
- Call `copy_mode` to get the relationship's copy mode.
- Call `versioning` to get the relationship's versioning mode.
- Call `propagation` to get the relationship's propagation behavior.
- Call `is_inline` to test whether the relationship is inline.
- Call `is_short` to test whether source objects store references to their associated destination objects as short object references.

EXAMPLE This example shows how the function `showProperties` describes a relationship.

```
…
if (curAttr.is_relationship()) {
   // Property is a relationship
   const d_Relationship &rel =
      (const d_Relationship &)curAttr;
   // Test whether the relationship is to-many
   if (rel.is_to_many()) {
      cout << " [to-many]";
   }
   cout << ":" << endl;

   if (rel.is_bidirectional()) {
      cout << "  bidirectional relationship to ";
      cout << rel.otherClass().name() << "; inverse: ";
      cout << rel.inverse().name();
   } // End bidirectional
   else {
      cout << "  unidirectional relationship to ";
      cout << rel.otherClass().name();
   } // End unidirectional
} // End if property is a relationship
…
```

Examining Types

To examine a particular type, you obtain a descriptor for the type, test the kind of data type it describes, cast it to the appropriate descriptor class, and call the descriptor's member functions to get information about the type.

Obtaining a Type Descriptor

You can obtain a descriptor for any type in the schema by calling member functions of a descriptor for the top-level module:

- To look up a type by name or type number:
  - Call `resolve_type` to get a type descriptor for the type.
  - Call `resolve` to get a generic descriptor for the type.
- To look up a type by name, call `resolve` to get a generic descriptor for the type.
- To get an iterator for all types:
  - Call `defines_types_begin` to get an iterator that returns type descriptors for all classes and property types.
  - Call `defines_begin` to get an iterator that returns generic descriptors for all modules, classes, and property types.

In addition, you can obtain a type descriptor for the type of a particular property of a class. To do so, call the `type_of` member function of a property descriptor for the property of interest.

If you obtain a generic descriptor, you can call its `is_type` member function to test whether it describes a type. If so, you can cast the generic descriptor to a type descriptor.

Testing the Kind of Data Type

A type descriptor can describe either a class or a property type. You can call its inherited `is_class` member function to test whether it describes a class. If so, you can cast the descriptor to a class descriptor and examine the described class as discussed in "Getting Information About a Class" on page 37.

If a type descriptor contains information about a property type, you must first determine the kind of data type it describes. You may then need to cast the descriptor to a more specific descriptor class before you can call member functions that get information about the described type.

Figure 2-4 shows the inheritance graph for property-type descriptor classes.

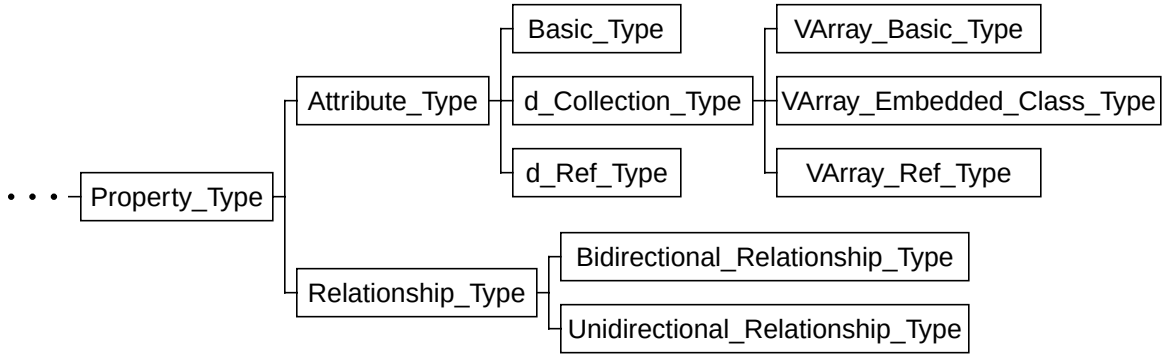

Figure 2-4 Property-Type Descriptor Classes

You can call member functions of a property-type descriptor to determine the kind of data type it describes.

Relationship Types

If `is_relationship_type` returns true, the described type is a relationship type. If `is_unidirectional_relationship_type` returns true, it is a unidirectional relationship type; if `is_bidirectional_relationship_type` returns true, it is a bidirectional relationship type. In all three cases, you can cast the type descriptor to a relationship-type descriptor (`Relationship_Type`) if you need to get information about the described relationship type.

Attribute Types

If both `is_class` and `is_relationship_type` return false, the described type is an attribute type and you can call addition member functions to determine what kind.

If `is_basic_type` returns true, the described type is a numeric type; you can cast the type descriptor to a numeric type descriptor (`Basic_Type`).

If `is_ref_type` returns true, the described type is an object-reference type; you can cast the type descriptor to a reference-type descriptor (`d_Ref_Type`).

If is_varray_type returns true, the described type is a VArray type.

- If is_varray_basic_type returns true, the described type is a numeric-VArray type; you can cast the type descriptor to a numeric-VArray type descriptor (VArray_Basic_Type).
- If is_varray_ref_type returns true, the described type is an object-reference-VArray type; you can cast the type descriptor to an object-reference-VArray type descriptor (VArray_Ref_Type).
- If is_varray_embedded_class_type returns true, the described type is an embedded-class-VArray type; you can cast the type descriptor to an embedded-class-VArray type descriptor (VArray_Embedded_Class_Type).

Getting Information About a Property Type

You can call member functions of a property-type descriptor to get information that is common to all property types.

- To get identifying information about the described type:
  - Call the inherited name member function to get the name of the type.
  - Call id to get the unique ID that identifies the type, namely its type number.
  - Call type_number to get the unique type number for the type. Unlike id, this member function returns an ooTypeNumber.
- Call the inherited dimension member function to get the layout size for a value of the type.

Additional information is available for different kinds of data types. The following table summarizes the kinds of information available for property types of various kinds.

Property Type	Information
Numeric type	Kind of numeric data in a value
Object-reference type	Referenced class Whether values are stored as short object references
Any collection type	Element type
Numeric VArray type	Kind of numeric data in an element
Object-reference VArray type	Object-reference type of an element
Embedded-class VArray type	Class of an element
Relationship type	Destination class

Refer to the descriptions of the individual property-type descriptor classes in Part 2 for details about the member functions that get information about the described property type.

EXAMPLE This example shows how the function `showProperties` describes the type of an attribute.

```
char textbuf[64];
...
// Describe the type of the attribute
const d_Type &curType = curAttr.type_of();

if (curType.is_basic_type()) {
   // Property is a numeric attribute
   const Basic_Type &bt =
      (const Basic_Type &)curType;
   cout << "  " << base_type_to_text(bt.base_type(), textbuf);
} // End numeric attribute

else if (curType.is_ref_type()) {
   // Property is an object-reference attribute
   const d_Ref_Type &ref =
      (const d_Ref_Type &)curType;
   cout << "  reference to ";
   cout << ref.referenced_type().name();
} // End object-reference attribute

else if (curType.is_class()) {
   // Property is an embedded attribute or a base class
   if (curAttr.is_base_class()) {
      cout << "  base class";
   } // End base class
   else {
      cout << "  embedded instance of ";
      cout << curType.name();
   }
} // End embedded or base class

else if (curType.is_varray_type()) {
   // Property is a VArray attribute
   cout << "  Varray of ";
   // Describe the element type
```

```
   if (curType.is_varray_basic_type()) {
      // Property is a numeric VArray attribute
      const VArray_Basic_Type &vbt =
         (const VArray_Basic_Type &)curType;
      cout << base_type_to_text(vbt.element_base_type(),
                                textbuf);
   } // End numeric VArray

   else if (curType.is_varray_ref_type()) {
      // Property is an object-reference VArray attribute
      const VArray_Ref_Type &vref =
         (const VArray_Ref_Type &)curType;
      const d_Ref_Type &ref = vref.element_ref_type();
      cout << "reference to ";
      cout << ref.referenced_type().name();
   } // End object-reference VArray

   else if (curType.is_varray_embedded_class_type()) {
      // Property is embedded-class VArray attribute
      const VArray_Embedded_Class_Type &vembd =
         (const VArray_Embedded_Class_Type &)curType;
      cout << "embedded instances of ";
      cout << vembd.element_class_type().name();
   } // End embedded-class VArray

   else {
      cout << "  unrecognized VArray type";
   }
} // End VArray attribute

else {
   cout << "  unrecognized attribute type";
}
...
```

Descriptions of numeric attributes and numeric VArray attributes use the function `base_type_to_text`, which converts a code of type `ooBaseType` to a text description of the corresponding numeric type. The function definition follows; repetitive details have been omitted. Appendix B, “Programming Examples,” contains the complete definition of `base_type_to_text`.

```
char *base_type_to_text(ooBaseType bt, char *textbuf) {
   switch (bt) {
      case ooCHAR: {
         sprintf(textbuf, "8-bit character");
         break;
      }
      case ooINT8: {
         sprintf(textbuf, "8-bit signed integer");
         break;
      }
      …
      case ooFLOAT64: {
         sprintf(textbuf,
                  "double-precision floating-point number");
         break;
      }
      case ooPTR: {
         sprintf(textbuf, "32-bit pointer");
         break;
      }
      default: {
         sprintf(textbuf, "unrecognized numeric type");
         break;
      }
   } // End switch
   return textbuf;
} // End base_type_to_text
```

Finding Entities That Use a Type

Once you have a type descriptor (for a class or a property type), you can call its member functions to get descriptors for entities that use the described type.

- Call `used_in_property_begin` to get an iterator that returns property descriptors (`d_Property`) for all properties in the schema that use the described type.
- Call `used_in_collection_type_begin` to get an iterator that returns collection-type descriptors (`d_Collection_Type`) for all collection types in the schema that are created from the described type.
- Call `used_in_ref_type_begin` to get an iterator that returns reference-type descriptors (`d_Ref_Type`) for all object-reference types in the schema that reference the described type.

EXAMPLE This example shows all properties that use the persistence-capable class `Library`, namely, properties that contain object-references to `Library` or VArrays of object-references to `Library`.

```
ooTrans trans;
trans.start();
...
// Get a descriptor for the class Library
const d_Class &lib = topMod.resolve_class("Library");
if (lib) {
   showUses(lib);
}
...
trans.commit();
```

The function `showUses` lists all properties that use a particular class. The relevant portion of this function follows. Appendix B, "Programming Examples," contains the complete definition of `showUses`.

```
void showUses(const d_Class &aClass) {
   const char *name = aClass.name();
   cout << "Properties that use the class ";
   cout << name << endl;

   // Iterate through all properties that use the class
   property_iterator itr = aClass.used_in_property_begin();
   while (itr != aClass.used_in_property_end()) {
      const d_Property &curProp = *itr;
      cout << "  " << curProp.name() << " of ";
      cout << curProp.defined_in_class().name();
      const d_Type &curType = curProp.type_of();

      if (aClass.persistent_capable()) {
         // Every property that uses a persistence-capable
         // class should be either an object-reference
         // attribute or an object-reference VArray attribute

         if (curType.is_ref_type()) {
            // Property is an object-reference attribute
            // Check whether it uses short or standard references
            const d_Ref_Type &ref =
               (const d_Ref_Type &)curType;
            if (ref.is_short()) {
               cout << " ooShortRef(";
            }
```

```
                else {
                    cout << " ooRef(";
                }
                cout << name << ")" << endl;
            } // End object-reference type

            else if (curType.is_varray_ref_type()) {
                // Property is an object-reference VArray
                // attribute
                // Check whether it uses short or standard
                // references
                const VArray_Ref_Type &vref =
                    (const VArray_Ref_Type &)curType;
                const d_Ref_Type &ref = vref.element_ref_type();
                if (ref.is_short()) {
                    cout << " ooVArray(ooShortRef(";
                }
                else {
                    cout << " ooVArray(ooRef(";
                }
                cout << name << "))" << endl;
            } // End object-reference VArray

            else {
                cout << " unexpected type" << endl;
            }
        } // End if persistence-capable
        else {
            ...
        } // End else non-persistence-capable
        ++itr;
    } // End while more properties
} // End showUses
```

Locking the Schema

Active Schema allows you to lock the schema of a federated database against unauthorized access. Once the schema has been locked, no process can access the schema or unlock it without the correct key. After the schema has been unlocked, no process can relock it or change its key without the key that unlocked it.

- To lock the schema, call the static member function `d_Module::lock_schema`. The parameter is the key that can be used to access or unlock the schema in the future.
- To access a locked schema, pass its key as the parameter to the static member function `d_Module::top_level`.
- To unlock the schema, pass its key as the parameter to the static member function `d_Module::unlock_schema`.
- To relock the schema, call the static member function `d_Module::lock_schema`, passing as the key the same key that was used to unlock it. If you want to change the key, specify the new key as the first parameter and the previous key as the second parameter.

EXAMPLE This example locks the schema, setting its key to `112233446677`, then accesses the locked schema.

```
ooTrans trans;
trans.start();
...
// Lock the schema
d_Module::lock_schema(11223344556677);

// Access the locked schema
const d_Module &topMod = d_Module::top_level(11223344556677);
```

The following statement unlocks the schema:

```
d_Module::unlock_schema(11223344556677);
```

The following statement relocks the schema, changing its key to `99887766554433`.

```
d_Module::lock_schema(
   99887766554433,    // New key
   11223344556677);   // Old key
```

3

Examining Persistent Data

Active Schema applications can examine the persistent objects in any Objectivity / DB federated database—even without C++ definitions of the objects' classes. You might use these capabilities to develop an object browser for Objectivity / DB federated databases.

In This Chapter

Persistent-Data Objects

Active Schema provides access to data in the federated database using *persistent-data objects*. A persistent data object is self-describing—that is, it contains both persistent data and information about the structure and content of that data. Different persistent-data classes provide access to persistent data of different kinds.

Figure 3-1 shows the persistent-data classes.

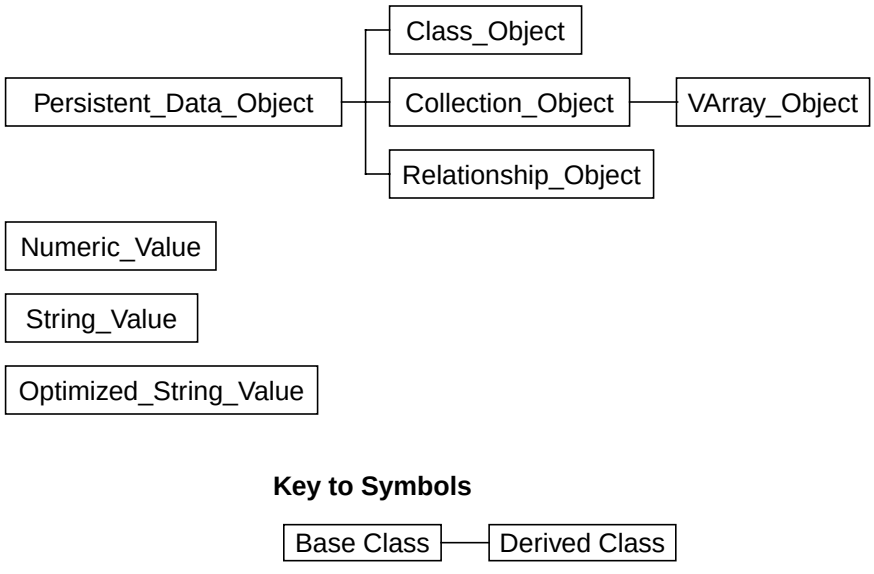

Figure 3-1 Persistent-Data Classes

Access to Persistent Data

Access to persistent data begins with a class object (`Class_Object`) that has a handle to a persistent object, called its *associated persistent object*. The class object uses a class descriptor for the persistent object's class to describe the object's data. The class of the persistent object is called the class object's *described class*.

The class object uses other persistent-data objects to provide access to the data in most properties of its associated persistent object:

- The value of a numeric attribute is available in a numeric value (`Numeric_Value`). A numeric value contains a code indicating the data type. An application can examine a numeric value's code to determine the type of numeric data it contains and then convert it to the appropriate C++ numeric data type.

- The value of an object-reference attribute is directly available through the class object. However, the referenced object is accessible only through another class object for that referenced object.
- The data for an instance of a non-persistence-capable class in an embedded-class attribute is returned in one of two forms.
  - If the embedded class is a recognized string class, the embedded instance is available in a string value (`String_Value`). A string value contains a code indicating the kind of string. An application can examine a string value's code to determine the kind of string and then convert the string value to the appropriate string class.
    - If the string is an instance of an Objectivity/C++ string class, such as `ooVString`, the string value can be converted directly to that class.
    - If the string is an instance of an application-defined optimized string class `ooString(`*`N`*`)`, the string value can be converted to an optimized string value (`Optimized_String_Value`), which lets an Active Schema application access the string even though it does not contain a definition of the string's class.
  - Otherwise, the embedded instance is available in a class object.
- The value of a VArray attribute is available in a VArray object (`VArray_Object`).
- The associated destination objects for a relationship are available in a relationship object (`Relationship_Object`).

If a particular attribute contains a fixed-size array of values, each element of the array is accessed through its own persistent-data object. If an attribute contains a VArray (or a fixed-size array of VArrays), each element of each VArray is accessed through its own persistent-data object.

For example, consider the following class definitions:

```
class Base1 {
   int32 a[10];
   ooVString b; };
class Base2 : public Base1 {
   ooVArray(uint16) c;
   float32 d; };
class Info {
   char e;
   ooVArray(ooVString) f; };
class Test : public ooObj, Base2 {
   ooRef(Test) x : copy(delete);
   Info y;
   ooRef(Test) z; };
```

Figure 3-3 illustrates the persistent-data objects that access the persistent data for an instance of the class `Test`.

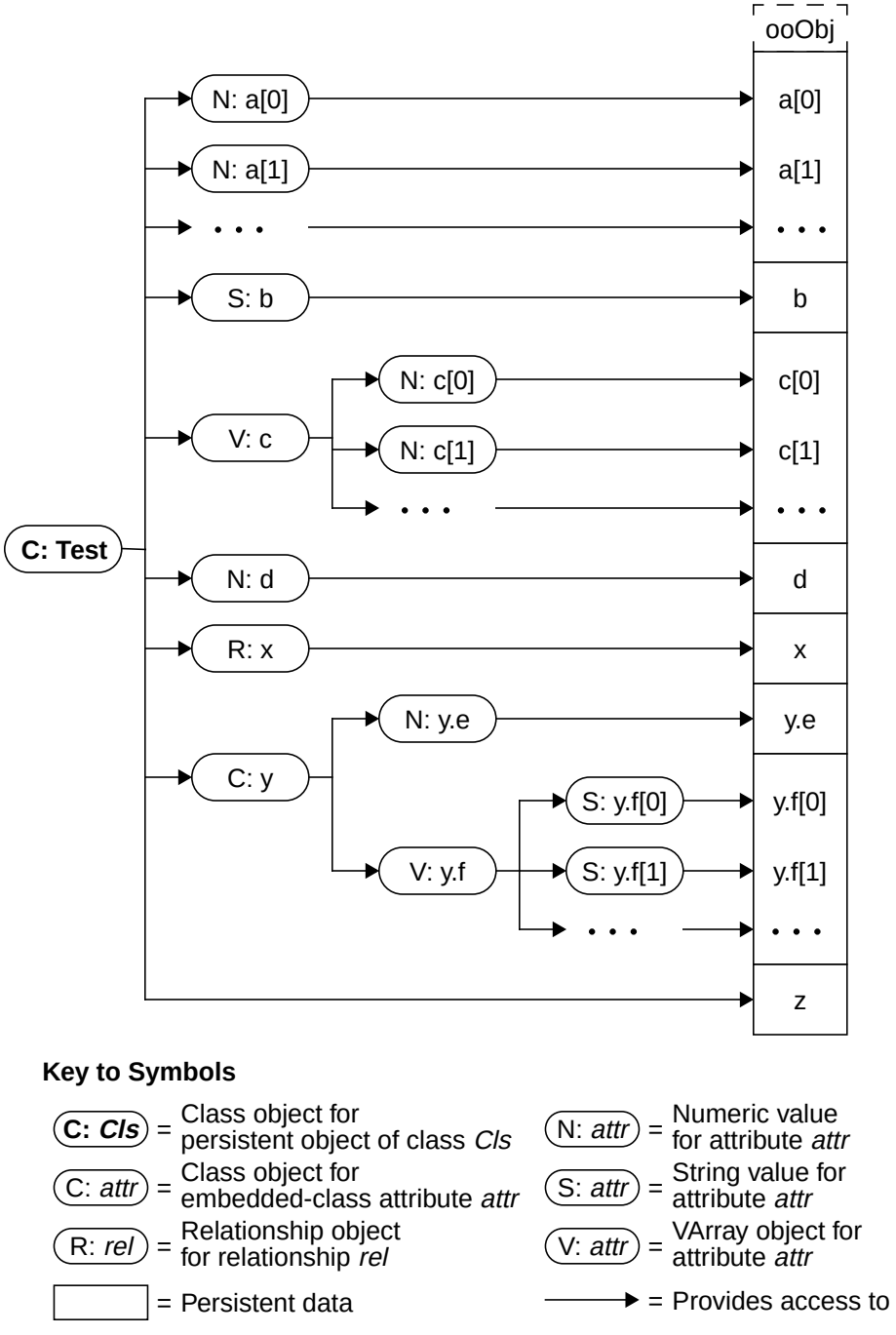

Figure 3-2 Access to a Persistent Object's Data

All persistent-data objects in Figure 3-2 have the same associated persistent object, namely, the instance of `Test` whose persistent data is illustrated at the right of the figure. All persistent-data objects are *contained-in* the class object for the instance of `Test`.

Direct and Indirect Access

A class object provides *direct* access to every property defined by its described class. It can provide *indirect* access to all the inherited properties of its described class by traversing the inheritance hierarchy. If your application does not repeatedly access inherited properties of a persistent object, you can access all properties (defined or inherited) through the class object for the associated persistent object.

If your application repeatedly accesses inherited properties, however, the most efficient approach is to access each inherited property through a class object for the ancestor class that defines the property. Following this approach, you obtain a class object for each base class, just as you do for an embedded-class attribute.

Figure 3-3 illustrates the persistent data objects that access an instance of the class `Test`, using a class object for each application-defined base class of the class `Test`. Instead of accessing data for an inherited property indirectly through the class object for the instance of `Test`, the application accesses it through the class object for the base class (`Base1` or `Base2`) that defines the property. All persistent-data objects in Figure 3-3 have the same associated persistent object, namely, the instance of `Test` whose persistent data is illustrated at the right of the figure. All persistent-data objects are contained-in the class object for the instance of `Test`.

NOTE Programs typically do not need to use class objects describing the Objectivity/C++ base classes `ooObj`, `ooContObj`, and `ooGCContObj`.

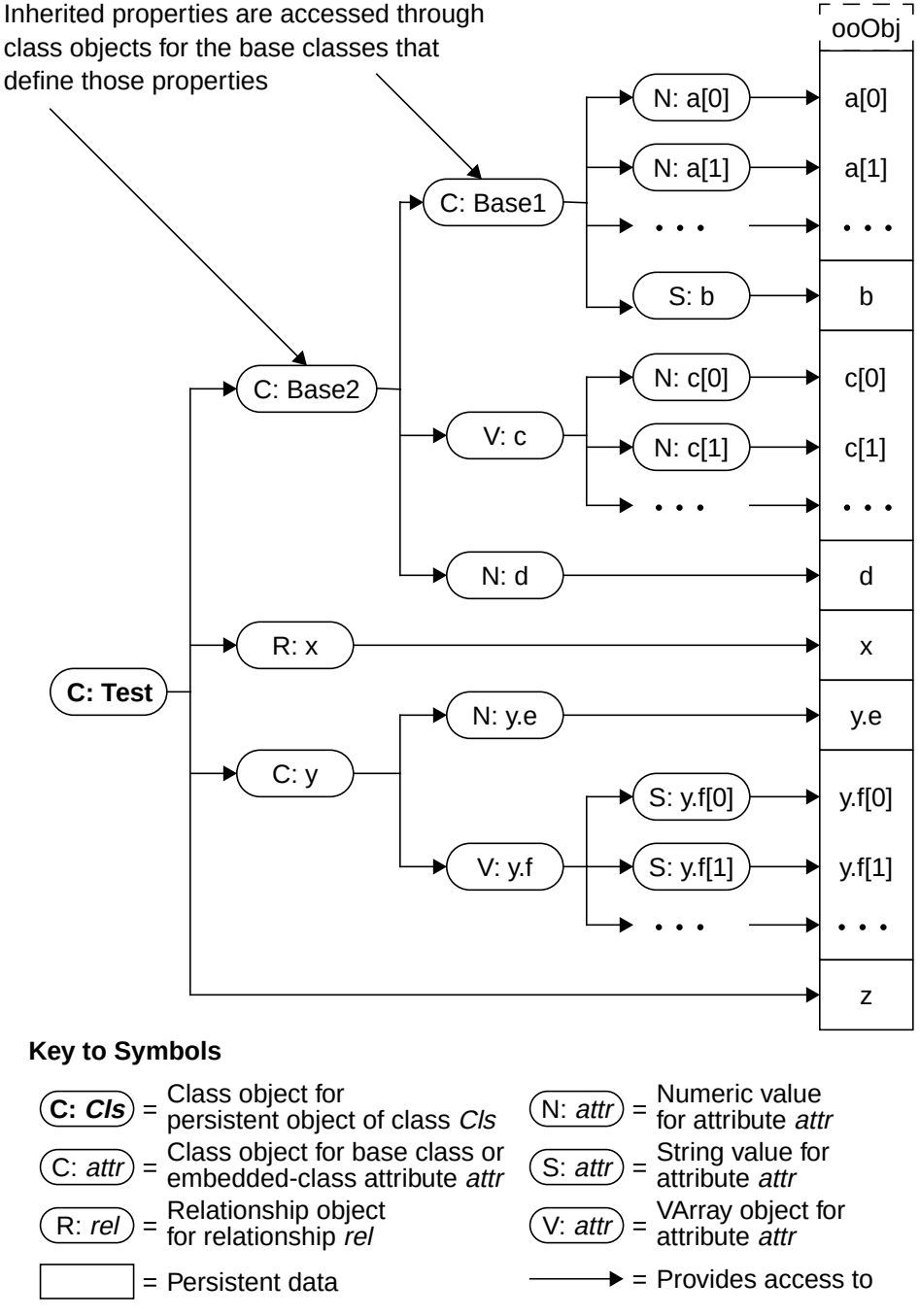

Figure 3-3 Using Class Objects for Base Classes

Examining A Persistent Object

If you want to examine a persistent object in a federated database, you first obtain a handle or an object reference for the object just as you would with any Objectivity/C++ application. For example, you might iterate through the objects in a particular database.

To examine the data for a persistent object, you perform these steps:

1. Construct a class object for the persistent object from its handle or object reference.
2. If necessary, call member functions of the class object or its associated class descriptor to find out what properties the object has.
3. Call member functions of the class object to get data for a property or to get another persistent-data object that can access the property's data.

NOTE If your application passes class objects as parameters to, or return values from, functions, the class objects should be passed and returned by reference wherever possible.

Constructing a Class Object

You can construct a class object for an existing persistent object using either a handle or a reference to that persistent object. If necessary, the constructor opens a handle for the persistent object, giving your application access to its persistent data.

EXAMPLE This example iterates through the objects in a particular database, getting a class object for each.

```
ooTrans trans;
trans.start();
...
ooHandle(ooDBObj) dbH;
... // Set dbH
ooItr(ooObj) objItr;
// Initialize an object iterator for persistent objects
// in the database
objItr.scan(dbH);
while (objItr.next()) {
   // Get a handle for the current object
   ooHandle(ooObj) curObjH(objItr);
```

```
   // Construct a class object to access the current object
   Class_Object curCO(curObjH);
   // Examine the object's data through class object curCO
   showData(curCO);
} // End while more objects in database
...
trans.commit();
```

The `Class_Object` constructor sets the new class object's class descriptor automatically. It gets the type number of the persistent object's class and looks up that type number in the top-level module.

If you know the class of a persistent object, you can specify its class with optional parameters to the constructor.

Getting Information About a Class Object

You can call a class object's `type_of` member function to get its class descriptor.

If a class object corresponds to a base class or an embedded-class attribute, you can call its `contained_in` member function to get the class object for the associated persistent object.

You can get a handle for the associated persistent object by calling a class object's `object_handle` member function. In addition, conversion operators allow you to convert a class object to a handle to the associated persistent object or to an object reference to the associated persistent object.

Identifying Components

To get data for a particular component of a persistent object, you identify the component by its position in the class object's described class.

- You can use attribute position to identify immediate base classes and properties defined by the described class. The class object has *direct* access to these components.
- You can use class position to identify any component, including ancestor classes and inherited properties.

  The class object has *indirect* access to inherited components. If you need to access many of them, consider doing so through class objects for their defining classes.

The form in which data is returned depends on the type of the component. If necessary, you can get information about a component from an attribute descriptor that describes it.

If you know the name of a component in the described class, you can call the `position_in_class` member function of a class object to finds its class position. You can call the `resolve_attribute` member function to get an attribute descriptor for the component.

EXAMPLE In this example, the class object provides access to an object of a class that is known to have a `length` attribute.

```
ooTrans trans;
trans.start();
...
// Set the class object
Class_Object CO = ... ;

// Get an attribute descriptor for length
const d_Attribute &lenDesc = CO.resolve_attribute("length");
... // Use lenDesc to get information about the attribute


// Get the class position of length
const Class_Position &lenPos = CO.position_in_class("length");
... // Use lenPos to identify the length attribute to CO
...
trans.commit();
```

If you don't know the names of the class's components, you can call member functions of the class object's class descriptor to get attribute descriptors for the components of the object's class. You can call the class descriptor's `position_in_class` member function to get an attribute's class position.

EXAMPLE In this example, the components of the class object are unknown. The application gets the associated class descriptor, then iterates through all components of the described class, getting an attribute descriptor and the class position for each.

```
ooTrans trans;
trans.start();
...
// Set the class object
Class_Object CO = ... ;
// Get the class descriptor
const d_Class &class = CO.type_of();
// Iterate through all properties (defined and inherited)
attributes_plus_inherited_iterator itr =
   class.attributes_plus_inherited_begin();
```

```
while (itr != class.attributes_plus_inherited_end()) {
   const d_Attribute &curAttr = *itr;
   // Use curAttr to get information about the current attribute
   // Get the class position of the current attribute
   const Class_Position &curPos =
      class.position_in_class(curAttr);
   … // Use curPos to identify the current attribute to CO
   ++itr;
} // End while more attributes
…
trans.commit();
```

Alternatively, you might simply iterate through all the attribute positions of the described class, getting the attribute descriptor at each position.

EXAMPLE In this example, the components of the class object are unknown. The application gets the associated class descriptor, then iterates through its attribute positions, getting an attribute descriptor for the component at each position.

```
ooTrans trans;
trans.start();
…
// Set the class object
Class_Object CO = … ;
// Get the class descriptor
const d_Class &class = CO.type_of();
// Iterate through all attribute positions in described class
size_t nComponents = class.number_of_attributes();
for (size_t pos = 0; pos < nComponents; ++pos) {
   // Use pos to identify the current attribute to CO

   // Get a descriptor for the current attribute
   const d_Attribute &curAttr =
      class.attribute_at_position(pos);
   …
} // End for all attribute positions
…
trans.commit();
```

Accessing Component Data

You can call member functions of a class object to access the components of its described class.

- You can get a class object for any base class and use that class object for direct access to the properties defined by the base class.
- You can get the data for an attribute or a persistent-data object that accesses the attribute's data.
- You can get a relationship object that accesses the destination objects associated with the persistent object by a particular relationship.

Different member functions return data or persistent-data objects for components of different kinds. If you know the components of the described class, you can call the appropriate member function for each component. Otherwise, you must use an attribute descriptor for each component to determine what kind of component it is (base class, attribute, or relationship). If the component is a property, you can use the attribute descriptor to determine what type of data the property contains.

Null Objects

A class object, a relationship object, or a VArray object can be null, indicating the lack of data. For example, suppose you get a class object containing the object referenced by an object-reference attribute. If that attribute doesn't contain an object reference, you get a null class object.

You can use a class object, a relationship object, or a VArray object as an integer expression to test whether that object is null. A null object is converted to zero; an object that contains data is converted to nonzero.

Base Classes

Call an attribute descriptor's `is_base_class` member function to test whether the described component is a base class. If so, you can call the class object's `get_class_obj` member function to get a class object that provides direct access to properties defined by that base class.

EXAMPLE This example shows part of the function `showData`, which displays the data for a class object. This function iterates through the components of the object's class. If a component is a base class, the function gets a class object for the base class and recursively shows the values of the properties defined in that class. Appendix B, "Programming Examples," contains the complete definition of `showData`.

```
ooStatus showData (
   Class_Object &CO, // Class object to display
   ooBoolean mbd = oocFalse, // True if base or embedded class
   char *prefix = "") // Prefix for attribute of embedded class
{
   ooStatus rc;

   // Check for null class object
   if (! CO) {
      cout << "(null)" << endl;
      return oocSuccess;
   }

   const d_Class &classOfObj = CO.type_of();
   if (! classOfObj) {
      cerr << "Can't find class of object" << endl;
      return oocError;
   }
   else if (! mbd) {
      cout << endl << "Object of class " << classOfObj.name();
      cout << endl;
   }
   // Iterate through components, showing data for each
   size_t nComponents = classOfObj.number_of_attributes();
   for (size_t pos = 0; pos < nComponents; ++pos) {

      // Get an attribute descriptor for the component
      const d_Attribute &curAttr =
         classOfObj.attribute_at_position(pos);

      // Get the type of the component
      const d_Type &curType = curAttr.typeOf();

      // Determine the kind of component
      if (curAttr.is_base_class()) {
         // Ignore internal base classes like ooObj
         if (! ((const d_Class &)curType).is_internal()) {
            // Recursively show properties of base class
            rc = showData(CO.get_class_obj(pos),
                          oocTrue,
                          prefix);
            if (rc != oocSuccess) {
               return rc;
            }
         } // End if not internal
      } // End if base class
```

```
      else if (curAttr.is_relationship()) {
         ... // Component is a relationship; show its data
      } // End if relationship
      else {
         ... // Component is an attribute; show its data
      } // End else component is attribute
   } // End for all components
   return oocSuccess;
} // End showData
```

Relationships

Call an attribute descriptor's inherited `is_relationship` member function to test whether the described component is a relationship. If so, you can call the class object's `get_relationship` member function to get a relationship object that provides access to the associated destination object(s). "Examining Relationship Data" on page 79 explains how to get data from a relationship object.

Attributes

If a component is neither a base class nor a relationship, it is an attribute. You can call the attribute descriptor's inherited `type_of` member function to get a type descriptor for the attribute's data type. You can call member functions of the type descriptor to find out what kind of data the attribute contains and call the corresponding member function of the class object to access the data.

The following table lists the member functions that get data for the various attribute types.

Type of Attribute	Member Function	Data Returned As
Numeric attribute	`get`	Numeric value
Object-reference attribute	`get_ooref`	Object reference
	`get_class_obj`	Class object
Embedded string class attribute	`get_string`	String value
Embedded non-string class attribute	`get_class_obj`	Class object
VArray attribute	`get_varray`	VArray object

As the third column shows, the data for an attribute is generally returned as a persistent-data object. The one exception is that `get_ooref` returns the value of an object-reference attribute as an object reference of type `ooRef(ooObj)`.

The member functions that get attribute data allow you to specify an index in the fixed-size array that the attribute contains. You can call the attribute descriptor's `array_size` member function to get the number of elements in the fixed-size array of values. If the attribute contains a single value instead of an array, you can omit the index parameter when you get the attribute data.

EXAMPLE This example shows how the function `showData` handles attribute data. Details for attributes of various types appear in subsequent examples.

```
...
else {
   // Component is an attribute
   cout << prefix << curAttr.name();

   // Get the number of values in the fixed-size array
   size_t nVals = curAttr.array_size();
   if (nVals > 1) {
      cout << " (" << nVals << ")";
   }
   cout << ":" << endl;

   // Test the attribute type to determine how to
   // access the attribute's data
   if (curType.is_basic_type()) {
      ... // Show numeric value(s)
   }
   else if (curType.is_ref_type()) {
      // Show the object reference(s)
   }
   else if (curType.is_string_type()) {
      // Show the string value(s)
   }
   else if (curType.is_class()) {
      // Show embedded instance(s)
   }
   else if (curType.is_varray_type()) {
      // Show the varray(s)
   }
   else {
      cout << "unrecognized attribute type" << endl;
   }
} // End else component is attribute
...
```

Numeric Attribute

Call the is_basic_type member function of the attribute's type descriptor to test whether the attribute is numeric. If so, call the class object's get member function to get a numeric value for each element of the attribute's fixed-size array. "Examining Numeric Data" on page 73 explains how to get data from a numeric value.

Object-Reference Attribute

Call the is_ref_type member function of the attribute's type descriptor to test whether the attribute is an object-reference attribute. If so, call the class object's get_ooref member function to get an object reference for each element of the attribute's fixed-size array. That member function does not open a handle to the referenced object. If you want to examine the data of the referenced object, however, you can call get_class_obj to obtain a class object for the referenced object.

EXAMPLE This example shows how the function showData handles object-reference attributes.

```
...
else if (curType.is_ref_type()) {
   // Get the object reference(s)
   if (nVals == 1) {
      ooRef(ooObj) ref = CO.get_ooref(pos);
      showRef(ref);
   } // End if one value
   else {
      for (size_t n = 0; n < nVals; ++n) {
         cout << n << ". ";
         ooRef(ooObj) ref = CO.get_ooref(pos, n);
         showRef(ref);
      } // End for each value
   } // End else array of values
   cout << endl;
} // End if object-reference attribute
...
```

The function showRef prints the class name and object identifier (OID) for the referenced object.

```
void showRef(ooRef(ooObj) ref) {
   if (ref.is_null()) {
      cout << "(null)" << endl;
   }
```

```
   else {
      cout << ref.typeName() << " object ";
      cout << ref.sprint() << endl;
   }
} // End showRef
```

Embedded String-Class Attribute

Call the `is_string_type` member function of the attribute's type descriptor to test whether the attribute is an embedded string class. If so, call the class object's `get_string` member function to get a string value for each element of the attribute's fixed-size array. "Examining String Data" on page 75 explains how to get data from a string value.

Embedded Non-String-Class Attribute

Call the inherited `is_class` member function of the attribute's type descriptor to test whether the attribute is an embedded-class attribute. Because embedded string classes are treated specially, you must also ensure that the type descriptor's `is_string_type` member function returns false.

If the attribute is an embedded-class attribute for a non-string class, call the class object's `get_class_obj` member function to get a class object for each element of the attribute's fixed-size array.

EXAMPLE This example shows how the function `showData` handles embedded non-string-class attributes.

```
...
else if (curType.is_string_type()) {
   ...
} // End if string attribute
else if (curType.is_class()) {
   // Set attrName to prefix for attributes
   // of the embedded class
   char *attrName =
      new char[strlen(prefix) +
               strlen(curAttr.name()) + 2];
   if (prefix) {
      sprintf(attrName, "%s.%s", prefix,
                                 curAttr.name());
   }
   else {
      sprintf(attrName, "%s", curAttr.name());
   }
```

```
   // Get the embedded instance(s)
   if (nVals == 1) {
      rc = showData(CO.get_class_obj(pos),
                    oocTrue,
                    attrName);
      if (rc != oocSuccess) {
         return rc;
      }
   } // End if one value
   else {
      for (size_t n = 0; n < nVals; ++n) {
         cout << n << ". ";
         rc = showData(CO.get_class_obj(pos, n),
                       oocTrue,
                       attrName);
         if (rc != oocSuccess) {
            return rc;
         }
      } // End for each value
   } // End else array of values
   delete [] attrName;
   cout << endl;
} // End if embedded non-string-class attribute
…
```

VArray Attribute

Call the `is_varray_type` member function of the attribute's type descriptor to test whether the attribute is a VArray attribute. If so, call the class object's `get_varray` member function to get a VArray object for each element of the attribute's fixed-size array. "Examining VArray Data" on page 76 explains how to get data from a varray object.

Examining Numeric Data

A numeric value provides access to data of a basic numeric type within the data of a persistent object. For example, a numeric value may contain the value of an attribute or an element of a VArray.

- You can call a class object's `get` member function to obtain a numeric value for a particular numeric attribute.
- If a VArray object provides access to a numeric VArray, you can call its `get` member function to obtain a numeric value for a particular element.

Constructors and conversion operators allow you to convert automatically between an Active Schema numeric value and a C++ numeric data type. If you know what type of data a numeric value contains, you can cast the numeric value to the corresponding C++ type. If not, you can call a numeric value's `type` member function to find out what type of numeric data it contains. The `type` member function returns a code of type `ooBaseType`.

EXAMPLE The function `showNumeric` prints the data for a numeric value. The function definition follows; repetitive details have been omitted. Appendix B, "Programming Examples," contains the complete definition of `showNumeric`.

```
ooStatus showNumeric (Numeric_Value numVal) {
   // Use the kind of numeric data to determine
   // how to print the value
   ooBaseType bt = numVal.type();
   switch (bt) {
      case ooCHAR: {
         cout << (char)numVal << endl;
         break;
      }
      case ooINT8: {
         cout << (int8)numVal << endl;
         break;
      }
      …
      case ooFLOAT64: {
         cout << (float64)numVal << endl;
         break;
      }
      case ooPTR: {
         cout << "(pointer)" << endl;
         break;
      }
      default: {
         cout << "(unrecognized numeric type)" << endl;
         break;
      }
   } // End switch
} // End showNumeric
```

Examining String Data

A string value provides access to an instance of a recognized string class embedded within the data of a persistent object. For example, a string value may contain the value of an attribute or an element of a VArray.

- You can call a class object's `get_string` member function to obtain a string value for a particular string attribute.
- If a VArray object provides access to a string VArray, you can call its `get_string` member function to obtain a string value for a particular element.

Once you have a string value, you can call its `type` member function to find out what type of string object the string value contains. The `type` member function returns a code of type `ooAsStringType`.

If a string value contains an instance of an internal string class (`ooVString`, `ooUtf8String`, or `ooSTString`), you can convert the string value to a pointer to an object of the appropriate internal string class. You can then examine the string as you would in any Objectivity/C++ application.

If a string value contains an instance of an application-defined optimized string class `ooString(N)`, you can use the string value to construct an optimized string value that accesses the string data. You can call the optimized string value's `get_copy` member function to get a transient copy of the string.

EXAMPLE The function `showString` prints the data for a string value.

```
ooStatus showString (String_Value strVal) {
   // Use the kind of string to determine how
   // to print the value
   switch (strVal.type()) {
      case ooAsStringVSTRING: {
         ooVString *vStr = strVal;
         cout << (const char *)(*vStr) << endl;
         break;
      }
      case ooAsStringUTF8: {
         ooUtf8String *utf8Str = strVal;
         cout << (const char *)(*utf8Str) << endl;
         break;
      }
      case ooAsStringOPTIMIZED: {
         Optimized_String_Value optStr(strVal);
         cout << optStr.get_copy() << endl;
         break;
      }
```

```
      case ooAsStringST: {
         cout << "(Smalltalk string)" << endl;
         break;
      }
      default: {
         cout << "(unrecognized string class)" << endl;
         break;
      }
   } // End switch kind of string
} // End showString
```

Examining VArray Data

A VArray object provides access to the data for a particular VArray in a particular VArray attribute of a particular persistent object. You can call a class object's `get_varray` member function to obtain a VArray object for a particular VArray attribute.

NOTE If your application passes VArray objects as parameters to, or return values from, functions, the VArray objects should be passed and returned by reference wherever possible.

Getting Information About a VArray Object

You can call a VArray object's `contained_in` member function to get the class object for the persistent object whose data contains the VArray.

You can call a VArray object's `type_of` member function to get a type descriptor for the element type of the associated VArray.

You can call a VArray object's `size` member function to get the number of elements in the associated VArray.

Getting an Element

Before you can access the elements of a VArray, you must examine the type descriptor for its element type to determine what kind of elements it contains. The following table lists the member functions that get data for VArray elements.

Element Type of VArray	Member Function	Data Returned As
Numeric type	`get`	Numeric value
Object-reference type	`get_ooref`	Object reference
	`get_class_obj`	Class object
Embedded string class	`get_string`	String value
Embedded non-string class	`get_class_obj`	Class object

As the third column shows, the persistent data is generally returned in a persistent-data object. The one exception is that `get_ooref` returns the value of an object-reference element as an object reference of type `ooRef(ooObj)`. That member function does not open a handle to the referenced object. If you want to examine the data for the referenced object, however, you need to call `get_class_obj` to obtain a class object for the referenced object.

EXAMPLE This example shows part of the function `showVArray`, which displays the data in a VArray object. Appendix B, "Programming Examples," contains the complete definition of `showVArray`.

```
ooStatus showVArray (VArray_Object VO) {

   // Check for null VArray object
   if (! VO) {
      cout << "(null VArray)" << endl;
      return oocSuccess;
   }

   // Get the number of elements in the VArray
   uint32 nVals = VO.size();
   if (nVals > 0) {
      cout << "VArray of " << nVals << "elements" << endl;
   }
   else {
      cout << "(empty VArray)" << endl;
      return oocSuccess;
   }
```

```
   // Test the element type to determine how to
   // access the elements
   const d_Type &elemType = VO.type_of();

   if (elemType.is_basic_type()) {
      ... // Show the numeric elements
   } // End if numeric VArray
   else if (elemType.is_ref_type()) {
      ... // Show the object-reference elements
   } // End if object-reference VArray
   else if (elemType.is_string_type()) {
      ... // Show the string elements
   } // End if string attribute
   else if (elemType.is_class()) {
      ... // Show the embedded-instance elements
   } // End if embedded non-string-class attribute
   else {
      cout << "unrecognized element type" << endl;
   }
   return oocSuccess;
} // End showVArray
```

Iterating Through the Elements

If the element type of the VArray is a numeric type, an object-reference type, or an internal Objectivity / C++ string class, you can iterate through its elements. To do so, first call the VArray object's `create_iterator` member function to get a VArray iterator for the elements of the VArray. This member function returns an Objectivity / C++ VArray iterator of the class `d_Iterator<ooObj>`. You must then cast the VArray iterator to the VArray iterator class for the element type of the associated VArray. For example, for a `VArray(float64)`, you should cast the VArray iterator to `d_Iterator<float64>`. Once you have a VArray iterator of the correct class, you can use it to iterate through the elements of the VArray. Refer to the reference documentation for Objectivity / C++ for information about working with a VArray iterator of class `d_Iterator<`*`elementType`*`>`.

If the elements of the VArray are embedded instances of an application-defined class that your application does not declare, you cannot use `create_iterator` because you would not be able to cast the returned VArray iterator to the appropriate class. For example, if the associated VArray is of type `VArray(Foo)` and your application does not declare the class `Foo`, you would not be able to cast the VArray iterator to `d_Iterator<Foo>`.

EXAMPLE This example iterates through the elements of an `ooVString` VArray.

```
ooTrans trans;
trans.start();
...
VArray_Object VO = ... ;

// Get the element type
const d_Type &elemType = VO.type_of();

if ((elemType.is_string_type() &&
   (!strcmp(elemType.name(), "ooVString")) {

   // Get VArray iterator for ooVString elements
   d_Iterator<ooVString> dit =
      (d_Iterator<ooVString> &)VO.create_iterator();

   // Use the VArray iterator to get each element
   while (dit.not_done()) {
      ooVString vStr = dit.get_element();
      cout << (const char *)vStr << endl;
      ++dit;
   }
} // End if element type is ooVString
...
trans.commit();
```

Examining Relationship Data

A relationship object provides access to the data for a particular relationship of a particular source object. That relationship is called the relationship object's *described relationship*; its data consists of one or more associations, each of which relates the source object to a particular destination object.

Once you have a class object for the source object, you obtain a relationship object for a particular relationship by calling the class object's `get_relationship` member function, specifying the position of the relationship of interest.

NOTE If your application passes relationship objects as parameters to, or return values from, functions, the relationship objects should be passed and returned by reference wherever possible.

Getting Information About a Relationship Object

You can call a relationship object's `relationship` member function to get a relationship descriptor for its described relationship.

You can call a relationship object's `contained_in` member function to get the class object for the source object of its associations.

You can call a relationship object's `other_class` member function to get a class descriptor for the destination class of its described relationship.

You can call a relationship object's `exist` member function to tests whether any association exists or whether an association exists to a particular destination object.

Testing the Kind of Relationship

The way you access the associated destination objects depends on whether the relationship is to-one or to-many. You can call the `is_to_many` member function of the relationship object's relationship descriptor to test whether the relationship is to-many.

Accessing a To-One Relationship

If the relationship is to-one, you can call the relationship object's `get_ooref` member function to get an object reference for the destination object. That member function does not open a handle to the destination object.

If you want to examine the data for the destination object, you can call the relationship object's `get_class_obj` member function to obtain a class object for the destination object.

Accessing a To-Many Relationship

If the relationship is to-many, you call the relationship object's `get_iterator` member function to initialize an Objectivity/C++ object iterator to find all associated destination objects. The parameter to this member function is the object iterator to be initialized. An optional second parameter allows you to specify the intended level of access to each destination object.

The Objectivity/C++ object iterator allows you to get a handle to each destination object. If you want to examine the data of a destination object, you can construct a class object from the handle.

EXAMPLE This example shows how the function showData handles relationships.

```
...
else if (curAttr.is_relationship()) {
   cout << prefix << curAttr.name() << ":" << endl;
   // Get the relationship object
   rc = showRelationship(CO.get_relationship(pos));
   if (rc != oocSuccess) {
      return rc;
   }
   cout << endl;
} // End if relationship
...
```

The function showRelationship prints the data for a relationship object; it prints an object reference for each destination object using the function showRef, which is shown on page 71.

```
ooStatus showRelationship (Relationship_Object RO) {

   // Check for null relationship object
   if (! RO) {
      cout << "(no associated object)" << endl;
      return oocSuccess;
   }

   // Get relationship descriptor
   const d_Relationship &rel = RO.relationship();

   // Test whether relationship is to-many
   if (rel.is_to_many()) {
      ooItr(ooObj) objItr;
      // Initialize an object iterator for destination objects
      RO.get_iterator(objItr);
      // Iterate through the destination objects
      int n = 0;
      while (objItr.next()) {
         cout << n << ". ";
         // Get handle for this destination object
         ooHandle(ooObj) curObjH(objItr);
         showRef(curObjH);
         ++n;
      } // End while more destination objects
   } // End if to-many
```

```
   else { // Relationship is to-one
      // Get object reference for destination object
      ooRef(ooObj) destination = RO.get_ooref();
      showRef(destination);
   } // End else relationship is to-one
   return oocSuccess;
} // End showRelationship
```

4

Modifying the Schema

Active Schema applications can modify the schema of any Objectivity/DB federated database, adding new modules or classes, and modify existing class descriptions.

In This Chapter

About Schema Modification

You can use Active Schema to add a named module to the schema and to add or modify the classes in an existing module. After making the modifications, your process can use the evolved schema without restarting. If another process has modified class descriptions, you can activate those remote schema changes, making them available to your process.

NOTE You cannot use Active Schema to delete class descriptions from the schema.

Modifying Class Descriptions

To modify the class descriptions in a module, you perform three steps:

1. Propose new classes, and / or new versions or modified definitions of existing application-defined classes.
2. Modify characteristics of proposed classes or modify the attributes, relationships, and / or base classes of the proposed classes.
3. Activate the proposals.

You can propose new and evolved classes and modify proposals without being in a transaction. When you activate proposed schema changes, Active Schema automatically opens a separate transaction in which to make the changes.

Any modification to an existing class description is called *schema evolution*. If the modification changes the layout shape for persistent objects of the class, all existing persistent objects of the class must be converted to the new shape.

NOTE After you modify the description of any class in the schema, you must rebuild any Objectivity/C++ application that uses the class. The application's class declaration must match the new schema description of the class. Active Schema applications need not be rebuilt.

Before you modify the schema of a federated database, you should understand the types of schema-evolution operations that Objectivity/DB supports and its limitations on those operations; Active Schema is subject to the same limitations. For details, see the chapter on schema evolution in the Objectivity/C++ Data Definition Language (DDL) book. If a schema-evolution operation must be performed in a separate cycle with the DDL processor, the corresponding schema modifications must be proposed and activated in a separate cycle with Active Schema. In each Active Schema cycle, you propose the modifications to the schema and you activate them.

If the operation requires object conversion, you can perform the necessary conversion as described in the chapter on object conversion in the Objectivity/C++ programmer's guide; alternatively, you can use Active Schema to modify the objects as described in Chapter 5.

WARNING You risk data corruption if you modify the schema while an Objectivity/C++ application is using persistent objects of the classes that you evolve. 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.

Extending Class Descriptions

In the Objectivity/DB object model, all properties in a class description represent information about an instance of the class, not about the class itself. Using Active Schema, you can extend a class description to contain properties that represent information about the class itself, analogous to the static data members of a C++ class. These properties are called *persistent static properties* of the class.

Replicating a Schema

Active Schema enables you to create a schema that is an exact replica of the schema of an existing federated database. If you need to replicate a schema, you must ensure that each module, class, and bidirectional relationship in the new schema has the same identifying number as the corresponding entity in the existing schema. To support this requirement:

- You can specify a schema number for any named module you add to the schema.
- You can specify a shape number for any class you propose.
- You can specify an association number for any bidirectional relationship you add to a proposed class.

If you are not trying to replicate a schema, you do not specify these identifying numbers, but instead allow Active Schema to assign the next available number to each new entity.

Proposal Descriptors

Proposal descriptors contain information about class definitions that have been proposed for inclusion in the schema. Different proposal-descriptor classes provide information about proposed classes and the various components of proposed classes.

Figure 4-1 shows the inheritance graph for the major proposal-descriptor classes.

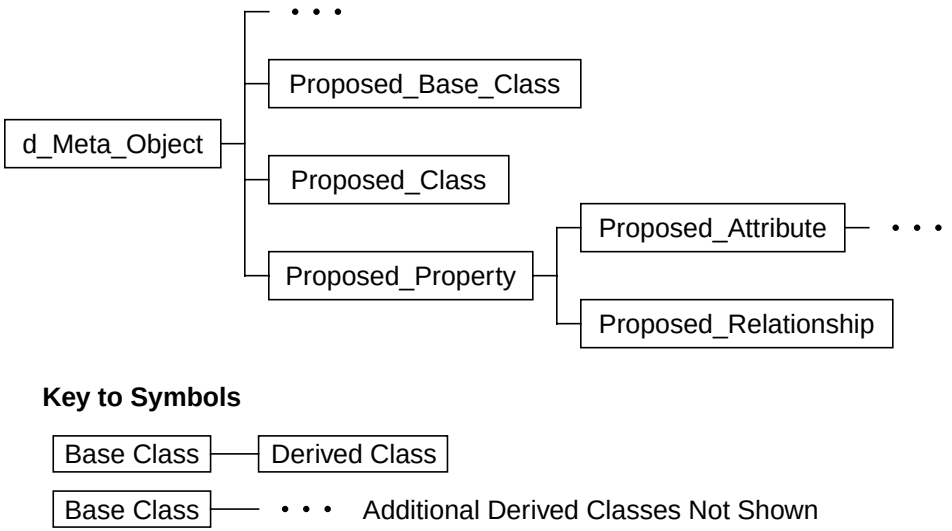

Figure 4-1 Proposal-Descriptor Classes

You propose changes to a module by creating proposed classes. To do so, you call member functions of a module descriptor. A proposed class (`Proposed_Class`) can represent either a new class to be added to the module, or an evolved definition of a class already defined in the module. It has an associated proposed base class (`Proposed_Base_Class`) for each of its immediate base classes, a proposed attribute (`Proposed_Attribute`) for each of its attributes, and a proposed relationship (`Proposed_Relationship`) for each of its relationships.

A proposed class is a descriptor that contains information about the class definition. Each proposed base class, proposed attribute, and proposed relationship is a descriptor that contains information about one component of the proposed class. You can call member functions of any of these proposal descriptors to see the current definition of the proposed entity and to modify its definition.

Proposal Lists

Each module descriptor has a *proposal list* that contains all proposed classes in the described module. Initially a module descriptor's proposal list is empty. Each time you propose a new or evolved class, the new proposed class is added to the module descriptor's proposal list. When proposals are successfully activated, the module descriptor's proposal list is emptied. If activation fails, the proposal list is left unchanged so that you can fix any inconsistencies in the proposals and try activation again. You can explicitly remove proposals from a module descriptor's proposal list. To delete a particular proposed class, call the module descriptor's `delete_proposal` member function; to delete all proposals from the module descriptor, call its `clear_proposals` member function.

NOTE Once a proposed class has been removed from a module descriptor's proposal list, whether by activation or by a call the `delete_proposal` or `clear_proposals`, the proposed class becomes invalid.

Any member function that obtains a module descriptor for an existing module returns a constant descriptor of the type:

```
const d_Module &
```

Because member functions that propose schema changes and activate proposals modify a module descriptor's proposal list, they cannot be called for a constant descriptor. If you want to modify classes in the described module, you must first cast your module descriptor to the non-constant type:

```
d_Module &
```

Adding a Module

To add a new named module to the schema, call the static member function `d_Module::add_module`. This member function returns a (non-constant) descriptor for the newly created module. You can call members of the returned module descriptor to add proposed classes to the new module.

EXAMPLE This code adds a module named `Manufacturing`. The module descriptor `newMod` is used in later examples to add proposed classes to the newly created module.

```
ooTrans trans;
trans.start();
...
d_Module &newMod = d_Module::add_module("Manufacturing");
...
trans.commit();
```

If you are replicating an existing schema, you may specify the schema number for the new module as the second parameter to `add_module`.

WARNING If a federated database is used by Objectivity for Java, Objectivity / Smalltalk, or Objectivity / SQL++ applications, you should not use Active Schema to define named modules. Those applications can access classes in the top-level module only.

Defining a New Class

To add a new class definition to the schema, you create a proposed class, add the class to the proposal list of the appropriate module, and fill in the class definition. Proposing the new class adds it to the module's proposal list. You can create a new proposed class and add it to the module's proposal list at the same time. Alternatively, you can create a proposed class independent of any module and later propose the class in the new module.

A newly created proposed class has an empty definition; you call its member functions to add base classes, attributes, and relationships.

The name of your new class must be unique within its own module. Although a schema may contain classes with the same name in different modules, only one of these classes can be accessed by any given Objectivity / C++ application.

NOTE You can avoid confusion by giving each new class a name that is unique *within the entire schema*, not just within its containing module.

Proposing a New Class

Call the `propose_new_class` member function of a module descriptor to propose a new class in the described module. You can pass the name of the new class as the parameter to `proposed_new_class` to create the proposed class and propose it to the module in a single operation. If you are replicating an existing schema, you may specify the type number for the new class as the second parameter to `propose_new_class`.

EXAMPLE This statement proposes a new class named `Factory` to the new `Manufacturing` module, whose descriptor is `newMod`. The proposal descriptor `factor` is used in later examples that add base classes and properties to the class definition.

```
Proposed_Class &factory = newMod.propose_new_class("Factory");
```

You can create a proposed class independent of a module by passing the class name to the `Proposed_Class` constructor. Typically, you pass 0 as the second parameter to the constructor, which assigns the class the next available type number. If you are replicating an existing schema, you may specify the type number for the new class as the second parameter. You can later propose the class to a module by passing a pointer to the proposed class as a parameter to the module's `propose_new_class` member function.

If you create a proposed class using the explicit constructor and later propose it in a module, you must ensure that the proposed class is available when you activate proposals; see "Activating Proposals" on page 112. If you allocate the proposed class on the stack, it must be in scope when you activate proposals. If you allocate the proposed class dynamically, you are responsible for deleting it, but you must not delete it until after you activate proposals.

EXAMPLE This example creates a proposed class named `Machine`, adds components to fill in its definition, then proposes the new class to the module whose descriptor is `newMod`.

```
Proposed_Class *machineP = new Proposed_Class("Machine", 0);
... // Add components to fill in the class definition
newMod.propose_new_class(machineP);
... // Propose other changes
```

```
newMod.activate_proposals(...); // Activate proposals
delete machineP; // Delete the proposed class
```

An alternative approach is to allocate the proposed class on the stack.

```
{  ...
   Proposed_Class machine("Machine", 0);
   ... // Add components to fill in the class definition
   newMod.propose_new_class(&machine);
   ... // Propose other changes
   newMod.activate_proposals(...); // Activate proposals
   ...
}
```

Adding Components to a Proposed Class

After you create a proposed class, you can call its member functions to add the necessary components.

Adding a Base Class

Call the `add_base_class` member function of a proposed class to add a base class. Parameters specify the position of the base class within the data of the proposed class, the access kind (public, protected, or private), and the base class name. The base class you specify can be an existing class in the schema, a proposed class, or a new class that has not yet been defined. In the latter case, you must create a proposed class for the new base class and fill in its definition before you activate proposals for the module that contains your proposed class.

EXAMPLE This statement makes `Factory` a persistence-capable class by adding `ooObj` as its base class. The constant `oocLast` indicates the next available position within the proposed class.

```
factory.add_base_class(
   oocLast,     // Position
   d_PUBLIC,    // Access kind
   "ooObj");    // Base class name
```

Adding an Attribute

Various member functions of a proposed class add attributes of different types. All these member functions take as parameters the position of the new attribute, its access kind, its name, and the number of elements in its fixed-size array of values. Each member function takes additional parameters providing information that is specific to the type of attribute being defined.

WARNING If a federated database is used by Objectivity for Java or Objectivity / Smalltalk applications, you should not use Active Schema to give a class an attribute with a fixed-size array of values. Objectivity for Java and Objectivity / Smalltalk applications can access an attribute *only* if it has a single value.

The following table lists the member functions that add attributes of the various types, and the information that is passed in additional parameters to each.

Type of Attribute	Member Function	Additional Parameters
Numeric	`add_basic_attribute`	Type of numeric data Default value
Object reference	`add_ref_attribute`	Referenced class Whether values are stored as short object references
Embedded class	`add_embedded_class_attribute`	Embedded class
Numeric VArray	`add_varray_attribute`	Type of numeric data
Object-reference VArray	`add_varray_attribute`	Whether elements are stored as short object references Referenced class
Embedded-class VArray	`add_varray_attribute`	Embedded class

When you specify an embedded class or a referenced class, the class may be an existing class in the schema, a proposed class, or a new class that has not yet been defined. In the latter case, you must create a proposed class for the new embedded or referenced class and fill in its definition before you activate proposals for the module that contains your proposed class.

A string attribute is treated the same as any embedded-class attribute; similarly, a string VArray class is treated like any embedded-class VArray. Just specify the string class name (for example, `ooVString`) as the embedded class.

EXAMPLE The following statements add attributes to the new class `Factory`.

```
// Add ASCII string attribute "name"
factory.add_embedded_class_attribute(
   oocLast,        // Position
   d_PUBLIC,       // Access kind
   "name",         // Attribute name
   1,              // # elements in fixed-size array
   "ooVString");   // Embedded class name

// Add 32-bit unsigned integer attribute "capacity"
factory.add_basic_attribute(
   oocLast,        // Position
   d_PUBLIC,       // Access kind
   "capacity",     // Attribute name
   1,              // # elements in fixed-size array
   ooUINT32,       // Type of numeric data
   10000);         // Default value

// Add object-reference attribute "manager"
factory.add_ref_attribute(
   oocLast,        // Position
   d_PUBLIC,       // Access kind
   "manager",      // Attribute name
   1,              // # elements in fixed-size array
   "Employee",     // Referenced class
   oocFalse);      // Whether stored as short references

// Add object-reference VArray attribute "shift_supervisors"
factory.add_varray_attribute(
   oocLast,              // Position
   d_PUBLIC,             // Access kind
   "shift_supervisors",  // Attribute name
   1,                    // # elements in fixed-size array
   oocFalse,             // Whether stored as short references
   "Employee");          // Referenced class
```

Adding a Relationship

Two member functions of a proposed class add relationships:

- Call `add_unidirectional_relationship` to add a unidirectional relationship.
- Call `add_bidirectional_relationship` to add a bidirectional relationship.

Both these member functions take as parameters the position of the new relationship, its access kind, its name, the destination class, whether the relationship is inline, whether it is short, whether it is to-many, its copy mode, its versioning mode, and its propagation behavior. When you create a bidirectional relationship, you also specify the name of the inverse relationship and whether the inverse relationship is to-many. If you are replicating an existing schema, you may specify the association number for a new bidirectional relationship as the final parameter to `add_bidirectional_relationship`.

The destination class for a new relationship may be an existing class in the schema, a proposed class, or a new class that has not yet been defined. In the latter case, you must create a proposed class for the new destination class and fill in its definition before you activate proposals for the module that contains your proposed class. When you create a new destination class for a bidirectional relationship, you must define the appropriate inverse relationship in the destination class.

EXAMPLE The following statements add two relationships to the class `Factory`. The bidirectional to-one relationship `coproducer` links a factory to its partner in a two-factory team that produces a particular product line. This relationship is its own inverse; if factory `A` is the coproducer of factory `B`; then factory `B` is the coproducer of factory `A`. The unidirectional to-many relationship `products` links a factory to the products that it produces.

```
// Add bidirectional to-one relationship "coproducer”
factory.add_bidirectional_relationship(
   oocLast,        // Position
   d_PUBLIC,       // Access kind
   "coproducer",   // Relationship name
   "Factory",      // Destination class
   oocFalse,       // Whether relationship is inline
   oocFalse,       // Whether relationship is short
   oocFalse,       // Whether relationship is to-many
   0,              // Copy mode
   0,              // Versioning mode
   0,              // Propagation behavior
   "coproducer",   // Inverse relationship
   oocFalse);      // Whether inverse relationship is to-many
```

```
// Add unidirectional to-many relationship "products"
factory.add_unidirectional_relationship(
   oocLast,     // Position
   d_PUBLIC,    // Access kind
   "products",  // Relationship name
   "Product",   // Destination class
   oocFalse,    // Whether relationship is inline
   oocFalse,    // Whether relationship is short
   oocFalse,    // Whether relationship is to-many
   0,           // copy mode
   0,           // versioning mode
   0);          // propagation behavior
```

The proposed description for the class `Factory` now corresponds to this DDL declaration:

```
class Factory : public ooObj {
   ooVString name;
   uint32 capacity;
   ooRef(Employee) manager;
   ooVArray(ooRef(Employee)) shift_supervisors;
   ooRef(Factory) coproducers <-> coproducers;
   ooRef(Product) products[] : copy(delete);
}
```

If the classes `Employee` and `Product` do not already exist in the schema, they must be proposed before proposals are activated for the module descriptor `newMod`, whose proposal list contains the proposed class `Factory`.

Copying a Property

You can call the `add_property` member function of a proposed class to add a property that is a copy of an existing property of an existing class. The parameters to this function are the position of the new property, its access kind, and a property descriptor for the property to be copied.

Modifying an Existing Class

You can modify the definition of an existing class either by *evolution* or by *versioning*:

- Evolution of a class changes its definition; if the class definition is modified in a way that affects its storage layout, a new description is added to the

schema with a new shape number for the class; otherwise, the description of the class (with its existing type number and shape number) is modified.

- Versioning a class creates a new description in the schema with a new type number. The definition of the new version can subsequently be modified by evolution, possibly creating new shapes for that version.

You should use evolution for most changes; use versioning only when necessary for legacy reasons.

NOTE The proposed changes to existing class definitions in a given transaction must either be all proposed evolution or all proposed versioning.

Proposing an Evolved Class

Call the `propose_evolved_class` member function of a module descriptor to propose an evolved definition of a specified class in the described module. The proposed class initially has a definition identical to the class being evolved; you call its member functions to modify, add, or remove base classes, attributes, and relationships.

EXAMPLE This example proposes an evolved definition of the class `Library` in the top-level module. Note that the constant descriptor `topMod` is cast to a non-constant descriptor `RWtopMod` (where `RW` indicates read / write). The proposed change is made to `RWtopMod`, not to `topMod`. The module descriptor `RWtopMod` is used in later examples that modify the top-level module. The proposal descriptor `library` is used in later examples that modify the definition of the class `Library`.

```
ooTrans trans;
trans.start();
...
const d_Module &topMod = d_Module::top_level();
d_Module &RWtopMod = const_cast<d_Module &>(topMod);

Proposed_Class &library =
   RWtopMod.propose_evolved_class("Library");
...
trans.commit();
```

If you are replicating an existing schema, you may specify the shape number for the evolved class as the second parameter to `propose_evolved_class`.

Proposing a New Version of a Class

Call the propose_versioned_class member function of a module descriptor to propose a new version a specified class in the described module. The proposed class initially has a definition identical to the most recent version of the specified class; you call its member functions to modify, add, or remove base classes, attributes, and relationships.

EXAMPLE This example proposes a new version of the class `Client` in the top-level module. Note that the constant descriptor `topMod` is cast to a non-constant descriptor `RWtopMod` (where `RW` indicates read / write). The proposed change is made to `RWtopMod`, not to `topMod`.

```
ooTrans trans;
trans.start();
...
const d_Module &topMod = d_Module::top_level();
d_Module &RWtopMod = const_cast<d_Module &>(topMod);

Proposed_Class &client =
   RWtopMod.propose_versioned_class("Client");
...
trans.commit();
```

If you are replicating an existing schema, you may specify the shape number for the new version as the second parameter to `propose_versioned_class`.

Adding Persistent Static Properties

Active Schema allows you to store persistently any information you choose to keep about a persistence-capable class, analogous to the information in its static data members. This information comprises the values of the *persistent static properties* of the class.

To define persistent static properties of a class and regular properties of an auxiliary persistence-capable class. You create a persistent instance of this auxiliary class and sets its properties appropriately. You then store the auxiliary object with the schema description of your class by passing an object reference to the object as a parameter to the set_static_ref member function of a descriptor for the class.

Any member function that obtains a class descriptor returns a constant descriptor of the type:

```
const d_Class &
```

Because `set_static_ref` modifies a class descriptor, it cannot be called for a constant descriptor. If you want to call this function, you must first cast your class descriptor to the non-constant type:

```
d_Class &
```

You can call the `get_static_ref` member function of a class descriptor to find the auxiliary object containing the persistent static properties for the described class. If the described class does not have persistent static properties, this member function returns a null object reference.

EXAMPLE The `Vat` class has static data members `min_volume` and `max_volume`, which represent the legal range for the volume of a vat. This is information about the class itself and is shared by all instances of `Vat`; that is, no instance can have a volume below `min_volume` or above `max_volume`.

The values for the two static data members vary at the different wineries in which the application is installed. Initially, they are set based on the sizes of vats used at the particular winery. If the winery later adds a new vat that is larger or smaller than its existing vats, or discontinues the use of its smallest or largest vat, the corresponding static data member is modified. Because static data members of a class are not saved persistently, however, changes to the minimum or maximum volume made in one execution of the application are not available the next time the application is run.

```
// DDL file winery.ddl
class Vat : public ooObj {
   ...
private:
   uint16 volume;
   static uint16 min_volume; // Can't be saved persistently
   static uint16 max_volume; // Can't be saved persistently
public:
   static void init_min_max(unit16 min, unint16 max);
   static uint16 get_min_volume();
   static uint16 get_max_volume();
   static void set_min_volume(unit16 min);
   static void set_max_volume(unit16 max) ;
   ...
};
```

```
// Application code file
#include "winery.h"

// Static member functions that get and set static
// data members are called outside a transaction

// Initialize min and max volume for vats in this winery
static void Vat::init_min_max(unit16 min, unint16 max) {
   Vat::min_volume = min;
   Vat::max_volume = max;
}

// Get minimum volume for vats in this winery
static uint16 Vat::get_min_volume() { return Vat::min_volume }

// Set minimum volume to add (or remove) a smaller vat size
static void Vat::set_min_volume(unit16 min) {
   Vat::min_volume = min;
}

// Get maximum volume for vats in this winery
static uint16 Vat::get_max_volume() { return Vat::max_volume }

// Set maximum volume to add (or remove) a larger vat size
static void Vat::set_max_volume(unit16 max) {
   Vat::max_volume = max;
}
```

To save the static properties of `Vat` persistently, an Active Schema application uses the auxiliary class `VatModel`, whose data members correspond to the static data members of `Vat`. The static data members of `Vat` contain its *transient* static properties. The data members of an instance of `VatModel` contain the *persistent* static properties of the class `Vat`.

```
// DDL file winery.ddl
class Vat : public ooObj {
   …
   static uint16 min_volume; // Transient static property
   static uint16 max_volume; // Transient static property
   …
};
```

```
class VatModel : public ooObj {
   friend class Vat;
protected:
   uint16 min_volume; // Persistent static property of class Vat
   uint16 max_volume; // Persistent static property of class Vat
};
```

When the static properties of `Vat` are initialized, the static member function `Vat::init_min_max` creates an instance of `VatModel`, sets its data members, and saves it with the schema description of `Vat`. The first time a subsequent execution of the application gets a static property of `Vat`, the static member function `Vat::get_min_volume` or `Vat::get_max_volume` sets the transient static properties of `Vat` from its persistent static properties. When the application modifies a static property of `Vat`, the static member function `Vat::set_min_volume` or `Vat::set_max_volume` sets both its transient static property and the corresponding persistent static property.

```
// Application code file
#include "winery.h"
#include <ooas.h>
...
// Static member functions that get and set static
// data members are called outside a transaction

// Initialize min and max volume for vats in this winery
static void Vat::init_min_max(unit16 min, unint16 max) {

   // Set the transient static properties of Vat
   Vat::min_volume = min;
   Vat::max_volume = max;

   ooTrans trans;
   trans.start();
   ... // Make this an update transaction

   // Create auxiliary object and set its properties, which
   // contain the persistent static properties of Vat
   ooHandle(VatModel) vmH = new(...) VatModel();
   vmH->min_volume = min;
   vmH->max_volume = max;

   // Get a descriptor for the top-level module
   const d_Module &topMod = d_Module::top_level();

   // Get a descriptor for the class Vat
   const d_Class &vat = topMod.resolve_class("Vat");
```

```
   // Cast vat to a non-const descriptor before modifying
   d_Class &vatRW = const_cast<d_Class&>(vat);

   // Store auxiliary object with the class description of Vat
   vatRW.set_static_ref(vmH);
   trans.commit();
} // End Vat::init_min_max

// Get minimum volume for vats in this winery
static uint16 Vat::get_min_volume() {
   ooRef(VatModel) vmR;
   ooHandle(VatModel) vmH;

   if (!Vat::min_volume) {
      // Transient static properties have not been set yet
      ooTrans trans;
      trans.start();
      …
      // Get a descriptor for the top-level module
      const d_Module &topMod = d_Module::top_level();

      // Get a descriptor for the class Vat
      const d_Class &vat = topMod.resolve_class("Vat");

      // Find the auxiliary object
      vmR = static_cast<ooRef(VatModel)>(vat.get_static_ref());
      vmH = vmR;

      // Set the transient static properties of Vat from
      // its persistent static properties
      Vat::min_volume = vmH->min_volume;
      Vat::max_volume = vmH->max_volume;
      trans.commit();
   }
   return Vat::min_volume;
} // End Vat::get_min_volume

// Set minimum volume to add (or remove) a smaller vat size
static void Vat::set_min_volume(unit16 min) {
   ooRef(VatModel) vmR;
   ooHandle(VatModel) vmH;

   // Set transient static property of Vat
   Vat::min_volume = min;
```

```
    ooTrans trans;
    trans.start();
    ... // Make this an update transaction

    // Get a descriptor for the top-level module
    const d_Module &topMod = d_Module::top_level();

    // Get a descriptor for the class Vat
    const d_Class &vat = topMod.resolve_class("Vat");

    // Find the auxiliary object
    vmR = static_cast<ooRef(VatModel)>(vat.get_static_ref());
    vmH = vmR;

    // Set the persistent static static property of Vat
    vmH->update();
    vmH->min_volume = min;
    trans.commit();
} // End Vat::set_min_volume

// Get maximum volume for vats in this winery
static uint16 Vat::get_max_volume() {
    ... // (Logic is similar to Vat::get_min_volume)
}

// Set maximum volume to add (or remove) a larger vat size
static void Vat::set_max_volume(unit16 max) {
    ... // (Logic is similar to Vat::set_min_volume)
}
```

Working With Proposed Classes

After creating proposed classes, you can find existing proposed classes in the proposal list of a module descriptor. You can examine and modify the class description for any proposed class.

Finding a Proposed Class

After you have created proposed classes, you can look up existing class proposals. You call member functions of a module descriptor to get the proposed classes in its proposal list.

- To look up a proposed class by name, call `resolve_proposed_class`.

- To get an iterator that finds all proposed classes in the proposal list, call `proposed_classes_begin`. The returned iterator gets constant proposed classes of the type:

  ```
  const Proposed_Class &
  ```

  If you intend to modify a proposed class, for example by changing its components, you must first cast your proposed class to the non-constant type:

  ```
  Proposed_Class &
  ```

EXAMPLE This statement finds the proposed class named `Library` in the proposal list of the top-level module.

```
Proposed_Class &proposed =
   RWtopMod.resolve_proposed_class("Library");
```

Getting Information From a Proposed Class

You can call member functions of a proposed class to get information about that proposal:

- Call the inherited `name` member function to get the name of the proposed class.
- Call `proposed_in_module` to get the module descriptor whose proposal list contains the proposed class.
- Call `persistent_capable` to test whether the proposed class is persistence-capable.
- Call `number_of_attribute_positions` to get the number of attribute positions in the storage layout for the proposed class; that is, the total number of its immediate base classes, attributes, and relationships.
- Call `number_of_base_classes` to get the number of immediate base classes of the proposed class.
- Call `has_added_virtual_table` to test whether storage for a virtual-table pointer has been added to the proposed class.
- If the proposed class has been renamed, call `previous_name` to get its previous name.
- If a shape number was specified when the proposed class was created, call `specified_shape_number` to get that shape number.
- Call `position_in_class` to get the class position of a particular component of the proposed class, that is, an attribute or relationship defined in or inherited by the class, an immediate base class, or an ancestor class at any level in the inheritance graph for the proposed class.

EXAMPLE This code iterates through the proposed classes in the proposal list of the module descriptor `newMod` and prints information about each proposed class.

```
proposed_class_iterator itr = newMod.proposed_classes_begin();
while (itr != newMod.proposed_classes_end()) {
   const Proposed_Class &curClass = *itr;
   cout << "Class " << curClass.name() << endl;
   if (curClass.persistent_capable()) {
      cout << "  persistence capable" << endl;
   }
   size_t all = curClass.number_of_attribute_positions();
   size_t base = curClass.number_of_base_classes();
   cout << "  " << base << " base classes" << endl;
   cout << "  " << all - base << " properties" << endl;
   if (curClass.has_added_virtual_table()) {
      cout << "  virtual table added" << endl;
   }
   const char *prevname = curClass.previous_name();
   if (prevname) {
      cout << "  previous name: " << prevname << endl;
   }
   ++itr;
}
```

Modifying a Proposed Class

You can call member functions of a proposed class to change characteristics of the class and to modify its list of components. To change the characteristics of a particular base class or property, you obtain the proposal descriptor for that base class or property and call its member functions. See “Working With Proposed Base Classes” on page 105 and “Working With Proposed Properties” on page 107.

Changing Class Characteristics

You can change the following characteristics of a proposed class:

- Call `rename` to change the name of the proposed class.
- Call `add_virtual_table` to give the proposed class room in its physical layout for a virtual-table pointer.

  A proposed new class does not have a virtual table; you can call this member function to add one. If a proposed evolved class already has a virtual table, you cannot remove it. If it doesn’t, you can call this member function to add a virtual table. Once you have added a virtual table to a proposed class, you cannot remove it.

Modifying List of Components

You can change the list of base classes, attributes, and relationships for a proposed class. “Adding Components to a Proposed Class” on page 90 explains how to add new components. In addition, you can delete and rearrange existing components:

- Call `delete_base_class` to delete a base class.
- Call `move_base_class` to move the position of a base class (relative to the other base classes of the proposed class).
- Call `change_base_class` to replace an existing base class with a different base class.
- Call `delete_property` to delete an attribute or a relationship.
- Call `move_property` to move the position of an attribute or a relationship relative to the other properties of the proposed class.

These member functions all return a status code. You should check the returned code to see whether the modification was successful.

EXAMPLE This example is from an interactive schema editor application. The user has issued a command to move a property. The illustrated code tries to move the property and notifies the user if the operation fails.

```
Proposed_Class &pClass;
const char *property;
size_t newPos;
... // Interaction with user sets pClass, property, and newPos
ooStatus rc = pClass.move_property(property, newPos);
if (rc != oocSuccess) {
   cerr << "Move failed" << endl;
}
```

Modifying in Multiple Cycles

If you need to modify a class description in more than one cycle, remember that a proposed class becomes invalid when you successfully activate proposals at the end of each cycle. As a consequence, you must obtain a new proposed class at the start of each new cycle by calling the module descriptor’s `propose_evolved_class` member function. A typical sequence of steps is as follows:

1. Obtain the necessary proposed classes for the first cycle by calling `proposed_new_class` and `proposed_evolved_class`.
2. Modify the class descriptions using the proposed classes.

3. Activate proposals, ending the current cycle.
4. Perform any necessary object conversion.
5. Obtain the necessary proposed classes for the next cycle. To further update a class that was defined or modified in an earlier cycle, *do not* use an existing proposed class. Instead, call `proposed_evolved_class` to obtain a new proposed class for the class to be modified.

Repeat steps 2 through 4 until the desired schema evolution is complete.

Working With Proposed Base Classes

A proposed base class is a component of a particular proposed class, called its *defining proposed class*. The proposed base class provides information about the base class within the description of its defining proposed class, for example, its position within the immediate base classes of the proposed class and its access kind. If two more proposed classes are derived from the same base class, each of the two proposed classes has its own proposed base class corresponding to their mutual base class.

Obtaining a Proposed Base Class

You call member functions of a proposed class to create proposed base classes for that proposed class or to obtain proposed base classes from the description of that proposed class.

- To create a new proposed base class, call `add_base_class`.
- To look up a proposed base class by name, call `resolve_base_class`.
- To get an iterator that finds all proposed base classes of that proposed class, call `base_class_list_begin`. The returned iterator gets constant proposed base classes of the type:

  ```
  const Proposed_Base_Class &
  ```

  If you intend to modify a proposed base class, for example by changing its access kind, you must first cast your proposed base class to the non-constant type:

  ```
  Proposed_Base_Class &
  ```

Getting Information From a Proposed Base Class

After you have obtained a proposed base class, you can call its member functions to get information about it:

- Call the inherited `name` member function to get the name of the proposed class.
- Call `defined_in_class` to get its defining proposed class.
- Call `persistent_capable` to test whether the proposed base class is persistence capable.
- Call `position` to get the attribute position of this proposed base class within the physical layout of its defining proposed class.
- Call `access_kind` to get the access kind of the proposed base class.
- If the proposed base class replaces a former base class in its defining proposed class, call `previous_name` to get the name of the former base class.

Modifying a Proposed Base Class

You can change the access kind of a proposed base class by calling its `change_access` member function.

You can change the position of a proposed base class by calling the `move_base_class` member function of its defining proposed class.

To change the definition of the base class itself, you work with a proposed class representing that base class. If the base class does not already exist in the schema, call the module descriptor's `propose_new_class` member function to obtain its proposed class; if the base class already exists, call the module descriptor's `propose_evolved_class` member function instead.

EXAMPLE This example gives public access to all base classes of the class `Library`.

```
proposed_base_class_iterator itr =
   library.base_class_list_begin();
while (itr != library.base_class_list_end()) {
   // Cast to non-constant to allow modification
   Proposed_Base_Class &curBase = (Proposed_Base_Class &)*itr;
   ooStatus rc = curBase.change_access(d_PUBLIC);
   if (rc != oocSuccess) {
      cerr << " Couldn't change access for ";
      cerr << curBase.name() << endl;
   }
   ++itr;
} // End while more base classes
```

Working With Proposed Properties

A proposed property is a descriptor for a particular property of a particular proposed class. You obtain a proposed property from its containing proposed class. If you want to examine or modify its definition, you typically need to test what kind of property is proposed and cast the proposed property to the corresponding proposal-descriptor class.

As Figure 4-1 on page 86 shows, a proposed property can be either a proposed attribute or a proposed relationship. Proposed attributes are further subdivided according to the data type of their values. Figure 4-2 shows the inheritance graph for proposed-attribute classes.

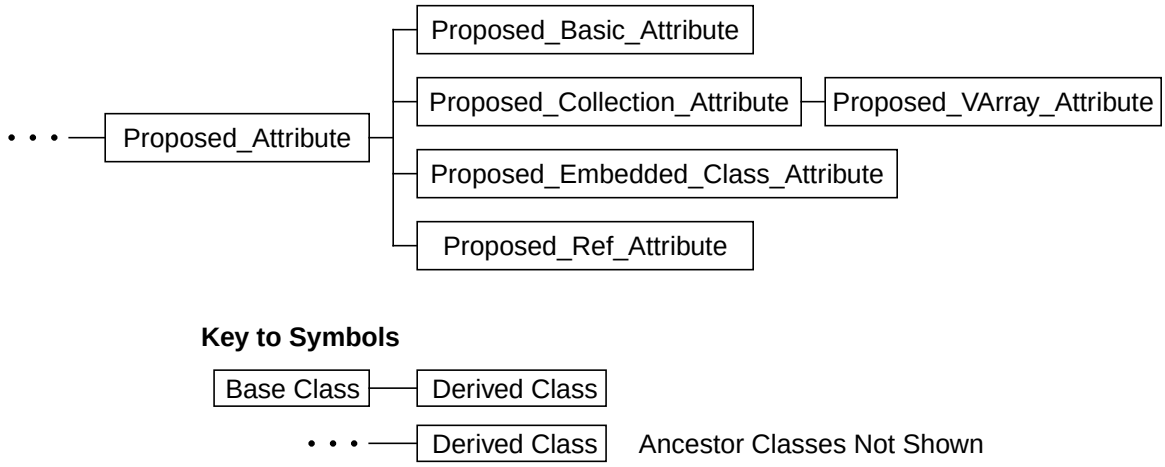

Figure 4-2 Proposed-Attribute Classes

Obtaining a Proposed Property

You call member functions of a proposed class to create proposed properties for that proposed class or to obtain proposed properties from the description of that proposed class.

- To create a new proposed property, call the appropriate `add_`*`propertyType`* member function for adding a property of desired kind; see "Adding an Attribute" on page 91 and "Adding a Relationship" on page 93. For example, call `add_basic_attribute` to add an attribute of a basic numeric type.
- To look up a proposed property by name, call `resolve_property`.

- To get an iterator that finds all proposed properties of the proposed class, call `defines_property_begin`. The returned iterator gets constant proposed properties of the type:

  ```
  const Proposed_Property &
  ```

  If you intend to modify a proposed property, for example by setting its value, you must first cast your proposed property to the non-constant type:

  ```
  Proposed_Property &
  ```

Testing the Kind of Proposed Property

When you look up a proposed property by name or iterate through all properties of a proposed class, you obtain a proposed property of the class `Proposed_Property`. You can call its member functions to determine what kind of property is proposed. You may then need to cast the proposed property to the particular proposal-descriptor class before you can call member functions that are specific to that kind of proposed property.

- Call `is_relationship_type` to test whether the proposed property is a relationship. If so, you can cast it to `Proposed_Relationship`; if not, the proposed property is an attribute. You can call other member functions to determine the type of attribute.
- Call `is_basic_type` to test whether the proposed property is a numeric attribute. If so, you can cast it to `Proposed_Basic_Attribute`.
- Call `is_ref_type` to test whether the proposed property is an object-reference attribute. If so, you can cast it to a `Proposed_Ref_Attribute`.
- Call `is_embedded_class_type` to test whether the proposed property is an embedded-class attribute. If so, you can cast it to a `Proposed_Embedded_Class_Attribute`.
- Call `is_varray_type` to test whether the proposed property is a VArray attribute. If so, you can cast it to a `Proposed_VArray_Attribute`. All proposed VArray attributes are represented by that class; you can call additional member functions to determine the kind of VArray:
  - Call `is_varray_basic_type` to test whether the proposed property is a numeric VArray attribute.
  - Call `is_varray_ref_type` to test whether the proposed property is an object-reference VArray attribute.
  - Call `is_varray_embedded_class_type` to test whether the proposed property is an object-reference VArray attribute.

Getting and Setting Information

After you have obtained a descriptor for a proposed property and cast it to the most specific class, you can call its member functions to examine and modify the definition of the property. The following table summarizes the kinds of information available for proposed properties of various kinds.

Property Type	Information	Can Be Changed
Any property	Name[a]	Yes
	Defining class	No
	Position in defining class	Yes[b]
	Access kind	Yes
	Number of elements in fixed-size array of values	Yes for attributes No for relationships
	Physical layout size of value	Not explicitly
	Previous name	No
Numeric attribute	Kind of numeric data in a value	Yes
	Default value	No
	Whether the proposed attribute has a default value	No
Object-reference attribute	Referenced class	Yes
	Whether values are stored as short object references	Yes
Embedded-class attribute	Class of a value	Yes
Numeric VArray attribute	Kind of numeric data in an element	Yes
Object-reference VArray attribute	Referenced class	Yes
	Whether elements are stored as short object references	Yes
Embedded-class VArray attribute	Class of an element	Yes

Property Type	Information	Can Be Changed
Relationship	Destination class	Yes
	Whether the relationship is bidirectional	Yes
	Whether the relationship is to-many	Yes
	Inverse relationship	Yes
	Copy mode	Yes
	Versioning mode	Yes
	Propagation behavior	Yes
	Whether the relationship is inline	Yes
	Whether the relationship is short	Yes
	Whether the inverse relationship is to-many	No
	Association number specified when proposed relationship was created	No

a. To get the name of a proposed property, call its inherited `name` member function.

b. To change the position of a proposed property, call the `move_property` member function of its containing proposed class.

Refer to the descriptions of the individual proposal-descriptor classes in Part 2 for details about the member functions that get and set information for the proposed property.

EXAMPLE This example changes all short references in properties of the class `Library` to standard references.

```
ooStatus rc;
proposed_property_iterator itr =
   library.defines_property_begin();
while (itr != library.defines_property_end()) {

   // Cast to non-constant proposed property so that
   // it can be modified.
   Proposed_Property &curProp = (Proposed_Property &)*itr;

   // Check for properties that use references
   if (curProp.is_relationship_type()) {
      // Relationship
      Proposed_Relationship &rel =
         (Proposed_Relationship &)curProp;
```

```
            if (rel.is_short())
               rc = rel.change_short(oocFalse);
            else {
               rc = oocSuccess;
         } // End if relationship

         else if (curProp.is_ref_type()) {
            // Object-reference attribute
            Proposed_Ref_Attribute &ref =
               (Proposed_Ref_Attribute &)curProp;
            if (ref.is_short())
               rc = ref.change_short(oocFalse);
            else {
               rc = oocSuccess;
         } // End if object-reference attribute

         else if (curProp.is_varray_ref_type()) {
            // Object-reference VArray attribute
            Proposed_VArray_Attribute &refArray =
               (Proposed_VArray_Attribute &)curProp;
            if (refArray.element_is_short())
               rc = refArray.change_element_short(oocFalse);
            else {
               rc = oocSuccess;
         } // End if object-reference VArray attribute

         else {
            // Attribute that doesn't use references
            rc = oocSuccess;
         }

         if (rc != oocSuccess) {
            cerr << " Couldn't change short references for ";
            cerr << curProp.name() << endl;
         }
         ++itr;
      } // End while more proposed properties
```

Activating Proposals

To activate all proposals in a module descriptor's proposal list, call the descriptor's `activate_proposals` member function. To activate all proposals in the proposal lists of all module descriptors, call the `activate_proposals` member function for a module descriptor for the top-level module. If you want to activate only the proposals in the top level module (and not all proposals), an optional final parameter to `activate_proposals` allows you to do so.

Activated proposals are first checked for consistency. Schema modification fails if any proposal or group of proposals is illogical or disallowed by Objectivity / DB. For example, if the proposals include a bidirectional relationship for which no inverse relationship is defined, the schema is not modified.

NOTE If your proposals include a bidirectional relationship with the source class in one module and the destination class in a different module, you *must* activate all proposals at once by calling `activate_proposals` for a module descriptor for the top-level module. The proposals for either of the two modules are not consistent by themselves, because they do not include a class that defines a necessary inverse relationship.

Active Schema performs the schema modification in a separate transaction. If a transaction is in process when you activate the proposals, Active Schema commits that transaction automatically before starting the schema-modification transaction. By default, Active Schema restarts the transaction in the same mode (MROW or not), with the same lock-waiting behavior and the same sensitivity of index updating as the transaction had when `active_proposals` was called. Optional parameters allow you to change these characteristics of the transaction when Active Schema restarts it after activating proposals.

EXAMPLE This example activates the proposals in the proposal list of the module descriptor `newMod`.

```
ooTrans trans;
ooHandle(ooFDObj) fdH;
...
ooStatus rc = newMod.activate_proposals(
   trans,    // Current transaction
   fdH);     // Handle for federated database
if (rc != oocSuccess) {
   cerr << "Proposal activation failed" << endl;
}
```

Activating Remote Schema Changes

If any other process has added classes to a module or caused evolution of existing classes in the module, you can use Active Schema to make those schema changes available to your process. To do so, you get a descriptor for the module and call its `activate_remote_schema_changes` member function. This member function returns a status code indicating whether activation succeeded. Its optional third parameter is a pointer to an unsigned integer. If you specify this parameter, the integer is set to the number of new shape descriptions that were added to the schema of the current process.

To activate remote changes to all modules, call the `activate_remote_schema_changes` member function for a module descriptor for the top-level module. If you want to activate only schema changes in the top level module (and not all schema changes), an optional final parameter to `activate_remote_schema_changes` allows you to do so.

NOTE A process's internal representation of the federated database schema is frozen during MROW transactions. As a consequence, you cannot activate remote schema if the current transaction is active and in MROW mode.

If the current transaction is not active when you activate the remote schema changes, Active Schema starts the transaction before attempting to modify the federated database schema and commits the transaction after the schema has been modified.

EXAMPLE This example activates the remote schema changes for the top-level module.

```
ooTrans trans;
ooHandle(ooFDObj) fdH;
size_t newShapes;
...
ooStatus rc = RWtopMod.activate_remote_schema_changes(
   trans,          // Current transaction
   fdH,            // Handle to federated database
   &newShapes);    // Pointer to result integer
if (rc == oocSuccess) {
   cout << newShapes << " new shapes added" << endl;
}
else {
   cerr << "Remote schema changes not activated" << endl;
}
```

Handling Evolution Messages

Various schema-evolution operations produce messages; Active Schema uses a function, called an *evolution message handler*, to handle messages that are produced by schema evolution. The default evolution message handler prints each message as a warning to the standard error stream. You can customize the behavior of Active Schema by defining an application-specific evolution message handler. Your evolution message handler should be a `void` function that takes one parameter of type `const char *`, the message string to be handled.

EXAMPLE The handler function `collect_evol_msg` does not produce output. Instead, it records schema evolution messages in the global string `evolMsgs`, which will contain all evolution messages that occur in a particular proposal activation.

```
char *evolMsgs;

static void collect_evol_msg(const char *newMsg)
{
   if (newMsg != NULL) {
      // Append the new message to global string
      char *oldMsgs = evolMsgs;
      evolMsgs = new char[strlen(oldMsgs) +
                          strlen(newMsg)  + 2];
      sprintf(evolMsgs, "%s%s\n", oldMsgs, newMsg);
   }
}
```

To install your message handler, you call the static member function `d_Module::set_evolution_message_handler`, passing a pointer to your function as the parameter. After doing so, your handler will be used instead of the default handler.

EXAMPLE The application that uses handler `collect_evol_msg` can install it with this statement:

```
// Install the message handler.
d_Module::set_evolution_message_handler(collect_evol_msg);
```

The evolution message handler is called for each message that results from schema modifications following a call to `activate_proposals`.

EXAMPLE Before activating proposals, the application that uses handler `collect_evol_msg` initializes the global string. If the proposals could not be activated successfully, the application writes the string to the standard error stream and then frees the string.

```
ooTrans trans;
ooHandle(ooFDObj) fdH;
...
// Initialize the message archive with a null string.
evolMsgs = new char[1];
*evolMsgs = '\0';

// Activate the proposals
try {
   ooStatus rc = RWtopMod.activate_proposals(trans, fdH);
}
catch (asException &exc) {
   cerr << exc << endl;
}
if (rc != oocSuccess) {
   cerr << "Proposal activation failed" << endl;
   // Write evolution error messages
   cerr << evolMsgs;
}
delete [] evolMsgs;
```

5

Modifying Persistent Data

Active Schema applications can modify the persistent objects in any Objectivity/DB federated database and add new persistent objects—even without C++ definitions of the objects' classes.

In This Chapter

Creating a New Basic Object

You can create a new basic object of any class defined in the schema by calling the static member function `Class_Object::new_persistent_object`. The parameters are a class descriptor for the class of the new object and a handle

to a database, container, or basic object. The handle is the clustering directive that controls where the new basic object is stored. The result is a class object for the new basic object. You can use this class object to set the properties of the new basic object.

All storage for the newly created basic object is initialized to contain zeros. No constructors are called (even if your application contains definitions of the class of the new basic object or its embedded classes). Thus:

- Numeric attributes are initialized to zero.
- Object-reference attributes are initialized to null object references.
- Attributes of recognized string classes are initialized to null strings.
- VArray attributes are initialized to null VArrays.
- The interpretation of zero data in an embedded-class attribute is dependent on that class.

If you create new basic objects, you are responsible for initializing the attributes to meaningful values.

EXAMPLE This example creates a basic object of the class `Test`, whose definition is shown on page 59. The class object `testCO` is used in later examples to set properties of the new `Test` object.

```
ooTrans trans;
trans.start();
...
ooHandle(ooDBObj) dbH;
... // Set dbH

// Get a class descriptor for the class of the new object
const d_Class &test = topMod.resolve_class("Test");

// Create the new Test object, clustering it in the
// database whose handle is dbH
Class_Object testCO =
   Class_Object::new_persistent_object(
      test, // Class descriptor for class of new object
      dbH); // Clustering directive
...
trans.commit();
```

Creating a New Container

If the federated database schema contains an application-defined container class (possibly a class that your Active Schema application defined), you can create a new container of that class by calling the static member function `Class_Object::new_persistent_container_object`. The parameters are:

- A class descriptor for the container class
- A handle to be used as a clustering directive for the new container
- The clustering factor for a hashed container (or zero for a non-hashed container)
- The initial number of pages allocated for the container
- The percentage by which the container should grow

Refer to Objectivity/C++ reference documentation for information about the characteristics of a container that are specified when it is created.

The `new_persistent_container_object` member function returns a class object that you can use to set application-defined properties of the container.

EXAMPLE This example defines a container class named `myContainer` that has a string attribute `usedFor`. It then creates a container of this class. The class object `containerCO` is used in a later example to set the `usedFor` attribute of the new container.

```
ooTrans trans;
ooHandle(ooFDObj) fdH;
trans.start();
...
ooHandle(ooDBObj) dbH;
... // Set dbH

// Get a modifiable descriptor for the top-level module
const d_Module &topMod = d_Module::top_level();
d_Module &RWtopMod = const_cast<d_Module &>(topMod);

// Define the new container class
Proposed_Class &propContainer =
   RWtopMod.propose_new_class("myContainer");
propContainer.add_base_class(
   oocLast,        // Position
   d_PUBLIC,       // Access kind
   "ooContObj");   // Base class name
```

```
propContainer.add_embedded_class_attribute(
   oocLast,        // Position
   d_PUBLIC,       // Access kind
   "usedFor",      // Attribute name
   1,              // # elements in fixed-size array
   "ooVString");   // Embedded class name

// Activate proposals, adding the new container
// class to the schema
ooStatus rc = RWtopMod.activate_proposals(trans, fdH);
... // Check that activation succeeded

// Get a class descriptor for the new container class.
const d_Class &container = topMod.resolve_class("myContainer");

// Create a new container of class myContainer, in the
// database whose handle is dbH
Class_Object containerCO =
   Class_Object::new_persistent_container_object(
      container,   // Class descriptor for class of new container
      dbH,         // Clustering directive
      0,           // Non-hashed container
      5,           // 5 pages initially
      10);         // Let container grow by 10% when needed to
                   // accommodate more basic objects
...
trans.commit();
```

Modifying a Persistent Object

You can use a class object for a persistent object to set the properties of that persistent object. The persistent object can be one that exists in the federated database that you are modifying or one that you just created and are initializing.

Automatic Updating

By default, Active Schema tries to obtain the necessary access for a persistent object that you modify by calling member functions of its class object, or member functions of a VArray object or a relationship object that accesses its data. Any such member function automatically calls the `update` member function on the handle for the associated persistent object, which opens that object for read / write access. This feature is called *automatic updating*.

Automatic updating is enabled by default. It can be disabled and re-enabled, respectively, with the static member functions `Persistent_Data_Object::disable_auto_update` and `Persistent_Data_Object::enable_auto_update`. You can test whether automatic updating is enabled by calling the static member function `Persistent_Data_Object::auto_update_is_enabled`.

When automatic updating is disabled, modification of persistent objects occurs as in any Objectivity/C++ application. That is, you can open an object in read mode and then write to it, but the change will not be committed unless some other operation has marked the page for update.

Setting Properties

Member functions of a class object allow you to set the values of some properties and to get persistent-data objects through which you can set the values of other properties. If a particular attribute contains a fixed-size array of values, each element of the array is set separately or accessed through a separate persistent-data object.

The following table shows how to set properties of various kinds.

Type of Property	Process for Setting Value
Numeric attribute	Call `set`
Object-reference attribute	Call `set_ooref`
Embedded string class attribute	■ Call `get_string` to get a string value ■ Use the string value to modify the string (page 124)
Embedded non-string class attribute	■ Call `get_class_obj` to get a class object ■ Use the class object to modify the embedded instance
VArray attribute	■ Call `get_varray` to get a VArray object ■ Use the VArray object to modify the VArray (page 126)
Relationship	■ Call `get_relationship` to get a relationship object ■ Use the relationship object to modify the associations (page 129)

You use position to identify the attribute whose value you want to set, as described in “Identifying Components” on page 64. If you need to test the type of a property, you can do so using an attribute descriptor for it, as described in “Accessing Component Data” on page 67.

The member functions that set attribute data allow you to specify an index in the fixed-size array of attribute values. If the attribute contains a single value instead of an array, you can omit the index parameter. These member functions return a status code that you can check to see whether the modification succeeded.

Numeric Attributes

To modify the value of a numeric attribute, call the class object's `set` member function for any element of the attribute's fixed-size array. You can specify the new value with a quantity of the correct Objectivity / C++ numeric type; the value is converted automatically into an Active Schema numeric value.

EXAMPLE This example sets the attribute `a` of the persistent object associated with the class object `testCO`. The inherited attribute `a` is declared as follows:

```
int32 a[10];
```

The example sets the fixed-size array of values to contain the numbers 100, 200, …, 1000.

```
ooTrans trans;
trans.start();
...
// Get position of attribute
const Class_Position aPos = testCO.position_in_class("a");
ooStatus rc = oocSuccess;
size_t i = 0;
int32 val = 100;
// Set each element of the fixed-size array
while ((i < 10) && (rc == oocSuccess)) {
   // Set current element
   rc = testCO.set(
         aPos,  // Position of attribute to be set
         i,     // Index of element to be set
         val);  // New value for element
   ++i;
   val += 100;
} // End while

if (rc != oocSuccess) {
   cerr << "Failed to set attribute a" << endl;
}
...
trans.commit();
```

Object-Reference Attribute

To modify the value of an object-reference attribute, call the class object's `set_ooref` member function for any element of the attribute's fixed-size array. The new value can be an object reference that you obtained from the federated database or one that you obtained by calling the `get_ooref` member function of some class object or VArray object. It can also be a class object; `Class_Object` defines conversion operators that can convert a class object to an object reference.

EXAMPLE This example sets the attribute `z` of the persistent object associated with the class object `testCO`. The attribute `z` is declared as follows:

```
ooRef(Test) z;
```

The example sets the attribute `z` to reference another newly created object of the `Test` class.

```
ooTrans trans;
trans.start();
...
// Get position of attribute
const Class_Position zPos =
       testCO.position_in_class("z");

// Create the new object to reference
Class_Object otherTestCO =
   Class_Object::new_persistent_object(
      test,     // Class descriptor for class of new object
      dbH);     // Clustering directive

// Set the attribute
ooStatus rc = testCO.set_ooref(
         zPos,           // Position of attribute to be set
         otherTestCO);   // New value for attribute
if (rc != oocSuccess) {
   cerr << "Failed to set attribute z" << endl;
}
...
trans.commit();
```

Modifying String Data

You can use a string value to modify the data in a string attribute or in an element of a string VArray. If necessary, call the string object's `type` member function to find out what type of string object the string value contains.

NOTE If you modify a persistent object's string data through an instance of `ooVString`, `ooUtf8String`, `ooSTString`, or `Optimized_String_Value`, you are responsible for opening the persistent object for read / write access. Automatic updating does not affect modifications made through instances of these classes.

Internal String Class

If a string value contains an instance of an internal string class (`ooVString`, `ooUtf8String`, or `ooSTString`), convert the string value to the appropriate internal string class, then modify the string as you would in any Objectivity / C++ application.

EXAMPLE This example sets the `usedFor` attribute of the container associated with the class object `containerCO`. The `usedFor` attribute is an embedded `ooVString` attribute; it is set to contain the string `"Objects added by Leslie Jones"`.

```
ooTrans trans;
trans.start();
...
// Get position of attribute
const Class_Position usedPos =
       containerCO.position_in_class("usedFor");

// Open the container for update access because automatic
// updating does not apply to changes made through string
// values
containerCO.object_handle().update();

// Get the string value for the usedFor attribute
String_Value strVal = containerCO.get_string(usedPos);

// Convert the string value to ooVString
ooVString *vStr = strVal;
```

```
// Set the ooVString, which sets the usedFor attribute
*vStr = "Objects added by Leslie Jones";
...
trans.commit();
```

Optimized String Class

If a string value contains an instance of an application-defined optimized string class `ooString(N)`, use the string value to the constructor as an optimized string value that accesses the string data. Then call the optimized string value's `set` member function to set the string's value.

EXAMPLE This example sets the `name` attribute of the persistent object associated with the class object `CO`. The `name` attribute is an embedded `ooString(20)` attribute; it is set to contain the string `"Marty Weiss"`.

```
ooTrans trans;
trans.start();
...
Class_Object CO = ...

// Get position of attribute
const Class_Position namePos = CO.position_in_class("name");

// Open the persistent object for update access because
// automatic updating does not apply to changes made
// through string values
CO.object_handle().update();

// Get the string value for the name attribute
String_Value strVal = CO.get_string(namePos);

// Convert the string value to an optimized string value
Optimized_String_Value optStr(strVal);

// Set the optimized string value, which sets the name
// attribute
optStr.set("Marty Weiss");
...
trans.commit();
```

Modifying VArray Data

You can use a VArray object to modify the data in a VArray attribute.

Changing the Array Size

If you obtain a VArray object from a class object for a newly created persistent object, the associated VArray is null. You may want to initialize the VArray to contain some elements. If so, you can change the size of the array to accommodate the desired number of elements, then set each element to the desired value.

To change the size of a VArray, call the VArray object's `resize` member function, specifying the desired number of elements. You can increase or decrease the array size.

You can add an element to the end of a numeric or object-reference VArray by calling the VArray object's `extend` member function.

Setting an Element

Each element of a VArray is set separately or accessed through a separate persistent-data object. Depending on the VArray's element type, you can call a member function of its VArray object to set an element or to get a persistent-data object through which you can set an element.

The following table shows how to set elements for VArrays of various element types.

Element Type of VArray	Process for Setting Value
Numeric type	Call `set` or `replace_element_at`
Object-reference type	Call `set_ooref` or `replace_element_at`
Embedded string class	■ Call `get_string` to get a string value ■ Use the string value to modify the string (page 124)
Embedded non-string class	■ Call `get_class_obj` to get a class object ■ Use the class object to modify the embedded instance

The member functions that set an element take as parameters the array index of the element to be changed and the new value for that element. These member functions return a status code that you can check to see whether the modification succeeded.

EXAMPLE This example sets the attribute `c` of the persistent object associated with the class object `testCO`. The inherited attribute `c` is declared as follows:

```
ooVArray(uint16) c;
```

The example sets the VArray to contain 10 elements: 10, 20, …, 100.

```
ooTrans trans;
trans.start();
...
// Get position of attribute
const Class_Position cPos = testCO.position_in_class("c");

// Get the VArray object for the c attribute
VArray_Object VO = testCO.get_varray(cPos);

// Set the size of the VArray to 10 elements
ooStatus rc = VO.resize(10);

size_t i = 0;
uint16 val = 10;

// Set each element of the VArray
while ((i < 10) && (rc == oocSuccess)) {
   // Set current element
   rc = VO.set(
          i,    // Index of element to be set
          val); // New value for element
   ++i;
   val += 10;
} // End while

if (rc != oocSuccess) {
   cerr << "Failed to set attribute c" << endl;
}
...
trans.commit();
```

Replacing Elements During Iteration

If the element type of the VArray is a numeric type or an object-reference type, you can iterate through its elements, and replace some or all of the elements. To do so, first call the VArray object's `create_iterator` member function to get a VArray iterator for the elements of the VArray. This member function returns an Objectivity/C++ VArray iterator of the class `d_Iterator<ooObj>`. As described

in “Iterating Through the Elements” on page 78, you must then cast the VArray iterator to the appropriate class for the element type of the associated VArray.

When you reach an element that you want to replace, you call the VArray object’s `replace_element_at` member function. The parameters are the new value for the current element and the VArray iterator. The VArray iterator parameter is of type:

```
const d_Iterator<ooObj> &
```

As a consequence, you must cast the VArray iterator to that type before passing it to `replace_element_at`.

EXAMPLE This example iterates through the elements of a `float32` VArray, increasing each element by ten percent.

```
ooTrans trans;
trans.start();
...
VArray_Object VO = ... ;

// Get VArray iterator for float32 elements
d_Iterator<float32> dit =
   (d_Iterator<float32> &)VO.create_iterator();

// Use the VArray iterator to increase each element
while (dit.not_done()) {
   // Replace current element by a value that is 10% larger
   VO.replace_element_at(
      dit.get_element() * 1.1,           // New value
      (const d_Iterator<ooObj> &)dit);   // VArray iterator
   ++dit;
}
...
trans.commit();
```

Modifying Relationship Data

You can use a relationship object to modify the associations in a relationship of a persistent object. The member functions that modify associations use a constant object handle to specify the new destination object. You can use a handle that you obtained from the federated database. Alternatively, you can use a class object for the destination object; `Class_Object` defines conversion operators that can convert a class object to a handle.

Modifying a To-One Relationship

If the relationship is to-one, you can call the relationship object's `set` member function to set the destination object. You can call its `del` member function to remove any existing association, leaving the source object with no associated destination object.

EXAMPLE The class `Test` has a to-one unidirectional relationship `x`, defined as follows:

```
ooRef(Test) x : copy(delete);
```

This example sets the relationship `x` for the persistent object associated with the class object `testCO` to contain an association to the persistent object associated with the class object `otherTestCO`. It also deletes any association in the `x` relationship of the latter persistent object.

```
ooTrans trans;
trans.start();
...
// Get position of relationship x in class Test
const Class_Position xPos = testCO.position_in_class("x");

// Get the relationship object for relationship x of testCO
Relationship_Object RO = testCO.get_relationship(xPos);

// Set the association from testCO to otherTestCO
RO.set(otherTestCO);

// Get the relationship object for relationship x of
// otherTestCO
Relationship_Object otherRO =
   otherTestCO.get_relationship(xPos);
```

```
// Delete any association from otherTestCO's relationship x
otherRO.del();
...
trans.commit();
```

Modifying a To-Many Relationship

If the relationship is to-many, you call the relationship object's `add` member function to add an association to the specified destination object. You can call its `sub` member function to remove the existing association to a specified destination object. You can call its `del` member function to remove all existing associations from the source object.

EXAMPLE This example removes widget from the products of the Salinas factory and adds warbler to the products of that factory. The class `Factory`, whose definition appears on page 94, has a to-many unidirectional relationship `products`, defined as follows:

```
ooRef(Product) products[] : copy(delete);
```

Factory names are used as scope names in a particular database. Product names are used as scope names in a different database. This example looks up the factory named `"Salinas"` and the products named `"Widget"` and `"Warbler"`. It then modifies the `products` relationship of the factory, adding an association to `Warbler` and removing the association to `Widget`.

The function `errorEnd` (not shown) prints an error message, aborts the transaction, and returns an error code.

```
ooTrans trans;
trans.start();
...
// Open the factory database
ooHandle(ooDBObj) factDbH;
if (dbH.open(factDbH, "factoryDB", oocUpdate) != oocSuccess) {
   return errorEnd("Failed to open factoryDB", &trans);
}
// Look up the Salinas factory
ooHandle(ooObj) salinasH;
ooStatus rc =
   salinasH.lookupObj(factDbH, "Salinas", oocUpdate);
if (rc != oocSuccess) {
   return errorEnd("Can't find Salinas factory", &trans);
}
```

```
// Open the product database
ooHandle(ooDBObj) prodDbH;
if (dbH.open(prodDbH, "productDB", oocUpdate) != oocSuccess) {
   return errorEnd("Failed to open productDB", &trans);
}
// Look up the Widget product
ooHandle(ooObj) widgetH;
ooStatus rc =
   widgetH.lookupObj(prodDbH, "Widget", oocRead);
if (rc != oocSuccess) {
   return errorEnd("Can't find Widget product", &trans);
}
// Look up the Warbler product
ooHandle(ooObj) warblerH;
ooStatus rc =
   warblerH.lookupObj(prodDbH, "Warbler", oocRead);
if (rc != oocSuccess) {
   return errorEnd("Can't find Warbler product", &trans);
}
// Construct a class object for the Salinas factory
Class_Object CO = Class_Object(salinasH);

// Get position of the products relationship in class Factory
const Class_Position pos = CO.position_in_class("products");

// Get the relationship object for the products relationship
Relationship_Object RO = CO.get_relationship(pos);

// Add an association from the Salinas factory to Warbler
RO.add(warblerH);

// Remove the association from the Salinas factory to Widget
RO.sub(widgetH);
…
trans.commit();
```

Object Conversion

A single Active Schema application can modify the schema description of a class and perform any necessary object conversion on persistent objects of that class. The application can perform as many schema-evolution cycles as necessary. For example, if the value of a new attribute is computed from the value of a deleted attribute, the first modification step could add the new attribute and initialize its value; the second step could delete the old attribute.

EXAMPLE This example changes the `readings` attribute of the class `Instrument` from a `float64` attribute with a fixed-size array of 10 elements to a `float64` VArray attribute. The `readings` attribute is defined by the class `Instrument`, not inherited, so its class position within `Instrument` can be converted to an attribute position.

The function `errorEnd` (not shown) prints an error message, aborts the transaction, and returns an error code.

```
ooHandle(ooFDObj) fdH;
ooTrans trans;
trans.start();
ooStatus rc;
...
// Lock the schema
rc = d_Module::lock_schema(11223344556677);
if (rc != oocSuccess) {
   return errorEnd("Couldn't lock schema");
}

// Get modifiable descriptor for the top-level module
const d_Module &topMod =
   d_Module::top_level(11223344556677);
d_Module &RWtopMod = const_cast<d_Module &>(topMod);

// CYCLE 1: Add a temporary VArray attribute

// Get a proposed class for Instrument
Proposed_Class &instr1 =
   RWtopMod.propose_evolved_class("Instrument");
```

```
// Get the class position of the readings attribute
Class_Position readPos =
   instr1.position_in_class("readings");
if (!readPos.is_convertible_to_uint()) {
   return errorEnd("Attribute readings is inherited");
}
size_t readAttrPos = (size_t)readPos;

// Add the new VArray attribute before readings
rc = instr1.add_varray_attribute(
   readAttrPos,       // Position
   d_PUBLIC,          // Access kind
   "temp_readings",   // Attribute name
   1,                 // # elements in fixed-size array
   ooFLOAT64 );       // Type of numeric elements
if (rc != oocSuccess) {
   return errorEnd("Couldn't add attribute");
}
// Get the new position of the readings attribute
// while instr1 is still valid
readPos = instr1.position_in_class("readings");

// Get the position of the temp_readings attribute
// while instr1 is still valid
const Class_Position tempPos =
   instr1.position_in_class("temp_readings");

// Activate the Cycle 1 proposals
rc = RWtopMod.activate_proposals(trans, fdH);
if (rc != oocSuccess) {
   return errorEnd("Cycle 1 failed");
}

// Convert objects

// Initialize an object iterator for all persistent objects in
// the federated database
ooItr(ooObj) objItr1;
objItr1.scan(fdH);

// Examine each object, updating Instrument objects
while (objItr1.next()) {

   // Get a handle for the current object
   ooHandle(ooObj) curObjH(objItr1);
```

```
      // Construct a class object to access the current object
      Class_Object curCO(curObjH);
      const d_Class &classOfObj = curCO.type_of();

      // If the object is an Instrument, update it
      if (!strcmp(classOfObj.name(), "Instrument")) {
         // Get the VArray object for the new attribute
         VArray_Object VO = curCO.get_varray(tempPos);

         // Set the size of the VArray to 10 elements
         rc = VO.resize(10);
         if (rc != oocSuccess) {
            objItr1.end();
            return errorEnd("Couldn't set the VArray size");
         }

         // Set each element of the temp_readings VArray
         // from the corresponding element of the readings array
         for (size_t i = 0; i < 10; ++i) {
            // Replace current element of VArray
            VO.replace_element_at(
               curCO.get(readPos, i), // New value
               i);                    // Index of VArray element
         } // End for each element
      } // End if object is an instrument
   } // End while more objects in federated database

   // CYCLE 2: Delete old readings attribute

   // Get a new proposed class for Instrument; instr1 is invalid
   Proposed_Class &instr2 =
      RWtopMod.propose_evolved_class("Instrument");

   // Delete the old attribute
   rc = instr2.delete_property("readings");
   if (rc != oocSuccess) {
      return errorEnd("Couldn't delete old attribute");
   }

   // Activate the Cycle 2 proposals
   rc = RWtopMod.activate_proposals(trans, fdH);
   if (rc != oocSuccess) {
      return errorEnd("Cycle 2 failed");
   }
```

```
// CYCLE 3: Rename temp_readings to readings

// Get a new proposed class for Instrument; instr1 and instr2
// are invalid
Proposed_Class &instr3 =
   RWtopMod.propose_evolved_class("Instrument");

// Get a proposed property for the temp_readings attribute
Proposed_Property &newProp =
   instr3.resolve_property("temp_readings");

// Rename the property
rc = newProp.rename("readings");
if (rc != oocSuccess) {
   return errorEnd("Couldn't rename new attribute");
}

// Activate the Cycle 3 proposals
rc = RWtopMod.activate_proposals(trans, fdH);
if (rc != oocSuccess) {
   return errorEnd("Cycle 3 failed");
}

// Unlock the schema
rc = d_Module::unlock_schema(11223344556677);
if (rc != oocSuccess) {
   return errorEnd("Couldn't unlock schema");
}
…
trans.commit();
```

6

Working With Iterators

All Active Schema iterators follow the ODMG convention of modeling the C++ standard template library (STL) interface.

In This Chapter

About Iterators

An *iterator* is an instance of an iterator class. An iterator allows you to step through a group of items, called the iterator's *iteration set*. During iteration, the iterator keeps track of its *current position* within its iteration set. The element at the current position is called the iterator's *current element*.

The iteration set for any Active Schema iterator contains either entities in the federated database schema or proposed changes to the schema. The iterator gets a descriptor for each element of the iteration set. The name of an iterator class typically indicates the kind of descriptor it returns. For example, an iterator of class `attribute_iterator` returns attribute descriptors; an iterator of class `proposed_class_iterator` returns proposed classes.

"Iterator Classes" on page 152 lists all iterator classes.

Actual and Loop-Control Iterators

To step through an iteration set, you use two instances of the same iterator class.

- The *actual iterator* that steps through the iteration set is initialized so that its current element is the first element in the iteration set.
- The *loop-control iterator* represents the termination condition for the actual iterator—that is, its current position is after the last element in the iteration set.

When the two iterators are equal, the actual iterator has finished stepping through its entire iteration set.

To get the actual iterator, you call a member function whose name ends with the `_begin` suffix; to get the corresponding loop-control iterator, you call a member function with a similar name that ends with the `_end` suffix. For example, to iterate through descriptors of all entities in the scope of a module, you call the module descriptor's `defines_begin` member function to obtain the actual iterator, and you call the same module descriptor's `defines_end` member function to obtain the loop-control iterator.

Returned Descriptors

Iterators of all classes return constant descriptors. If you use the descriptor only to examine the schema, a constant descriptor type will suffice. However, if you obtain a module descriptor for which you want to propose schema changes or if you obtain a proposal descriptor that you want to modify, you must first cast the descriptor to the non-constant type.

For example, the class `proposed_class_iterator` returns constant proposed classes of the type:

```
const Proposed_Class &
```

If you want to modify a proposed class that you obtain from the iterator, you must first cast it to the non-constant type:

```
Proposed_Class &
```

Stepping Through the Iteration Set

All iterator classes overload the dereference operator (*) to return the iterator's current element. At each step through the iteration set, you use this operator to get the current element.

All iterator classes also overload both the prefix and the postfix increment operator (++). After processing the current element, you use this operator to advance the iterator's position.

NOTE You typically use the prefix increment operator, which advances the iterator and then returns it. The postfix operator returns a *copy* of the iterator before its position is advanced.

As you step through the iteration set, you test for termination by comparing the actual iterator with the loop control operator. The iterator classes overload both the equality operator (==) and the inequality operator (!=). You can use whichever of these two operators is appropriate for the logic of your loop. For example, you can continue a loop while the two iterators are different or until the two iterators are the same.

EXAMPLE A descriptor iterator of the class `meta_object_iterator` steps through the entities in the scope of some particular module or class. The following code fragment obtains a descriptor iterator for the entities in the scope of the top-level module. It obtains a descriptor for each entity and passes that descriptor to the function `display_descriptor` (not shown).

```
const d_Module &topMod = d_Module::top_level();
meta_object_iterator itr = topMod.defines_begin();
while (itr != topMod.defines_end()) {
   const d_Meta_Object &descriptor = *itr;
   display_descriptor(descriptor);
   ++itr;
}
```

Iteration Order

The order in which an iterator finds elements of its iteration set is not specified. However, all iterators for a particular iteration set are guaranteed to find elements in the same order. For example, suppose you get a descriptor for a particular module and call its `defines_begin` member function to get a descriptor iterator. That iterator finds the entities in the module's scope in some particular order. If you obtain a different descriptor for the same module and call that module descriptor's `defines_begin` member function, the resulting iterator will find the entities in the module's scope in the same order as did the first iterator.

Modifications to the schema can change iteration order. In the preceding example, if the application modified the schema between iterating with the first iterator and with the second, the iteration order may be different. Similarly, if two processes that started with the schema in different states iterate over the same scope, they may find the entities in that scope in two different orders.

7

Error Handling

Any inappropriate attempt to access or modify persistent data or a schema description results in an error condition. System-level error conditions result in standard Objectivity/C++ errors; user-level error conditions can result in either C++ exceptions or standard Objectivity/C++ errors.

In This Chapter

Errors and Exceptions

System-level error conditions result in standard Objectivity/C++ errors. By default, Active Schema throws a C++ exception when a user-level error occurs. Interactive applications built with Active Schema can provide robust behavior by catching exceptions corresponding to user errors that can be anticipated.

If you prefer, you can configure Active Schema to signal standard Objectivity/C++ errors instead of throwing exceptions.

Error and Exception Classes

The class asError is the abstract base class for all Active Schema error and exception classes; it defines the data and behavior common to both system- and user-level error conditions. You can call an error object's is_system_error member function to test whether it represents a system-level or user-level error; you can call its code member function to obtain the corresponding Objectivity / DB error code. This class defines operator const char *, which converts an error object into its error message string.

Error classes correspond to system-level error conditions; they are direct derived classes of asError. When a system-level error occurs, Active Schema signals an error with an ooError structure containing the error code of the corresponding error class, and the error message describes the situation that caused the error.

Exception classes correspond to user-level error conditions; they are derived classes of asException.

Enabling and Disabling Exceptions

When a user-level error condition occurs, if exceptions are enabled (the default), Active Schema throws an exception of the appropriate class; its error message describes the situation that caused the error. If exceptions are disabled, Active Schema signals an error with an ooError structure containing the error code of the corresponding exception class and the error message.

- To disable exceptions, call the static member function asException::disable_exceptions. After doing so, Active Schema will signal standard Objectivity / C++ errors.
- To enable exceptions after they have been disabled, call the static member function asException::enable_exceptions. After doing so, Active Schema will once again throw exceptions.
- To test whether exceptions are enabled, call the static member function asException::exceptions_are_enabled.

Checking Status Codes

Any Active Schema member function that modifies the schema or modifies persistent data returns an Objectivity/C++ status code of type `ooStatus`. You should follow Objectivity/C++ programming conventions when calling these functions. In particular, you should check the returned code and proceed only if the code is `oocSuccess`.

EXAMPLE This code fragment activates proposed modifications to the schema. It indicates an error if the modifications failed.

```
if (module.activate_proposals(trans, fdH) != oocSuccess) {
   cerr << "Evolution failed" << endl;
   return 1;
}
```

Catching Exceptions

If your application enables exceptions, you should catch any exceptions that can be anticipated. In Part 2 of this book, descriptions of member functions list the exceptions that they may throw. In addition, you can catch `asException` to handle any exception that may arise.

EXAMPLE In this code fragment, the VArray object `vObj` contains a VArray of 32-bit integer values. The `index` variable is set to the user-specified index of an element of the VArray. The application looks up the indicated element, catching `VArrayBoundsError` exceptions in case the user entered an inappropriate index.

```
VArray_Object vObj;
size_t index;
uint32 vbVal;
... // Set vObj and get index from the user
try {
   vbVal = vObj.get(index);
}
catch(VArrayBoundsError &exc) {
   ... // Report exception
}
```

Schema Failures

A problem or failure in a different application may leave the federated database schema in a corrupted state. When that happens, your application may experience inexplicable schema failure, such as a crash or inability to open or close a container in the system database.

You can attempt to recover from such a failure by calling the static member function `d_Module::sanitize`. That member function uses your process' internal representation of the federated database schema to restore any corrupted class descriptions.

Part 2 REFERENCE

Active Schema Programming Interface

The Active Schema interface for examining the schema of a federated database adheres as closely as possible to the ODMG 2.0 schema-access specification. Active Schema omits the ODMG concepts of persistent member functions and class exceptions because there is no Objectivity / DB equivalent. Active Schema approximates other ODMG concepts, giving them different names because their specifications do not correspond exactly to the functionality that Objectivity / DB provides. For example, there is no class `d_Primitive_Type`, but rather a class `Basic_Type` encapsulating the Objectivity built-in primitives, such as `int32` and `float64`.

Classes described in the ODMG specification have names that begin with the `d_` prefix. Active Schema augments the definitions of most of these classes with additional member functions that are not part of the ODMG specifications. Classes that are unique to Active Schema have names that do not begin with the `d_` prefix.

Global Types and Constants

Active Schema introduces several non-class types and constants that are used as parameters to, and return values from, various member functions. Those types and constants are described in detail in “Global Types and Constants” on page 153 and are listed in “Types and Constants Index” on page 559.

Classes

The classes in the Active Schema programming interface can be classified as follows:

1. Classes that describe the schema of the federated database
2. Classes that access persistent data
3. Classes that describe proposed changes to the federated database schema
4. Iterator classes
5. Error and exception classes

The following tables contain summaries of classes in the first four categories. Detailed descriptions of those classes appear in alphabetical order, one per chapter, starting on page 165.

Descriptions of the error and exceptions classes appear in alphabetical order in the chapter “Error and Exception Classes” starting on page 457.

“Classes Index” on page 537 contains an alphabetical list of all classes, with member functions listed under each class. “Functions Index” on page 547 contains an alphabetical list of all functions, including member functions.

Classes that Describe the Schema

The following classes are used to describe the contents of the federated database schema. They consist of the schema-descriptor classes, which describe entities in the schema, plus two classes (`d_Inheritance` and `Class_Position`) that describe interrelationships between entities in the schema.

Class	Description
`Attribute_Type`	Abstract base class of descriptor classes for attribute types
`Basic_Type`	Descriptor class for basic numeric types
`Bidirectional_Relationship_Type`	Descriptor class for bidirectional relationship types
`Class_Position`	Class describing the class position of an attribute within the physical layout of a class that defines or inherits the attribute
`d_Attribute`	Descriptor class for attributes
`d_Class`	Descriptor class for classes
`d_Collection_Type`	Abstract base class of descriptors for collection types
`d_Inheritance`	Descriptor for inheritance connections between classes

Class	Description
d_Meta_Object	Abstract base class for all schema descriptors and all proposal descriptors
d_Module	Descriptor class for modules
d_Property	Abstract base class of descriptors for properties
d_Ref_Type	Descriptor class for object-reference types
d_Relationship	Descriptor class for relationships
d_Scope	Abstract base class of classes that organize the entities in the federated database schema
d_Type	Abstract base class of descriptors for all types
Property_Type	Abstract base class of descriptors for property types
Relationship_Type	Abstract base class of descriptors for relationship types
Top_Level_Module	Descriptor class for the top-level module in a schema
Unidirectional_Relationship_Type	Descriptor class for unidirectional relationship types
VArray_Basic_Type	Descriptor class for numeric VArray types
VArray_Embedded_Class_Type	Descriptor class for embedded-class VArray types
VArray_Ref_Type	Description class for object-reference Varray types

Persistent-Data Classes

The following classes serve as self-describing data types for persistent data.

Class	Description
`Class_Object`	Self-describing data type for persistent objects
`Collection_Object`	Abstract base class for classes that serve as self-describing data types for collections
`Numeric_Value`	Self-describing data type for numeric values
`Optimized_String_Value`	Self-describing data type for optimized strings of an application-defined class `ooString(N)`
`Persistent_Data_Object`	Abstract base class for classes that serve as self-describing data types for structured persistent data
`Relationship_Object`	Self-describing data type for relationships between persistent objects
`String_Value`	Self-describing data type for strings
`VArray_Object`	Self-describing data type for variable-size arrays of elements of the same data type

Proposal-Descriptor Classes

The following classes are used to describe proposed changes to the federated database schema.

Class	Description
Proposed_Attribute	Abstract base class for descriptors of the attributes of a proposed class
Proposed_Base_Class	Descriptor class for base classes of a proposed class
Proposed_Basic_Attribute	Descriptor class for numeric attributes of a proposed class
Proposed_Class	Descriptor class for proposed classes to be added to the schema
Proposed_Collection_Attribute	Abstract base class for descriptors of the collection attributes of a proposed class
Proposed_Embedded_Class_Attribute	Descriptor class for embedded-class attributes of a proposed class
Proposed_Property	Abstract base class for descriptors of the properties of a proposed class
Proposed_Ref_Attribute	Descriptor class for object-reference attributes of a proposed class
Proposed_Relationship	Descriptor class for relationships of a proposed class
Proposed_VArray_Attribute	Descriptor class for VArray attributes of a proposed class

Iterator Classes

The following iterator classes provide access to descriptors of entities within the schema and descriptors of proposed schema changes.

Iterator Class	Iterates Through
`attribute_iterator`	Attributes defined in a particular class (including relationships and embedded-class attributes corresponding to base classes)
`attribute_plus_inherited_iterator`	Attributes defined in or inherited by a particular class (including relationships and embedded-class attributes corresponding to base classes)
`base_class_plus_inherited_iterator`	Embedded-class attributes corresponding to all ancestor classes of a particular class
`collection_type_iterator`	Collection types using a particular component type (for example, as their element type)
`inheritance_iterator`	Inheritance connections between a particular class and either its immediate parent classes or its child classes
`list_iterator<element_type>`	List of elements of type *`element_type`*
`meta_object_iterator`	Entities in a particular scope
`module_iterator`	Named modules in the schema
`property_iterator`	Properties that use a particular type
`proposed_base_class_iterator`	Base classes of a proposed class
`proposed_class_iterator`	Proposed classes in the proposal list of a particular module descriptor
`proposed_property_iterator`	Properties of a particular proposed class
`ref_type_iterator`	Reference types that use a particular type
`relationship_iterator`	Relationships defined in a particular class
`type_iterator`	Types in a particular scope

Global Types and Constants

This chapter describes the global types and constants used in the Active Schema programming interface. Types and constants of standard C++ data types are listed alphabetically. Constants of Active Schema data types are listed under the corresponding type.

See:

- “Reference Index” on page 153 for an alphabetical list of global names
- “Reference Descriptions” on page 155 for individual descriptions

Reference Index

oocCurrentMrow	Indicates the concurrent access policy for current transaction.
oocCurrentSensitivity	Indicates the sensitivity of index updating for the current transaction; that is, when indexes are updated relative to when indexed objects are updated.
oocCurrentTransWait	Indicates the lock-waiting behavior for the current transaction.
oocLast	Indicates the last attribute position in a proposed class.
oocLatestVersion	Indicates the version number for the latest version of a class.
oocNoID	Indicates a described entity has no ID (because it is a proposed schema change and not an existing entity in the schema).
ooFloatType	The kind of floating-point data type for a proposed attribute of a proposed class.
ooIntegerType	The kind of integer data type for a proposed attribute of a proposed class.
ooNumberType	The kind of integer data type for a proposed attribute of a proposed class.
ooPTR_t	The kind of integer data type for a proposed attribute of a proposed class.
ooUINT64_t	The kind of integer data type for a proposed attribute of a proposed class.

Reference Descriptions

d_Access_Kind

global type

The access kind or visibility declared for a base class or attribute of a class.

Constants

`d_INVALID`
: An invalid access kind.

`d_PRIVATE`
: Private access.

`d_PROTECTED`
: Protected access.

`d_PUBLIC`
: Public access.

d_Kind

global type

The kind of collection in an attribute of a class.

Constants

`ARRAY`
: Variable-size array. (This is the only kind of collection attribute that Objectivity / DB supports.)

`BAG`
: Bag.

`DICTIONARY`
: Dictionary.

`LIST`
: List.

`SET`
: Set.

`STL_LIST`
: Standard Template Library (STL) list.

`STL_MAP`
: STL map.

`STL_MULTIMAP`
: STL multimap.

`STL_MULTISET`
STL multiset.

`STL_SET`
STL set.

`STL_VECTOR`
STL vector.

d_Ref_Kind

global type

The kind of reference described by an attribute descriptor.

Constants

`REF`
Object reference to a persistent object. (This is the only reference kind that Objectivity / DB supports.)

`POINTER`
Pointer to the referenced data.

d_Rel_Kind

global type

The kind of relationship described by a relationship descriptor.

Constants

`REL_REF`
Relationship to a single object; that is, a to-one relationship.

`REL_SET`
Relationship to a set of objects. Objectivity / DB does not support this kind of relationship.

`REL_LIST`
Relationship to a list of objects; that is, a to-many relationship.

ooAsAddModuleErrorCode

global type

The reason why an attempt to add a new module failed.

Constants

`NULL_NAME`
No name for the module was given.

`NAME_ALREADY_USED`
The federated database already contains a module with the name specified for the new module.

`CREATE_FAILED`

A valid name was given, but Active Schema was unable to create the new module.

ooAsStringType

global type

The kind of string class of an attribute.

Constants

`ooAsStringOPTIMIZED`

An optimized string class `ooString(`*`N`*`)`

`ooAsStringNONE`

Not a string class

`ooAsStringST`

The Smalltalk string class `ooSTString`

`ooAsStringUTF8`

The Unicode string class `ooUtf8String`

`ooAsStringVSTRING`

The ASCII string class `ooVString`

ooAsType

global type

The kind of data required by an Active Schema operation.

Constants

`Basic_Type_t`

A basic numeric type.

`Bidirectional_Relationship_Type_t`

A bidirectional relationship type.

`Class_Object_t`

Self-describing data containing an instance of a class.

`Class_Or_Ref_Type_t`

An embedded-class type or an object-reference type.

`d_Alias_Type_t`

An alias type. (Objectivity / DB currently does not support alias types in the schema.)

`d_Attribute_t`

An attribute of a class in the schema.

`d_Class_t`

A class in the schema.

`d_Collection_Type_t`

A collection type.

`d_Constant_t`

A constant. (Objectivity / DB currently does not support constants in the schema.)

`d_Exception_t`

An exception. (Objectivity / DB currently does not support exceptions in the schema.)

`d_Inheritance_t`

An inheritance connection between a parent class and a child class in the schema.

`d_Keyed_Collection_Type_t`

A keyed collection type. (Objectivity / DB currently does not support keyed collection types in the schema.)

`d_Meta_Object_t`

A descriptor.

`d_Module_t`

A module.

`d_Operation_t`

An operation. (Objectivity / DB currently does not support operations in the schema.)

`d_Parameter_t`

A parameter to an operation. (Objectivity / DB currently does not support operations or their parameters in the schema.)

`d_Property_t`

A property of a class in the schema.

`d_Ref_Type_t`

An object-reference type.

`d_Relationship_t`

A relationship between two classes in the schema.

`d_Scope_t`

A scope that organizes entities in the schema.

`d_Type_t`
: A type used in the schema.

`None_t`
: An unrecognized type.

`Numeric_Value_t`
: A numeric value.

`Proposed_Base_Class_t`
: A proposed base class of a proposed class.

`Proposed_Basic_Attribute_t`
: A proposed numeric attribute of a proposed class.

`Proposed_Class_t`
: A proposed class.

`Proposed_Embedded_Class_Attribute_t`
: A proposed embedded-class attribute of a proposed class.

`Proposed_Ref_Attribute_t`
: A proposed object-reference attribute of a proposed class.

`Proposed_Relationship_t`
: A proposed relationship of a proposed class.

`Proposed_VArray_Attribute_t`
: A proposed VArray attribute of a proposed class.

`Relationship_Object_t`
: Self-describing data containing a relationship between instances.

`Relationship_Type_t`
: A relationship type.

`Short_Ref_Type_t`
: A short object-reference type.

`Unidirectional_Relationship_Type_t`
: A unidirectional relationship type.

`VArray_Basic_Type_t`
: A numeric VArray type.

`VArray_Class_Or_Ref_Type_t`
: An embedded-class or object-reference VArray type.

`VArray_Embedded_Class_Type_t`
An embedded-class VArray type.

`VArray_Object_t`
Self-describing data containing a VArray.

`VArray_Ref_Type_t`
An object-reference VArray type.

ooBaseType

global type

The kind of numeric data type for an attribute of a class.

Constants

`ooCHAR`
8-bit character.

`ooINT8`
8-bit signed integer.

`ooINT16`
16-bit signed integer.

`ooINT32`
32-bit signed integer.

`ooINT64`
64-bit signed integer.

`ooFLOAT32`
32-bit (single-precision) floating-point number.

`ooFLOAT64`
64-bit (double-precision) floating-point number.

`ooNONE`
Invalid or unrecognized numeric type.

`ooPTR`
32-bit pointer.

`ooUINT8`
8-bit unsigned integer.

`ooUINT16`
16-bit unsigned integer.

`ooUINT32`
32-bit unsigned integer.

ooUINT64

64-bit unsigned integer.

oocCurrentMrow global constant

Indicates the concurrent access policy for current transaction.

oocCurrentSensitivity global constant

Indicates the sensitivity of index updating for the current transaction; that is, when indexes are updated relative to when indexed objects are updated.

oocCurrentTransWait global constant

Indicates the lock-waiting behavior for the current transaction.

oocLast global constant

Indicates the last attribute position in a proposed class.

oocLatestVersion global constant

Indicates the version number for the latest version of a class.

oocNoID global constant

Indicates a described entity has no ID (because it is a proposed schema change and not an existing entity in the schema).

ooFloatType global type

The kind of floating-point data type for a proposed attribute of a proposed class.

Constants

ooFLOAT32

32-bit (single-precision) floating-point number.

ooFLOAT64

64-bit (double-precision) floating-point number.

Discussion

Although ooFloatType is used like a non-class type, it is implemented as a class (a derived class of ooNumberType) whose constructors convert the indicated constants to instances of ooFloatType.

ooIntegerType

global type

The kind of integer data type for a proposed attribute of a proposed class.

Constants

`ooCHAR`
8-bit character.

`ooINT8`
8-bit signed integer.

`ooINT16`
16-bit signed integer.

`ooINT32`
32-bit signed integer.

`ooINT64`
64-bit signed integer.

`ooUINT8`
8-bit unsigned integer.

`ooUINT16`
16-bit unsigned integer.

`ooUINT32`
32-bit unsigned integer.

`ooUINT64`
64-bit unsigned integer.

Discussion

Although `ooIntegerType` is used like a non-class type, it is implemented as a class (a derived class of `ooNumberType`) whose constructors convert the indicated constants to instances of `ooIntegerType`.

ooNumberType

global type

The kind of integer data type for a proposed attribute of a proposed class.

Constants

`ooCHAR`
8-bit character.

`ooFLOAT32`
32-bit (single-precision) floating-point number.

`ooFLOAT64`
64-bit (double-precision) floating-point number.

`ooINT8`
8-bit signed integer.

`ooINT16`
16-bit signed integer.

`ooINT32`
32-bit signed integer.

`ooINT64`
64-bit signed integer.

`ooPTR`
32-bit pointer.

`ooUINT8`
8-bit unsigned integer.

`ooUINT16`
16-bit unsigned integer.

`ooUINT32`
32-bit unsigned integer.

`ooUINT64`
64-bit unsigned integer.

Discussion

Although `ooNumberType` is used like a non-class type, it is implemented as a class whose constructors convert the indicated constants to instances of `ooNumberType`.

ooPTR_t

global type

The kind of integer data type for a proposed attribute of a proposed class.

Constants

`ooPTR`
32-bit pointer.

Discussion

Although `ooPTR_t` is used like a non-class type, it is implemented as a class (a derived class of `ooNumberType`) whose constructors convert the indicated constant to an instance of `ooPTR_t`.

ooUINT64_t

global type

The kind of integer data type for a proposed attribute of a proposed class.

Constants

`ooUINT64`

64-bit unsigned integer.

Discussion

Although `ooUINT64_t` is used like a non-class type, it is implemented as a class (a derived class of `ooNumberType`) whose constructors convert the indicated constant to an instance of `ooUINT64_t`.

attribute_plus_inherited_iterator Class

Inheritance: **`attribute_plus_inherited_iterator`**

The class `attribute_plus_inherited_iterator` represents iterators for attributes of a class. An instance of this class is called an *inherited-attribute iterator*.

See:

- "Reference Summary" on page 166 for an overview of member functions
- "Reference Index" on page 166 for a list of member functions

About Inherited-Attribute Iterators

An inherited-attribute iterator steps through all attributes of a particular class, including relationships and embedded-class attributes corresponding to base classes. It finds all attributes of the class, whether they are defined in that class or inherited. That collection of attributes is called the iterator's *iteration set*; during iteration, the inherited-attribute iterator keeps track of its position within its iteration set. The element at the current position is called the iterator's *current element*. The inherited-attribute iterator allows you to step through the iteration set, obtaining a descriptor for the current element at each step.

Chapter 6, "Working With Iterators," contains additional information about iterators.

Obtaining an Inherited-Attribute Iterator

You should not instantiate this class directly. Instead, you call the `attributes_plus_inherited_begin` member function of a class descriptor to get an inherited-attribute iterator for all attributes of the described class. You can test for that iterator's termination condition by comparing it with the inherited-attribute iterator returned by the same class descriptor's `attributes_plus_inherited_begin` member function.

Including Attributes of Internal Base Classes

By default, an inherited-attribute iterator treats the Objectivity/C++ application classes `ooObj`, `ooContObj`, `ooDBObj`, and `ooFDObj` as if they were root base classes, inheriting from no other classes. Any ancestor classes of those application classes are considered internal; as a consequence, the iteration set does *not* include attributes of internal attribute classes.

If desired, you can override this default behavior, allowing access to attributes of ancestor classes at all levels. To do so, you call the `d_Class::enable_root_descent` static member function.

Reference Summary

Assigning	`operator=`
Getting the Current Element	`operator*`
Advancing the Current Position	`operator++`
Comparing	`operator==` `operator!=`

Reference Index

`attribute_plus_inherited_iterator`	Reserved for internal use.
`operator++`	Increment operator; advances this inherited-attribute iterator's current position.
`operator*`	Dereference operator; gets this inherited-attribute iterator's current element.
`operator=`	Assignment operator; sets this inherited-attribute iterator to be a copy of the specified inherited-attribute iterator.

operator==	Equality operator; tests whether this inherited-attribute iterator is the same as the specified inherited-attribute iterator.
operator!=	Inequality operator; tests whether this inherited-attribute iterator is different from the specified inherited-attribute iterator.

Constructors

attribute_plus_inherited_iterator

Reserved for internal use.

```
attribute_plus_inherited_iterator (
     const attribute_plus_inherited_iterator &itrR);
```

Discussion You should not copy an inherited-attribute iterator; the behavior of a copied iterator is undefined.

Operators

operator++

Increment operator; advances this inherited-attribute iterator's current position.

```
1.   attribute_plus_inherited_iterator  &operator++();
2.   attribute_plus_inherited_iterator operator++(int n);
```

Parameters *n*

This parameter is not used in calling this operator; its presence in the function declaration specifies a postfix operator.

Returns (Variant 1) This inherited-attribute iterator, advanced to the next attribute. (Variant 2) A new inherited-attribute iterator, set to this iterator before its position is advanced.

Discussion Variant 1 is the prefix increment operator, which advances this inherited-attribute iterator and then returns it.

Variant 2 is the postfix increment operator, which returns a new inherited-attribute iterator set to this iterator, and then advances this iterator.

If the current position is already after the last attribute in the iteration set, neither variant advances this iterator.

operator*

Dereference operator; gets this inherited-attribute iterator's current element.

```
const d_Attribute  &operator*() const;
```

Returns An attribute descriptor for the current element.

Discussion You should ensure that iteration has not terminated before calling this member function. The return value is undefined if the current position is after the last attribute in the iteration set.

operator=

Assignment operator; sets this inherited-attribute iterator to be a copy of the specified inherited-attribute iterator.

```
attribute_plus_inherited_iterator  &operator=(
     const attribute_plus_inherited_iterator &itrR);
```

Parameters *itrR*

The inherited-attribute iterator specifying the new value for this inherited-attribute iterator.

Returns This inherited-attribute iterator after it has been set to a copy of *itrR*.

operator==

Equality operator; tests whether this inherited-attribute iterator is the same as the specified inherited-attribute iterator.

```
int operator==(
     const attribute_plus_inherited_iterator &other) const;
```

Parameters *other*

The inherited-attribute iterator with which to compare this inherited-attribute iterator.

Returns Nonzero if the two inherited-attribute iterators are equal and zero if they are different.

Discussion Two inherited-attribute iterators are equal if they iterate over the same iteration set and they have the same current position.

See also operator!=

operator!=

Inequality operator; tests whether this inherited-attribute iterator is different from the specified inherited-attribute iterator.

```
int operator!=(
     const attribute_plus_inherited_iterator &other) const;
```

Parameters *other*

The inherited-attribute iterator with which to compare this inherited-attribute iterator.

Returns Nonzero if the two inherited-attribute iterators are different and zero if they are equal.

Discussion Two inherited-attribute iterators are different if they iterate over different iteration sets or if they are at different positions in the same iteration set.

See also operator==

Attribute_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Attribute_Type**

The abstract class `Attribute_Type` represents descriptors for attribute types in the schema of the federated database. An instance of any concrete derived class is called an *attribute-type descriptor*; it provides information about a particular attribute type, called its *described type*.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

base_class_plus_inherited_iterator Class

Inheritance: **base_class_plus_inherited_iterator**

The class base_class_plus_inherited_iterator represents iterators for ancestor classes of a class. An instance of this class is called a *base-class iterator.*

See:

- "Reference Summary" on page 174 for an overview of member functions
- "Reference Index" on page 174 for a list of member functions

About Base_Class Iterators

A base-class iterator steps through embedded-class attributes corresponding to all ancestor classes of a particular class. That collection of attributes is called the iterator's *iteration set*; during iteration, the base-class iterator keeps track of its position within its iteration set. The element at the current position is called the iterator's *current element*. The base-class iterator allows you to step through the iteration set, obtaining a descriptor for the current element at each step.

Chapter 6, "Working With Iterators," contains additional information about iterators.

Obtaining a Base_Class Iterator

You should not instantiate this class directly. Instead, you call the base_classes_plus_inherited_begin member function of a class descriptor to get a base-class iterator for all ancestor classes of the described class. You can test for that iterator's termination condition by comparing it with the base-class iterator returned by the same class descriptor's base_classes_plus_inherited_end member function.

Including Internal Base Classes

By default, a base-class iterator treats the Objectivity/C++ application classes `ooObj`, `ooContObj`, `ooDBObj`, and `ooFDObj` as if they were root base classes, inheriting from no other classes. Any ancestor classes of those application classes are considered internal; as a consequence, the iteration set does *not* include the internal attribute classes.

If desired, you can override this default behavior, allowing access to ancestor classes at all levels. To do so, you call the `d_Class::enable_root_descent` static member function.

Reference Summary

Assigning	`operator=`
Getting the Current Element	`operator*`
Advancing the Current Position	`operator++`
Comparing	`operator==` `operator!=`

Reference Index

`base_class_plus_inherited_iterator`	Reserved for internal use.
`operator++`	Increment operator; advances this base-class iterator's current position.
`operator*`	Dereference operator; gets this base-class iterator's current element.
`operator=`	Assignment operator; sets this base-class iterator to be a copy of the specified base-class iterator.

operator==	Equality operator; tests whether this base-class iterator is the same as the specified base-class iterator.
operator!=	Inequality operator; tests whether this base-class iterator is different from the specified base-class iterator.

Constructors

base_class_plus_inherited_iterator

Reserved for internal use.

```
base_class_plus_inherited_iterator (
     const base_class_plus_inherited_iterator &itrR);
```

Discussion — You should not copy a base-class iterator; the behavior of a copied iterator is undefined.

Operators

operator++

Increment operator; advances this base-class iterator's current position.

```
1.   base_class_plus_inherited_iterator  &operator++();
2.   base_class_plus_inherited_iterator operator++(int n);
```

Parameters — *n*

This parameter is not used in calling this operator; its presence in the function declaration specifies a postfix operator.

Returns — (Variant 1) This base-class iterator, advanced to the next attribute.
(Variant 2) A new base-class iterator, set to this iterator before its position is advanced.

Discussion — Variant 1 is the prefix increment operator, which advances this base-class iterator and then returns it.

Variant 2 is the postfix increment operator, which returns a new base-class iterator set to this iterator, and then advances this iterator.

If the current position is already after the last attribute in the iteration set, neither variant advances this iterator.

operator*

Dereference operator; gets this base-class iterator's current element.

```
const d_Attribute  &operator*() const;
```

Returns An attribute descriptor for the current element.

Discussion The base classes of a class are described as if they were embedded-class attributes. As a consequence, this member function returns an *attribute descriptor*, not a class descriptor. To obtain a class descriptor for the base class itself, call the `class_type_of` member function of the returned attribute descriptor.

You should ensure that iteration has not terminated before calling this member function. The return value is undefined if the current position is after the last attribute in the iteration set.

operator=

Assignment operator; sets this base-class iterator to be a copy of the specified base-class iterator.

```
base_class_plus_inherited_iterator  &operator=(
     const base_class_plus_inherited_iterator &itrR);
```

Parameters *itrR*

The base-class iterator specifying the new value for this base-class iterator.

Returns This base-class iterator after it has been set to a copy of *itrR*.

operator==

Equality operator; tests whether this base-class iterator is the same as the specified base-class iterator.

```
int operator==(
     const base_class_plus_inherited_iterator &other) const;
```

Parameters *other*

The base-class iterator with which to compare this base-class iterator.

Returns Nonzero if the two base-class iterators are equal and zero if they are different.

Discussion Two base-class iterators are equal if they iterate over the same iteration set and they have the same current position.

See also operator!=

operator!=

Inequality operator; tests whether this base-class iterator is different from the specified base-class iterator.

```
int operator!=(
     const base_class_plus_inherited_iterator &other) const;
```

Parameters *other*

The base-class iterator with which to compare this base-class iterator.

Returns Nonzero if the two base-class iterators are different and zero if they are equal.

Discussion Two base-class iterators are different if they iterate over different iteration sets or if they are at different positions in the same iteration set.

See also operator==

Basic_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Attribute_Type ->Basic_Type**

The class `Basic_Type` represents descriptors for basic numeric types. This document uses the term *numeric type* to include any fundamental character, integer, floating-point, or pointer type. An instance of `Basic_Type` is called a *numeric-type descriptor*; it provides information about a particular numeric type, called its *described type*.

You should never instantiate this class directly. Instead, you can obtain a numeric-type descriptor either from the module descriptor for the top-level module or from an attribute descriptor for a numeric attribute. Typically, you obtain an instance by calling the inherited `type_of` member function of an attribute descriptor.

Member Functions

base_type

Gets the numeric type described by this numeric-type descriptor.

```
ooBaseType base_type() const;
```

Returns A code identifying the described numeric type; one of:

- `ooCHAR` indicates an 8-bit character.
- `ooINT8` indicates an 8-bit signed integer.
- `ooINT16` indicates a 16-bit signed integer.
- `ooINT32` indicates a 32-bit signed integer.
- `ooINT64` indicates a 64-bit signed integer.
- `ooUINT8` indicates an 8-bit unsigned integer.

- ooUINT16 indicates a 16-bit unsigned integer.
- ooUINT32 indicates a 32-bit unsigned integer.
- ooUINT64 indicates a 64-bit unsigned integer.
- ooFLOAT32 indicates a 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates a 64-bit (double-precision) floating-point number.
- ooPTR indicates a 32-bit pointer.

is_basic_type

Overrides the inherited member function. Indicates that the described type is a basic numeric type.

```
virtual ooBoolean is_basic_type() const;
```

Returns ooc True.

Bidirectional_Relationship_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Relationship_Type ->Bidirectional_Relationship_Type**

The class `Bidirectional_Relationship_Type` represents descriptors for bidirectional relationship types. An instance of this class provides information about a particular bidirectional relationship type, called its *described type*.

You should never instantiate this class directly. Instead, you can obtain an instance of this class either from the module descriptor for the top-level module or from a relationship descriptor for a bidirectional relationship. Typically, you obtain an instance by calling the inherited `type_of` member function of a relationship descriptor.

Member Functions

is_bidirectional_relationship_type

Overrides the inherited member function. Indicates that the described type is a bidirectional relationship type.

```
virtual ooBoolean is_bidirectional_relationship_type() const;
```

Returns `oocTrue`.

Class_Object Class

Inheritance: **`Persistent_Data_Object->Class_Object`**

The class `Class_Object` is a self-describing data type for persistent objects. An instance of this class is called a *class object*; it provides access to persistent data contained within some persistent object, called its *associated persistent object*.

See:

- "Reference Summary" on page 184 for an overview of member functions
- "Reference Index" on page 185 for a list of member functions

About Class Objects

Each class object provides access to values of the properties defined in one particular class, called its *described class*. A class object uses a class descriptor for its described class to guide its access to the associated data.

You can construct a class object for an existing persistent object using either a handle or a reference to that persistent object. You can create a class object for a new object to be added to the database by calling the static member function `Class_Object::new_persistent_object`. You can create a class object for a new container to be added to the database by calling the static member function `Class_Object::new_persistent_container_object`.

From one class object, you can obtain class objects above and below it in the hierarchy of class objects for the associated persistent object:

- You can call the `contained_in` member function to get the parent class object.
- You can call the `get_class_obj` member function to get the child class object corresponding to a particular immediate parent class or embedded class of the described class.

From a class object, you can access the data for properties of the associated persistent object.

Chapter 3, “Examining Persistent Data,” contains additional information about class objects.

Reference Summary

Constructing and Creating Class Objects	Class_Object new_persistent_object new_persistent_container_object
Copying Class Objects	Class_Object operator=
Getting Information About the Class Object	contained_in resolve_attribute position_in_class type_of
Getting Properties of the Persistent Object	get get_ooref get_string get_class_obj get_varray get_relationship
Setting Properties of the Persistent Object	set set_ooref
Getting the Persistent Object	operator_const_ooHandle(ooObj) operator_const_ooRef(ooObj) operator_ooHandle(ooObj) operator_ooRef(ooObj) object_handle
Static Utilities	new_persistent_object new_persistent_container_object

Reference Index

operator const ooRef(ooObj)	Converts this class object to a constant object reference.
operator ooHandle(ooObj)	Converts this class object to an object handle.
operator ooRef(ooObj)	Converts this class object to an object reference.
position_in_class	Gets the class position of the specified attribute within this class object's described class.
resolve_attribute	Looks up an attribute defined by this class object's described class.
set	Sets the specified numeric attribute of the persistent object.
set_ooref	Sets the specified object-reference attribute of the persistent object.
type_of	Gets this class object's described class.

Constructors

Class_Object

Constructs a class object.

```
1.  Class_Object();
2.  Class_Object(
      const Class_Object &otherCOR);
3.  Class_Object(
      ooHandle(ooObj) &objH,
      const d_Class &classR);
4.  Class_Object(
      ooHandle(ooObj) &objH,
      ooTypeNumber tnum,
      const d_Module &modR);
5.  Class_Object(
      const ooRef(ooObj) &objR);
6.  Class_Object(
      ooHandle(ooObj) &objH);
```

Parameters

otherCOR

The class object to be copied.

objH

An object handle to the persistent object for the new class object.

classR

A class descriptor for the class of the specified persistent object.

tnum

The type number for the class of the specified persistent object.

modR

A module descriptor for the module containing the specified type number.

objR

An object reference to the persistent object for the new class object.

Discussion

The first variant is the default constructor, which creates a class object with no associated class descriptor or persistent object. You can set a newly created class object using `operator=`.

The second variant is the copy constructor, which creates a new class object with the same class descriptor and persistent object as the specified class object. Both copies access the same persistent object. Any change made with one class object will be seen by the other class object.

The third and fourth variants create a class object for the specified persistent object. Both variants open a handle for the persistent object, if necessary. These variants throw an `InvalidShape` exception if the specified object is not an instance of the specified class. They throw an `InvalidHandle` exception if the specified handle is not valid.

The fifth variant creates a class object for the referenced persistent object. It creates and opens a handle from the specified object reference. It sets the class descriptor for the new class object by looking up the type number of the specified object in the top-level module.

The sixth variant creates a class object for the specified persistent object; it opens a handle for the persistent object, if necessary. It sets the class descriptor for the new class object by looking up the type number of specified object in the top-level module.

Operators

operator=

Assignment operator; sets this class object to a copy of the specified class object.

```
Class_Object &operator=(const Class_Object &otherCOR);
```

Parameters *otherCOR*

The class object to be copied.

Returns This class object after it has been updated to be a copy of *otherCOR*.

Discussion Both copies access the same persistent object. Any change made with one class object will be seen by the other class object.

operator const ooHandle(ooObj)

Converts this class object to a constant object handle.

```
operator const ooHandle(ooObj)();
```

Returns A handle to this class object's associated persistent object, or a null object handle if this is the null class object.

Discussion This operator is valid only for the root class object in the hierarchy of class objects for the associated persistent object.

This operator throws exceptions:

- `NotHandleClassObject` if this class object describes a parent class or an embedded class
- `InvalidHandle` if this is a null class object

See also `operator const ooRef(ooObj)`
`operator ooHandle(ooObj)`
`operator ooRef(ooObj)`
`object handle`

operator const ooRef(ooObj)

Converts this class object to a constant object reference.

```
operator const ooRef(ooObj)() const;
```

Returns A constant object reference to this class object's associated persistent object, or a null constant object reference if this is the null class object.

Discussion This operator is valid only for the root class object in the hierarchy of class objects for the associated persistent object.

This operator throws exceptions:

- NotHandleClassObject if this class object describes a base class or an embedded class
- InvalidHandle if this is a null class object

See also
operator const ooHandle(ooObj)
operator const ooHandle(ooObj)
operator ooRef(ooObj)
object handle

operator ooHandle(ooObj)

Converts this class object to an object handle.

```
operator ooHandle(ooObj)();
```

Returns A handle to this class object's associated persistent object, or a null object handle if this is the null class object.

Discussion This operator is valid only for the root class object in the hierarchy of class objects for the associated persistent object.

This operator throws exceptions:

- NotHandleClassObject if this class object describes a parent class or an embedded class
- InvalidHandle if this is a null class object

See also
operator const ooHandle(ooObj)
operator const ooRef(ooObj)
operator ooRef(ooObj)
object handle

operator ooRef(ooObj)

Converts this class object to an object reference.

```
operator ooRef(ooObj)();
```

Returns An object reference to this class object's associated persistent object.

Discussion This operator is valid only for the root class object in the hierarchy of class objects for the associated persistent object.

This operator throws exceptions:

- NotHandleClassObject if this class object describes a parent class or an embedded class
- InvalidHandle if this is a null class object

See also
operator const ooHandle(ooObj)
operator const ooHandle(ooObj)
operator const ooRef(ooObj)
object_handle

Member Functions

contained_in

Gets this class object's containing class object.

```
Class_Object &contained_in() const;
```

Returns The class object for the persistent object whose data this class object accesses.

Discussion If this class object corresponds to an embedded or base class within the data of a persistent object, the returned class object is the class object for that persistent object.

If this is the class object for a persistent object, this member function returns this class object itself.

If this class object is null, this member function returns this null class object.

get

Gets the data for the specified numeric attribute of the persistent object.

```
1.  Numeric_Value get(
      const Class_Position &classPosR,
      size_t fixedArrayIndex = 0) const;
2.  Numeric_Value get(
      size_t attributePos) const;
3.  Numeric_Value get(
      size_t attributePos,
      size_t fixedArrayIndex) const;
```

Parameters

classPosR

The class position of the desired attribute within this class object's described class.

fixedArrayIndex

The index of the desired value in the fixed-size array in the specified attribute, or 0 if the attribute contains a single numeric value.

attributePos

The attribute position of the desired attribute in this class object's described class.

Returns

The numeric value at the specified index in the specified attribute of the associated persistent object.

Discussion

Because the first variant identifies the attribute by class position, it can access attributes defined in or inherited by this class object's described class. In contrast, the second and third variants, which use attribute position, can access only those attributes defined in the class object's described class.

All variants throw an `AttributeTypeError` exception if the specified attribute is not a numeric attribute.

The first and third variants throw an `ArrayBoundsError` exception if *fixedArrayIndex* exceeds the upper bound for the array in the specified attribute.

See also

`set`

get_class_obj

Gets the data for the specified base class, embedded-class attribute, or object-reference attribute of the persistent object.

```
1.  Class_Object get_class_obj(
      const Class_Position &classPosR,
      size_t fixedArrayIndex = 0) const;
2.  Class_Object get_class_obj(
      size_t attributePos) const;
3.  Class_Object get_class_obj(
      size_t attributePos,
      size_t fixedArrayIndex) const;
```

Parameters

classPosR

The class position of the desired attribute within this class object's described class.

fixedArrayIndex

The index of the desired value in the fixed-size array in the specified attribute, or 0 if the attribute contains a single embedded object. An array index should not be used if the attribute is a base class.

attributePos

The attribute position of the desired attribute in this class object's described class.

Returns

For a base class or an embedded-class attribute, a class object for the embedded object element at the specified index; for an object-reference attribute, a class object for the persistent object referenced by the value at the specified index.

Discussion

Because the first variant identifies the attribute by class position, it can access attributes defined in or inherited by this class object's described class. In contrast, the second and third variants, which use attribute position, can access only those attributes defined in the class object's described class.

To obtain a value from an object-reference attribute without opening a handle for the referenced object, call `get_ooref` instead of this member function.

All variants throw an `AttributeTypeError` exception if the specified attribute is not a base class, an embedded-class attribute, or an object-reference attribute.

The first and third variants throw an `ArrayBoundsError` exception if *fixedArrayIndex* exceeds the upper bound for the array in the specified attribute.

get_ooref

Gets the object reference in the specified object-reference attribute of the persistent object.

```
1.   ooRef(ooObj) get_ooref(
       const Class_Position &classPosR,
       size_t fixedArrayIndex = 0) const;
2.   ooRef(ooObj) get_ooref(
       size_t attributePos) const;
3.   ooRef(ooObj) get_ooref(
       size_t attributePos,
       size_t fixedArrayIndex) const;
```

Parameters

classPosR

The class position of the desired attribute within this class object's described class.

fixedArrayIndex

The index of the desired value in the fixed-size array in the specified attribute, or 0 if the attribute contains a single object reference.

attributePos

The attribute position of the desired attribute in this class object's described class.

Returns

A class object for the persistent object referenced by the value at the specified index of the specified attribute.

Discussion

Because the first variant identifies the attribute by class position, it can access attributes defined in or inherited by this class object's described class. In contrast, the second and third variants, which use attribute position, can access only those attributes defined in the class object's described class.

To open a handle for the reference object and obtain a class object for it, call `get_class_obj` instead of this member function.

All variants throw an `AttributeTypeError` exception if the specified attribute is not an object-reference attribute.

The first and third variants throw an `ArrayBoundsError` exception if *fixedArrayIndex* exceeds the upper bound for the array in the specified attribute.

See also

`set_ooref`

get_relationship

Gets the data for the specified relationship of the persistent object.

```
1.   Relationship_Object get_relationship(
       const Class_Position &classPosR) const;
2.   Relationship_Object get_relationship(
       size_t attributePos) const;
```

Parameters

classPosR

The class position of the desired relationship within this class object's described class.

attributePos

The attribute position of the desired relationship in this class object's described class.

Discussion

Because the first variant identifies the relationship by class position, it can access any relationship defined in or inherited by this class object's described class. In contrast, the second variant, which uses attribute position, can access only those relationships defined in the class object's described class.

All variants throw an AttributeTypeError exception if the specified property is not a relationship.

get_string

Gets the data for the specified string attribute of the persistent object.

```
1.  String_Value get_string(
      const Class_Position &classPosR,
      size_t fixedArrayIndex = 0) const;
2.  String_Value get_string(
      size_t attributePos) const;
3.  String_Value get_string(
      size_t attributePos,
      size_t fixedArrayIndex) const;
```

Parameters

classPosR

The class position of the desired attribute within this class object's described class.

fixedArrayIndex

The index of the desired value in the fixed-size array in the specified attribute, or 0 if the attribute contains a single string.

attributePos

The attribute position of the desired attribute in this class object's described class.

Discussion

Because the first variant identifies the attribute by class position, it can access attributes defined in or inherited by this class object's described class. In contrast, the second and third variants, which use attribute position, can access only those attributes defined in the class object's described class.

All variants throw an AttributeTypeError exception if the specified attribute is not a string attribute.

The first and third variants throw an ArrayBoundsError exception if *fixedArrayIndex* exceeds the upper bound for the array in the specified attribute.

get_varray

Gets the data for the specified embedded-class or VArray attribute of the persistent object.

```
1.  VArray_Object get_varray(
      const Class_Position &classPosR,
      size_t fixedArrayIndex = 0) const;
2.  VArray_Object get_varray(
      size_t attributePos) const;
3.  VArray_Object get_varray(
      size_t attributePos,
      size_t fixedArrayIndex) const;
```

Parameters

classPosR

The class position of the desired attribute within this class object's described class.

fixedArrayIndex

The index of the desired value in the fixed-size array in the specified attribute, or 0 if the attribute contains a single VArray.

attributePos

The attribute position of the desired attribute in this class object's described class.

Discussion

Because the first variant identifies the attribute by class position, it can access attributes defined in or inherited by this class object's described class. In contrast, the second and third variants, which use attribute position, can access only those attributes defined in the class object's described class.

All variants throw an `AttributeTypeError` exception if the specified attribute is not a VArray attribute.

The first and third variants throw an `ArrayBoundsError` exception if *fixedArrayIndex* exceeds the upper bound for the array in the specified attribute.

is_class_object

Overrides the inherited member function; indicates that this is a class object.

```
virtual ooBoolean is_class_object() const;
```

Returns

`oocTrue`.

new_persistent_container_object

Creates a class object whose associated persistent object is a newly created instance of the specified container class.

```
static Class_Object new_persistent_container_object(
     const d_Class &classR,
     const ooHandle(ooObj) &nearHandle
     uint32 hash,
     uint32 initPages,
     uint32 percentGrow);
```

Parameters

classR

Class descriptor for the described class of the new class object. The described class must be a container class (derived from `ooContObj`).

nearHandle

Clustering directive for the new persistent object.

hash

Indicates whether the new container should be a hashed container.

- If *hash* = 0, then the container is non-hashed. Neither the container nor any basic objects it contains may be used as a scope when naming an object.
- If *hash* > 0, then the container is hashed. The value of *hash* is used as a clustering factor (number of sequentially indexed objects to place into a page) when you add scope names to this container. If you do not care about the clustering factor, you should set the value of *hash* to 1.

initPages

The initial number of pages allocated for the container.

percentGrow

The percentage by which the container should grow when needed to accommodate more basic objects.

Returns

The newly created class object.

Discussion

This static member function creates a persistent object of the specified container class, but does not call constructors to initialize the data members of the new object. The caller is responsible for initializing the members.

This member function throws exceptions:

- NonPersistentClassObject if the specified class is not persistence capable.
- WrongCategoryOfNewObject if the specified class is not a container class.

See also new_persistent_object

new_persistent_object

Creates a class object whose associated persistent object is a newly created instance of the specified class.

```
static Class_Object new_persistent_object(
     const d_Class &classR,
     const ooHandle(ooObj) &nearHandle);
```

Parameters

classR

Class descriptor for the described class of the new class object. The described class must be persistence capable (derived from ooObj) and may not be a container class (derived from ooContObj)

nearHandle

Clustering directive for the new persistent object.

Returns The newly created class object.

Discussion This static member function creates a persistent object of the specified class, but does not call constructors to initialize the data members of the new object. The caller is responsible for initializing the members.

This member function throws exceptions:

- NonPersistentClassObject if the specified class is not persistence capable.
- WrongCategoryOfNewObject if the specified class is a container class.

See also new_persistent_container_object

object_handle

Gets a handle to this class object's associated persistent object.

```
ooHandle(ooObj) &object_handle() const;
```

Returns A handle to this class object's associated persistent object, or a null object handle if this is the null class object.

Discussion This member function is valid only for the root class object in the hierarchy of class objects for the associated persistent object.

This operator throws exceptions:

- NotHandleClassObject if this class object describes a base class or an embedded class
- InvalidHandle if this is a null class object

See also
operator const ooHandle(ooObj)
operator const ooRef(ooObj)
operator ooRef(ooObj)

position_in_class

Gets the class position of the specified attribute within this class object's described class.

```
const Class_Position position_in_class(
     const char *memName) const;
```

Parameters

memName

The name of the attribute whose position is desired. This string can be a qualified name (such as `foo::base::x`) to disambiguate attributes of the same name inherited from different base classes. You should specify a qualified name only if necessary, because it takes more time to look up a qualified name than an unqualified one.

Returns

A class position that gives the layout position of the specified attribute within the described class.

resolve_attribute

Looks up an attribute defined by this class object's described class.

```
const d_Attribute &resolve_attribute(
     const char *memName) const;
```

Parameters

memName

The name of the attribute to be looked up.

Returns

The attribute descriptor for the described class's attribute with the specified name, or the null descriptor if *memName* is not the name of an immediate base class of the described class or the name of an attribute or a relationship defined by the described class.

set

Sets the specified numeric attribute of the persistent object.

```
1.  ooStatus set(
      const Class_Position &classPosR,
      const Numeric_Value val);
2.  ooStatus set(
      const Class_Position &classPosR,
      size_t fixedArrayIndex,
      const Numeric_Value val);
3.  ooStatus set(
      size_t attributePos,
      const Numeric_Value val);
4.  ooStatus set(
      size_t attributePos,
      size_t fixedArrayIndex,
      const Numeric_Value val);
```

Parameters

classPosR

The class position of the desired attribute within this class object's described class.

val

The new numeric value for the specified index in the specified attribute of the associated persistent object.

fixedArrayIndex

The index of the desired value in the fixed-size array in the specified attribute, or 0 if the attribute contains a single numeric value.

attributePos

The attribute position of the desired attribute in this class object's described class.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

Because the first and second variants identify the attribute by class position, they can be used to set any numeric attribute defined in or inherited by this class object's described class. In contrast, the third and fourth variants, which use attribute position, can set only those attributes defined in the class object's described class.

All variants throw an `AttributeTypeError` exception if the specified attribute is not a numeric attribute.

The second and fourth variants throw an ArrayBoundsError exception if *fixedArrayIndex* exceeds the upper bound for the array in the specified attribute.

See also get

set_ooref

Sets the specified object-reference attribute of the persistent object.

```
1.  ooStatus set_ooref(
      const Class_Position &classPosR,
      const ooRef(ooObj) objR);
2.  ooStatus set_ooref(
      const Class_Position &classPosR,
      size_t fixedArrayIndex,
      const ooRef(ooObj) objR);
3.  ooStatus set_ooref(
      size_t attributePos,
      const ooRef(ooObj) objR);
4.  ooStatus set_ooref(
      size_t attributePos,
      size_t fixedArrayIndex,
      const ooRef(ooObj) objR);
```

Parameters

classPosR

The class position of the desired attribute within this class object's described class.

objR

The new object reference for the specified index in the specified attribute of the associated persistent object.

fixedArrayIndex

The index of the desired value in the fixed-size array in the specified attribute, or 0 if the attribute contains a single object reference.

attributePos

The attribute position of the desired attribute in this class object's described class.

Returns

ocSuccess if successful; otherwise ocError.

Discussion

Because the first and second variants identify the attribute by class position, they can be used to set any object-reference attribute defined in or inherited by this class object's described class. In contrast, the third and fourth variants, which use attribute position, can set only those attributes defined in this class object's described class.

All variants throw an `AttributeTypeError` exception if the specified attribute is not an object-reference attribute.

The second and fourth variants throw an `ArrayBoundsError` exception if *`fixedArrayIndex`* exceeds the upper bound for the array in the specified attribute.

See also `get_ooref`

type_of

Gets this class object's described class.

```
const d_Class &type_of() const;
```

Returns

A class descriptor for this class object's described class.

Class_Position Class

Inheritance: **`Class_Position`**

`Class_Position` represents the positions of attributes within classes. Each instance of this class, called a *class position*, gives the position of a particular attribute in the physical layout for objects of a particular class.

See:

- "Reference Summary" on page 204 for an overview of member functions
- "Reference Index" on page 204 for a list of member functions

About Class Positions

A class position indicates nesting of data inherited from base classes; see "Class Position" on page 35. You do not instantiate this class directly, instead, you obtain the class position for a particular attribute of a particular class by calling the `position_in_class` member function of a class descriptor, a class object, or a proposed class; the parameter to the function specifies the attribute of interest.

The class position for an immediate base class or an attribute that is not inherited contains a single number. Such a class position can be converted to and from an unsigned integer. You can call the `is_convertible_to_uint` member function to see whether such conversion is possible.

If a class position is one or more levels of inheritance deep, you may not convert it to an integer. An attempt to do so throws a `ConvertDeepPositionToInt` exception.

Reference Summary

Copying Class Positions	operator=
Testing	operator== is_convertible_to_uint
Converting to Attribute Position	operator size_t

Reference Index

Operators

operator=

Assignment operator; sets this class position to a copy of the specified class position.

```
Class_Position  &operator=(const Class_Position &otherCPR);
```

Parameters

otherCPR

The class position to be copied.

Returns

This class position after it has been updated to be a copy of *otherCPR*.

operator==

Equality operator; tests whether this class position is equal to the specified class position.

```
int operator==(const Class_Position &pR) const;
```

Parameters *pR*

The class position to be compared with this class position.

Returns Nonzero if the two class positions indicate the same path of attribute positions; otherwise, zero.

Discussion A class position does not keep track of the class in which it specifies a position. As a consequence, this member function would return true if the two class positions indicated the same path even if the class positions obtained were from descriptors for two different classes.

operator size_t

Conversion operator that returns an integral position.

```
operator size_t() const;
```

Returns This class position converted to the integral position, or -1 if this is the null class position.

Discussion If this is the class position for an immediate base class or an attribute that is not inherited, the returned integer is the attribute position of the base class or attribute within its defining class. If this class position is one or more levels of inheritance deep, an attempt to convert it to an integer throws a `ConvertDeepPositionToInt` exception.

You can call the `is_convertible_to_uint` member function to see whether this class position can be converted to an integer.

Member Functions

is_convertible_to_uint

Tests whether this class position can be converted to an unsigned integer.

```
ooBoolean is_convertible_to_uint() const;
```

Returns `oocTrue` if this class position gives the position of an immediate base class or an attribute defined in the class; `oocFalse` if it gives the position of an ancestor class or an inherited attribute.

Collection_Object Class

Inheritance: **`Persistent_Data_Object->Collection_Object`**

The class `Collection_Object` is the abstract base class for classes that serve as self-describing data types for collections.

Currently Objectivity / DB supports only one kind of collection attribute, namely VArray attributes, which contain variable-size arrays of elements of the same type. The subclass `VArray_Object` represents this kind of persistent data.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

d_Attribute Class

Inheritance: **d_Meta_Object->d_Property->d_Attribute**

The class `d_Class` represents descriptors for attributes of classes in the schema of the federated database. An instance of `d_Attribute` is called an *attribute descriptor*.

See:

- "Reference Summary" on page 210 for an overview of member functions
- "Reference Index" on page 211 for a list of member functions

About Attribute Descriptors

An attribute descriptor provides information about a particular attribute, called its *described attribute*. The described attribute is defined by some class. It stores a particular piece of data for a persistent instance of that class and its derived classes. The described attribute holds either a single value of some type, or a fixed-size array of values of the same type.

Obtaining an Attribute Descriptor

You should never instantiate this class directly. Instead, you can obtain an attribute descriptor by calling a member function on either a class descriptor for the class that defines the attribute or a class descriptor for a class that inherits the attribute.

Member Function	Call on Descriptor for	Description
attribute_at_position	Defining class	Looks up attribute by attribute position (page 34) in defining class.
	Inheriting class	Looks up attribute by class position (page 35) in described class.
attributes_plus_inherited_begin	Defining class Inheriting class	Gets iterator for all attributes of described class
attribute_with_id	Defining class	Looks up attribute by Objectivity/DB attribute ID (page 35)
defines_attribute_begin	Defining class	Gets iterator for all attributes defined in class
resolve_attribute	Defining class	Looks up attribute by name

Reference Summary

Getting Information About the Described Attribute	dimension array_size element_size id position default_value class_type_of
Testing the Described Attribute	operator== has_default_value is_base_class
Getting Related Descriptors	class_type_of

Reference Index

Operators

operator==

Equality operator; tests whether this attribute descriptor is equal to the specified attribute descriptor.

```
int operator==(const d_Attribute &otherAR) const;
```

Parameters

otherAR

The attribute descriptor to be compared with this attribute descriptor.

Returns

Nonzero if this attribute descriptor and *otherAR* both describe the same attribute of the same class; otherwise, zero.

Member Functions

array_size

Gets the array size for the described attribute.

```
size_t array_size() const;
```

Returns The number of elements in the fixed-size array of values for the described attribute (or one if the attribute contains a single value instead of an array).

class_type_of

Gets the class for the described embedded-class attribute.

```
const d_Class  &class_type_of() const;
```

Returns A class descriptor for the class that is the type of the described attribute.

Discussion You should call this member function only if you know that the described attribute is an embedded-class attribute (or a base class). If the attribute's type is not a class, this member function throws an `AttributeTypeError` exception.

default_value

Gets the default value for the described numeric attribute.

```
Numeric_Value default_value() const;
```

Returns A numeric value containing the default value for the described attribute, or an invalid numeric value if the described attribute type is not a basic numeric type or if the attribute has no default value.

Discussion You can call `has_default_value` to test whether the described attribute has a default value.

dimension

Gets the physical layout size of the described attribute.

```
unsigned long dimension() const;
```

Returns The number of bytes required to store the attribute's data on the platform where the current application is running.

Discussion If the described attribute is a fixed-size array, the returned number is the total layout size for all elements of the array.

element_size

Gets the physical layout size for a single value of the attribute's type.

```
size_t element_size() const;
```

Returns The number of bytes required to store a single value of the attribute's type on the platform where the current application is running.

Discussion If the described attribute is not a fixed-size array, this member function returns the same number as `dimension`.

has_default_value

Tests whether the described attribute has a default value.

```
ooBoolean has_default_value() const;
```

Returns `oocTrue` if the described attribute has a default value; otherwise, `oocFalse`.

Discussion Only attributes of basic numeric types can have default values.

id

Gets the attribute ID of the described attribute.

```
virtual uint32 id() const;
```

Returns The attribute ID that permanently identifies the described attribute within its class.

is_base_class

Tests whether the described attribute is a base class.

```
ooBoolean is_base_class() const;
```

Returns `oocTrue` if the described attribute is a base class; otherwise, `oocFalse`.

Discussion A base class is described like an embedded-class attribute; this member function allows you to test whether an embedded-class attribute is a base class.

is_read_only

Tests whether the described attribute is read only.

```
d_Boolean is_read_only() const;
```

Returns `oocFalse`.

Discussion Objectivity/C++ does not support persistent read-only attributes, so this test fails for all attributes.

is_static

Tests whether the described attribute is static.

```
d_Boolean is_static() const;
```

Returns `oocFalse`.

Discussion Objectivity/C++ does not support persistent static attributes, so this test fails for all attributes.

position

Gets the position of the described attribute within its defining class.

```
size_t position() const;
```

Returns The attribute position of the described attribute within the physical layout of its defining class.

d_Class Class

Inheritance: **d_Meta_Object->d_Type->d_Class, d_Scope->d_Class**

The class `d_Class` represents descriptors for classes in the schema of the federated database. An instance of `d_Class` is called a *class descriptor*.

See:

- "Reference Summary" on page 217 for an overview of member functions
- "Reference Index" on page 218 for a list of member functions

About Class Descriptors

A class descriptor provides information about a class in the schema, called its *described class*. The described class is actually a particular shape of a particular version of a particular class.

Because `d_Class` inherits from `d_Scope`, a class descriptor can look up or iterate through properties defined in the scope of the described class, obtaining descriptors for the properties in the class's scope.

Obtaining a Class Descriptor

You should never instantiate this class directly. Instead, you can obtain a class descriptor either from the module descriptor for the top-level module or from the module descriptor for the module in which the class is defined:

- Call the `resolve_class` member function of the module descriptor to look up the class by name. Alternatively, you can call the module descriptor's `resolve_type` or `resolve` member function and cast the result to a class descriptor.
- Call the `defines_types_begin` member function of the module descriptor to get an iterator for all types in the module's scope. You can obtain type descriptors from the iterator. Call the `is_class` member function of a type descriptor to see whether it describes a class; if so, you can safely cast it to a class descriptor.

Specifying a Version

By default, when you call the `resolve_class`, `resolve_type`, or `resolve` member function of a module descriptor, you get a class descriptor for the *most recent version* of the specified class. However, an optional parameter to these member functions allows you to specify the desired version number. When you use that parameter, you obtain a class descriptor for the specified version of the specified class.

Specifying a Shape

When you call the `resolve_class`, `resolve_type`, or `resolve` member function of a module descriptor, you obtain a class descriptor for the *most recent shape* of the specified class and version. If the class has evolved, you can look up the previous shape by calling the class descriptor's `previous_shape` member function. If you have a descriptor for an older shape, you can call its `next_shape` member function to get the descriptor for the next shape.

Reference Summary

Getting Information About the Described Class	id type_number number_of_attributes shape_number next_shape previous_shape version_number latest_version position_in_class get_static_ref
Testing the Described Class	persistent_capable is_internal is_string_type has_extent has_base_class has_virtual_table is_deleted
Testing for the Null Descriptor	operator_size_t
Setting Information About the Described Class	set_static_ref
Getting Descriptors from the Schema	resolve_attribute resolve_relationship resolve attribute_at_position attribute_with_id attributes_plus_inherited_begin defines_attribute_begin defines_relationship_begin defines_begin base_class_list_begin base_classes_plus_inherited_begin sub_class_list_begin
Static Utilities	enable_root_descent disable_root_descent root_descent_is_enabled

Reference Index

defines_end	Gets an iterator representing the termination condition for iteration through the properties defined in the described class.
defines_relationship_begin	Gets an iterator for the relationships defined in the described class.
defines_relationship_end	Gets an iterator representing the termination condition for iteration through the relationships defined in the described class.
disable_root_descent	Disables access to ancestors of Objectivity/C++ persistent-object and storage-object classes when iterating through inherited attributes or base classes.
enable_root_descent	Enables access to ancestors of Objectivity/C++ persistent-object and storage-object classes when iterating through inherited attributes or base classes.
has_base_class	Tests whether the described class is derived from the specified base class.
has_extent	Tests whether the described class has a nonzero physical size.
has_virtual_table	Tests whether the described class has a virtual table.
id	Gets the unique ID that identifies the described class.
is_class	Overrides the inherited member function; indicates that this is a class descriptor.
is_deleted	Tests whether the described class is deleted.
is_internal	Tests whether the described class is an internal Objectivity/DB class.
is_string_type	Tests whether the described class is a string class.

latest_version	Gets the latest version of the described class.
next_shape	Gets the next shape of the described class.
number_of_attributes	Gets the number of attributes in the described class.
operator size_t	Conversion operator that tests whether this class descriptor is null.
persistent_capable	Tests whether the described class is persistence-capable.
position_in_class	Gets the class position of the specified attribute within the described class.
previous_shape	Gets the previous shape of the described class.
resolve	Looks up a property defined by the described class.
resolve_attribute	Looks up an attribute defined by the described class.
resolve_relationship	Looks up a relationship defined by the described class.
root_descent_is_enabled	Tests whether iteration through inherited attributes or base classes includes access to ancestors of Objectivity/C++ persistent-object and storage-object classes.
shape_number	Gets the shape number for the described class.
sub_class_list_begin	Gets an iterator for the inheritance connections between the described class and its child classes.

sub_class_list_end	Gets an iterator representing the termination condition for iteration through the inheritance connections between the described class and its child classes.
type_number	Gets the unique type number for the described class and version.
version_number	Gets the version number for the described class.

Operators

operator size_t

Conversion operator that tests whether this class descriptor is null.

```
virtual operator size_t() const;
```

Returns Zero if this class descriptor is null; otherwise, the nonzero type number of the described class.

Discussion Any member function that looks up a class descriptor returns a class descriptor object; unsuccessful searches return a null class descriptor. This operator allows you to use a class descriptor as an integer expression to test whether that class descriptor is valid (not null).

Member Functions

attribute_at_position

Gets the attribute at the specified position in the described class.

```
1.   const d_Attribute  &attribute_at_position(
       const Class_Position &posR) const;
2.   const d_Attribute  &attribute_at_position(
       size_t pos) const;
```

Parameters *posR*

The class position of the desired attribute in the described class.

pos

The attribute position of the desired attribute in the described class.

Returns An attribute descriptor for the attribute at the specified position.

Discussion The first variant can get a descriptor for any attribute defined in or inherited by the described class. The second variant can get a descriptor for any attribute defined in the described class.

This member function throws an `AttributeOutOfRange` exception if *pos* or an attribute position within *posR* is not a valid position in the containing class. It throws an `AttributeTypeError` exception if *posR* indicates a position within an attribute that is not an embedded base class.

attribute_with_id

Gets the attribute with the specified ID in the described class.

```
const d_Attribute  &attribute_with_id(uint32 IDtoMatch) const;
```

Parameters *IDtoMatch*

The attribute ID of the desired attribute.

Returns An attribute descriptor for the attribute with the specified ID, or the null descriptor if the described class has no such attribute.

attributes_plus_inherited_begin

Gets an iterator for the attributes of the described class.

```
attribute_plus_inherited_iterator
     attributes_plus_inherited_begin() const;
```

Returns An inherited-attribute iterator that finds all base classes, attributes, and relationships in the described class, whether they are defined in that class or inherited.

See also `attributes_plus_inherited_end`
`defines_attribute_begin`

attributes_plus_inherited_end

Gets an iterator representing the termination condition for iteration through the attributes of the described class.

```
attribute_plus_inherited_iterator
     attributes_plus_inherited_end() const;
```

Returns An inherited-attribute iterator that is positioned after the last attribute of the described class.

Discussion You can compare the iterator returned by `attributes_plus_inherited_begin` with the one returned by this member function to test whether iteration has finished.

base_class_list_begin

Gets an iterator for the inheritance connections between the described class and its parent classes.

```
inheritance_iterator base_class_list_begin() const;
```

Returns An inheritance iterator that finds all inheritance connections in which the described class is the child (derived) class.

Discussion The returned iterator gets an inheritance descriptor for each inheritance connection from an immediate base class to the described class. To get the base class from one of these inheritance descriptors, call its `derives_from` member function.

See also `base_class_list_end`
`base_classes_plus_inherited_begin`
`sub_class_list_begin`

base_class_list_end

Gets an iterator representing the termination condition for iteration through the inheritance connections between the described class and its parent classes.

```
inheritance_iterator base_class_list_end() const;
```

Returns An inheritance iterator that is positioned after the last inheritance connection between the described class and a base class.

Discussion You can compare the iterator returned by `base_class_list_begin` with the one returned by this member function to test whether iteration has finished.

base_classes_plus_inherited_begin

Gets an iterator for the ancestor classes of the described class.

```
base_class_plus_inherited_iterator
     base_classes_plus_inherited_begin() const;
```

Returns A base-class iterator that finds all ancestor classes of the described class.

See also base_class_list_begin
base_classes_plus_inherited_end

base_classes_plus_inherited_end

Gets an iterator representing the termination condition for iteration through the ancestor classes of the described class.

```
base_class_plus_inherited_iterator
     base_classes_plus_inherited_end() const;
```

Returns A base-class iterator that is positioned after the last ancestor class of the described class.

Discussion You can compare the iterator returned by base_classes_plus_inherited_begin with the one returned by this member function to test whether iteration has finished.

defines_attribute_begin

Gets an iterator for the attributes defined in the described class.

```
attribute_iterator defines_attribute_begin() const;
```

Returns An attribute iterator that finds all immediate base classes, attributes, and relationships in the described class.

Discussion The returned iterator finds the same descriptors as the iterator returned by defines_begin, but it gets them typed as attribute descriptors instead of generic descriptors.

See also attributes_plus_inherited_begin
defines_attribute_end

defines_attribute_end

Gets an iterator representing the termination condition for iteration through the attributes defined in the described class.

```
attribute_iterator defines_attribute_end() const;
```

Returns An attribute iterator that is positioned after the last attribute defined in the described class.

Discussion You can compare the iterator returned by `defines_attribute_begin` with the one returned by this member function to test whether iteration has finished.

defines_begin

Gets an iterator for the properties defined in the described class.

```
virtual meta_object_iterator defines_begin() const;
```

Returns A descriptor iterator that finds all immediate base classes, attributes, and relationships defined in the described class.

Discussion The returned iterator gets generic descriptors for each property defined in the described class. An alternative to calling this member function is to call more specific functions that find properties of some particular kind.

See also `defines_end`
`defines_attribute_begin`
`defines_relationship_begin`

defines_end

Gets an iterator representing the termination condition for iteration through the properties defined in the described class.

```
virtual meta_object_iterator defines_end() const;
```

Returns A descriptor iterator that is positioned after the last property defined in the described class.

Discussion You can compare the iterator returned by `defines_begin` with the one returned by this member function to test whether iteration has finished.

defines_relationship_begin

Gets an iterator for the relationships defined in the described class.

```
relationship_iterator defines_relationship_begin() const;
```

Returns A relationship iterator that finds all relationships defined in the described class.

See also `defines_relationship_end`

defines_relationship_end

Gets an iterator representing the termination condition for iteration through the relationships defined in the described class.

```
relationship_iterator defines_relationship_end() const;
```

Returns A relationship iterator that is positioned after the last relationship defined in the described class.

Discussion You can compare the iterator returned by `defines_relationship_begin` with the one returned by this member function to test whether iteration has finished.

disable_root_descent

Disables access to ancestors of Objectivity/C++ persistent-object and storage-object classes when iterating through inherited attributes or base classes.

```
static void disable_root_descent();
```

Discussion By default, the iterators returned by `attributes_plus_inherited_begin` and `base_classes_plus_inherited_begin` treat the Objectivity/C++ persistent-object base class `ooObj` and the storage-object classes `ooContObj`, `ooDBObj`, and `ooFDObj` as if they were root base classes, inheriting from no other classes. You can call `enable_root_descent` to override this behavior, allowing access to ancestor classes at all levels; after doing so, you can call this member function to disable access once again.

See also `root_descent_is_enabled`

enable_root_descent

Enables access to ancestors of Objectivity/C++ persistent-object and storage-object classes when iterating through inherited attributes or base classes.

```
static void enable_root_descent();
```

Discussion By default, the iterators returned by `attributes_plus_inherited_begin` and `base_classes_plus_inherited_begin` treat the Objectivity/C++ persistent-object base class `ooObj` and the storage-object classes `ooContObj`, `ooDBObj`, and `ooFDObj` as if they were root base classes, inheriting from no other classes. You can call this member function to override this behavior, allowing access to ancestor classes at all levels.

See also `disable_root_descent`
`root_descent_is_enabled`

get_static_ref

Gets the persistent object containing the persistent static properties of the described class.

```
ooRef(ooObj) get_static_ref() const;
```

Returns This method must be called within a transaction.

See also set_static_ref

has_base_class

Tests whether the described class is derived from the specified base class.

```
ooBoolean has_base_class(const char *nameToMatch) const;
```

Parameters *nameToMatch*

The name of the base class of interest.

Returns `oocTrue` if the described class is derived from *nameToMatch*; otherwise, `oocFalse`.

has_extent

Tests whether the described class has a nonzero physical size.

```
d_Boolean has_extent() const;
```

Returns `oocTrue` if the described class has a nonzero physical size; otherwise, `oocFalse`.

has_virtual_table

Tests whether the described class has a virtual table.

```
ooBoolean has_virtual_table() const;
```

Returns `oocTrue` if the described class has a virtual table; otherwise, `oocFalse`.

id

Gets the unique ID that identifies the described class.

```
virtual uint32 id() const;
```

Returns The ID for the described class.

Discussion The ID for a class (or any type) is the same as its type number.

is_class

Overrides the inherited member function; indicates that this is a class descriptor.

```
virtual ooBoolean is_class() const;
```

Returns `oocTrue`.

is_deleted

Tests whether the described class is deleted.

```
ooBoolean is_deleted() const;
```

Returns `oocTrue` if the described class has been deleted from the schema; otherwise, `oocFalse`.

Discussion Active Schema cannot delete classes. However, another application could have deleted the class before Active Schema started. In that case, the class description remains in the schema but is marked as deleted.

is_internal

Tests whether the described class is an internal Objectivity / DB class.

```
ooBoolean is_internal() const;
```

Returns `oocTrue` if the described class is an internal Objectivity / DB class; `oocFalse` if the described class is an application-defined class (including an optimized string class).

See also Appendix A, “Internal Classes”

is_string_type

Tests whether the described class is a string class.

```
virtual ooBoolean is_string_type() const;
```

Returns `oocTrue` if the described class is a string class; otherwise, `oocFalse`.

The string classes are:

- The ASCII string class `ooVString`
- The optimized string classes `ooString(`*`N`*`)`
- The Unicode string class `ooUTF8String`
- The Smalltalk string class `ooSTString`

latest_version

Gets the latest version of the described class.

```
const d_Class  &latest_version() const;
```

Returns A class descriptor for the latest version of the described class.

next_shape

Gets the next shape of the described class.

```
const d_Class  &next_shape() const;
```

Returns A class descriptor for the next shape of the described class, or the null descriptor if the described shape is the last shape of the described class and version.

See also previous_shape

number_of_attributes

Gets the number of attributes in the described class.

```
size_t number_of_attributes() const;
```

Returns The number of immediate base classes, attributes, and relationships in the defined class.

Discussion The returned number does not include inherited attributes.

persistent_capable

Tests whether the described class is persistence-capable.

```
d_Boolean persistent_capable() const;
```

Returns oocTrue if the described class is persistence-capable; otherwise, oocFalse.

position_in_class

Gets the class position of the specified attribute within the described class.

```
1.  const Class_Position position_in_class(
      const char *memName) const;
2.  const Class_Position position_in_class(
      const d_Attribute &attR) const;
```

Parameters *memName*

The name of the attribute whose position is desired. This string can be a qualified name (such as `foo::base::x`) to disambiguate attributes of the same name inherited from different base classes. You should specify a qualified name only if necessary because it takes more time to look up a qualified name than an unqualified one.

attR

An attribute descriptor for the attribute whose position is desired.

Returns A class position that gives the layout position of the specified attribute within the described class.

See also `attribute_at_position`

previous_shape

Gets the previous shape of the described class.

```
const d_Class  &previous_shape() const;
```

Returns A class descriptor for the previous shape of the described class, or the null descriptor if the schema does not contain a description of the previous shape of the described class and version.

See also `next_shape`

resolve

Looks up a property defined by the described class.

```
virtual const d_Meta_Object  &resolve(
     const char *n,
     int32 version = oocLatestVersion) const;
```

Parameters *n*

The name of the property to be looked up.

version

The desired version of the entity named *n*. This parameter should be omitted because it is only relevant for looking up classes, not properties.

Returns The descriptor for the described class's property with the specified name, or the null descriptor if *n* is not the name of an immediate base class of the described class or the name of an attribute or a relationship defined by the described class.

Discussion The returned generic descriptor can be cast to the appropriate descriptor class (`d_Attribute` or `d_Relationship`). An alternative to calling this member function is to call more specific functions that look up properties of a particular kind.

See also `resolve_attribute`
`resolve_relationship`

resolve_attribute

Looks up an attribute defined by the described class.

```
const d_Attribute  &resolve_attribute(
     const char *nameToMatch) const;
```

Parameters *nameToMatch*

The name of the attribute to be looked up.

Returns The attribute descriptor for the described class's attribute with the specified name, or the null descriptor if *nameToMatch* is not the name of an immediate base class of the described class or the name of an attribute or a relationship defined by the described class.

resolve_relationship

Looks up a relationship defined by the described class.

```
const d_Relationship  &resolve_relationship(
     const char *nameToMatch) const;
```

Parameters *nameToMatch*

The name of the relationship (association) to be looked up.

Returns The relationship descriptor for the described class's relationship with the specified name, or the null descriptor if the described class does not define a relationship named *nameToMatch*.

root_descent_is_enabled

Tests whether iteration through inherited attributes or base classes includes access to ancestors of Objectivity / C++ persistent-object and storage-object classes.

```
static ooBoolean root_descent_is_enabled();
```

Returns `oocTrue` if root descent is enabled; otherwise, `oocFalse`.

Discussion By default, access is disabled; the iterators returned by `attributes_plus_inherited_begin` and `base_classes_plus_inherited_begin` treat the Objectivity/C++ persistent-object base class `ooObj` and the storage-object classes `ooContObj`, `ooDBObj`, and `ooFDObj` as if they were root base classes, inheriting from no other classes.

See also `disable_root_descent`
`enable_root_descent`

set_static_ref

Stores the specified persistent object with the described class to represent its persistent static properties.

```
ooStatus set_static_ref(const ooRef(ooObj) &objR);
```

Parameters *objR*

Object reference to the persistent object containing the persistent static properties for the described class.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This method must be called within a transaction; its actions become visible to other Active Schema applications when the transaction is committed.

Note: The member functions that get class descriptors return `const` objects. You must cast such a class descriptor to a non-`const` descriptor and call this member function on the non-`const` descriptor.

See also `get_static_ref`

shape_number

Gets the shape number for the described class.

```
ooTypeNumber shape_number() const;
```

Returns The shape number for this class descriptor.

Discussion If the described shape is the original shape of the described class and version, the shape number is identical to the type number.

See also `next_shape`
`previous_shape`

sub_class_list_begin

Gets an iterator for the inheritance connections between the described class and its child classes.

```
inheritance_iterator sub_class_list_begin() const;
```

Returns An inheritance iterator that finds all inheritance connections in which the described class is the parent (base class).

Discussion The returned iterator gets an inheritance descriptor for each inheritance connection from the described class to an immediate derived class. To get the derived class from one of these inheritance descriptors, call its `inherits_to` member function.

See also `base_class_list_begin`
`sub_class_list_end`

sub_class_list_end

Gets an iterator representing the termination condition for iteration through the inheritance connections between the described class and its child classes.

```
inheritance_iterator sub_class_list_end() const;
```

Returns An inheritance iterator that is positioned after the last inheritance connection between the described class and a subclass.

Discussion You can compare the iterator returned by `sub_class_list_begin` with the one returned by this member function to test whether iteration has finished.

type_number

Gets the unique type number for the described class and version.

```
virtual ooTypeNumber type_number() const;
```

Returns The unique type number for the described class and version.

See also `shape_number`

version_number

Gets the version number for the described class.

```
int32 version_number() const;
```

Returns The version number for the described class.

d_Collection_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Attribute_Type ->d_Collection_Type**

The abstract class `d_Collection_Type` represents descriptors for collection types for attributes. An instance of any concrete derived class is called a *collection-type descriptor*; it provides information about a particular collection type, called its *described collection type*.

Concrete derived classes represent descriptors for:

- Numeric variable-size array types
- Embedded-class variable-size array types
- Object-reference variable-size array types

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

Member Functions

element_type

Gets the type of elements in the described collection type.

```
virtual const d_Type  &element_type() const;
```

Returns A type descriptor for the elements in the described collection type.

kind

Gets the ODMG collection kind of the described collection type.

```
virtual d_Kind kind() const = 0;
```

Returns The ODMG collection kind of the described collection type.

Discussion The only ODMG collection kind that Objectivity / DB supports is variable-size arrays of elements of the same type.

d_Inheritance Class

Inheritance: **d_Inheritance**

The class d_Inheritance represents descriptors for inheritance connections between classes. An instance of d_Inheritance is called an *inheritance descriptor*.

See:

- "Reference Summary" on page 238 for an overview of member functions
- "Reference Index" on page 238 for a list of member functions

About Inheritance Descriptors

An inheritance descriptor provides information about a particular connection in an inheritance graph between one particular parent or base class and one child or derived class.

Obtaining an Inheritance Descriptor

You should never instantiate this class directly. Instead, you can obtain an inheritance descriptor by iterating through the parent classes or the child classes of a class.

- Call the base_class_list_begin member function of a class descriptor to get an iterator that finds all inheritance connections in which the described class is the child class.
- Call the sub_class_list_begin member function of a class descriptor to get an iterator that finds all inheritance connections in which the described class is the parent class.

Getting Information About the Inheritance Connection

Member functions return information about the described inheritance connection:

- The access kind (public, private, or protected)
- The parent class from which the inheriting class is derived
- The child class that inherits from the parent class
- The layout position of the parent class data within the storage of a persistent instance of the child class

Reference Summary

Getting the Parent Class	`derives_from`
Getting the Child Class	`inherits_to`

Getting Information About the Inheritance Collection	`access_kind`
	`is_virtual`
	`position`

Testing for the Null Descriptor	`operator size_t`

Reference Index

`access_kind`	Gets the access kind of the described inheritance connection.
`derives_from`	Gets the parent class in the described inheritance connection.
`inherits_to`	Gets the child class in the described inheritance connection.
`is_virtual`	Tests whether the described inheritance connection is virtual.
`position`	Gets the layout position of data for the parent class within the storage of a persistent instance of the child class.
`operator size_t`	Conversion operator that tests whether this inheritance descriptor is null.

Operators

operator size_t

Conversion operator that tests whether this inheritance descriptor is null.

```
virtual operator size_t() const;
```

Returns Zero if this inheritance descriptor is null; otherwise, nonzero.

Discussion Any member function that looks up an inheritance descriptor returns an inheritance descriptor object; unsuccessful searches return a null inheritance descriptor. This operator allows you to use an inheritance descriptor as an integer expression to test whether that inheritance descriptor is valid (not null).

Member Functions

access_kind

Gets the access kind of the described inheritance connection.

```
d_Access_Kind access_kind() const;
```

Returns The access kind (or visibility) of the parent base class as specified in the declaration of the child or derived class; one of the following:

- `d_PUBLIC` indicates a public base class.
- `d_PROTECTED` indicates a protected base class.
- `d_PRIVATE` indicates a private base class.

derives_from

Gets the parent class in the described inheritance connection.

```
const d_Class  &derives_from() const;
```

Returns A class descriptor for the parent or base class (from which the child class derives).

See also `inherits_to`

inherits_to

Gets the child class in the described inheritance connection.

```
const d_Class  &inherits_to() const;
```

Returns A class descriptor for the child or derived class (which inherits from the parent class).

See also `derives_from`

is_virtual

Tests whether the described inheritance connection is virtual.

```
d_Boolean is_virtual() const;
```

Returns `oocFalse`.

Discussion Objectivity/C++ does not support virtual inheritance, so this test fails for all inheritance connections.

position

Gets the layout position of data for the parent class within the storage of a persistent instance of the child class.

```
size_t position() const;
```

Returns The zero-based layout position of data for the parent class within the storage of a persistent instance of the child class.

d_Meta_Object Class

Inheritance: **d_Meta_Object**

The class d_Meta_Object is the abstract base class for descriptor classes that describe named entities (modules, classes, attributes, and so on) in the federated database schema, and for descriptor classes that describe proposed additions or modifications to the schema. Each concrete class derived from this class describes one particular kind of schema entity or proposal. An instance of any concrete class derived from d_Meta_Object is called a *descriptor*.

See:

- "Reference Summary" on page 242 for an overview of member functions
- "Reference Index" on page 242 for a list of member functions

About Descriptors

As the name of this class implies, a descriptor is a meta-object—that is, an object that provides information about a "real" object. Each descriptor provides information about a particular named entity in a federated database called its *described entity*. In addition to the persistent information from the federated database schema, a descriptor can have a transient comment. The comment is associated with a descriptor only during the interaction in which the descriptor was obtained; it is not saved persistently with the described entity in the federated database schema.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

Chapter 2, "Examining the Schema," contains additional information about descriptors.

Reference Summary

Getting Information About the Described Entity	name defined_in id comment
Setting Information About the Described Entity	set_comment
Testing the Described Entity	is_class is_module is_type
Testing for the Null Descriptor	operator size_t

Reference Index

Operators

operator=

Overrides the assignment operator (=). Disallows assigning a new value to a descriptor.

```
d_Meta_Object  &operator=(const d_Meta_Object &val);
```

Discussion This member function always throws an exception. If this descriptor is non-null, it throws an AssignToMO exception; if this is the null descriptor, it throws an AssignToNullMO exception.

operator size_t

Conversion operator that tests whether this descriptor is null.

```
virtual operator size_t() const;
```

Returns Zero if this descriptor is null; otherwise, nonzero.

Discussion Any member function that looks up a descriptor returns a descriptor object; unsuccessful searches return a null descriptor. This operator allows you to use a descriptor as an integer expression to test whether that descriptor is valid (not null).

When this member function is called for a valid descriptor of an entity in the schema, it returns the unique ID of that entity.

Member Functions

comment

Gets the transient comment associated with this descriptor.

```
const char  *comment() const;
```

Returns The transient comment associated with this descriptor.

See also set_comment

defined_in

Gets the scope in which the described entity is defined.

```
const d_Scope  &defined_in() const;
```

Returns The scope for the entity described by this descriptor, or null if the described entity is not defined in the scope of any entity in the federated database schema.

Discussion The following table lists the various kinds of entity that a descriptor can describe and identifies the scope of each kind.

Described Entity	Scope
Top-level module	Top-level module
Named module	Top-level module
Non-class type	Top-level module
Class	Module in which the described class is defined
Property	Class in which the described property is defined
Proposed class	None
Proposed property	None
Proposed base class	None

id

Gets the unique ID that identifies the described entity within its scope.

```
virtual uint32 id() const;
```

Returns The ID for the described entity, or `oocNoID` if the described entity does not have an ID. If this is the null descriptor, this member function returns 0.

Discussion Existing entities in the schema have IDs; proposed changes to the schema do not.

Active Schema uses an ID to identify an existing entity uniquely within its scope. If the described entity is a class or non-class type, its ID is the same as its type number. If the described entity is a property, its ID is the same as its Objectivity attribute ID.

is_class

Tests whether the described entity is a class.

```
virtual ooBoolean is_class() const;
```

Returns `oocTrue` if this is a class descriptor; otherwise `oocFalse`.

is_module

Tests whether the described entity is a module.

```
virtual ooBoolean is_module() const;
```

Returns `oocTrue` if this is a module descriptor; otherwise `oocFalse`.

is_type

Tests whether the described entity is a type.

```
virtual ooBoolean is_type() const;
```

Returns `oocTrue` if the described entity is a type (class or non-class type); otherwise `oocFalse`.

name

Gets the name of the described entity.

```
const char *name() const;
```

Returns The name of the entity about which this descriptor provides information.

set_comment

Sets the transient comment for this descriptor, replacing any existing comment.

```
void set_comment(const char *com);
```

Parameters *com*

The new comment to be associated with this descriptor.

Discussion The comment is associated with this descriptor only during the interaction in which this descriptor was obtained; it is not saved persistently with the described entity in the federated database schema.

See also `comment`

type_number

Gets the unique type number for the described entity.

```
virtual ooTypeNumber type_number();
```

Returns The unique type number for the described entity.

d_Module Class

Inheritance: **d_Meta_Object->d_Module, d_Scope->d_Module**

The class `d_Module` represents descriptors for modules in the schema of the federated database. An instance of `d_Module` is called a *module descriptor*.

See:

- "Reference Summary" on page 248 for an overview of member functions
- "Reference Index" on page 249 for a list of member functions

About Module Descriptors

A module descriptor is both a descriptor and a scope. As a descriptor, it provides information about a particular module, called its *described module*. As a scope, it allows you to obtain descriptors for the entities defined in the scope of the described module, either by looking up a particular entity or by iterating through all entities in the module's scope.

You should never instantiate this class directly; instead:

- Call the `d_Module::top_level` static member function to obtain a descriptor for the top-level module.
- Call the `resolve_module` member function of the top-level module's descriptor to look up another module by name.
- Call the `named_modules_begin` member function of top-level module's descriptor to get an iterator for all named modules.

Reference Summary

Getting Information About the Described Module	schema_number id next_type_number next_assoc_number
Setting Information About the Described Module	set_next_type_number set_next_assoc_number
Testing the Described Module	is_top_level
Getting Descriptors from the Schema	top_level resolve_class resolve_type resolve_module resolve defines_types_begin named_modules_begin defines_begin
Schema Evolution	propose_new_class propose_evolved_class propose_versioned_class add_module activate_proposals activate_remote_schema_changes delete_proposal clear_proposals
Getting Proposed Classes	resolve_proposed_class proposed_classes_begin
Static Utilities	top_level lock_schema unlock_schema add_module evolution_message_handler set_evolution_message_handler sanitize
Application-Defined Functions Used by this Class	evolution message handler

Reference Index

named_modules_end	Gets an iterator representing the termination condition for iteration through the modules defined in the described module's scope.
next_assoc_number	Gets the next available association number for the described module.
next_type_number	Gets the next available type number for the described module.
propose_evolved_class	Proposes an evolved definition of the specified class in the described module.
propose_new_class	Proposes a new class to be added to the described module.
propose_versioned_class	Proposes a new version of the specified class in the described module.
proposed_classes_begin	Gets an iterator for the proposed classes in this module descriptor's proposal list.
proposed_classes_end	Gets an iterator representing the termination condition for iteration through the proposed classes in this module descriptor's proposal list.
resolve	Looks up an entity in the described module's scope.
resolve_class	Looks up a class in the described module's scope.
resolve_module	Looks up a module in the described module's scope.
resolve_proposed_class	Looks up a proposed class in this module descriptor's proposal list.
resolve_type	Looks up a type in the described module's scope.
sanitize	Updates the federated database schema, restoring any class descriptions that may have become corrupted.
schema_number	Gets the schema number of the described module.
set_evolution_message_handler	Installs the specified evolution message handler.

set_next_assoc_number	Sets the next available association number for the described module.
set_next_type_number	Sets the next available type number for the described module.
top_level	Gets a descriptor for the top-level module in the federated database.
unlock_schema	Unlocks the schema of the federated database.

Member Functions

activate_proposals

Activates all proposed changes in this module descriptor's proposal list.

```
ooStatus activate_proposals(
     ooTrans &trans,
     ooHandle(ooFDObj) &fdH,
     const ooMode modeOption = (ooMode)oocCurrentMrow,
     const int32 waitOption = oocCurrentTransWait,
     const ooIndexMode indexModeOption =
               (ooIndexMode) oocCurrentSensitivity,
     ooBoolean alsoActivateSubmodules = oocTrue);
```

Parameters

trans

The current transaction.

fdH

A handle to the federated database.

modeOption

The concurrent access policy for the restarted transaction; relevant only if this member function commits the transaction; one of the following:

- `oocCurrentMrow` (the default) restarts the transaction with the same concurrent access policy as it had before this member function committed it.
- `oocMROW` enables the *multiple readers, one writer* (MROW) concurrent access policy.
- `oocNoMROW` disables MROW; enables the exclusive concurrent access policy.

waitOption

The lock-waiting behavior for the restarted transaction; relevant only if this member function commits the transaction; one of the following:

- `oocCurrentTransWait` (the default) restarts the transaction with the same lock-waiting behavior that the transaction had before this member function committed it.
- `oocTransNoWait` uses the default lock-waiting option currently in effect for the Objectivity context (see the `ooSetLockWait` global function).
- `oocNoWait` or 0 turns off lock waiting for the restarted transaction.
- `oocWait` causes the restarted transaction to wait indefinitely for locks.
- An integer *n* in the range 1 <= *n* <= 14400 causes the restarted transaction to wait for the specified number of seconds. If *n* = 0, it is treated as `oocNoWait`. If *n* is less than 0 or greater than 14400, it is treated as `oocWait`.

indexModeOption

The sensitivity of index updating; relevant only if this member function commits the transaction. Specifies when indexes are updated relative to when indexed objects are updated; one of the following:

- `oocCurrentSensitivity` (the default) restarts the transaction with the same sensitivity that the transaction had before this member function committed it.
- `oocInsensitive` updates all applicable indexes automatically when the restarted transaction commits.
- `oocSensitive` updates all applicable indexes immediately after an indexed field is created or modified. You should use this value if the restarted transaction will modify indexed fields and then perform predicate scans on the relevant indexes.
- `oocExplicitUpdate` updates all applicable indexes only by explicit calls to the `ooUpdateIndexes` global function.

alsoActivateSubmodules

`oocTrue` to activate proposals of all submodules of the described module; otherwise, `oocFalse`. This parameter is ignored unless the described module is the top-level module.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This member function tries to activate the proposals in this module descriptor's proposal list. If the described module is the top-level module and *alsoActivateSubmodules* is `oocTrue`, it activates the proposals in the proposal list of all module descriptors.

If the current transaction is active, this member function commits the transaction before attempting to modify the federated database schema and restarts it after the schema has been modified. This member function throws a FailedToRestartTransaction exception if it is unable to restart the transaction.

If successful, this member function updates the runtime Objectivity / DB schema table as indicated by this module descriptor's proposal list and clears the proposal list. If the described module is the top-level module and *alsoActivateSubmodules* is oocTrue, it clears the proposal list of all module descriptors. If unsuccessful, it leaves the proposal list(s) unchanged and throws an EvolutionError exception containing an Objectivity diagnostic message.

activate_remote_schema_changes

Activates any schema changes in the described module made by other processes, making them available to the calling process.

```
ooStatus activate_remote_schema_changes(
     ooTrans &trans,
     ooHandle(ooFDObj) &fdH,
     size_t  *numShapes = NULL,
     ooBoolean alsoActivateSubmodules = oocTrue);
```

Parameters

trans

The current transaction.

fdH

Handle to the federated database

numShapes

Pointer to an unsigned integer to be set to the number of new shape descriptions added to the schema of the current process. If this member function fails, the unsigned integer is set to zero.

alsoActivateSubmodules

oocTrue to activate remote changes to all submodules of the described module; otherwise, oocFalse. This parameter is ignored unless the described module is the top-level module.

Returns

oocSuccess if any remote schema changes were activated successfully; otherwise, oocError.

Discussion

If any other process has added classes to the described module or caused evolution of existing classes in the described module, this member function reads the updated schema, making the remote schema changes available to the calling process. If the described module is the top-level module and

alsoActivateSubmodules is oocTrue, this member function makes all schema changes to all modules available to the calling process.

If the current transaction is active, this member function reads the federated database schema and commits the transaction. It then restarts the transaction using the same settings that were in effect when this member function was called.

If the current transaction is not active, this member function starts the transaction before attempting to read the federated database schema and commits it after the schema has been read.

NOTE A process's internal representation of the federated database schema is frozen during MROW transactions. As a consequence, this member function fails if the current transaction is active and using the MROW concurrent access policy.

---

add_module

Adds a new module to the federated database schema.

```
static d_Module  &add_module(
     const char *schemaName,
     uint32 schemaNumber = 0);
```

Parameters

schemaName

The name for the new module.

schemaNumber

The unique schema number for the new module. If this parameter is omitted, the new module is assigned the next available schema number.

This parameter is typically omitted. It may be specified in applications that need to recreate another schema exactly.

Returns A module descriptor for the new module.

Discussion This member function throws a CantAddModule exception if, for any reason, it is unable to add a module to the federated database schema.

clear_proposals

Clears this module descriptor's proposal list.

```
void clear_proposals();
```

See also delete_proposal

defines_begin

Gets an iterator for the entities in the scope of the described module.

```
virtual meta_object_iterator defines_begin() const;
```

Returns A descriptor iterator that finds all entities in the scope of the described module.

Discussion If the described module is the top-level module, the returned iterator finds all modules, classes, and non-class types in the schema. If not, the returned iterator finds all classes defined in the described module.

The returned iterator gets generic descriptors for each entity in the described module's scope. An alternative to calling this member function is to call more specific functions that find entities of some particular kind.

See also `defines_end`
`defines_types_begin`
`named_modules_begin`
`proposed_classes_begin`

defines_end

Gets an iterator representing the termination condition for iteration through the entities defined in the described module's scope.

```
virtual meta_object_iterator defines_end() const;
```

Returns A descriptor iterator that is positioned after the last entity in the described module's scope.

Discussion You can compare the iterator returned by `defines_begin` with the one returned by this member function to test whether iteration has finished.

defines_types_begin

Gets an iterator for the types in the scope of the described module.

```
type_iterator defines_types_begin() const;
```

Returns A type iterator that finds all types in the scope of the described module.

Discussion If the described module is the top-level module, the returned iterator finds all classes and non-class types in the schema. If not, the returned iterator finds all classes defined in the described module.

See also `defines_types_end`

defines_types_end

Gets an iterator representing the termination condition for iteration through the types defined in the described module's scope.

```
type_iterator defines_types_end() const;
```

Returns A type iterator that is positioned after the last type in the described module's scope.

Discussion You can compare the iterator returned by `defines_types_begin` with the one returned by this member function to test whether iteration has finished.

delete_proposal

Deletes a proposed class from the proposal list of the described module.

```
ooStatus delete_proposal(const char *proposalName);
```

Parameters *`proposalName`*

The name of the proposed class to be deleted.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `clear_proposals`

evolution_message_handler

Gets the currently installed evolution message handler.

```
static void (*evolution_message_handler())(const char *);
```

Returns A function pointer to the currently installed evolution message handler.

See also `set_evolution_message_handler`
"Evolution Message Handler" on page 268

id

Gets the unique ID that identifies the described module within its scope.

```
virtual uint32 id() const;
```

Returns The ID for the described module.

is_module

Overrides the inherited member function. Indicates that this is a module descriptor.

```
virtual ooBoolean is_module() const;
```

Returns `oocTrue`.

is_top_level

Tests whether the described module is the top-level module of the federated database.

```
virtual ooBoolean is_top_level() const;
```

Returns `oocTrue` if the described module is the top-level module; otherwise, `oocFalse`.

lock_schema

Locks the schema of the federated database.

```
1.  static ooStatus lock_schema(uint64 key);
2.  static ooStatus lock_schema(uint64 key, uint64 oldKey);
```

Parameters *`key`*

The key with which the schema can be unlocked or accessed in the future.

`oldKey`

The key with which the schema was locked previously.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion The first variant locks the schema. If the schema is being locked for the first time, *`key`* can be any key; if the schema was locked previously, *`key`* must be the key with which the schema was last locked.

The second variant relocks the schema and changes the key; *`oldKey`* must be the key with which the schema was last locked.

See also `top_level`
`unlock_schema`

named_modules_begin

Gets an iterator for the modules defined in the described module's scope.

```
virtual module_iterator named_modules_begin() const;
```

Returns A module iterator that finds all modules defined in the described module.

Discussion If the described module is the top-level module, the returned iterator finds all other modules; otherwise, the returned iterator has an empty iteration set.

See also named_modules_end

named_modules_end

Gets an iterator representing the termination condition for iteration through the modules defined in the described module's scope.

```
virtual module_iterator named_modules_end() const;
```

Returns A module iterator that is positioned after the last module in the described module's scope.

Discussion You can compare the iterator returned by named_modules_begin with the one returned by this member function to test whether iteration has finished.

next_assoc_number

Gets the next available association number for the described module.

```
ooAssocNumber next_assoc_number() const;
```

Returns The next available association number for the described module.

Discussion The result is the association number to be assigned to the next relationship that is added to a class in the described module.

See also set_next_assoc_number

next_type_number

Gets the next available type number for the described module.

```
ooTypeNumber next_type_number() const;
```

Returns The next available type number for the described module.

Discussion The result is the type number to be assigned to the next class that is added to the described module.

See also set_next_type_number

propose_evolved_class

Proposes an evolved definition of the specified class in the described module.

```
Proposed_Class  &propose_evolved_class(
     const char *name,
     ooTypeNumber tnum = 0,
     ooBoolean isRecursiveInternalCall = oocFalse);
```

Parameters *name*

The name of the class to be evolved.

tnum

The shape number for the evolved shape the class. If this parameter is omitted, the new shape is assigned the next available type number.

This parameter is typically omitted. It may be specified in applications that need to recreate another schema exactly.

isRecursiveInternalCall

`oocFalse` if called by an application; `oocTrue` if called internally by Active Schema. You should always omit this parameter (or pass `oocFalse`).

Returns A proposed class that describes the proposed new definition of the class *name*.

Discussion This member function adds the new proposed class to this module descriptor's proposal list. The proposed class is created with proposed properties and base classes that describe the current definition of the specified class. You can call member functions of the returned proposed class to modify this description.

This member function throws an exception:

- UnnamedObjectError if name is null
- NameNotInModule if the described module does not contain a class named name
- ProposeEvolutionOfInternal if name is the name of an internal Objectivity / DB class
- ProposeEvolAndVers if any class versioning has already been proposed in the current transaction

See also propose_new_class
propose_versioned_class

propose_new_class

Proposes a new class to be added to the described module.

```
1.  Proposed_Class  &propose_new_class(
      const char *name,
      ooTypeNumber tnum = 0);
2.  Proposed_Class propose_new_class(
      Proposed_Class *newClass);
```

Parameters

name

The name of the new class.

tnum

The type number for the new class. If this parameter is omitted, the new class is assigned the next available type number.

This parameter is typically omitted. It may be specified in applications that need to recreate another schema exactly.

newClass

A pointer to an existing proposed class to be added to this module descriptor's proposal list.

Returns

A proposed class that describes the proposed new class.

Discussion

This member function adds the new proposed class to this module descriptor's proposal list. The proposed class is created "empty" in that it has no proposed properties or base classes. You can call member functions of the returned proposed class to modify this empty description.

If `name` is invalid, this member function throws an exception:

- `UnnamedObjectError` if `name` is null
- `NameAlreadyInModule` if the described module already contains a class (or type) named `name`
- `NameAlreadyProposedInModule` if the described module already has a proposed class named `name`

See also

`propose_evolved_class`
`propose_versioned_class`

propose_versioned_class

Proposes a new version of the specified class in the described module.

```
Proposed_Class  &propose_versioned_class(
     const char *name,
     ooTypeNumber tnum = 0,
     ooBoolean isRecursiveInternalCall = oocFalse);
```

Parameters

name

The name of the class for which a new version is to be created.

tnum

The shape number for the new version of the class. If this parameter is omitted, the new version is assigned the next available type number.

This parameter is typically omitted. It may be specified in applications that need to recreate another schema exactly.

isRecursiveInternalCall

`oocFalse` if called by an application; `oocTrue` if called internally by Active Schema. You should always omit this parameter (or pass `oocFalse`).

Returns

A proposed class that describes the proposed new version of the class *name*.

Discussion

This member function adds the new proposed class to this module descriptor's proposal list. The proposed class is created with proposed properties and base classes that describe the most recent version of the specified class. You can call member functions of the returned proposed class to modify this description.

This member function throws an exception:

- `UnnamedObjectError` if `name` is null
- `NameNotInModule` if the described module does not contain a class named `name`
- `ProposeEvolAndVers` if any class evolution has already been proposed in the current transaction

See also

`propose_new_class`
`propose_versioned_class`

proposed_classes_begin

Gets an iterator for the proposed classes in this module descriptor's proposal list.

```
proposed_class_iterator proposed_classes_begin() const;
```

Returns A proposed-class iterator that finds all proposed classes in this module descriptor's proposal list.

See also proposed_classes_end

proposed_classes_end

Gets an iterator representing the termination condition for iteration through the proposed classes in this module descriptor's proposal list.

```
proposed_class_iterator proposed_classes_end() const;
```

Returns A proposed-class iterator that is positioned after the last proposed class in this module descriptor's proposal list.

Discussion You can compare the iterator returned by proposed_classes_begin with the one returned by this member function to test whether iteration has finished.

resolve

Looks up an entity in the described module's scope.

```
const d_Meta_Object  &resolve(
     const char *n,
     int32 version = oocLatestVersion) const;
```

Parameters *n*

The name of the entity to be looked up.

version

The desired version of the entity named *n*. This optional parameter can be specified when looking up a class that was created using the Objectivity/C++ class-versioning feature.

Returns The descriptor for the entity in the described module's scope with the specified name and version, or the null descriptor if no such entity exists.

Discussion If the described module is the top-level module, this member function can look up any module, class, or non-class type in the schema. If not, it can look up any class defined in the described module.

The returned generic descriptor can be cast to the appropriate descriptor class (for example d_Module or d_Class). An alternative to calling this member function is to call more specific functions that look up entities of a particular kind.

See also
resolve_class
resolve_module
resolve_proposed_class
resolve_type

resolve_class

Looks up a class in the described module's scope.

```
1.  const d_Class  &resolve_class(
      const char *str,
      int32 version = oocLatestVersion) const;
2.  const d_Class  &resolve_class(
      ooTypeNumber n) const;
```

Parameters

str

The class name to be looked up.

version

The desired version of the class named *str*. This optional parameter can be specified when looking up a class that was created using the Objectivity / C++ class-versioning feature.

n

The type number of the class to be looked up.

Returns

The descriptor for the specified class in the described module's scope, or the null descriptor if no such class exists.

Discussion

If the described module is the top-level module, this member function can look up any class in the schema. If not, it can look up any class defined in the described module.

See also
resolve
resolve_module
resolve_proposed_class
resolve_type

resolve_module

Looks up a module in the described module's scope.

```
const d_Module  &resolve_module(
     const char *schemaName) const;
```

Parameters *schemaName*

The name of the module to be looked up.

Returns The descriptor for the module with the specified name in the described module's scope, or the null descriptor if no such module exists.

Discussion If the described module is the top-level module, this member function can look up any module in the schema. If not, this member function returns the null descriptor (because named modules cannot contain other modules).

See also resolve
resolve_class
resolve_proposed_class
resolve_type

resolve_proposed_class

Looks up a proposed class in this module descriptor's proposal list.

```
Proposed_Class  &resolve_proposed_class(
     const char *strToMatch) const;
```

Parameters *strToMatch*

The name of the proposed class to be looked up.

Returns The proposed class with the specified name in this module descriptor's proposal list, or the null descriptor if no such proposed class exists.

See also resolve
resolve_class
resolve_module
resolve_type

resolve_type

Looks up a type in the described module's scope.

```
1.   const d_Type  &resolve_type(const char *str) const;
2.   const d_Type  &resolve_type(ooTypeNumber tn) const;
```

Parameters *str*

The type name to be looked up.

tn

The type number of the type to be looked up.

Returns The descriptor for the specified type in the described module's scope, or the null descriptor if no such type exists.

Discussion If the described module is the top-level module, this member function can look up any class or non-class type in the schema. If not, it can look up any class defined in the described module.

Typically this member function is used only to look up non-class types in the scope of the top-level module. The `resolve_class` member function is used instead of this member function to look up classes.

See also `resolve`
`resolve_class`
`resolve_module`
`resolve_proposed_class`

sanitize

Updates the federated database schema, restoring any class descriptions that may have become corrupted.

```
static ooStatus sanitize(
     ooTrans &trans,
     ooHandle(ooFDObj) &fdH);
```

Parameters *trans*

The current transaction.

fdH

Handle to the federated database

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function is a last resort for applications that experience inexplicable schema failure, such as a crash or inability to open or close a container in the system database. The system database, which contains the schema, has the identifier 1; the internal Objectivity / DB objects that represent class descriptions in the schema are stored in that database and, thus, have object identifiers of the form `1-`*n*`-`*n*`-`*n*. A problem or failure in a different application can leave these schema objects in a corrupted state in the federated database; when that happens, your application may be unable to open or close the container for the corrupted objects. In that case, you can call this member function, which restores the class descriptions in the schema. This member function will fail if it is unable to obtain an update lock on the schema.

If the current transaction is active, this member function repairs the federated database schema and commits the transaction. It then restarts the transaction

using the same settings that were in effect when this member function was called.

If the current transaction is not active, this member function starts the transaction before attempting to repair the federated database schema and commits it after the schema has been modified.

schema_number

Gets the schema number of the described module.

```
ooTypeNumber schema_number() const;
```

Returns The type number that uniquely identifies this module within the federated database schema.

set_evolution_message_handler

Installs the specified evolution message handler.

```
static void set_evolution_message_handler(
     void (*handler)(const char *));
```

Parameters *handler*

Function pointer to the evolution message handler to be installed. As the member function signature indicates, the handler is a function that takes one parameter of type `const char *` and returns no value.

See also evolution_message_handler
"Evolution Message Handler" on page 268

set_next_assoc_number

Sets the next available association number for the described module.

```
ooStatus set_next_assoc_number(ooAssocNumber n);
```

Parameters *n*

The new next available association number for the described module; may not be less than the current next available association number (which is returned by next_assoc_number).

Returns `oocSuccess` if successful; `oocError` if *n* is lower than the current next association number.

Discussion Most applications will not need to call this member function. It is provided to enable an application to recreate the exact state of another schema.

See also next_assoc_number

set_next_type_number

Sets the next available type number for the described module.

```
ooStatus set_next_type_number(ooTypeNumber n);
```

Parameters *n*

The new next available type number for the described module; may not be less than the current next available type number (which is returned by next_type_number).

Returns oocSuccess on success; oocError if *n* is lower than the current next type number.

Discussion Most applications will not need to call this member function. It is provided to enable an application to recreate the exact state of another schema.

See also next_type_number

top_level

Gets a descriptor for the top-level module in the federated database.

```
static const d_Module  &top_level(uint64 key = 0);
```

Parameters *key*

The key with which the schema was last locked. This parameter can be omitted if the schema is not currently locked.

Returns A module descriptor that provides information about the top-level module.

Discussion If the federated database schema is currently locked, you must specify the appropriate key as the parameter to this member function. An AccessDenied error occurs if you attempt to access a locked schema without the correct key.

Note that this member function returns a const object; if you want to call any non-const member functions (for example, propose_new_class), you must cast the result to the type:

```
d_Module &
```

See also lock_schema

unlock_schema

Unlocks the schema of the federated database.

```
static ooStatus unlock_schema(uint64 key);
```

Parameters *key*

The key with which the schema was locked.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion Once the schema has been unlocked, it can be accessed by any process. The schema may be relocked, but only by a process that supplies *key* as the key with which it was last locked.

See also `lock_schema`

Application-Defined Functions

You can customize the behavior of Active Schema by defining an application-specific function to handle messages that are produced by schema evolution. The form of your evolution message handler is as follows:

Evolution Message Handler

Handles a message produced when schema evolution occurs.

```
static void my_evolution_message_handler(const char *newMsg);
```

Parameters *newMsg*

The message string to be handled.

Discussion The evolution message handler is called for each message that results from schema modifications following a call to `activate_proposals`.

See also `d_Module::set_evolution_message_handler`

d_Property Class

Inheritance: **d_Meta_Object->d_Property**

The abstract class `d_Property` represents descriptors for properties of classes in the schema of the federated database.

See:

- "Reference Summary" on page 270 for an overview of member functions
- "Reference Index" on page 270 for a list of member functions

About Property Descriptors

An instance of any concrete class derived from `d_Property` is called a *property descriptor*; it provides information about a particular property, called its *described property*. Concrete derived classes represent descriptors for two different kinds of properties that a class can have:

- Attributes or component data of the class
- Relationships (or associations) from the class defining the relationship (or source class) to a destination class. The destination class can be any persistence-capable class, including the source class itself.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

Reference Summary

Getting Information About the Described Property	defined_in_class type_of access_kind
Testing the Described Property	is_relationship
Getting Related Descriptors	defined_in_class type_of

Reference Index

Member Functions

access_kind

Gets the access kind of the described property.

```
d_Access_Kind access_kind() const;
```

Returns The access kind (or visibility) of the described property as specified in the declaration of the class in which it is defined; one of the following:

- d_PUBLIC indicates a public property.
- d_PROTECTED indicates a protected property.
- d_PRIVATE indicates a private property.

defined_in_class

Gets the class in which the described property is defined.

```
const d_Class  &defined_in_class() const;
```

Returns A class descriptor for the class in which the described property is defined.

is_relationship

Tests whether the described property is a relationship.

```
virtual ooBoolean is_relationship() const;
```

Returns `oocTrue` if the described property is a relationship (association) and `oocFalse` if it is an attribute.

type_of

Gets the type of the described property.

```
const d_Type  &type_of() const;
```

Returns A type descriptor for the declared type of the described property.

d_Ref_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Attribute_Type ->d_Ref_Type**

The class `d_Ref_Type` represents descriptors for reference types. An instance of `d_Ref_Type` is called a *reference-type descriptor*.

See:

- "Reference Summary" on page 274 for an overview of member functions
- "Reference Index" on page 274 for a list of member functions

About Reference-Type Descriptors

A reference-type descriptor provides information about a particular object-reference type, called its *described type*.

NOTE The only reference types that Active Schema supports are object references to instances of a particular persistence-capable referenced class.

You should never instantiate this class directly. Instead, you can obtain a reference-type descriptor either from the module descriptor for the top-level module or from an attribute descriptor for an object-reference attribute. Typically, you obtain an instance by calling the inherited `type_of` member function of an attribute descriptor.

Reference Summary

Getting the Referenced Class	referenced_type
Getting Information About the Described Type	is_short ref_kind

Reference Index

is_ref_type	Overrides the inherited member function. Indicates that the described type is an object-reference type.
is_short	Tests whether the described type is a short object-reference type.
ref_kind	Gets the ODMG reference kind of the described type.
referenced_type	Gets the type referenced by the described type.

Member Functions

is_ref_type

Overrides the inherited member function. Indicates that the described type is an object-reference type.

```
virtual ooBoolean is_ref_type() const;
```

Returns `oocTrue`.

is_short

Tests whether the described type is a short object-reference type.

```
ooBoolean is_short() const;
```

Returns `oocTrue` if the described type is a short object-reference type `ooShortRef(`*`Class`*`)`; otherwise, `oocFalse`.

ref_kind

Gets the ODMG reference kind of the described type.

```
d_Ref_Kind ref_kind() const;
```

Returns REF.

Discussion Object references are the only kind of ODMG reference that Objectivity / DB supports.

referenced_type

Gets the type referenced by the described type.

```
const d_Type  &referenced_type() const;
```

Returns A type descriptor for the class referenced by the described type.

Discussion You can cast the returned type descriptor to a class descriptor if you need to call member functions defined by `d_Class`.

d_Relationship Class

Inheritance: **d_Meta_Object->d_Property->d_Attribute->d_Relationship**

The class `d_Relationship` represents descriptors for relationships between classes in the schema of the federated database. An instance of `d_Relationship` is called a *relationship descriptor*.

See:

- "Reference Summary" on page 278 for an overview of member functions
- "Reference Index" on page 278 for a list of member functions

About Relationship Descriptors

A relationship descriptor provides information about a particular relationship, called its *described relationship*.

You should never instantiate this class directly. Instead, you can obtain a relationship descriptor from a class descriptor for the class that defines the relationship:

- Call the `resolve_relationship` member function of the class descriptor to look up the relationship by name.
- Call the `defines_relationship_begin` member function of the class descriptor to get an iterator for all relationships defined in the class.

Because a relationship descriptor is a special kind of attribute descriptor, you can also obtain a relationship descriptor as you would obtain any attribute descriptor. If you do so, however, you need to cast the resulting attribute descriptor to a relationship descriptor before you can call any member function defined in this class.

Reference Summary

Getting Information About the Described Relationship	other_class inverse copy_mode versioning propagation encoded_assoc_number rel_kind
Testing the Described Relationship	is_to_many is_bidirectional is_inline is_short
Getting Related Descriptors	other_class inverse

Reference Index

copy_mode	Gets the copy mode of the described relationship.
encoded_assoc_number	Gets the type number encoding characteristics of the described relationship.
inverse	Gets the inverse relationship of the described bidirectional relationship.
is_bidirectional	Tests whether the described relationship is bidirectional.
is_inline	Tests whether the described relationship is inline.
is_relationship	Overrides the inherited member function. Indicates that this is a relationship descriptor.
is_short	Tests whether the described relationship is a short relationship.
is_to_many	Tests whether the described relationship is to-many.
other_class	Gets the destination class of the described relationship.
propagation	Gets the propagation behavior of the described relationship.

rel_kind	Gets the ODMG relationship kind of the described relationship.
versioning	Gets the versioning mode of the described relationship.

Member Functions

copy_mode

Gets the copy mode of the described relationship.

```
uint8 copy_mode() const;
```

Returns The copy mode of the described relationship, which specifies what happens to an association from a source object to a destination object when the source object is copied; one of the following:

- 0 indicates that this relationship descriptor has no information about the copy mode.
- `oocCopyDrop` indicates that the association is deleted.
- `oocCopyMove` indicates that the association is moved from the source object to its new copy.
- `oocCopyCopy` indicates that the association is copied from the source object to the new object.

encoded_assoc_number

Gets the type number encoding characteristics of the described relationship.

```
ooTypeNumber encoded_assoc_number() const;
```

Returns The type number encoding characteristics of the described relationship.

Discussion The bidirectional relationships in the schema are assigned serially-allocated 32-bit integers, called their *encoded association numbers*. Certain high-order bits of an encoded association number are set to encrypt the relationship's direction and other characteristics.

Most applications do not need to work with encoded association numbers. However, if you need to exactly recreate another schema description, you can call this member function of a relationship descriptor for a bidirectional relationship that you want to recreate. You can pass the resulting number as the *`specifiedAssocNum`* parameter to the add_bidirectional_relationship

member function of the proposed class in which you are duplicating the existing description.

inverse

Gets the inverse relationship of the described bidirectional relationship.

```
const d_Relationship  &inverse() const;
```

Returns A relationship descriptor for the inverse relationship of the described bidirectional relationship, or the null descriptor if the described relationship is unidirectional.

This member function signals a `CantFindRelInverse` error if it fails to find the inverse of the described bidirectional relationship.

is_bidirectional

Tests whether the described relationship is bidirectional.

```
ooBoolean is_bidirectional() const;
```

Returns `oocTrue` if the described relationship is bidirectional; otherwise, `oocFalse`.

is_inline

Tests whether the described relationship is inline.

```
ooBoolean is_inline() const;
```

Returns `oocTrue` if the described relationship is an inline relationship; otherwise, `oocFalse`.

is_relationship

Overrides the inherited member function. Indicates that this is a relationship descriptor.

```
virtual ooBoolean is_relationship() const;
```

Returns `oocTrue`.

is_short

Tests whether the described relationship is a short relationship.

```
ooBoolean is_short() const;
```

Returns `oocTrue` if the described relationship is a short relationship; otherwise, `oocFalse`.

is_to_many

Tests whether the described relationship is to-many.

```
ooBoolean is_to_many() const;
```

Returns `oocTrue` if the described relationship is a to-many relationship and `oocFalse` if it is a to-one relationship.

other_class

Gets the destination class of the described relationship.

```
const d_Class  &other_class() const;
```

Returns A class descriptor for the destination class of the described relationship.

Discussion To get the source class of the described relationship, call this relationship descriptor's inherited `defined_in_class` member function.

propagation

Gets the propagation behavior of the described relationship.

```
uint8 propagation() const;
```

Returns The propagation behavior of the described relationship, which specifies whether the locking and deletion operations are propagated from locked or deleted source objects to their related destination objects; one of the following:

- 0 indicates that neither locks nor deletions are propagated.
- `oocLockPropagationYesDeletePropagationNo` indicates that locks are propagated, but deletions are not.
- `oocLockPropagationNoDeletePropagationYes` indicates that deletions are propagated, but locks are not.
- `oocLockPropagationYesDeletePropagationYes` indicates that both locks and deletions are propagated.

rel_kind

Gets the ODMG relationship kind of the described relationship.

```
d_Rel_Kind rel_kind() const;
```

Returns The ODMG relationship kind of the described relationship; one of the following:

- REL_REF indicates a to-one relationship.
- REL_LIST indicates a to-many relationship.

Discussion Because Objectivity / DB does not support the ODMG relationship type REL_SET, this member function simply tells whether the relationship is to-one or to-many. The is_to_many member function is a simpler way to obtain the same information.

versioning

Gets the versioning mode of the described relationship.

```
uint8 versioning() const;
```

Returns The versioning mode of the described relationship, which specifies what happens to an association from a source object to a destination object when a new version of the source object is created; one of the following:

- 0 indicates that this relationship descriptor has no information about the versioning mode.
- oocVersionDrop indicates that the association is deleted.
- oocVersionMove indicates that the association is moved from the source object to its new version.
- oocVersionCopy indicates that the association is copied from the source object to its new version.

d_Scope Class

Inheritance: **d_Scope**

The abstract class d_Scope represents objects that organize the entities in the federated database schema. An instance of any concrete class derived from d_Scope is called a *scope*.

See:

- "Reference Summary" on page 283 for an overview of member functions
- "Reference Index" on page 284 for a list of member functions

About Scopes

Each scope contains definitions of entities in the schema. This class defines member functions for iterating through the entities defined in a scope and for looking up a particular entity in the scope.

Because d_Scope is abstract, you never instantiate it. You should not derive your own classes from this class.

For additional information, see "Scope" on page 23.

Reference Summary

Looking up an Entity	resolve
Iterating Through Entities in a Scope	defines_begin
Testing	is_class is_module

Reference Index

Member Functions

defines_begin

Gets an iterator for this scope.

```
virtual meta_object_iterator defines_begin() const = 0;
```

Returns A descriptor iterator that finds all entities defined in this scope.

See also defines_end

defines_end

Gets an iterator representing the termination condition for iteration through this scope.

```
virtual meta_object_iterator defines_end() const = 0;
```

Returns A descriptor iterator that is positioned after the last entity defined in this scope.

Discussion You can compare the iterator returned by defines_begin with the one returned by this member function to test whether iteration has finished.

is_class

Tests whether this scope is a class.

```
virtual ooBoolean is_class() const;
```

Returns oocTrue if this scope is a class; otherwise oocFalse.

is_module

Tests whether this scope is a module.

```
virtual ooBoolean is_module() const;
```

Returns `oocTrue` if this scope is a module; otherwise `oocFalse`.

resolve

Looks up a name in this scope.

```
virtual const d_Meta_Object  &resolve(
     const char *n,
     int32 version = oocLatestVersion) const = 0;
```

Parameters

n

The name to be looked up.

version

The desired version of the entity named *n*. This optional parameter can be specified when looking up a class that was created using the Objectivity/C++ versioning feature.

Returns A descriptor for the entity with the specified name and version that is defined in this scope, or the null descriptor if no such entity exists.

Discussion The returned generic descriptor can be cast to the appropriate descriptor class (for example, `d_Module` or `d_Class`). An alternative to calling this member function is to call more specific functions defined by derived classes.

See also
`d_Class::resolve_attribute`
`d_Class::resolve_relationship`
`d_Module::resolve_class`
`d_Module::resolve_module`
`d_Module::resolve_type`

d_Type Class

Inheritance: **d_Meta_Object->d_Type**

The abstract class d_Type represents descriptors for types in the schema of the federated database.

See:

- "Reference Summary" on page 288 for an overview of member functions
- "Reference Index" on page 288 for a list of member functions

About Type Descriptors

An instance of any concrete class derived from d_Type is called a *type descriptor*; it provides information about a particular type, called its *described type*. Concrete derived classes represent descriptors for:

- Classes
- Attribute types
- Relationship types

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

Reference Summary

Getting Information About the Described Type	type_number dimension defined_in_module
Testing the Described Type	is_basic_type is_string_type is_ref_type is_varray_type is_varray_basic_type is_varray_ref_type is_varray_embedded_class_type is_relationship_type is_unidirectional_relationship_type is_bidirectional_relationship_type
Getting Related Descriptors	defined_in_module used_in_property_begin used_in_ref_type_begin used_in_collection_type_begin

Reference Index

is_type	Overrides the inherited member function. Indicates that this is a type descriptor.
is_unidirectional_relationship_type	Tests whether the described type is a unidirectional relationship type.
is_varray_basic_type	Tests whether the described type is a variable-size array type with numeric elements.
is_varray_embedded_class_type	Tests whether the described type is a variable-size array type whose elements are instances of some non-persistence-capable class.
is_varray_ref_type	Tests whether the described type is a variable-size array type whose elements are object-references to instances of some persistence-capable class.
is_varray_type	Tests whether the described type is a variable-size array type.
type_number	Gets the unique type number for the described type.
used_in_collection_type_begin	Gets an iterator for the collection types in the schema that are created from the described type.
used_in_collection_type_end	Gets an iterator representing the termination condition for iteration through the collection types that are created from the described type.
used_in_property_begin	Gets an iterator for the properties in the schema that use the described type.
used_in_property_end	Gets an iterator representing the termination condition for iteration through the properties that use the described type.

used_in_ref_type_begin	Gets an iterator for the object-reference types in the schema that reference the described type.
used_in_ref_type_end	Gets an iterator representing the termination condition for iteration through the object-reference types that reference the described type.

Member Functions

defined_in_module

Gets the module in which the described type is defined.

```
const d_Module  &defined_in_module() const;
```

Returns A module descriptor for the module containing the described type.

dimension

Gets the layout size for a value of the described type.

```
size_t dimension() const;
```

Returns The layout size in bytes for a value of the described type on the platform where the current application is running.

Discussion If the described type is a class, the result is the number of bytes required to store an instance of that class.

If the described type is a property type, the result is the number of bytes required to store a property of this type within the data of an instance of a class that contains the property.

is_basic_type

Tests whether the described type is a basic numeric type.

```
virtual ooBoolean is_basic_type() const;
```

Returns `oocTrue` if the described type is a basic numeric type; otherwise, `oocFalse`.

is_bidirectional_relationship_type

Tests whether the described type is a bidirectional relationship type.

```
virtual ooBoolean is_bidirectional_relationship_type() const;
```

Returns `oocTrue` if the described type is a bidirectional relationship type; otherwise, `oocFalse`.

is_ref_type

Tests whether the described type is an object-reference type.

```
virtual ooBoolean is_ref_type() const;
```

Returns `oocTrue` if the described type is an object-reference type `ooRef(`*`PCclass`*`)` or `ooShortRef(`*`PCclass`*`)`; otherwise, `oocFalse`.

is_relationship_type

Tests whether the described type is a relationship type.

```
virtual ooBoolean is_relationship_type() const;
```

Returns `oocTrue` if the described type is a relationship type; otherwise, `oocFalse`.

is_string_type

Tests whether the described type is a string class.

```
virtual ooBoolean is_string_type() const;
```

Returns `oocTrue` if the described type is a string class; otherwise, `oocFalse`.

The string classes are:

- The ASCII string class `ooVString`
- The optimized string classes `ooString(`*`N`*`)`
- The Unicode string class `ooUTF8String`
- The Smalltalk string class `ooSTString`

is_type

Overrides the inherited member function. Indicates that this is a type descriptor.

```
ooBoolean is_type() const;
```

Returns `oocTrue`.

is_unidirectional_relationship_type

Tests whether the described type is a unidirectional relationship type.

```
virtual ooBoolean is_unidirectional_relationship_type() const;
```

Returns `oocTrue` if the described type is a unidirectional relationship type; otherwise, `oocFalse`.

is_varray_basic_type

Tests whether the described type is a variable-size array type with numeric elements.

```
virtual ooBoolean is_varray_basic_type() const;
```

Returns `oocTrue` if the described type is a VArray of numeric elements; otherwise, `oocFalse`.

is_varray_embedded_class_type

Tests whether the described type is a variable-size array type whose elements are instances of some non-persistence-capable class.

```
virtual ooBoolean is_varray_embedded_class_type() const;
```

Returns `oocTrue` if the described type is an embedded-class VArray type; otherwise, `oocFalse`.

is_varray_ref_type

Tests whether the described type is a variable-size array type whose elements are object-references to instances of some persistence-capable class.

```
virtual ooBoolean is_varray_ref_type() const;
```

Returns `oocTrue` if the described type is an object-reference VArray type; otherwise, `oocFalse`.

is_varray_type

Tests whether the described type is a variable-size array type.

```
virtual ooBoolean is_varray_type() const;
```

Returns `oocTrue` if the described type is a VArray type; otherwise, `oocFalse`.

type_number

Gets the unique type number for the described type.

```
virtual ooTypeNumber type_number();
```

Returns The unique type number for the described type.

used_in_collection_type_begin

Gets an iterator for the collection types in the schema that are created from the described type.

```
collection_type_iterator used_in_collection_type_begin() const;
```

Returns A collection-type iterator that finds all collection types created from the described type.

Discussion The following table lists the collection types that can be created from the various described types. The returned iterator will find any relevant collection type that is used in the schema.

Described Type	Possible Collection Types
Numeric type *N*	`ooVArray(`*N*`)`
Persistence-capable class *C*	`ooVArray(ooRef(`*C*`))` `ooVArray(ooShortRef(`*C*`))`
Non-persistence-capable class *F*	`ooVArray(`*F*`)`
Object-reference type `ooRef(`*C*`)`	`ooVArray(ooRef(`*C*`))`
Object-reference type `ooShortRef(`*C*`)`	`ooVArray(ooShortRef(`*C*`))`
Collection type `ooVArray(`*X*`)`	(none)
Relationship type *R*	(none)

See also `used_in_collection_type_end`

used_in_collection_type_end

Gets an iterator representing the termination condition for iteration through the collection types that are created from the described type.

```
collection_type_iterator used_in_collection_type_end() const;
```

Returns A collection-type iterator that is positioned after the last related collection type.

Discussion You can compare the iterator returned by `used_in_collection_type_begin` with the one returned by this member function to test whether iteration has finished.

used_in_property_begin

Gets an iterator for the properties in the schema that use the described type.

```
property_iterator used_in_property_begin() const;
```

Returns A property iterator that finds all properties in the schema that use the described type.

Discussion The following table lists properties that use the various described types. The returned iterator will find any relevant property that is used in the schema.

Described Type	Relevant Properties
Numeric type *N*	Attributes of types: *N* ooVArray(*N*)
Persistence-capable class *C*	Attributes of types: ooRef(*C*) ooShortRef(*C*) ooVArray(ooRef(*C*)) ooVArray(ooShortRef(*C*))
Non-persistence-capable class *F*	Attributes of types: *F* ooVArray(*F*)
Object-reference type ooRef(*C*)	Attributes of types: ooRef(*C*) ooVArray(ooRef(*C*))
Object-reference type ooShortRef(*C*)	Attributes of types: ooShortRef(*C*) ooVArray(ooShortRef(*C*))
Collection type ooVArray(*X*)	Attributes of type: ooVArray(*X*)
Unidirectional relationship type with destination class *C*	Unidirectional relationships to class *C*
Bidirectional relationship type with destination class *C*	Bidirectional relationships to class *C*

The returned iterator gets generic property descriptors for each property that uses the described type. You can call the `is_relationship_type` member function to find out whether the described type is a relationship type. If so, you can cast the property descriptors to relationship descriptors; if not, you can cast the property descriptors to attribute descriptors.

See also `used_in_property_end`

used_in_property_end

Gets an iterator representing the termination condition for iteration through the properties that use the described type.

```
property_iterator used_in_property_end() const;
```

Returns A property iterator that is positioned after the last related property.

Discussion You can compare the iterator returned by `used_in_property_begin` with the one returned by this member function to test whether iteration has finished.

used_in_ref_type_begin

Gets an iterator for the object-reference types in the schema that reference the described type.

```
ref_type_iterator used_in_ref_type_begin() const;
```

Returns A reference-type iterator that finds all object-reference types that reference the described type.

Discussion The following table lists the possible object-reference types for each of the various described types. The returned iterator will find any relevant object-reference type that is used in the schema.

Described Type	Possible Object-Reference Types
Numeric type *N*	(none)
Persistence-capable class *C*	`ooRef(`*C*`)` `ooShortRef(`*C*`)`
Non-persistence-capable class *F*	(none)
Object-reference type `ooRef(`*C*`)`	(none)
Object-reference type `ooShortRef(`*C*`)`	(none)

Described Type	Possible Object-Reference Types
Collection type `ooVArray(`*X*`)`	(none)
Relationship type *R*	(none)

See also `used_in_ref_type_end`

used_in_ref_type_end

Gets an iterator representing the termination condition for iteration through the object-reference types that reference the described type.

```
ref_type_iterator used_in_ref_type_end() const;
```

Returns A reference-type iterator that is positioned after the last related object-reference type.

Discussion You can compare the iterator returned by `used_in_ref_type_begin` with the one returned by this member function to test whether iteration has finished.

list_iterator<*element_type*> Class

Inheritance: **list_iterator<***element_type***>**

The class template `list_iterator` generates constant forward iterator classes that iterate through lists of elements of the same data type. That list is called the iterator's *iteration list*. During iteration, the list iterator keeps track of its position within its iteration list. The element at the current position is called the iterator's *current element*.

See:

- "Reference Summary" on page 300 for an overview of member functions
- "Reference Index" on page 301 for a list of member functions

About List Iterators

A list iterator of the class `list_iterator<`*`element_type`*`>` iterates through a list of elements whose type is *`element_type`*. For example, a list iterator of the class `list_iterator<d_Module>` iterates through a list of module descriptors.

Predefined List-Iterator Classes

Active Schema includes the following predefined synonyms for the classes created from this template. For example, an instance of the class `module_iterator` is an iterator for a list of module descriptors.

Synonym Class Name	*element_type*	Iterator for Lists of
`attribute_iterator`	`d_Attribute`	Attribute descriptors
`collection_type_iterator`	`d_Collection_Type`	Collection-type descriptors
`inheritance_iterator`	`d_Inheritance`	Inheritance descriptors

Synonym Class Name	*element_type*	Iterator for Lists of
module_iterator	d_Module	Module descriptors
property_iterator	d_Property	Property descriptors
proposed_base_class_iterator	Proposed_Base_Class	Proposed base classes
proposed_class_iterator	Proposed_Class	Proposed classes
proposed_property_iterator	Proposed_Property	Proposed properties
ref_type_iterator	d_Ref_Type	Reference-type descriptors
relationship_iterator	d_Relationship	Relationship descriptors

Obtaining List Iterators

You should not directly instantiate any of the classes created from this template. Instead, you work with iterators returned by various member functions. An iterator of any class in the preceding table is obtained from a descriptor and iterates over an internal list maintained by that descriptor.

Chapter 6, “Working With Iterators,” contains additional information about iterators.

Attribute Iterators

You can call the defines_attribute_begin member function of a class descriptor to obtain an attribute iterator for attributes defined in the described class (including relationships and embedded-class attributes corresponding to base classes). You can test for that iterator’s termination condition by comparing it with the attribute iterator returned by the same class descriptor’s defines_attribute_end member function.

Collection-Type Iterators

You can call the used_in_collection_type_begin member function of a type descriptor to obtain a collection-type iterator for collection types using the described type (for example, as their element type). You can test for that iterator’s termination condition by comparing it with the collection-type iterator returned by the same type descriptor’s used_in_collection_type_end member function.

Inheritance Iterators

You can call the base_class_list_begin member function of a class descriptor to obtain an inheritance iterator for inheritance connections between the described class and its immediate parent classes. You can test for that iterator's termination condition by comparing it with the inheritance iterator returned by the same class descriptor's base_class_list_begin member function.

You can call the sub_class_list_begin member function of a class descriptor to obtain an inheritance iterator for inheritance connections between the described class and its child classes. You can test for that iterator's termination condition by comparing it with the inheritance iterator returned by the same class descriptor's sub_class_list_end member function.

Module Iterators

You can call the named_modules_begin member function of a module descriptor for the top-level module to obtain a module iterator for named modules in the schema. You can test for that iterator's termination condition by comparing it with the module iterator returned by the same module descriptor's named_modules_begin member function.

Property Iterators

You can call the used_in_property_begin member function of a type descriptor to obtain a property iterator for properties using the described type. You can test for that iterator's termination condition by comparing it with the property iterator returned by the same type descriptor's used_in_property_end member function.

Proposed-Base-Class Iterators

You can call the base_class_list_begin member function of a proposed class to obtain a proposed-base-class iterator for the base classes of the proposed class. You can test for that iterator's termination condition by comparing it with the proposed-base-class iterator returned by the same proposed class's base_class_list_begin member function.

Proposed-Class Iterators

You can call the proposed_classes_begin member function of a module descriptor to obtain a proposed-class iterator for the proposed classes in the proposal list of the described module. You can test for that iterator's termination condition by comparing it with the proposed-class iterator returned by the same module descriptor's proposed_classes_begin member function.

Proposed-Property Iterators

You can call the `defines_property_begin` member function of a proposed class to obtain a proposed-property iterator for the properties of the proposed class. You can test for that iterator's termination condition by comparing it with the proposed-base-class iterator returned by the same proposed class's `defines_property_end` member function.

Reference-Type Iterators

You can call the `used_in_ref_type_begin` member function of a type descriptor to obtain a reference-type iterator for reference types using the described type. You can test for that iterator's termination condition by comparing it with the reference-type iterator returned by the same type descriptor's `used_in_ref_type_end` member function.

Relationship Iterators

You can call the `defines_relationship_begin` member function of a class descriptor to obtain a relationship iterator for relationships defined in the described class. You can test for that iterator's termination condition by comparing it with the relationship iterator returned by the same class descriptor's `defines_relationship_end` member function.

Reference Summary

Assigning	`operator=`
Getting the Current Element	`operator*`
Advancing the Current Position	`operator++`
Comparing	`operator==` `operator!=`

Reference Index

Operators

operator++

Increment operator; advances this list iterator's current position.

```
1.   list_iterator<element_type>  &operator++();
2.   list_iterator<element_type> operator++(int n);
```

Parameters

n

This parameter is not used in calling this operator; its presence in the function declaration specifies a postfix operator.

Returns

(Variant 1) This list iterator, advanced to the next element.
(Variant 2) A new list iterator, set to this iterator before its position is advanced.

Discussion

Variant 1 is the prefix increment operator, which advances this list iterator and then returns it.

Variant 2 is the postfix increment operator, which returns a new list iterator set to this iterator, and then advances this iterator.

If the current position is already after the last element in the iteration list, neither variant advances this iterator.

operator*

Dereference operator; gets the element at this list iterator's current position.

```
const element_type  &operator*() const;
```

Returns The element at this list iterator's current position.

Discussion You should ensure that iteration has not terminated before calling this member function. The return value is undefined if the current position is after the last element in the iteration list.

operator=

Assignment operator; sets this list iterator to be a copy of the specified list iterator.

```
list_iterator<element_type>  &operator=(
     const list_iterator<element_type> &otherIteratorR);
```

Parameters *otherIteratorR*

The list iterator specifying the new value for this list iterator.

Returns This list iterator after it has been set to a copy of *otherIteratorR*.

operator==

Equality operator; tests whether this list iterator is equal to the specified list iterator.

```
int operator==(
     const list_iterator<element_type> &otherIteratorR) const;
```

Parameters *otherIteratorR*

The list iterator with which to compare this list iterator.

Returns Nonzero if the two list iterators are equal and zero if they are different.

Discussion Two list iterators are equal if they iterate over the same list and they have the same current position.

See also operator!=

operator!=

inequality operator; tests whether this list iterator is different from the specified list iterator.

```
int operator!= (
     const list_iterator<element_type> &otherIteratorR) const;
```

Parameters

otherIteratorR

The list iterator with which to compare this list iterator.

Returns

Nonzero if the two list iterators are different and zero if they are equal.

Discussion

Two list iterators are different if they iterate over different lists or if they are at different positions in the same list.

See also

operator==

meta_object_iterator Class

Inheritance: **meta_object_iterator**

The class `meta_object_iterator` represents iterators for descriptors of entities in a given scope. An instance of this class is called a *descriptor iterator.*

See:

- "Reference Summary" on page 306 for an overview of member functions
- "Reference Index" on page 306 for a list of member functions

About Descriptor Iterators

A descriptor iterator steps through the entities in the scope of some particular module or class. That collection of entities is called the iterator's iteration set; during iteration, the descriptor iterator keeps track of its position within its iteration set. The element at the current position is called the iterator's current element. The descriptor iterator allows you to step through the iteration set, obtaining a descriptor for the current element at each step.

You should not instantiate this class directly. Instead, you work with descriptor iterators returned by the following member functions:

- The `defines_begin` member function of a module descriptor returns a descriptor iterator for the entities in the scope of the described module. You can test for that iterator's termination condition by comparing it with the descriptor iterator returned by the same module descriptor's `defines_end` member function.
- The `defines_begin` member function of a class descriptor returns a descriptor iterator for the properties of the described class. You can test for that iterator's termination condition by comparing it with the descriptor iterator returned by the same class descriptor's `defines_end` member function.

Chapter 6, “Working With Iterators,” contains additional information about iterators.

Reference Summary

Copying	operator=
Getting the Current Element	operator*
Advancing the Current Position	operator++
Comparing	operator== operator!=

Reference Index

is_attr_iterator	Tests whether this descriptor iterator steps through the attributes of a class.
operator++	Increment operator; advances this descriptor iterator’s current position.
operator*	Dereference operator; gets this descriptor iterator’s current element.
operator=	Assignment operator; sets this descriptor iterator to be a copy of the specified descriptor iterator.
operator==	Equality operator(==). Tests whether this descriptor iterator is the same as the specified descriptor iterator.
operator!=	Inequality operator; tests whether this descriptor iterator is different from the specified descriptor iterator.

Operators

operator++

Increment operator; advances this descriptor iterator's current position.

```
1.   virtual meta_object_iterator  &operator++();
2.   virtual meta_object_iterator operator++(int n);
```

Parameters *n*

This parameter is not used in calling this operator; its presence in the function declaration specifies a postfix operator.

Returns (Variant 1) This descriptor iterator, advanced to the next entity.
(Variant 2) A new descriptor iterator, set to this iterator before its position is advanced.

Discussion Variant 1 is the prefix increment operator, which advances this descriptor iterator and then returns it.

Variant 2 is the postfix increment operator, which returns a new descriptor iterator set to this iterator, and then advances this iterator.

If the current position is already after the last entity in the iteration set, neither variant advances this iterator.

operator*

Dereference operator; gets this descriptor iterator's current element.

```
virtual const d_Meta_Object  &operator*() const;
```

Returns A descriptor for the current element.

Discussion You should ensure that iteration has not terminated before calling this member function. The return value is undefined if the current position is after the last entity in the iteration set.

operator=

Assignment operator; sets this descriptor iterator to be a copy of the specified descriptor iterator.

```
meta_object_iterator  &operator=(
     const meta_object_iterator &moI);
```

Parameters *moI*

The descriptor iterator specifying the new value for this descriptor iterator.

Returns This descriptor iterator after it has been set to a copy of *moI*.

operator==

Equality operator(==). Tests whether this descriptor iterator is the same as the specified descriptor iterator.

```
virtual int operator==(const meta_object_iterator &moI) const;
```

Parameters *moI*

The descriptor iterator with which to compare this descriptor iterator.

Returns Nonzero if the two descriptor iterators are equal and zero if they are different.

Discussion Two descriptor iterators are equal if they iterate over the same iteration set and they have the same current position.

See also operator!=

operator!=

Inequality operator; tests whether this descriptor iterator is different from the specified descriptor iterator.

```
virtual int operator!=(
     const meta_object_iterator &moI) const;
```

Parameters *moI*

The descriptor iterator with which to compare this descriptor iterator.

Returns Nonzero if the two descriptor iterators are different and zero if they are equal.

Discussion Two descriptor iterators are different if they iterate over different iteration sets or if they are at different positions in the same iteration set.

See also operator==

Member Functions

is_attr_iterator

Tests whether this descriptor iterator steps through the attributes of a class.

```
ooBoolean is_attr_iterator() const;
```

Returns `oocTrue` if this descriptor iterator steps through the attributes of a class; `oocFalse` if it steps through the types in a module.

Discussion This member function is used internally; applications typically do not need to call it.

Numeric_Value Class

Inheritance: **`Numeric_Value`**

The class `Numeric_Value` is a self-describing data type for persistent numeric values. An instance of this class, called a *numeric value*, contains up to 64 bits of raw data and a code indicating the basic numeric type of the data. The data can be any fundamental character, integer, floating-point, or pointer type.

See:

- "Reference Summary" on page 312 for an overview of member functions
- "Reference Index" on page 313 for a list of member functions

About Numeric Values

All numeric data in persistent objects is transferred to an Active Schema application as numeric values. Encapsulating the data within a numeric value avoids the inherent risk that the data may be transferred to program memory under mistaken assumptions about alignment, precision, or integral versus floating-point representation.

Typically, an Active Schema application does not work with numeric values explicitly, but instead works with the basic numeric data types such as `int8`, `float32`, `uint64`.

- When you pass a number of a basic numeric type as a parameter to a member function that expects a numeric value, the appropriate constructor converts the parameter to an instance of `Numeric_Value`.
- Certain member functions return an instance of `Numeric_Value` on the stack. If you call such a function, assigning its returned value to a variable of the correct basic numeric type, the appropriate conversion function converts the return value to the required type.
- If you examine persistent data in the federated database, you may obtain a numeric value (for example, the value of an attribute) without knowing what

type of data it contains. In that situation, you can call its `type` member function and then cast the numeric value to the correct basic numeric type. For example, if a numeric value's `type` member function returns `ooUINT32`, you can cast it to the basic numeric type `uint32`.

Reference Summary

Constructors	`Numeric_Value`
Getting Information About the Numeric Value	`type`
Testing the Numeric Value	`is_valid`

Comparing	`operator==`
	`operator!=`
	`operator<`
	`operator<=`
	`operator>`
	`operator>=`

Converting to Basic Numeric Types	`operator char`
	`operator int8`
	`operator uint8`
	`operator int16`
	`operator uint16`
	`operator int32`
	`operator uint32`
	`operator int64`
	`operator uint64`
	`operator float32`
	`operator float64`
	`operator void*`

Writing	`::operator<<`

Reference Index

operator void*	Converts this numeric value to a 32-bit pointer.
type	Gets the type of numeric data that this numeric value contains.

Constructors

Numeric_Value

Constructs a numeric value from a number of some basic numeric type.

```
1.  Numeric_Value(char n);
2.  Numeric_Value(int8 n);
3.  Numeric_Value(uint8 n);
4.  Numeric_Value(int16 n);
5.  Numeric_Value(uint16 n);
6.  Numeric_Value(int32 n);
7.  Numeric_Value(uint32 n);
8.  Numeric_Value(int64 n);
9.  Numeric_Value(uint64 n);
10. Numeric_Value(float32 n);
11. Numeric_Value(float64 n);
12. Numeric_Value(void *n);
```

Parameters *n*

The number to be converted to a numeric value.

Discussion If you pass a number of a basic numeric type as a parameter to a member function that expects a numeric value, one of these constructors will perform the necessary conversion.

Operators

operator==

Equality operator; tests whether this numeric value is equal to the specified numeric value.

```
int operator==(const Numeric_Value &otherVal);
```

Parameters

otherVal

The numeric value to be compared with this numeric value.

Returns

Nonzero if this numeric value is equal to *otherVal*; otherwise, zero.

Discussion

This member function performs any necessary type conversion before comparing the data in the two numeric values. Thus, if the two numeric values represent the same numeric quantity, they are considered equal even if their data is of different numeric types. For example, an equal comparison of an `int8` value 2 with a `float32` value 2.0 succeeds (returns nonzero).

This member function throws an `IllegalNumericCompare` exception if the two numeric values cannot be compared. This situation occurs when comparing a floating-point value with a 64-bit integer value or when comparing an unsigned 64-bit integer value with a signed 64-bit integer value.

See also

`operator!=`

operator!=

Inequality operator; tests whether this numeric value is different from the specified numeric value.

```
int operator!=(const Numeric_Value &otherVal);
```

Parameters

otherVal

The numeric value to be compared with this numeric value.

Returns

Nonzero if the two numeric values are different and zero if they are equal.

Discussion

This member function performs any necessary type conversion before comparing the data in the two numeric values. Thus, if the two numeric values represent the same numeric quantity, they are considered equal even if their data is of different numeric types. For example, an inequality comparison of an `int8` value 2 with a `float32` value 2.0 fails (returns zero).

This member function throws an `IllegalNumericCompare` exception if the two numeric values cannot be compared. This situation occurs when comparing a floating-point value with a 64-bit integer value or when comparing an unsigned 64-bit integer value with a signed 64-bit integer value.

See also `operator==`

operator<

Less-than operator; tests whether this numeric value is less than the specified numeric value.

```
int operator<(const Numeric_Value &otherVal);
```

Parameters *otherVal*

The numeric value to be compared with this numeric value.

Returns Nonzero if this numeric value is less than *otherVal*; otherwise, zero.

Discussion This member function performs any necessary type conversion before comparing the data in the two numeric values. It throws an `IllegalNumericCompare` exception if the two numeric values cannot be compared. This situation occurs when comparing a floating-point value with a 64-bit integer value or when comparing an unsigned 64-bit integer value with a signed 64-bit integer value.

See also `operator<=`
`operator>`

operator<=

Less-than-or-equal-to operator; tests whether this numeric value is less than or equal to the specified numeric value.

```
int operator<=(const Numeric_Value &otherVal);
```

Parameters *otherVal*

The numeric value to be compared with this numeric value.

Returns Nonzero if this numeric value is less than or equal to *otherVal*; otherwise, zero.

Discussion This member function performs any necessary type conversion before comparing the data in the two numeric values. Thus, if the two numeric values represent the same numeric quantity, they are considered equal even if their data is of different numeric types. For example, a less-than-or-equal-to comparison of an `int8` value 2 with a `float32` value 2.0 succeeds (returns nonzero).

This member function throws an IllegalNumericCompare exception if the two numeric values cannot be compared. This situation occurs when comparing a floating-point value with a 64-bit integer value or when comparing an unsigned 64-bit integer value with a signed 64-bit integer value.

See also operator<
operator>=

::operator<< global function

Stream insertion operator; writes the specified numeric value to the specified output stream.

```
ostream &::operator<<(
     ostream &stream,
     const Numeric_Value &value)
```

Parameters *stream*

The output stream to which the numeric value is to be written

value

The numeric value to be written.

Returns The output stream.

operator>

Greater-than operator; tests whether this numeric value is greater than the specified numeric value.

```
int operator>(const Numeric_Value &otherVal);
```

Parameters *otherVal*

The numeric value to be compared with this numeric value.

Returns Nonzero if this numeric value is greater than *otherVal*; otherwise, zero.

Discussion This member function performs any necessary type conversion before comparing the data in the two numeric values. It throws an IllegalNumericCompare exception if the two numeric values cannot be compared. This situation occurs when comparing a floating-point value with a 64-bit integer value or when comparing an unsigned 64-bit integer value with a signed 64-bit integer value.

See also operator<
operator>=

operator>=

Greater-than-or-equal-to operator; tests whether this numeric value is greater than or equal to the specified numeric value.

```
int operator>=(const Numeric_Value &otherVal);
```

Parameters

otherVal

The numeric value to be compared with this numeric value.

Returns

Nonzero if this numeric value is greater than or equal to *otherVal*; otherwise, zero.

Discussion

This member function performs any necessary type conversion before comparing the data in the two numeric values. Thus, if the two numeric values represent the same numeric quantity, they are considered equal even if their data is of different numeric types. For example, a greater-than-or-equal-to comparison of an `int8` value 2 with a `float32` value 2.0 succeeds (returns nonzero).

This member function throws an `IllegalNumericCompare` exception if the two numeric values cannot be compared. This situation occurs when comparing a floating-point value with a 64-bit integer value or when comparing an unsigned 64-bit integer value with a signed 64-bit integer value.

See also

`operator<=`
`operator>`

operator char

Converts this numeric value to an 8-bit character.

```
operator char() const;
```

Returns

This numeric value's data converted (if necessary) to an 8-bit character.

operator float32

Converts this numeric value to a single-precision floating-point number.

```
operator float32() const;
```

Returns

This numeric value's data converted (if necessary) to a single-precision floating-point number.

Discussion

This member function throws an `IllegalNumericConvert` exception if this numeric value contains an unsigned 64-bit integer that cannot be converted to floating-point.

operator float64

Converts this numeric value to a double-precision floating-point number.

```
operator float64() const;
```

Returns This numeric value's data converted (if necessary) to a double-precision floating-point number.

Discussion This member function throws an `IllegalNumericConvert` exception if this numeric value contains an unsigned 64-bit integer that cannot be converted to floating-point.

operator int8

Converts this numeric value to an 8-bit signed integer.

```
operator int8() const;
```

Returns This numeric value's data converted (if necessary) to an 8-bit signed integer.

operator int16

Converts this numeric value to a 16-bit signed integer.

```
operator int16() const;
```

Returns This numeric value's data converted (if necessary) to a 16-bit signed integer.

operator int32

Converts this numeric value to a 32-bit signed integer.

```
operator int32() const;
```

Returns This numeric value's data converted (if necessary) to a 32-bit signed integer.

operator int64

Converts this numeric value to a 64-bit signed integer.

```
operator int64() const;
```

Returns This numeric value's data converted (if necessary) to a 64-bit signed integer.

operator uint8

Converts this numeric value to an 8-bit unsigned integer.

```
operator uint8() const;
```

Returns This numeric value's data converted (if necessary) to an 8-bit unsigned integer.

operator uint16

Converts this numeric value to a 16-bit unsigned integer.

```
operator uint16() const;
```

Returns This numeric value's data converted (if necessary) to a 16-bit unsigned integer.

operator uint32

Converts this numeric value to a 32-bit unsigned integer.

```
operator uint32() const;
```

Returns This numeric value's data converted (if necessary) to a 32-bit unsigned integer.

operator uint64

Converts this numeric value to a 64-bit unsigned integer.

```
operator uint64() const;
```

Returns This numeric value's data converted (if necessary) to a 64-bit unsigned integer.

operator void*

Converts this numeric value to a 32-bit pointer.

```
operator void*() const;
```

Returns This numeric value's data converted (if necessary) to a 32-bit pointer.

Discussion You should not attempt to dereference the pointer you obtain with this operator.

Typically, a C++ pointer attribute is used for transient data because a pointer value saved by one process will not be meaningful (or valid) in a process that retrieves that value. Although pointer attributes contain transient data, the schema description of a class includes those attributes so that the shape of the class will be correct.

Member Functions

is_valid

Tests whether this is a valid numeric value.

```
ooBoolean is_valid() const;
```

Returns `oocTrue` if this is a valid numeric value; otherwise, `oocFalse`.

Discussion A numeric value is valid if it has a valid numeric type; it is invalid if its type is `ooNONE`.

type

Gets the type of numeric data that this numeric value contains.

```
ooBaseType type() const;
```

Returns The type of numeric data; one of the following:

- `ooCHAR` indicates an 8-bit character.
- `ooINT8` indicates an 8-bit signed integer.
- `ooINT16` indicates a 16-bit signed integer.
- `ooINT32` indicates a 32-bit signed integer.
- `ooINT64` indicates a 64-bit signed integer.
- `ooUINT8` indicates an 8-bit unsigned integer.
- `ooUINT16` indicates a 16-bit unsigned integer.
- `ooUINT32` indicates a 32-bit unsigned integer.
- `ooUINT64` indicates a 64-bit unsigned integer.
- `ooFLOAT32` indicates a 32-bit (single-precision) floating-point number.
- `ooFLOAT64` indicates a 64-bit (double-precision) floating-point number.
- `ooPTR` indicates a 32-bit pointer.
- `ooNONE` indicates that this numeric value is invalid.

Optimized_String_Value Class

Inheritance: **`Optimized_String_Value`**

The class `Optimized_String_Value` is a self-describing data type for optimized strings. An instance of this class is called an *optimized string value*.

See:

- "Reference Summary" on page 324 for an overview of member functions
- "Reference Index" on page 324 for a list of member functions

About Optimized String Values

An optimized string value provides access to an optimized string embedded in the data of some persistent object. The embedded string is an instance of an application-defined optimized string class `ooString(`*`N`*`)`.

You obtain an optimized string value for a particular optimized string attribute of a particular persistent object by converting the string value for that attribute to an optimized string value.

Reference Summary

Constructors	Optimized_String_Value
Copying Optimized String Values	Optimized_String_Value operator=
Getting Information About the String's Class	fixed_length
Getting Information About the String	length
Examining the String	operator[] get_copy
Modifying the String	set resize

Reference Index

fixed_length	Gets the size of the fixed-size array for this optimized string value.
get_copy	Gets a transient copy of the string from this optimized string value.
length	Gets the length of the string in this optimized string value.
operator[]	Subscript operator; gets the character at the specified index from this optimized string value.
operator=	Assignment operator; sets this optimized string value to a copy of the specified optimized string value.
Optimized_String_Value	Constructs an optimized string value.
resize	Adjusts the allocated storage for this optimized string value as necessary to accommodate strings of the specified length.
set	Sets the string for this optimized string value.

Constructors

Optimized_String_Value

Constructs an optimized string value.

```
1.  Optimized_String_Value(
      const String_Value &svR);
2.  Optimized_String_Value(
      const Optimized_String_Value &otherROR);
```

Parameters

svR

The string value from which to construct the optimized string value.

otherROR

The optimized string value to be copied.

Discussion

The first variant allows you to convert a string value containing an optimized string to an optimized string value with which you can view and modify the string data.

The second variant is the copy constructor. It creates a new optimized string value with the same persistent string data as the specified optimized string value. Both copies access the same persistent data. Any change to the string made with one optimized string value will be seen by the other optimized string value.

Operators

operator[]

Subscript operator; gets the character at the specified index from this optimized string value.

```
1.  const char &operator[](uint32 index) const;
2.  char &operator[](uint32 index);
```

Parameters

index

The zero-based index of the character of interest.

Returns

The character at the specified index.

Discussion The first variant allows you to get a particular character. The second variant allows you to replace a particular character (by using the operator in the left-hand operand of an assignment statement).

This operator throws a StringBoundsError exception if *index* is invalid (greater than or equal to the length of the string).

operator=

Assignment operator; sets this optimized string value to a copy of the specified optimized string value.

```
Optimized_String_Value &operator=(
     const Optimized_String_Value &otherROR);
```

Parameters *otherROR*

The optimized string value to be copied.

Returns This optimized string value after it has been updated to be a copy of *otherROR*.

Discussion Both copies access the same persistent data. Any change to the string made with one optimized string value will be seen by the other optimized string value.

Member Functions

fixed_length

Gets the size of the fixed-size array for this optimized string value.

```
size_t fixed_length() const;
```

Returns The number of elements in the fixed-size array for this optimized string value.

Discussion This member function returns *N* for a value of the class ooString(*N*), which is optimized for storing strings of fewer than *N* characters.

get_copy

Gets a transient copy of the string from this optimized string value.

```
char *get_copy(char *buffer = NULL)) const;
```

Parameters *buffer*

Pointer to the memory buffer to which the string should be copied. If this parameter is omitted, the string is copied to newly allocated memory.

Returns Pointer to a copy of the string from this optimized string value.

Discussion This member function copies the persistent string data, either to memory supplied by the caller or to newly allocated memory. In the latter case, the caller is responsible for freeing the copy's memory when it is no longer needed.

NOTE Any change to the returned string affects that transient copy only; the persistent string data is not changed.

See also `set`

length

Gets the length of the string in this optimized string value.

```
size_t length() const;
```

Returns The number of characters in the string that this optimized string value contains.

resize

Adjusts the allocated storage for this optimized string value as necessary to accommodate strings of the specified length.

```
ooStatus resize(uint32 newLength);
```

Parameters *`newLength`*

The length of strings that this optimized string value should be able to accommodate.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function ensures that this optimized string value has enough allocated storage to accommodate a string of *`newLength`* characters (including its terminating null character). If this optimized string value does not currently have enough storage (between its fixed-size character array and its VArray), its VArray is extended to *`newLength`* characters.

If *`newLength`* is less than the number of characters in the string, the string is truncated to the indicated number of characters, the last of which is the null character. Otherwise, the length of the string is not modified.

If *`newLength`* is less than the number of elements in the fixed-size character array, any storage allocated to this optimized string value's VArray is freed.

set

Sets the string for this optimized string value.

```
void set(const char *newString);
```

Parameters

newString

Pointer to the new string for this optimized string value.

Discussion

This member function sets the persistent string data for this optimized string value to a copy of the specified string.

Persistent_Data_Object Class

Inheritance: **`Persistent_Data_Object`**

The class `Persistent_Data_Object` is the abstract base class for classes that serve as self-describing data types for structured persistent data.

See:

- "Reference Summary" on page 330 for an overview of member functions
- "Reference Index" on page 330 for a list of member functions

About Persistent-Data Objects

An instance of any concrete class derived from `Persistent_Data_Object` is a *self-describing object* because it contains persistent data and a descriptor that provides detailed information about the structure and content of that data.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

The various concrete derived classes contain the following kinds of persistent data:

- `Class_Object` represents persistent objects and embedded objects.
- `VArray_Object` represents VArray attributes in the data of persistent objects.
- `Relationship_Object` represents relationships between persistent objects.

Related Classes

Active Schema represents persistent data with additional classes that are not derived from `Persistent_Data_Object`:

- `Numeric_Value` represents persistent data of basic numeric types.
- `String_Value` represents persistent data of string classes.
- `Optimized_String_Value` represents persistent data of the application-defined optimized string classes `ooString(`*N*`)` where *N* is the string length for which the class is optimized.

Reference Summary

Testing the Type of Persistent Data	`is_class_object`
	`is_relationship_object`
	`is_varray_object`

Testing for a Null Persistent-Data Object	`operator size_t`

Static Utilities	`enable_auto_update`
	`disable_auto_update`
	`auto_update_is_enabled`

Reference Index

`auto_update_is_enabled`	Tests whether automatic updating of persistent-data objects is enabled.
`disable_auto_update`	Disables automatic updating of persistent-data objects.
`enable_auto_update`	Enables automatic updating of persistent-data objects.
`is_class_object`	Tests whether this persistent-data object is a class object.
`is_relationship_object`	Tests whether this persistent-data object is a relationship object.

is_varray_object	Tests whether this persistent-data object is a VArray object.
operator size_t	Conversion operator that tests whether this persistent-data object is null.

Operators

operator size_t

Conversion operator that tests whether this persistent-data object is null.

```
operator size_t() const;
```

Returns Zero if this persistent-data object is null; otherwise, nonzero.

Discussion Many member functions return persistent-data objects. When such a member function fails, it returns a null persistent-data object. This operator allows you to use a persistent-data object as an integer expression to test whether that persistent-data object is valid (not null).

Member Functions

auto_update_is_enabled

Tests whether automatic updating of persistent-data objects is enabled.

```
static ooBoolean auto_update_is_enabled();
```

Returns `oocTrue` if automatic updating of persistent-data objects is enabled; otherwise, `oocFalse`.

disable_auto_update

Disables automatic updating of persistent-data objects.

```
static void disable_auto_update();
```

enable_auto_update

Enables automatic updating of persistent-data objects.

```
static void enable_auto_update();
```

is_class_object

Tests whether this persistent-data object is a class object.

```
virtual ooBoolean is_class_object() const;
```

Returns `oocTrue` if this persistent-data object is a class object; otherwise, `oocFalse`.

is_relationship_object

Tests whether this persistent-data object is a relationship object.

```
virtual ooBoolean is_relationship_object() const;
```

Returns `oocTrue` if this persistent-data object is a relationship object; otherwise, `oocFalse`.

is_varray_object

Tests whether this persistent-data object is a VArray object.

```
virtual ooBoolean is_varray_object() const;
```

Returns `oocTrue` if this persistent-data object is a VArray object; otherwise, `oocFalse`.

Property_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type**

The abstract class `Property_Type` represents descriptors for property types in the schema of the federated database.

Concrete derived classes represent descriptors for attribute types and relationship types. An instance of any concrete derived class is called a *property-type descriptor*; it provides information about a particular property type, called its *described type*.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

Reference Index

`id`	Gets the unique ID that identifies the described type within its scope.
`operator size_t`	Conversion operator that tests whether this property-type descriptor is null.
`type_number`	Gets the unique type number for the described type.

Operators

operator size_t

Conversion operator that tests whether this property-type descriptor is null.

```
virtual operator size_t() const;
```

Returns Zero if this property-type descriptor is null; otherwise, the nonzero type number of the described type.

Discussion Any member function that looks up a property-type descriptor returns a property-type descriptor object; unsuccessful searches return a null property-type descriptor. This operator allows you to use a property-type descriptor as an integer expression to test whether that property-type descriptor is valid (not null).

Member Functions

id

Gets the unique ID that identifies the described type within its scope.

```
virtual uint32 id() const;
```

Returns The ID for the described type.

Discussion The ID of a property type is the same as its type number.

type_number

Gets the unique type number for the described type.

```
virtual ooTypeNumber type_number() const;
```

Returns The unique type number for the described property type.

Proposed_Attribute Class

Inheritance: **d_Meta_Object->Proposed_Property->Proposed_Attribute**

The class `Proposed_Attribute` is the abstract base class for descriptors of the attributes of a proposed class. An instance of any concrete derived class is called a *proposed attribute*; it provides information about a particular attribute within the description of a particular proposed class.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

Member Functions

change_array_size

Changes the number of elements in this proposed attribute's fixed-size array of elements.

```
ooStatus change_array_size(size_t newSize);
```

Parameters

`newSize`

The new number of elements in the fixed-size array of values for this proposed attribute (or one if the attribute is to contain a single value instead of an array).

Returns `oocSuccess` if successful; otherwise `oocError`.

Proposed_Base_Class Class

Inheritance: **d_Meta_Object->Proposed_Base_Class**

The class `Proposed_Base_Class` represents the base classes of a proposed class. An instance of this class is called a *proposed base class*.

See:

- "Reference Summary" on page 338 for an overview of member functions
- "Reference Index" on page 338 for a list of member functions

About Proposed Base Classes

A proposed base class provides information about a particular base class within the description of a particular proposed class. The proposed base class can correspond to a class that exists in the schema, a proposed class, or a new class that will be proposed before proposals are activated.

You should never instantiate this class directly. Instead, you can obtain a proposed base class from the proposed class whose description contains the proposed base class.

- Call the `resolve_base_class` member function of the proposed class to look up the proposed base class by name.
- Call the `base_class_list_begin` member function of the proposed class to get an iterator for all proposed base classes of the proposed class.

Reference Summary

Getting Information About the Proposed Base Class	defined_in_class position access_kind previous_name
Testing the Proposed Base Class	operator== operator!= persistent_capable
Testing for the Null Descriptor	operator size_t
Modifying the Proposed Base Class	change_access
Getting Descriptors from the Proposed Base Class	defined_in_class

Reference Index

access_kind	Gets the access kind of this proposed base class.
change_access	Changes the access kind of this proposed base class.
defined_in_class	Gets the proposed class whose description contains this proposed base class.
operator==	Equality operator; tests whether this proposed base class is equal to the specified proposed base class.
operator!=	Inequality operator; tests whether this proposed base class is different from the specified proposed base class.
operator size_t	Conversion operator that tests whether this proposed base class is null.
persistent_capable	Tests whether this proposed base class corresponds to an existing or proposed persistence-capable class.

position	Gets the position of this proposed base class within the proposed class that contains it.
previous_name	Gets the name of the former base class that this proposed base class replaces.

Operators

operator==

Equality operator; tests whether this proposed base class is equal to the specified proposed base class.

```
virtual int operator==(const Proposed_Base_Class &other) const;
```

Parameters

other

The proposed base class with which to compare this proposed base class.

Returns

Nonzero if this proposed base class and *other* are the same object in memory; otherwise, zero.

See also

operator!=

operator!=

Inequality operator; tests whether this proposed base class is different from the specified proposed base class.

```
virtual int operator!=(const Proposed_Base_Class &other) const;
```

Parameters

other

The proposed base class with which to compare this proposed base class.

Returns

Nonzero if this proposed base class and *other* are different objects in memory; otherwise, zero.

See also

operator size_t

operator size_t

Conversion operator that tests whether this proposed base class is null.

```
virtual operator size_t() const;
```

Returns

Zero if this proposed base class is null; otherwise, nonzero.

Discussion Any member function that looks up a proposed base class returns a proposed base class object; unsuccessful searches return a null descriptor. This operator allows you to use a proposed base class as an integer expression to test whether that proposed base class is valid (not null).

Member Functions

access_kind

Gets the access kind of this proposed base class.

```
d_Access_Kind access_kind() const;
```

Returns The visibility or access kind of this proposed base class; one of the following:

- `d_PUBLIC` indicates public access.
- `d_PROTECTED` indicates protected access.
- `d_PRIVATE` indicates private access.

See also `change_access`

change_access

Changes the access kind of this proposed base class.

```
ooStatus change_access(d_Access_Kind newAccess);
```

Parameters *`newAccess`*

The new visibility or access kind of this proposed base class; one of the following:

- `d_PUBLIC` indicates public access.
- `d_PROTECTED` indicates protected access.
- `d_PRIVATE` indicates private access.

See also `access_kind`

defined_in_class

Gets the proposed class whose description contains this proposed base class.

```
Proposed_Class  &defined_in_class() const;
```

Returns The proposed class whose description contains this proposed base class.

persistent_capable

Tests whether this proposed base class corresponds to an existing or proposed persistence-capable class.

```
d_Boolean persistent_capable() const;
```

Returns `oocTrue` if this proposed base class corresponds to an existing or proposed persistence-capable class; `oocFalse` if it corresponds to a class that has not yet been proposed or to an existing or proposed non-persistence-capable class.

position

Gets the position of this proposed base class within the proposed class that contains it.

```
size_t position() const;
```

Returns The attribute position of this proposed base class within the physical layout of the proposed class whose description contains this proposed base class.

previous_name

Gets the name of the former base class that this proposed base class replaces.

```
const char *previous_name() const;
```

Returns The previous name of this proposed base class.

Discussion A proposed base class has a previous name if it was added to the description of its proposed class by a call to the `change_base_class` member function of the containing proposed class. In that case, the previous name is the name of the proposed base class that was replaced by this proposed base class.

Proposed_Basic_Attribute Class

Inheritance: **d_Meta_Object->Proposed_Property->Proposed_Attribute ->Proposed_Basic_Attribute**

The class `Proposed_Basic_Attribute` represents the basic numeric attributes of a proposed class. An instance of this class is called a *proposed numeric attribute*.

See:

- "Reference Summary" on page 344 for an overview of member functions
- "Reference Index" on page 344 for a list of member functions

About Proposed Numeric Attributes

A proposed numeric attribute provides information about a particular numeric attribute within the description of a particular proposed class. A numeric attribute contains either a single numeric value or a fixed-size array of numeric values; the data type of the attribute can be any fundamental character, integer, floating-point, or pointer type.

You should never instantiate this class directly. Instead, you can obtain a proposed numeric attribute just as you would obtain any kind of proposed property.

- Call the `resolve_property` member function of the proposed class to look up the proposed numeric attribute by name.
- Call the `defines_property_begin` member function of the proposed class to get an iterator for all proposed properties of the proposed class.

If you need to call member functions defined in this class, you must cast the resulting proposed property to `Proposed_Basic_Attribute`.

Reference Summary

Getting Information About the Proposed Numeric Attribute	base_type default_value
Testing the Proposed Numeric Attribute	has_default_value
Modifying the Proposed Numeric Attribute	change_base_type

Reference Index

Member Functions

base_type

Gets the numeric type of this proposed numeric attribute.

```
ooBaseType base_type() const;
```

Returns A code identifying the numeric type; one of:

- ooCHAR indicates an 8-bit character.
- ooINT8 indicates an 8-bit signed integer.
- ooINT16 indicates a 16-bit signed integer.
- ooINT32 indicates a 32-bit signed integer.
- ooINT64 indicates a 64-bit signed integer.
- ooUINT8 indicates an 8-bit unsigned integer.

- ooUINT16 indicates a 16-bit unsigned integer.
- ooUINT32 indicates a 32-bit unsigned integer.
- ooUINT64 indicates a 64-bit unsigned integer.
- ooFLOAT32 indicates a 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates a 64-bit (double-precision) floating-point number.
- ooPTR indicates a 32-bit pointer.

See also change_base_type

change_base_type

Changes the numeric type of this proposed numeric attribute.

```
ooStatus change_base_type(ooBaseType newType);
```

Parameters *newType*

The new type for this proposed numeric attribute; one of:

- ooCHAR indicates an 8-bit character.
- ooINT8 indicates an 8-bit signed integer.
- ooINT16 indicates a 16-bit signed integer.
- ooINT32 indicates a 32-bit signed integer.
- ooINT64 indicates a 64-bit signed integer.
- ooUINT8 indicates an 8-bit unsigned integer.
- ooUINT16 indicates a 16-bit unsigned integer.
- ooUINT32 indicates a 32-bit unsigned integer.
- ooUINT64 indicates a 64-bit unsigned integer.
- ooFLOAT32 indicates a 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates a 64-bit (double-precision) floating-point number.
- ooPTR indicates a 32-bit pointer.

Returns ocSuccess if successful; otherwise ocError.

See also base_type

default_value

Gets the default value of this proposed numeric attribute.

```
Numeric_Value default_value() const;
```

Returns The default value of this proposed numeric attribute.

See also has_default_value

has_default_value

Tests whether this proposed numeric attribute has a default value.

```
ooBoolean has_default_value() const;
```

Returns `oocTrue` if this proposed numeric attribute has a default value; otherwise, `oocFalse`.

See also default_value

is_basic_type

Overrides the inherited member function. Indicates that this is a proposed numeric attribute.

```
virtual ooBoolean is_basic_type() const;
```

Returns `oocTrue`.

Proposed_Class Class

Inheritance: **d_Meta_Object->Proposed_Class**

The class `Proposed_Class` represents proposals to add new class descriptions to the schema or to evolve the existing definition of a particular version of a particular class. An instance of `Proposed_Class` is called a *proposed class*.

See:

- "Reference Summary" on page 348 for an overview of member functions
- "Reference Index" on page 349 for a list of member functions

About Proposed Classes

A proposed class provides information about the proposed class description and allows you to modify that description.

You should never instantiate this class directly. Instead, you create a proposed class by calling a member function of a module descriptor:

- To propose a new class to be added to the schema, call the `propose_new_class` member function of the module descriptor for the module in which you want to define a new class.
- To evolve an existing class, call the `propose_evolved_class` member function of the module descriptor for the module in which the class is defined.

In either case, the new proposed class is added to the module descriptor's proposal list. A module descriptor allows you to look up or iterate through the proposed classes in its proposal list:

- Call the `resolve_proposed_class` member function of the module descriptor to look up the proposed class by name.

- Call the proposed_classes_begin member function of the module descriptor to get an iterator for all proposed classes in the module's proposal list.

Reference Summary

Getting Information About the Proposed Class	number_of_attribute_positions number_of_base_classes position_in_class proposed_in_module previous_name specified_shape_number
Testing the Proposed Class	operator== operator!= persistent_capable has_added_virtual_table
Modifying the Proposed Class	rename add_base_class change_base_class delete_base_class move_base_class add_basic_attribute add_ref_attribute add_embedded_class_attribute add_varray_attribute add_unidirectional_relationship add_bidirectional_relationship add_property delete_property move_property add_virtual_table
Testing for the Null Descriptor	operator size_t
Getting Descriptors from the Proposed Class	resolve_base_class base_class_list_begin resolve_property defines_property_begin proposed_in_module

Reference Index

has_added_virtual_table	Tests whether storage for a virtual-table pointer has been added to this proposed class.
move_base_class	Moves a proposed base class of this proposed class.
move_property	Moves a proposed property of this proposed class.
number_of_attribute_positions	Gets the number of attribute positions in the storage layout for this proposed class.
number_of_base_classes	Gets the number of immediate base classes of this proposed class.
operator==	Equality operator; tests whether this proposed class is equal to the specified proposed class.
operator!=	Inequality operator; tests whether this proposed class is different from the specified proposed class.
operator size_t	Conversion operator that tests whether this proposed class is null.
persistent_capable	Tests whether this proposed class is persistence-capable.
position_in_class	Gets the class position of the specified attribute within this proposed class.
previous_name	Gets the previous name of this proposed class.
proposed_in_module	Gets the module descriptor whose proposal list contains this proposed class.
rename	Renames this proposed class.
resolve_base_class	Looks up a proposed base class of this proposed class.
resolve_property	Looks up a proposed property of this proposed class.
specified_shape_number	Gets the shape number specified for this proposed class when it was created.

Constructors

Proposed_Class

Constructs a proposed class.

```
Proposed_Class(
     const char *name,
     ooTypeNumber tnum);
```

Parameters

name

The name of the new class.

tnum

The type number for the new class. Specify 0 if you want the new class to be assigned the next available type number.

If you need to recreate another schema exactly, specify the type number of the class that you are recreating.

Operators

operator==

Equality operator; tests whether this proposed class is equal to the specified proposed class.

```
virtual int operator==(const Proposed_Class &other) const;
```

Parameters

other

The proposed class with which to compare this proposed class.

Returns

Nonzero if this proposed class and *other* are the same object in memory; otherwise, zero.

See also

operator!=

operator!=

Inequality operator; tests whether this proposed class is different from the specified proposed class.

```
virtual int operator!=(const Proposed_Class &other) const;
```

Parameters *other*

The proposed class with which to compare this proposed class.

Returns Nonzero if this proposed class and *other* are different objects in memory; otherwise, zero.

See also operator==

operator size_t

Conversion operator that tests whether this proposed class is null.

```
virtual operator size_t() const;
```

Returns Zero if this proposed class is null; otherwise, nonzero.

Discussion Any member function that looks up a proposed class returns a proposed class object; unsuccessful searches return a null descriptor. This operator allows you to use a proposed class as an integer expression to test whether that proposed class is valid (not null).

Member Functions

add_base_class

Adds a base class to this proposed class.

```
ooStatus add_base_class(
     int32 position,
     d_Access_Kind visibility,
     const char *name);
```

Parameters *position*

The desired attribute position of the new base class within the physical layout of this proposed class, or `oocLast` to position the base class after all currently proposed base classes, attributes, and relationships of this proposed class. Because base classes must come before attributes and relationships, you should specify `oocLast` only if this proposed class is a new class to which you have not added any attributes or relationships.

visibility

The visibility or access kind for the base class; one of the following:

- `d_PUBLIC` indicates public access.
- `d_PROTECTED` indicates protected access.

- d_PRIVATE indicates private access.

name

The name of the base class, which can be an existing class in the schema, a proposed class, or a new class that will be proposed before proposals are activated for this proposed class's module.

If *name* is the name of this proposed class, this member function throws an InheritsFromSelfError exception.

Returns oocSuccess if successful; otherwise oocError.

See also change_base_class
delete_base_class
move_base_class

add_basic_attribute

Adds a numeric attribute to this proposed class.

```
1.  ooStatus add_basic_attribute(
      int32 position,
      d_Access_Kind visibility,
      const char *name,
      size_t arraySize,
      ooNumberType btype);
2.  ooStatus add_basic_attribute(
      int32 position,
      d_Access_Kind visibility,
      const char *name,
      size_t arraySize,
      ooIntegerType btype,
      int64 defaultVal);
3.  ooStatus add_basic_attribute(
      int32 position,
      d_Access_Kind visibility,
      const char *name,
      size_t arraySize,
      ooUINT64_t btype,
      uint64 defaultVal);
```

```
4.  ooStatus add_basic_attribute(
      int32 position,
      d_Access_Kind visibility,
      const char *name,
      size_t arraySize,
      ooFloatType btype,
      float64 defaultVal);

5.  ooStatus add_basic_attribute(
      int32 position,
      d_Access_Kind visibility,
      const char *name,
      size_t arraySize,
      ooPTR_t btype,
      void *defaultVal);
```

Parameters

position

The desired attribute position of the new attribute within the physical layout of this proposed class, or `oocLast` to position the attribute after all currently proposed base classes, attributes, and relationships of this proposed class.

visibility

The visibility or access kind for the attribute; one of the following:

- `d_PUBLIC` indicates public access.
- `d_PROTECTED` indicates protected access.
- `d_PRIVATE` indicates private access.

name

The name of the attribute.

arraySize

The number of elements in the fixed-size array of values for the new attribute, or 1 for an attribute that stores a single numeric value.

btype

The kind of numeric data in the attribute; one of the following:

- `ooCHAR` indicates 8-bit character.
- `ooINT8` indicates 8-bit signed integer.
- `ooINT16` indicates 18-bit signed integer.
- `ooINT32` indicates 32-bit signed integer.
- `ooINT64` indicates 64-bit signed integer.
- `ooUINT8` indicates 8-bit unsigned integer.
- `ooUINT16` indicates 18-bit unsigned integer.
- `ooUINT32` indicates 32-bit unsigned integer.

- ooUINT64 indicates 64-bit unsigned integer.
- ooFLOAT32 indicates 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates 64-bit (double-precision) floating-point number.
- ooPTR indicates 32-bit pointer.

defaultVal

The default value for existing instances of the class that are converted to the new shape described by this proposed class.

Returns oocSuccess if successful; otherwise oocError.

Discussion Variants 2 through 5 ensure that the default value *defaultVal* is a legal value for the numeric type indicated by the *btype* parameter. You should use one of those variants only if this proposed class is an evolved shape of an existing class. When existing objects of the class are converted to the evolved shape, they are given the specified default value for the new attribute being added. If you specify a default value for a proposed new class, this member function throws a DefaultValueForUnevolvedClass exception.

See also add_embedded_class_attribute
add_ref_attribute
add_varray_attribute

add_bidirectional_relationship

Adds a bidirectional relationship to this proposed class.

```
ooStatus add_bidirectional_relationship(
     int32 position,
     d_Access_Kind visibility,
     const char *name,
     const char *referencedClassName,
     ooBoolean isInline,
     ooBoolean isShort,
     ooBoolean isToMany,
     uint8 copyMode,
     uint8 versioning,
     uint8 propagation,
     const char *inverseName,
     ooBoolean inverseIsToMany,
     ooAssocNumber specifiedAssocNum = 0);
```

Parameters *position*

The desired attribute position of the new relationship within the physical layout of this proposed class, or oocLast to position the relationship after all

currently proposed base classes, attributes, and relationships of this proposed class.

visibility

The visibility or access kind for the relationship; one of the following:

- d_PUBLIC indicates public access.
- d_PROTECTED indicates protected access.
- d_PRIVATE indicates private access.

name

The name of the relationship.

referencedClassName

The name of the destination class for the new relationship.

isInline

oocTrue if the new relationship is to be inline; otherwise, false.

isShort

oocTrue if the new relationship is to store references to related objects with short object references; otherwise, false.

isToMany

oocTrue if the new relationship is to be a to-many relationship; otherwise, false.

copyMode

The copy mode of the new relationship, which specifies what happens to an association from a source object to a destination object when the source object is copied; one of the following:

- 0 indicates that the association is deleted.
- oocCopyDrop indicates that the association is deleted.
- oocCopyMove indicates that the association is moved from the source object to its new copy.
- oocCopyCopy indicates that the association is copied from the source object to the new object.

versioning

The versioning mode of the new relationship, which specifies what happens to an association from a source object to a destination object when a new version of the source object is created; one of the following:

- 0 indicates that the association is deleted.
- oocVersionDrop indicates that the association is deleted.
- oocVersionMove indicates that the association is moved from the source object to its new version.

- oocVersionCopy indicates that the association is copied from the source object to its new version.

propagation

The propagation behavior of the new relationship, which specifies whether the locking and deletion operations are propagated from locked or deleted source objects to their related destination objects; one of the following:

- 0 indicates that neither locks or deletions are propagated.
- oocLockPropagationYesDeletePropagationNo indicates that locks are propagated, but deletions are not.
- oocLockPropagationNoDeletePropagationYes indicates that deletions are propagated, but locks are not.
- oocLockPropagationYesDeletePropagationYes indicates that both locks and deletions are propagated.

inverseName

The name of the inverse relationship.

inverseIsToMany

oocTrue if the inverse relationship is a to-many relationship; otherwise, false.

specifiedAssocNum

The association number encoding characteristics of the new relationship.

Returns

oocSuccess if successful; otherwise oocError.

Discussion

The bidirectional relationships in the schema are assigned serially-allocated 32-bit integers, called their *encoded association numbers*. Certain high-order bits of an encoded association number are set to encrypt the relationship's direction and other characteristics.

Most applications do not need to work with encoded association numbers. However, if you need to exactly recreate another schema description, you can call the encoded_assoc_number member function of a relationship descriptor for a bidirectional relationship that you want to recreate. You can pass the resulting number as the *specifiedAssocNum* parameter to this member function.

See also

add_unidirectional_relationship

add_embedded_class_attribute

Adds an embedded-class attribute to this proposed class.

```
ooStatus add_embedded_class_attribute(
     int32 position,
     d_Access_Kind visibility,
     const char *name,
     size_t arraySize,
     const char *otherClassName);
```

Parameters

position

The desired attribute position of the new attribute within the physical layout of this proposed class, or `oocLast` to position the attribute after all currently proposed base classes, attributes, and relationships of this proposed class.

visibility

The visibility or access kind for the attribute; one of the following:

- `d_PUBLIC` indicates public access.
- `d_PROTECTED` indicates protected access.
- `d_PRIVATE` indicates private access.

name

The name of the attribute.

arraySize

The number of elements in the fixed-size array of values for the new attribute, or 1 for an attribute that stores a single embedded object.

otherClassName

The name of the embedded non-persistence-capable class, which can be an existing class in the schema, a proposed class, or a new class that will be proposed before proposals are activated for this proposed class's module.

Returns

`oocSuccess` if successful; otherwise `oocError`.

See also

`add_basic_attribute`
`add_ref_attribute`
`add_varray_attribute`

add_property

Adds the specified property to this proposed class.

```
ooStatus add_property(
     int32 position,
     d_Access_Kind visibility,
     const d_Property &newProperty);
```

Parameters

position

The desired attribute position of the new property within the physical layout of this proposed class, or `oocLast` to position the property after all currently proposed base classes, attributes, and relationships of this proposed class.

visibility

The visibility or access kind for the attribute; one of the following:

- `d_PUBLIC` indicates public access.
- `d_PROTECTED` indicates protected access.
- `d_PRIVATE` indicates private access.

newProperty

Property descriptor for the property to be added.

Returns

`oocSuccess` if successful; otherwise `oocError`.

Discussion

This member function allows you to add a property whose specification is identical to an existing property of an existing class.

add_ref_attribute

Adds an object-reference attribute to this proposed class.

```
ooStatus add_ref_attribute(
     int32 position,
     d_Access_Kind visibility,
     const char *name,
     size_t arraySize,
     const char *referencedClassName,
     ooBoolean isShort);
```

Parameters

position

The desired attribute position of the new attribute within the physical layout of this proposed class, or `oocLast` to position the attribute after all currently proposed base classes, attributes, and relationships of this proposed class.

visibility

The visibility or access kind for the attribute; one of the following:

- d_PUBLIC indicates public access.
- d_PROTECTED indicates protected access.
- d_PRIVATE indicates private access.

name

The name of the attribute.

arraySize

The number of elements in the fixed-size array of values for the new attribute, or 1 for an attribute that stores a single object reference.

referencedClassName

The name of the referenced persistence-capable class, which can be an existing class in the schema, a proposed class, or a new class that will be proposed before proposals are activated for this proposed class's module.

isShort

ooTrue if the proposed attribute should use short object references and ooFalse if it should use standard object references.

Returns ooSuccess if successful; otherwise ooError.

See also add_basic_attribute
add_embedded_class_attribute
add_varray_attribute

add_unidirectional_relationship

Adds a unidirectional relationship to this proposed class.

```
ooStatus add_unidirectional_relationship(
     int32 position,
     d_Access_Kind visibility,
     const char *name,
     const char *otherClassName,
     ooBoolean isInline,
     ooBoolean isShort,
     ooBoolean isToMany,
     uint8 copyMode,
     uint8 versioning,
     uint8 propagation);
```

Parameters

position

The desired attribute position of the new relationship within the physical layout of this proposed class, or `oocLast` to position the relationship after all currently proposed base classes, attributes, and relationships of this proposed class.

visibility

The visibility or access kind for the relationship; one of the following:

- `d_PUBLIC` indicates public access.
- `d_PROTECTED` indicates protected access.
- `d_PRIVATE` indicates private access.

name

The name of the relationship.

otherClassName

The name of the destination class for the new relationship.

isInline

`oocTrue` if the new relationship is to be inline; otherwise, false.

isShort

`oocTrue` if the new relationship is to store references to related objects with short object references; otherwise, false.

isToMany

`oocTrue` if the new relationship is to be a to-many relationship; otherwise, false.

copyMode

The copy mode of the new relationship, which specifies what happens to an association from a source object to a destination object when the source object is copied; one of the following:

- 0 indicates that the association is deleted.
- `oocCopyDrop` indicates that the association is deleted.
- `oocCopyMove` indicates that the association is moved from the source object to its new copy.
- `oocCopyCopy` indicates that the association is copied from the source object to the new object.

versioning

The versioning mode of the new relationship, which specifies what happens to an association from a source object to a destination object when a new version of the source object is created; one of the following:

- 0 indicates that the association is deleted.
- `oocVersionDrop` indicates that the association is deleted.
- `oocVersionMove` indicates that the association is moved from the source object to its new version.
- `oocVersionCopy` indicates that the association is copied from the source object to its new version.

propagation

The propagation behavior of the new relationship, which specifies whether the locking and deletion operations are propagated from locked or deleted source objects to their related destination objects; one of the following:

- 0 indicates that neither locks or deletions are propagated.
- `oocLockPropagationYesDeletePropagationNo` indicates that locks are propagated, but deletions are not.
- `oocLockPropagationNoDeletePropagationYes` indicates that deletions are propagated, but locks are not.
- `oocLockPropagationYesDeletePropagationYes` indicates that both locks and deletions are propagated.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `add_bidirectional_relationship`

add_varray_attribute

Adds a VArray attribute to this proposed class.

```
1.  ooStatus add_varray_attribute(
      int32 position,
      d_Access_Kind visibility,
      const char *name,
      size_t arraySize,
      const char *embeddedClassName);
2.  ooStatus add_varray_attribute(
      int32 position,
      d_Access_Kind visibility,
      const char *name,
      size_t arraySize,
      ooNumberType btype);
```

3.
```
ooStatus add_varray_attribute(
  int32 position,
  d_Access_Kind visibility,
  const char *name,
  size_t arraySize,
  ooBoolean isShort,
  const char *referencedClassName);
```

Parameters

position

The desired attribute position of the new attribute within the physical layout of this proposed class, or `oocLast` to position the attribute after all currently proposed base classes, attributes, and relationships of this proposed class.

visibility

The visibility or access kind for the attribute; one of the following:

- `d_PUBLIC` indicates public access.
- `d_PROTECTED` indicates protected access.
- `d_PRIVATE` indicates private access.

name

The name of the attribute.

arraySize

The number of elements in the fixed-size array of values for the new attribute, or 1 for an attribute that stores a single VArray.

embeddedClassName

The name of the embedded non-persistence-capable class in the new embedded-class VArray attribute. The embedded class can be an existing class in the schema, a proposed class, or a new class that will be proposed before proposals are activated for this proposed class's module.

btype

The type of numeric elements in the new numeric VArray attribute; one of the following:

- `ooCHAR` indicates 8-bit character.
- `ooINT8` indicates 8-bit signed integer.
- `ooINT16` indicates 18-bit signed integer.
- `ooINT32` indicates 32-bit signed integer.
- `ooINT64` indicates 64-bit signed integer.
- `ooUINT8` indicates 8-bit unsigned integer.
- `ooUINT16` indicates 18-bit unsigned integer.
- `ooUINT32` indicates 32-bit unsigned integer.

- ooUINT64 indicates 64-bit unsigned integer.
- ooFLOAT32 indicates 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates 64-bit (double-precision) floating-point number.

isShort

oocTrue if the new object-reference VArray attribute should use short object references and oocFalse if it should use standard object references.

referencedClassName

The name of the referenced persistence-capable class in the new object-reference VArray attribute. The referenced class can be an existing class in the schema, a proposed class, or a new class that will be proposed before proposals are activated for this proposed class's module.

Returns oocSuccess if successful; otherwise oocError.

Discussion The first variant adds an embedded-class VArray attribute. The second variant adds a numeric VArray attribute. The third variant adds an object-reference VArray attribute.

See also add_basic_attribute
add_embedded_class_attribute
add_ref_attribute

add_virtual_table

Adds space for a virtual table to the storage layout for this proposed class.

```
ooStatus add_virtual_table();
```

Returns oocSuccess if successful; otherwise oocError.

See also has_added_virtual_table

base_class_list_begin

Gets an iterator for the proposed base classes of this proposed class.

```
proposed_base_class_iterator base_class_list_begin() const;
```

Returns A proposed-base-class iterator that finds all proposed base classes of this proposed class.

See also base_class_list_end

base_class_list_end

Gets an iterator representing the termination condition for iteration through the proposed base classes of this proposed class.

```
proposed_base_class_iterator base_class_list_end() const;
```

Returns A proposed-base-class iterator that is positioned after the last proposed base class of this proposed class.

Discussion You can compare the iterator returned by `base_class_list_begin` with the one returned by this member function to test whether iteration has finished.

change_base_class

Replaces a given base class of this proposed class with a different base class.

```
ooStatus change_base_class(
     const char *previousName,
     const char *name);
```

Parameters *previousName*

The name of the current base class to be replaced.

name

The name of the new base class that is to replace *previousName.*

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function is particularly useful if all of the following conditions are true:

- This proposed class is an evolved shape for an existing class
- The class *name* is an ancestor class of the class *previousName*.
- Data for properties inherited from *name* are to be preserved when objects of the evolved class are converted to the new shape.

Calling this member function is analogous to using an `oochangebase` pragma in a DDL file. See the Objectivity / C++ Data Definition Language book.

See also `add_base_class`
`delete_base_class`
`move_base_class`

defines_property_begin

Gets an iterator for the proposed properties of this proposed class.

```
proposed_property_iterator defines_property_begin() const;
```

Returns A proposed-property iterator that finds all proposed properties of this proposed class.

See also `defines_property_end`

defines_property_end

Gets an iterator representing the termination condition for iteration through the proposed properties of this proposed class.

```
proposed_property_iterator defines_property_end() const;
```

Returns A proposed-property iterator that is positioned after the last proposed property of this proposed class.

Discussion You can compare the iterator returned by `defines_property_begin` with the one returned by this member function to test whether iteration has finished.

delete_base_class

Deletes a proposed base class of this proposed class.

```
ooStatus delete_base_class(const char *baseClassName);
```

Parameters *`baseClassName`*

The name of the proposed base class to be deleted.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `add_base_class`
`change_base_class`
`move_base_class`

delete_property

Deletes a proposed property of this proposed class.

```
ooStatus delete_property(const char *attName);
```

Parameters *`attName`*

The name of the proposed attribute or relationship to be deleted.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `move_property`

has_added_virtual_table

Tests whether storage for a virtual-table pointer has been added to this proposed class.

```
ooBoolean has_added_virtual_table() const;
```

Returns `oocTrue` if storage for a virtual-table pointer has been added to this proposed class; otherwise, `oocFalse`.

See also `add_virtual_table`

move_base_class

Moves a proposed base class of this proposed class.

```
ooStatus move_base_class(const char *bcName, size_t newPos);
```

Parameters

bcName

The name of the proposed base class to be moved.

newPos

The new attribute position of the base class within the physical layout of this proposed class, or `oocLast` to position the base class after all currently proposed base classes, attributes, and relationships of this proposed class.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `add_base_class`
`change_base_class`
`delete_base_class`

move_property

Moves a proposed property of this proposed class.

```
ooStatus move_property(const char *attName, size_t newPos);
```

Parameters

attName

The name of the proposed attribute or relationship to be moved.

newPos

The new attribute position of the property within the physical layout of this proposed class, or `oocLast` to position the property after all currently proposed base classes, attributes, and relationships of this proposed class.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `delete_property`

number_of_attribute_positions

Gets the number of attribute positions in the storage layout for this proposed class.

```
size_t number_of_attribute_positions() const;
```

Returns The number of attribute positions in the physical layout for this proposed class, namely, the total number of its immediate base classes, attributes, and relationships.

Discussion The returned number does not include inherited attributes.

number_of_base_classes

Gets the number of immediate base classes of this proposed class.

```
size_t number_of_base_classes() const;
```

Returns The number of proposed immediate base classes in this proposed class.

persistent_capable

Tests whether this proposed class is persistence-capable.

```
d_Boolean persistent_capable() const;
```

Returns `oocTrue` if this proposed class is persistence-capable; otherwise, `oocFalse`.

position_in_class

Gets the class position of the specified attribute within this proposed class.

```
const Class_Position position_in_class(
     const char *name) const;
```

Parameters *name*

The name of the attribute whose position is desired. This string can be a qualified name to disambiguate attributes of the same name inherited from different base classes. You should specify a qualified name only if necessary, because it takes more time to look up a qualified name than an unqualified one.

Returns A class position that gives the layout position of the specified attribute within the described class.

Discussion You can use this member function to find the class position of any of the following:

- Any attribute or relationship defined in this proposed class.
- Any attribute or relationship inherited by this proposed class.
- An immediate base class of this proposed class.
- An ancestor class at any level in the inheritance graph for this proposed class.

previous_name

Gets the previous name of this proposed class.

```
const char *previous_name() const;
```

Returns The previous name of this proposed class, or null if this proposed class has never been renamed.

Discussion If this proposed class has been renamed more than once, this member function gets the name before the current name; there is no way to get any earlier names.

See also rename

proposed_in_module

Gets the module descriptor whose proposal list contains this proposed class.

```
const d_Module  &proposed_in_module() const;
```

Returns The module descriptor whose proposal list contains this proposed class.

rename

Renames this proposed class.

```
ooStatus rename(const char *newName);
```

Parameters *newName*

The new name for this proposed class.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `previous_name`

resolve_base_class

Looks up a proposed base class of this proposed class.

```
Proposed_Base_Class  &resolve_base_class(
     const char *nameToMatch);
```

Parameters *nameToMatch*

The name of the proposed base class to be looked up.

Returns The proposed base class of this proposed class with the specified name, or the null descriptor if *nameToMatch* is not the name of a proposed base class of this proposed class.

See also `resolve_property`

resolve_property

Looks up a proposed property of this proposed class.

```
Proposed_Property  &resolve_property(const char *nameToMatch);
```

Parameters *nameToMatch*

The name of the proposed property to be looked up.

Returns The proposed property of this proposed class with the specified name, or the null descriptor if *nameToMatch* is not the name of a proposed property of this proposed class.

See also `resolve_base_class`

specified_shape_number

Gets the shape number specified for this proposed class when it was created.

```
ooTypeNumber specified_shape_number() const;
```

Returns The shape number specified for this proposed class when it was created, or zero if no shape number was specified.

Discussion Typically, a shape number is specified for a proposed class only by an application that needs to recreate an existing schema exactly.

Proposed_Collection_Attribute Class

Inheritance: **d_Meta_Object->Proposed_Property->Proposed_Attribute ->Proposed_Collection_Attribute**

The class `Proposed_Collection_Attribute` is the abstract base class for descriptors of the collection attributes of a proposed class. An instance of any concrete derived class, called a *proposed collection attribute*, provides information about a particular collection attribute within the description of a particular proposed class.

Currently Objectivity / DB supports only one kind of collection attribute, namely VArray attributes, which contain variable-size arrays of elements of the same type. The derived class `Proposed_VArray_Attribute` represents this kind of proposed collection attribute.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

Member Functions

kind

Gets the ODMG collection kind of this proposed collection attribute.

```
virtual d_Kind kind() const = 0;
```

Returns — The ODMG collection kind of this proposed collection attribute.

Discussion — The only ODMG collection kind that Objectivity / DB supports is variable-size arrays of elements of the same type.

Proposed_Embedded_Class_Attribute Class

Inheritance: **d_Meta_Object->Proposed_Property->Proposed_Attribute ->Proposed_Embedded_Class_Attribute**

The class `Proposed_Embedded_Class_Attribute` represents the embedded-class attributes of a proposed class. An instance of this class is called a *proposed embedded-class attribute*.

See:

- "Reference Summary" on page 376 for an overview of member functions
- "Reference Index" on page 376 for a list of member functions

About Proposed Embedded-Class Attributes

A proposed embedded-class attribute provides information about a particular embedded-class attribute within the description of a particular proposed class. An embedded-class attribute contains either a single embedded instance or a fixed-size array of embedded instances. The embedded instances are instances of a particular non-persistence-capable embedded class.

You should never instantiate this class directly. Instead, you can obtain a proposed embedded-class attribute just as you would obtain any kind of proposed property.

- Call the `resolve_property` member function of the proposed class to look up the proposed embedded-class attribute by name.
- Call the `defines_property_begin` member function of the proposed class to get an iterator for all proposed properties of the proposed class.

If you need to call member functions defined in this class, you must cast the resulting proposed property to `Proposed_Embedded_Class_Attribute`.

Reference Summary

Getting Information About the Proposed Embedded-Class Attribute	embedded_class_name
Modifying the Proposed Embedded-Class Attribute	change_embedded_class

Reference Index

Member Functions

change_embedded_class

Changes the embedded class in this proposed embedded-class attribute.

```
ooStatus change_embedded_class(const char *embClass);
```

Parameters

embClass

The name of the embedded class in this embedded-class attribute.

Returns

`oocSuccess` if successful; otherwise `oocError`.

See also

embedded_class_name

embedded_class_name

Gets the name of the embedded class in this proposed embedded-class attribute.

```
const char  *embedded_class_name() const;
```

Returns The name of the embedded class in this embedded-class attribute.

See also change_embedded_class

is_embedded_class_type

Overrides the inherited member function; indicates that this is a proposed embedded-class attribute.

```
virtual ooBoolean is_embedded_class_type() const;
```

Returns `oocTrue`.

Proposed_Property Class

Inheritance: **d_Meta_Object->Proposed_Property**

The class `Proposed_Property` is the abstract base class for descriptors of the properties of a proposed class. An instance of any concrete derived class is called a *proposed property.*

See:

- "Reference Summary" on page 380 for an overview of member functions
- "Reference Index" on page 380 for a list of member functions

About Proposed Properties

A proposed property provides information about a particular property within the description of a particular proposed class.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

You can obtain a proposed property from the proposed class whose description contains the proposed property.

- Call the `resolve_property` member function of the proposed class to look up the proposed property by name.
- Call the `defines_property_begin` member function of the proposed class to get an iterator for all proposed properties of the proposed class.

Reference Summary

Getting Information About the Proposed Property	defined_in_class position array_size access_kind previous_name
Testing the Proposed Property	operator== operator!= is_basic_type is_ref_type is_embedded_class_type is_varray_type is_varray_basic_type is_varray_ref_type is_varray_embedded_class_type is_relationship_type
Testing for the Null Descriptor	operator_size_t
Modifying the Proposed Property	rename change_access
Getting Descriptors from the Proposed Property	defined_in_class

Reference Index

is_embedded_class_type	Tests whether this proposed property is an embedded-class attribute.
is_ref_type	Tests whether this proposed property is an object-reference attribute.
is_relationship_type	Tests whether this proposed property is a relationship.
is_varray_basic_type	Tests whether this proposed property is a variable-size array attribute with numeric elements.
is_varray_embedded_class_type	Tests whether this proposed property is a variable-size array attribute whose elements are instances of some non-persistence-capable class.
is_varray_ref_type	Tests whether this proposed property is a variable-size array attribute whose elements are object-references to instances of some persistence-capable class.
is_varray_type	Tests whether this proposed property is a variable-size array attribute.
operator==	Equality operator; tests whether this proposed property is equal to the specified proposed property.
operator!=	inequality operator; tests whether this proposed property is different from the specified proposed property.
operator_size_t	Conversion operator that tests whether this proposed property is null.
position	Gets the position of this proposed property within the proposed class that contains it.
previous_name	Gets the previous name of this proposed property.
rename	Renames this proposed property.

Operators

operator==

Equality operator; tests whether this proposed property is equal to the specified proposed property.

```
virtual int operator==(const Proposed_Property &other) const;
```

Parameters *other*

The proposed property with which to compare this proposed property.

Returns Nonzero if this proposed property and *other* are the same object in memory; otherwise, zero.

See also operator!=

operator!=

inequality operator; tests whether this proposed property is different from the specified proposed property.

```
virtual int operator!=(const Proposed_Property &other) const;
```

Parameters *other*

The proposed property with which to compare this proposed property.

Returns Nonzero if this proposed property and *other* are different objects in memory; otherwise, zero.

See also operator==

operator size_t

Conversion operator that tests whether this proposed property is null.

```
virtual operator size_t() const;
```

Returns Zero if this proposed property is null; otherwise, nonzero.

Discussion Any member function that looks up a proposed base class returns a proposed property object; unsuccessful searches return a null descriptor. This operator allows you to use a proposed property as an integer expression to test whether that proposed property is valid (not null).

Member Functions

access_kind

Gets the access kind of this proposed property.

```
d_Access_Kind access_kind() const;
```

Returns The visibility or access kind of this proposed property; one of the following:

- d_PUBLIC indicates public access.
- d_PROTECTED indicates protected access.
- d_PRIVATE indicates private access.

See also change_access

array_size

Gets the array size for this proposed property.

```
size_t array_size() const;
```

Returns If this proposed property is a proposed attribute, this member function returns the number of elements in the fixed-size array of values for this proposed attribute (or one if the attribute is to contain a single value instead of an array).

If this proposed property is a proposed relationship, this member function returns one.

change_access

Changes the access kind of this proposed property.

```
ooStatus change_access(d_Access_Kind newAccess);
```

Parameters *newAccess*

The new visibility or access kind of this proposed property; one of the following:

- d_PUBLIC indicates public access.
- d_PROTECTED indicates protected access.
- d_PRIVATE indicates private access.

See also access_kind

defined_in_class

Gets the proposed class whose description contains this proposed property.

```
Proposed_Class  &defined_in_class() const;
```

Returns The proposed class whose description contains this proposed property.

is_basic_type

Tests whether this proposed property is a basic numeric attribute.

```
virtual ooBoolean is_basic_type() const;
```

Returns `oocTrue` if this proposed property is a proposed numeric attribute; otherwise, `oocFalse`.

is_embedded_class_type

Tests whether this proposed property is an embedded-class attribute.

```
virtual ooBoolean is_embedded_class_type() const;
```

Returns `oocTrue` if this proposed property is a proposed embedded-class attribute; otherwise, `oocFalse`.

is_ref_type

Tests whether this proposed property is an object-reference attribute.

```
virtual ooBoolean is_ref_type() const;
```

Returns `oocTrue` if this proposed property is a proposed object-reference attribute; otherwise, `oocFalse`.

is_relationship_type

Tests whether this proposed property is a relationship.

```
virtual ooBoolean is_relationship_type() const;
```

Returns `oocTrue` if this proposed property is a proposed relationship; otherwise, `oocFalse`.

is_varray_basic_type

Tests whether this proposed property is a variable-size array attribute with numeric elements.

```
virtual ooBoolean is_varray_basic_type() const;
```

Returns `oocTrue` if this proposed property is a proposed numeric VArray attribute; otherwise, `oocFalse`.

is_varray_embedded_class_type

Tests whether this proposed property is a variable-size array attribute whose elements are instances of some non-persistence-capable class.

```
virtual ooBoolean is_varray_embedded_class_type() const;
```

Returns `oocTrue` if this proposed property is a proposed embedded-class VArray attribute; otherwise, `oocFalse`.

is_varray_ref_type

Tests whether this proposed property is a variable-size array attribute whose elements are object-references to instances of some persistence-capable class.

```
virtual ooBoolean is_varray_ref_type() const;
```

Returns `oocTrue` if this proposed property is a proposed object-reference VArray attribute; otherwise, `oocFalse`.

is_varray_type

Tests whether this proposed property is a variable-size array attribute.

```
virtual ooBoolean is_varray_type() const;
```

Returns `oocTrue` if this proposed property is a proposed VArray attribute; otherwise, `oocFalse`.

position

Gets the position of this proposed property within the proposed class that contains it.

```
size_t position() const;
```

Returns The attribute position of this proposed property within the physical layout of the proposed class whose description contains this proposed property.

previous_name

Gets the previous name of this proposed property.

```
const char  *previous_name() const;
```

Returns — The previous name of this proposed property, or null if this proposed property has never been renamed.

Discussion — If this proposed property has been renamed more than once, this member function gets the name before the current name; there is no way to get any earlier names.

See also — rename

rename

Renames this proposed property.

```
ooStatus rename(const char *newName);
```

Parameters

newName

The new name for this proposed property.

Returns — `oocSuccess` if successful; otherwise `oocError`.

See also — previous_name

Proposed_Ref_Attribute Class

Inheritance: **d_Meta_Object->Proposed_Property->Proposed_Attribute ->Proposed_Ref_Attribute**

The class `Proposed_Ref_Attribute` represents the object-reference attributes of a proposed class. An instance of this class is called a *proposed object-reference attribute*.

See:

- "Reference Summary" on page 388 for an overview of member functions
- "Reference Index" on page 388 for a list of member functions

About Proposed Object-Reference Attributes

A proposed object-reference attribute provides information about a particular object-reference attribute within the description of a particular proposed class. An *object-reference attribute* contains either a single object reference or a fixed-size array of object references. The referenced objects are instances of a particular persistence-capable referenced class.

You should never instantiate this class directly. Instead, you can obtain a proposed object-reference attribute just as you would obtain any kind of proposed property.

- Call the `resolve_property` member function of the proposed class to look up the proposed object-reference attribute by name.
- Call the `defines_property_begin` member function of the proposed class to get an iterator for all proposed properties of the proposed class.

If you need to call member functions defined in this class, you must cast the resulting proposed property to `Proposed_Ref_Attribute`.

Reference Summary

Getting Information About the Proposed Object-Reference Attribute	referenced_class_name
Testing the Proposed Object-Reference Attribute	is_short
Modifying the Proposed Object-Reference Attribute	change_referenced_class change_short

Reference Index

Member Functions

change_referenced_class

Changes the name of the referenced class for this proposed object-reference attribute.

```
ooStatus change_referenced_class(const char *refClassName);
```

Parameters *refClassName*

The name of the referenced class for this proposed object-reference attribute.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `referenced_class_name`

change_short

Changes the reference type for this proposed object-reference attribute.

```
ooStatus change_short(ooBoolean isShort);
```

Parameters *`isShort`*

`oocTrue` if this proposed attribute should use short object references and `oocFalse` if it should use standard object references.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `is_short`

is_ref_type

Overrides the inherited member function. Indicates that this is a proposed object-reference attribute.

```
virtual ooBoolean is_ref_type() const;
```

Returns `oocTrue`.

is_short

Gets the reference type for this proposed object-reference attribute.

```
ooBoolean is_short() const;
```

Returns `oocTrue` if the proposed attribute uses short object references and `oocFalse` if it uses standard object references.

See also `change_short`

referenced_class_name

Gets the name of the referenced class for this proposed object-reference attribute.

```
const char  *referenced_class_name() const;
```

Returns The name of the referenced class for this proposed object-reference attribute.

See also `change_referenced_class`

Proposed_Relationship Class

Inheritance: **d_Meta_Object->Proposed_Property->Proposed_Relationship**

The class `Proposed_Relationship` represents the relationships of a proposed class. An instance of this class is called a *proposed relationship*.

See:

- "Reference Summary" on page 392 for an overview of member functions
- "Reference Index" on page 392 for a list of member functions

About Proposed Relationships

A proposed relationship provides information about a particular relationship within the description of a particular proposed class.

You should never instantiate this class directly. Instead, you can obtain a proposed relationship just as you would obtain any kind of proposed property.

- Call the `resolve_property` member function of the proposed class to look up the proposed relationship by name.
- Call the `defines_property_begin` member function of the proposed class to get an iterator for all proposed properties of the proposed class.

If you need to call member functions defined in this class, you must cast the resulting proposed property to `Proposed_Relationship`.

Reference Summary

Getting Information About the Proposed Relationship	related_class_name inverse_name copy_mode versioning propagation specified_assoc_number
Testing the Proposed Relationship	is_to_many is_bidirectional is_inline is_short inverse_is_to_many
Modifying the Proposed Relationship	change_related_class change_to_many change_to_unidirectional change_to_bidirectional change_inverse change_copy_mode change_versioning change_propagation change_inline change_short

Reference Index

change_short	Changes whether this proposed relationship is stored with short OIDs.
change_to_bidirectional	Changes this proposed relationship to a bidirectional relationship.
change_to_many	Changes whether this proposed relationship is to-many.
change_to_unidirectional	Changes this proposed relationship to a unidirectional relationship.
change_versioning	Changes the versioning mode of this proposed relationship.
copy_mode	Gets the copy mode of this proposed relationship.
inverse_is_to_many	Tests whether this proposed relationship is a bidirectional relationship whose inverse is a to-many relationship.
inverse_name	Gets the name of the inverse relationship of this proposed relationship.
is_bidirectional	Tests whether this proposed relationship is bidirectional.
is_inline	Tests whether this proposed relationship is stored inline.
is_relationship_type	Overrides the inherited member function. Indicates that this is a proposed relationship.
is_short	Tests whether this proposed relationship is stored with short OIDs.
is_to_many	Tests whether this proposed relationship is to-many.
propagation	Gets the propagation behavior of this proposed relationship.
related_class_name	Gets the name of the destination class of this proposed relationship.
specified_assoc_number	Gets the type number encoding characteristics of this proposed relationship.
versioning	Gets the versioning mode of this proposed relationship.

Member Functions

change_copy_mode

Changes the copy mode of this proposed relationship.

```
ooStatus change_copy_mode(uint8 newMode);
```

Parameters *newMode*

The new copy mode for this proposed relationship, which specifies what happens to an association from a source object to a destination object when the source object is copied; one of the following:

- 0 indicates that the association is deleted.
- `oocCopyDrop` indicates that the association is deleted.
- `oocCopyMove` indicates that the association is moved from the source object to its new copy.
- `oocCopyCopy` indicates that the association is copied from the source object to the new object.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `copy_mode`

change_inline

Changes whether this proposed relationship is stored inline.

```
ooStatus change_inline(ooBoolean isInline);
```

Parameters *isInline*

`oocTrue` if this proposed relationship is to be stored inline; otherwise, `oocFalse`.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `is_inline`

change_inverse

Changes the inverse relationship of this proposed relationship.

```
ooStatus change_inverse(const char  *inverse);
```

Parameters *inverse*

The name of the new inverse relationship for this proposed bidirectional relationship.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function returns `oocError` if this is a proposed unidirectional relationship.

See also `inverse_name`

change_propagation

Changes the propagation behavior of this proposed relationship.

```
ooStatus change_propagation(uint8 newMode);
```

Parameters *newMode*

The new propagation behavior of this proposed relationship, which specifies whether the locking and deletion operations are propagated from locked or deleted source objects to their related destination objects; one of the following:

- 0 indicates that neither locks nor deletions are propagated.
- `oocLockPropagationYesDeletePropagationNo` indicates that locks are propagated, but deletions are not.
- `oocLockPropagationNoDeletePropagationYes` indicates that deletions are propagated, but locks are not.
- `oocLockPropagationYesDeletePropagationYes` indicates that both locks and deletions are propagated.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `propagation`

change_related_class

Changes the destination class of this proposed relationship.

```
ooStatus change_related_class(const char *relClass);
```

Parameters *relClass*

The name of the destination class of this proposed relationship.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `related_class_name`

change_short

Changes whether this proposed relationship is stored with short OIDs.

```
ooStatus change_short(ooBoolean isShort);
```

Parameters *isShort*

`oocTrue` if this proposed relationship is to be stored with short OIDs; otherwise, `oocFalse`.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `is_short`

change_to_bidirectional

Changes this proposed relationship to a bidirectional relationship.

```
ooStatus change_to_bidirectional(
     const char *inverseName,
     ooBoolean inverseIsToMany,
     ooAssocNumber specifiedAssocNum = 0);
```

Parameters *inverseName*

The name of the inverse relationship.

inverseIsToMany

`oocTrue` if the inverse relationship is a to-many relationship; otherwise, false.

specifiedAssocNum

The association number encoding characteristics of the relationship.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion

The bidirectional relationships in the schema are assigned serially-allocated 32-bit integers, called their *encoded association numbers*. Certain high-order bits of an encoded association number are set to encrypt the relationship's direction and other characteristics.

Most applications do not need to work with encoded association numbers. However, if you need to exactly recreate another schema description, you can call the `encoded_assoc_number` member function of a relationship descriptor for a bidirectional relationship that you want to recreate. You can pass the resulting number as the *`specifiedAssocNum`* parameter to this member function.

See also

`is_bidirectional`
`change_to_unidirectional`

change_to_many

Changes whether this proposed relationship is to-many.

```
ooStatus change_to_many(ooBoolean isToMany);
```

Parameters

`isToMany`

`oocTrue` if this proposed relationship is a to-many relationship and `oocFalse` if it is a to-one relationship.

Returns

`oocSuccess` if successful; otherwise `oocError`.

See also

`is_to_many`

change_to_unidirectional

Changes this proposed relationship to a unidirectional relationship.

```
ooStatus change_to_unidirectional();
```

Returns

`oocSuccess` if successful; otherwise `oocError`.

See also

`is_bidirectional`
`change_to_bidirectional`

change_versioning

Changes the versioning mode of this proposed relationship.

```
ooStatus change_versioning(uint8 newMode);
```

Parameters *newMode*

The new versioning mode for this proposed relationship, which specifies what happens to an association from a source object to a destination object when a new version of the source object is created; one of the following:

- 0 indicates that the association is deleted.
- `oocVersionDrop` indicates that the association is deleted.
- `oocVersionMove` indicates that the association is moved from the source object to its new version.
- `oocVersionCopy` indicates that the association is copied from the source object to its new version.

Returns `oocSuccess` if successful; otherwise `oocError`.

See also `versioning`

copy_mode

Gets the copy mode of this proposed relationship.

```
uint8 copy_mode() const;
```

Returns The copy mode of this proposed relationship, which specifies what happens to an association from a source object to a destination object when the source object is copied; one of the following:

- 0 indicates that the association is deleted.
- `oocCopyDrop` indicates that the association is deleted.
- `oocCopyMove` indicates that the association is moved from the source object to its new copy.
- `oocCopyCopy` indicates that the association is copied from the source object to the new object.

See also `change_copy_mode`

inverse_is_to_many

Tests whether this proposed relationship is a bidirectional relationship whose inverse is a to-many relationship.

```
ooBoolean inverse_is_to_many() const;
```

Returns `oocTrue` if this proposed relationship is a bidirectional relationship whose inverse is a to-many relationship; and `oocFalse` if it is a to-one relationship or a bidirectional relationship whose inverse is a to-one relationship.

See also `change_inverse`
`inverse_name`

inverse_name

Gets the name of the inverse relationship of this proposed relationship.

```
const char  *inverse_name() const;
```

Returns If this proposed relationship is bidirectional, this member function returns the name of its inverse relationship; if it is unidirectional, this member function returns null.

See also `change_inverse`
`inverse_is_to_many`

is_bidirectional

Tests whether this proposed relationship is bidirectional.

```
ooBoolean is_bidirectional() const;
```

Returns `oocTrue` if this proposed relationship is bidirectional; otherwise, `oocFalse`.

See also `change_to_bidirectional`
`change_to_unidirectional`

is_inline

Tests whether this proposed relationship is stored inline.

```
ooBoolean is_inline() const;
```

Returns `oocTrue` if this proposed relationship is stored inline; otherwise, `oocFalse`.

See also `change_inline`

is_relationship_type

Overrides the inherited member function. Indicates that this is a proposed relationship.

```
virtual ooBoolean is_relationship_type() const;
```

Returns `oocTrue`.

is_short

Tests whether this proposed relationship is stored with short OIDs.

```
ooBoolean is_short() const;
```

Returns `oocTrue` if this proposed relationship is stored with short OIDs; otherwise, `oocFalse`.

See also `change_short`

is_to_many

Tests whether this proposed relationship is to-many.

```
ooBoolean is_to_many() const;
```

Returns `oocTrue` if this proposed relationship is a to-many relationship and `oocFalse` if it is a to-one relationship.

See also `change_to_many`

propagation

Gets the propagation behavior of this proposed relationship.

```
uint8 propagation() const;
```

Returns The propagation behavior of this proposed relationship, which specifies whether the locking and deletion operations are propagated from locked or deleted source objects to their related destination objects; one of the following:

- 0 indicates that neither locks nor deletions are propagated.
- `oocLockPropagationYesDeletePropagationNo` indicates that locks are propagated, but deletions are not.
- `oocLockPropagationNoDeletePropagationYes` indicates that deletions are propagated, but locks are not.

- `oocLockPropagationYesDeletePropagationYes` indicates that both locks and deletions are propagated.

See also `change_propagation`

related_class_name

Gets the name of the destination class of this proposed relationship.

```
const char  *related_class_name() const;
```

Returns The name of the destination class of this proposed relationship.

Discussion To get the source class of this proposed relationship, call its inherited `defined_in_class` member function.

See also `change_related_class`

specified_assoc_number

Gets the type number encoding characteristics of this proposed relationship.

```
ooAssocNumber specified_assoc_number() const;
```

Returns The encoded association number that was specified when this proposed relationship was added to its proposed class, or zero if no such number was specified.

Discussion The bidirectional relationships in the schema are assigned serially-allocated 32-bit integers, called their *encoded association numbers*. Certain high-order bits of an encoded association number are set to encrypt the relationship's direction and other characteristics.

Most applications do not need to work with encoded association numbers. However, if you need to exactly recreate another schema description, you can call the `encoded_assoc_number` member function of a relationship descriptor for a bidirectional relationship that you want to recreate. To create a corresponding proposed relationship, you pass the resulting number as the *`specifiedAssocNum`* parameter to the `add_bidirectional_relationship` member function of the proposed class. After you have created the proposed relationship, you can call this member function to retrieve the specified association number.

versioning

Gets the versioning mode of this proposed relationship.

```
uint8 versioning() const;
```

Returns

The versioning mode of this proposed relationship, which specifies what happens to an association from a source object to a destination object when a new version of the source object is created; one of the following:

- 0 indicates that the association is deleted.
- `oocVersionDrop` indicates that the association is deleted.
- `oocVersionMove` indicates that the association is moved from the source object to its new version.
- `oocVersionCopy` indicates that the association is copied from the source object to its new version.

See also

`change_versioning`

Proposed_VArray_Attribute Class

Inheritance: **d_Meta_Object->Proposed_Property->Proposed_Attribute**
->Proposed_Collection_Attribute
->Proposed_VArray_Attribute

The class `Proposed_VArray_Attribute` represents the VArray attributes of a proposed class. An instance of this class is called a *proposed VArray attribute*.

See:
- "Reference Summary" on page 404 for an overview of member functions
- "Reference Index" on page 404 for a list of member functions

About Proposed VArray Attributes

A proposed VArray attribute provides information about a particular VArray attribute within the description of a particular proposed class. A VArray attribute contains either a single VArray or a fixed-size array of VArray values. The VArray elements can be any one of the following:

- Numeric values of a particular fundamental character, integer, floating-point, or pointer type
- Object references to instances of a particular persistence-capable referenced class
- Embedded instances of particular non-persistence-capable embedded class

You should never instantiate this class directly. Instead, you can obtain a proposed VArray attribute just as you would obtain any kind of proposed property.

- Call the `resolve_property` member function of the proposed class to look up the proposed VArray attribute by name.

- Call the `defines_property_begin` member function of the proposed class to get an iterator for all proposed properties of the proposed class.

If you need to call member functions defined in this class, you must cast the resulting proposed property to `Proposed_VArray_Attribute`.

Reference Summary

Getting Information About the Proposed VArray Attribute	`element_base_type`
	`element_referenced_class_name`
	`element_embedded_class_name`

Testing the Proposed VArray Attribute	`is_varray_basic_type`
	`is_varray_ref_type`
	`is_varray_embedded_class_type`
	`element_is_short`

Modifying the Proposed VArray Attribute	`change_element_base_type`
	`change_element_referenced_class`
	`change_element_embedded_class`
	`change_element_short`

Reference Index

`change_element_base_type`	Changes the numeric type for elements of this proposed VArray attribute.
`change_element_embedded_class`	Changes the embedded class for elements of this proposed VArray attribute.
`change_element_referenced_class`	Changes the referenced class for elements of this proposed VArray attribute.
`change_element_short`	Changes the reference type for elements of this proposed VArray attribute.
`element_base_type`	Gets the numeric type for elements of this proposed VArray attribute.
`element_embedded_class_name`	Gets the embedded class for elements of this proposed VArray attribute.

element_is_short	Tests the reference type for elements of this proposed VArray attribute.
element_referenced_class_name	Gets the referenced class for elements of this proposed VArray attribute.
is_varray_basic_type	Tests whether this proposed VArray attribute has numeric elements.
is_varray_embedded_class_type	Tests whether the elements of this proposed VArray attribute are instances of some non-persistence-capable class.
is_varray_ref_type	Tests whether the elements of this proposed VArray attribute are object-references to instances of some persistence-capable class.
is_varray_type	Overrides the inherited member function; indicates that this is a proposed VArray attribute.
kind	Gets the ODMG collection kind of this proposed VArray attribute.

Member Functions

change_element_base_type

Changes the numeric type for elements of this proposed VArray attribute.

```
ooStatus change_element_base_type(ooBaseType newType);
```

Parameters

newType

The new type for this proposed numeric VArray attribute; one of:

- ooCHAR indicates an 8-bit character.
- ooINT8 indicates an 8-bit signed integer.
- ooINT16 indicates a 16-bit signed integer.
- ooINT32 indicates a 32-bit signed integer.
- ooINT64 indicates a 64-bit signed integer.
- ooUINT8 indicates an 8-bit unsigned integer.
- ooUINT16 indicates a 16-bit unsigned integer.
- ooUINT32 indicates a 32-bit unsigned integer.
- ooUINT64 indicates a 64-bit unsigned integer.

- ooFLOAT32 indicates a 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates a 64-bit (double-precision) floating-point number.
- ooPTR indicates a 32-bit pointer.

Returns ocSuccess if successful; otherwise ocError.

Discussion This member function returns ocError if this is not a proposed numeric VArray attribute.

See also element_base_type
is_varray_basic_type

change_element_embedded_class

Changes the embedded class for elements of this proposed VArray attribute.

```
ooStatus change_element_embedded_class(const char *embClass);
```

Parameters *embClass*

The name of the new embedded class for this proposed embedded-class VArray attribute.

Returns ocSuccess if successful; otherwise ocError.

Discussion This member function returns ocError if this is not a proposed embedded-class VArray attribute.

See also element_embedded_class_name
is_varray_embedded_class_type

change_element_short

Changes the reference type for elements of this proposed VArray attribute.

```
ooStatus change_element_short(ooBoolean isShort);
```

Parameters *isShort*

ocTrue if this proposed object-reference VArray attribute should use short object references and ocFalse if it should use standard object references.

Returns ocSuccess if successful; otherwise ocError.

Discussion This member function returns `oocError` if this is not a proposed object-reference VArray attribute.

See also `element_is_short`
`is_varray_ref_type`

change_element_referenced_class

Changes the referenced class for elements of this proposed VArray attribute.

```
ooStatus change_element_referenced_class(const char *refClass);
```

Parameters *`refClass`*

The name of the new referenced class for this proposed object-reference VArray attribute.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion This member function returns `oocError` if this is not a proposed object-reference VArray attribute.

See also `element_referenced_class_name`
`is_varray_ref_type`

element_base_type

Gets the numeric type for elements of this proposed VArray attribute.

```
ooBaseType element_base_type() const;
```

Returns The type of numeric value in the elements of this proposed VArray attribute; one of the following:

- `ooCHAR` indicates 8-bit character.
- `ooINT8` indicates 8-bit signed integer.
- `ooINT16` indicates 18-bit signed integer.
- `ooINT32` indicates 32-bit signed integer.
- `ooINT64` indicates 64-bit signed integer.
- `ooUINT8` indicates 8-bit unsigned integer.
- `ooUINT16` indicates 18-bit unsigned integer.
- `ooUINT32` indicates 32-bit unsigned integer.
- `ooUINT64` indicates 64-bit unsigned integer.
- `ooFLOAT32` indicates 32-bit (single-precision) floating-point number.
- `ooFLOAT64` indicates 64-bit (double-precision) floating-point number.

- ooPTR indicates a 32-bit pointer.
- ooNONE indicates that the elements of this proposed VArray attribute are either object references or instances of an embedded class.

See also change_element_base_type
is_varray_basic_type

element_embedded_class_name

Gets the embedded class for elements of this proposed VArray attribute.

```
const char  *element_embedded_class_name() const;
```

Returns If this is a proposed embedded-class VArray attribute, this member function returns the name of the embedded class; otherwise, it returns null.

See also change_element_embedded_class
is_varray_embedded_class_type

element_is_short

Tests the reference type for elements of this proposed VArray attribute.

```
ooBoolean element_is_short() const;
```

Returns oocTrue if the this proposed object-reference VArray attribute uses short object references and oocFalse if it uses standard object references. This member function also returns oocFalse if this is not an object-reference VArray attribute.

See also change_element_short
is_varray_ref_type

element_referenced_class_name

Gets the referenced class for elements of this proposed VArray attribute.

```
const char  *element_referenced_class_name() const;
```

Returns If this is a proposed object-reference VArray attribute, this member function returns the name of the referenced class; otherwise, it returns null.

is_varray_type

Overrides the inherited member function; indicates that this is a proposed VArray attribute.

```
virtual ooBoolean is_varray_type() const;
```

Returns `oocTrue`.

is_varray_basic_type

Tests whether this proposed VArray attribute has numeric elements.

```
virtual ooBoolean is_varray_basic_type() const;
```

Returns `oocTrue` if this proposed VArray attribute is a proposed numeric VArray attribute; otherwise, `oocFalse`.

See also `change_element_base_type`
`element_base_type`

is_varray_embedded_class_type

Tests whether the elements of this proposed VArray attribute are instances of some non-persistence-capable class.

```
virtual ooBoolean is_varray_embedded_class_type() const;
```

Returns `oocTrue` if this proposed VArray attribute is a proposed embedded-class VArray attribute; otherwise, `oocFalse`.

See also `change_element_embedded_class`
`element_embedded_class_name`

is_varray_ref_type

Tests whether the elements of this proposed VArray attribute are object-references to instances of some persistence-capable class.

```
virtual ooBoolean is_varray_ref_type() const;
```

Returns `oocTrue` if this proposed VArray attribute is a proposed object-reference VArray attribute; otherwise, `oocFalse`.

See also `change_element_short`
`change_element_referenced_class`
`element_is_short`
`element_referenced_class_name`

kind

Gets the ODMG collection kind of this proposed VArray attribute.

```
d_Kind kind() const;
```

Returns ARRAY.

Relationship_Object Class

Inheritance: **`Persistent_Data_Object->Relationship_Object`**

The class `Relationship_Object` is a self-describing data type for relationships between persistent objects. An instance of this class is called a *relationship object*.

See:

- “Reference Summary” on page 412 for an overview of member functions
- “Reference Index” on page 412 for a list of member functions

About Relationship Objects

Each relationship object provides access to persistent data for a particular relationship, called its *described relationship*. The persistent data consists of zero or more *associations* for a particular persistent object, called *the source object*. Each association relates the source object to a particular *destination object*. If the described relationship is to-one, the source object can be associated with at most one destination object; if the described relationship is to-many, the source object can be associated with more than one destination object.

A relationship object uses a relationship descriptor for its described relationship to guide its access to the associated persistent data.

You obtain a relationship object for a particular relationship of a particular source object from a class object for that source object. To obtain the relationship object, you call the class object’s `get_relationship` member function, specifying the relationship of interest.

Reference Summary

Copying Relationship Objects	Relationship_Object operator=
Getting Information About the Relationship Object	contained_in relationship other_class
Getting the Destination Objects	get_class_obj get_ooref get_iterator
Setting the Destination Objects	set del add sub
Testing for Destination Objects	exist

Reference Index

add	Adds an association between the source object and the specified destination object.
contained_in	Gets this relationship object's containing class object.
del	Removes all existing associations between the source object and any destination objects.
exist	Tests whether an association exists between the source object and a destination object.
get_class_obj	Gets a class object for the destination object that is related to the source object by the described to-one relationship.
get_iterator	Gets an object iterator that finds all destination objects that are associated to the source object by the described to-many relationship.
get_ooref	Gets an object reference for the destination object that is related to the source object by the described to-one relationship.

is_relationship_object	Overrides the inherited member function. Indicates that this is a relationship object.
operator=	Assignment operator; sets this relationship object to a copy of the specified relationship object.
other_class	Gets the destination class of the described relationship.
relationship	Gets the described relationship.
Relationship_Object	Constructs a relationship object that is a copy of the specified relationship object.
set	Forms an association between the source object and the specified destination object.
sub	Removes the association(s) between the source object and the specified destination object.

Constructors

Relationship_Object

Constructs a relationship object that is a copy of the specified relationship object.

```
Relationship_Object(const Relationship_Object &otherROR);
```

Parameters

otherROR

The relationship object to be copied.

Discussion

The copy constructor creates a new relationship object with the same relationship descriptor and persistent relationship data as the specified relationship object. Both copies access the same persistent data. Any change to associations made with one relationship object will be seen by the other relationship object.

Operators

operator=

Assignment operator; sets this relationship object to a copy of the specified relationship object.

```
Relationship_Object &operator=(
     const Relationship_Object &otherROR);
```

Parameters *otherROR*

The relationship object to be copied.

Returns This relationship object after it has been updated to be a copy of *otherROR*.

Discussion Both copies access the same persistent data. Any change to associations made with one relationship object will be seen by the other relationship object.

Member Functions

add

Adds an association between the source object and the specified destination object.

```
void add(const ooHandle(ooObj) &newObjH);
```

Parameters *newObjH*

Handle for the destination object to be added.

Discussion The application must be able to obtain an update lock for the source object.

If the described relationship is bidirectional, this member function also adds the inverse association from the specified object to the source object. In that case, the application must be able to obtain update locks on both objects.

No error is signaled if an association already exists between the source object and the specified destination object. That is, you can create duplicate associations between the two objects (even though it could be semantically meaningless to do so).

This member function throws exceptions:

- `AddAssocError` if it is unable to add the specified association
- `DynRelAccessError` if the described relationship is to-one

See also `get_iterator`
`del`
`sub`

contained_in

Gets this relationship object's containing class object.

```
Class_Object &contained_in() const;
```

Returns The class object for the persistent object whose data this relationship object accesses.

Discussion The returned class object provides access to the source object for this relationship object.

del

Removes all existing associations between the source object and any destination objects.

```
void del();
```

Discussion The application must be able to obtain an update lock for the source object.

If the described relationship is bidirectional, this member function also removes the inverse association to this object from each of its formerly associated destination objects. In that case, the application must be able to obtain update locks on all of the destination objects.

If this member function is unable to remove the association(s), it throws a `DelAssocError` exception.

See also `add`
`set`
`sub`

exist

Tests whether an association exists between the source object and a destination object.

```
1.  ooBoolean exist() const;
2.  ooBoolean exist(const ooHandle(ooObj) &objHandle) const;
```

Parameters *`objHandle`*

Handle to the destination object to be tested.

Returns `oocTrue` if the specified association exists; otherwise, `oocFalse`.

Discussion The first variant tests whether any associations exist. The second tests whether an association exists to the specified destination object.

get_class_obj

Gets a class object for the destination object that is related to the source object by the described to-one relationship.

```
Class_Object get_class_obj() const;
```

Returns A class object containing data for the destination object, or a null class object if the source object is not associated with any destination object.

Discussion To obtain a reference to the destination object without opening a handle to it, call `get_ooref` instead of this member function.

This member function throws exceptions:

- `GetAssocError` if it is unable to get the destination object
- `DynRelAccessError` if the described relationship is to-many

See also `del`
`set`

get_iterator

Gets an object iterator that finds all destination objects that are associated to the source object by the described to-many relationship.

```
ooStatus get_iterator(
     ooItr(ooObj) &itrR,
     ooMode openMode = oocNoOpen);
```

Parameters *`itrR`*

An Objectivity/C++ object iterator to be initialized to find the associated destination objects.

`openMode`

Intended level of access to each destination object found by the object iterator's `next` member function:

- `oocNoOpen` (the default) causes `next` to set the object iterator to the next associated object without opening it.
- `oocRead` causes `next` to open the next associated object for read.
- `oocUpdate` causes `next` to open the next associated object for update.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion

On the successful completion of this member function, *itrR* is initialized to find the destination objects. That object iterator finds persistent objects, not class objects. You can construct a class object from any of these persistent objects if you want to examine its persistent data.

This member function throws a DynRelAccessError exception if the described relationship is to-one.

get_ooref

Gets an object reference for the destination object that is related to the source object by the described to-one relationship.

```
ooRef(ooObj) get_ooref() const;
```

Returns

An object reference for the destination object, or a null object reference if the source object is not associated with any destination object.

Discussion

To open a handle for the destination object and obtain a class object for it, call get_class_obj instead of this member function.

This member function throws exceptions:

- GetAssocError if it is unable to get the destination object
- DynRelAccessError if the described relationship is to-many

See also

del
set

is_relationship_object

Overrides the inherited member function. Indicates that this is a relationship object.

```
virtual ooBoolean is_relationship_object() const;
```

Returns

oocTrue.

other_class

Gets the destination class of the described relationship.

```
const d_Class &other_class() const;
```

Returns

A class descriptor for the destination class of the described relationship.

relationship

Gets the described relationship.

```
d_Relationship &relationship() const;
```

Returns A relationship descriptor for the described relationship.

set

Forms an association between the source object and the specified destination object.

```
void set(const ooHandle(ooObj) &newObjH);
```

Parameters *newObjH*

Handle for the new destination object of the described to-one relationship.

Discussion The application must be able to obtain an update lock for the source object.

If the described relationship is bidirectional, this member function also forms the inverse association from the specified destination object to the source object. In that case, the application must be able to obtain update locks on both objects.

Because this member function forms to-one associations, it throws an exception if the source object is already associated with a destination object by the described relationship. If you want to replace the existing destination object, you should first call `del` and then call this member function.

This member function throws exceptions:

- `SetAssocError` if it is unable to form the specified association
- `DynRelAccessError` if the described relationship is to-many

See also `get_class_obj`
`get_ooref`
`del`

sub

Removes the association(s) between the source object and the specified destination object.

```
void sub(
     const ooHandle(ooObj) &subObjH,
     const uint32 number = 1);
```

Parameters

subObjH

Handle for the destination object whose association is to be removed.

number

Number of associations to removed between the source object and the specified destination object:

- If you specify 0, all such associations are removed.
- If you specify 1 (the default), the first or only such association is removed.
- If you specify a number greater than 1, this member function removes the first *number* associations encountered.

You can use this parameter only if the described relationship is a many-to-many bidirectional association or a one-to-many unidirectional association.

Discussion

The application must be able to obtain an update lock for the source object.

If the described relationship is bidirectional, this member function also removes the inverse association(s) from the specified destination object to the source object. In that case, the application must be able to obtain update locks on both objects.

This member function throws exceptions:

- `SubAssocError` if it is unable to remove the specified association(s)
- `DynRelAccessError` if the described relationship is to-one

See also

`add`
`get_iterator`
`del`

Relationship_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Relationship_Type**

The abstract class `Relationship_Type` represents descriptors for relationship types. An instance of any concrete derived class is called a *relationship-type descriptor*; it provides information about a particular relationship type, called its *described type*.

Each relationship type has two defining characteristics:

- The directionality of relationships of this type (unidirectional or bidirectional)
- The destination class for relationships of this type.

Concrete derived classes represent descriptors for unidirectional and bidirectional relationships.

Because this class is abstract, you never instantiate it; instead, you work with instances of its concrete derived classes. You should not derive your own classes from this class.

Member Functions

is_relationship_type

Overrides the inherited member function. Indicates that the described type is a relationship type.

```
virtual ooBoolean is_relationship_type() const;
```

Returns `oocTrue`.

other_class

Gets the destination class of the described relationship type.

```
const d_Class  &other_class() const;
```

Returns A class descriptor for the destination class of the described relationship type.

String_Value Class

Inheritance: **`String_Value`**

The class `String_Value` is a self-describing data type for string values. An instance of this class is called a *string value*.

See:

- "Reference Summary" on page 424 for an overview of member functions
- "Reference Index" on page 424 for a list of member functions

About String Values

A string value provides access to a string object embedded in the data of some persistent object.

You obtain a string value for a particular string attribute of a particular persistent object from a class object for that persistent object. To obtain the string value, you call the class object's `get_string` member function, specifying the string attribute of interest.

Member functions allow you to determine what kind of string object the string value contains. Once you know the string type, you can convert the string value to an object that lets you access the string data.

- If a string value contains an instance of an internal string class, conversion operators allow you to convert it to an instance of the appropriate class: `ooVString`, `ooUtf8String`, or `ooSTString`. You can view or modify the persistent string data using the member functions of the internal class.
- If a string value contains an instance of an application-defined optimized string class `ooString(`*`N`*`)`, the `Optimized_String_Value` constructor allows you to convert the string value to an optimized string value. You can view or modify the string data with member functions of the optimized string value.

Reference Summary

Copying Relationship Objects	String_Value operator=
Getting Information About the String Value	type
Testing the String Value	is_vstring is_optimized_string is_utf8string is_ststring
Accessing the String Data	operator ooVString * operator ooUtf8String * operator ooSTString *

Reference Index

is_optimized_string	Tests whether this string value contains an optimized string.
is_ststring	Tests whether this string value contains a Smalltalk string.
is_utf8string	Tests whether this string value contains a Unicode string.
is_vstring	Tests whether this string value contains an ASCII string.
operator=	Assignment operator; sets this string value to a copy of the specified string value.
operator ooSTString *	Conversion operator; returns a pointer to the Smalltalk string contained by this string value.
operator ooUtf8String *	Conversion operator; returns a pointer to the Unicode string contained by this string value.
operator ooVString *	Conversion operator; returns a pointer to the ASCII string contained by this string value.

String_Value	Constructs a string value that is a copy of the specified string value.
type	Gets the type of string object that this string value contains.

Constructors

String_Value

Constructs a string value that is a copy of the specified string value.

```
String_Value(const String_Value &otherROR);
```

Parameters *otherROR*

The string value to be copied.

Discussion The copy constructor creates a new string value with the same persistent string data as the specified string value. Both copies access the same persistent data. Any change to the string made with one string value will be seen by the other string value.

Operators

operator=

Assignment operator; sets this string value to a copy of the specified string value.

```
String_Value &operator=(
     const String_Value &otherROR);
```

Parameters *otherROR*

The string value to be copied.

Returns This string value after it has been updated to be a copy of *otherROR*.

Discussion Both copies access the same persistent data. Any change to the string made with one string value will be seen by the other string value.

operator ooSTString *

Conversion operator; returns a pointer to the Smalltalk string contained by this string value.

```
operator ooSTString *() const;
```

Returns Pointer to an instance of `ooSTString` containing this string value's data.

Discussion This member function throws a `WrongStringType` exception if this string value does not contain a Smalltalk string.

operator ooUtf8String *

Conversion operator; returns a pointer to the Unicode string contained by this string value.

```
operator ooUtf8String *() const;
```

Returns Pointer to an instance of `ooUtf8String` containing this string value's data.

Discussion This member function throws a `WrongStringType` exception if this string value does not contain a Unicode string.

operator ooVString *

Conversion operator; returns a pointer to the ASCII string contained by this string value.

```
operator ooVString *() const;
```

Returns Pointer to an instance of `ooVString` containing this string value's data.

Discussion This member function throws a `WrongStringType` exception if this string value does not contain an ASCII string.

Member Functions

is_optimized_string

Tests whether this string value contains an optimized string.

```
ooBoolean is_optimized_string() const;
```

Returns `oocTrue` if this string value contains an optimized string of class `ooString(N)`; otherwise, `oocFalse`.

is_ststring

Tests whether this string value contains a Smalltalk string.

```
ooBoolean is_ststring() const;
```

Returns `oocTrue` if this string value contains a Smalltalk string of class `ooSTString`; otherwise, `oocFalse`.

is_utf8string

Tests whether this string value contains a Unicode string.

```
ooBoolean is_utf8string() const;
```

Returns `oocTrue` if this string value contains a Unicode string of class `ooUtf8String`; otherwise, `oocFalse`.

is_vstring

Tests whether this string value contains an ASCII string.

```
ooBoolean is_vstring() const;
```

Returns `oocTrue` if this string value contains an ASCII string of class `ooVString`; otherwise, `oocFalse`.

type

Gets the type of string object that this string value contains.

```
ooAsStringType type() const;
```

Returns The type of string object; one of the following:

- `ooAsStringOPTIMIZED` indicates an optimized string of class `ooString(`*`N`*`)`.
- `ooAsStringST` indicates a Smalltalk string of class `ooSTString`.
- `ooAsStringUTF8` indicates a Unicode string of class `ooUtf8String`.
- `ooAsStringVSTRING` indicates an ASCII string of class `ooVString`.

Top_Level_Module Class

Inheritance: **d_Meta_Object->d_Module, d_Scope->d_Module->Top_Level_Module**

The class `Top_Level_Module` represents descriptors for the top-level module in the schema of a federated database. An instance of this class is called a *top-level module descriptor.*

This class overrides member functions inherited from `d_Module`, but does not introduce any new functionality. Your programs should not need to use this class explicitly. That is, program variables for module descriptors can all use the declared type `d_Module` even if the described module is the top-level module.

You should never instantiate this class directly; instead, call the `d_Module::top_level` static member function to obtain a descriptor for the top-level module.

Reference Index

`is_top_level`	Overrides the inherited member function. Indicates that the described module is the top-level module.
`named_modules_begin`	Gets an iterator for all named modules in the federated database.
`named_modules_end`	Gets an iterator representing the termination condition for iteration through the named modules in the federated database.

Member Functions

is_top_level

Overrides the inherited member function. Indicates that the described module is the top-level module.

```
virtual ooBoolean is_top_level() const;
```

Returns `oocTrue`.

named_modules_begin

Gets an iterator for all named modules in the federated database.

```
virtual module_iterator named_modules_begin() const;
```

Returns A module iterator that finds all named (non-top-level) modules in the federated database.

See also `named_modules_end`

named_modules_end

Gets an iterator representing the termination condition for iteration through the named modules in the federated database.

```
virtual module_iterator named_modules_end() const;
```

Returns A module iterator that is positioned after the last named module in the federated database.

Discussion You can compare the iterator returned by `named_modules_begin` with the one returned by this member function to test whether iteration has finished.

type_iterator Class

Inheritance: **`type_iterator`**

The class `type_iterator` represents iterators for type descriptors. An instance of this class is called a *type iterator*.

See:

- "Reference Summary" on page 432 for an overview of member functions
- "Reference Index" on page 432 for a list of member functions

About Type Iterators

A type iterator steps through the types in the scope of some particular module. That collection of types is called the iterator's *iteration set*; during iteration, the type iterator keeps track of its position within its iteration set. The element at the current position is called the iterator's *current element*. The type iterator allows you to step through the iteration set, obtaining a descriptor for the current element at each step.

You should not instantiate this class directly. Instead, you call the `defines_types_begin` member function of a module descriptor to get a type iterator for the types in the scope of the described module. You can test for that iterator's termination condition by comparing it with the type iterator returned by the same module descriptor's `defines_types_end` member function.

Chapter 6, "Working With Iterators," contains additional information about iterators.

Reference Summary

Assigning	operator=
Getting the Current Element	operator*
Advancing the Current Position	operator++
Comparing	operator== operator!=

Reference Index

operator++	Increment operator; advances this type iterator's current position.
operator*	Dereference operator; gets this type iterator's current element.
operator=	Assignment operator; sets this type iterator to be a copy of the specified type iterator.
operator==	Equality operator; tests whether this type iterator is the same as the specified type iterator.
operator!=	Inequality operator; tests whether this type iterator is different from the specified type iterator.

Operators

operator++

Increment operator; advances this type iterator's current position.

```
1.   type_iterator  &operator++();
2.   type_iterator operator++(int n);
```

Parameters *n*

This parameter is not used in calling this operator; its presence in the function declaration specifies a postfix operator.

Returns (Variant 1) This type iterator, advanced to the next type.
(Variant 2) A new type iterator, set to this iterator before its position is advanced.

Discussion

Variant 1 is the prefix increment operator, which advances this type iterator and then returns it.

Variant 2 is the postfix increment operator, which returns a new type iterator set to this iterator, and then advances this iterator.

If the current position is already after the last type in the iteration set, neither variant advances this iterator.

operator*

Dereference operator; gets this type iterator's current element.

```
const d_Type  &operator*() const;
```

Returns

A type descriptor for the current element, or the null descriptor if the current position is after the last type in the iteration set.

Discussion

You should ensure that iteration has not terminated before calling this member function. The return value is undefined if the current position is after the last type in the iteration set.

operator=

Assignment operator; sets this type iterator to be a copy of the specified type iterator.

```
type_iterator  &operator=(const type_iterator &itrR);
```

Parameters

itrR

The type iterator specifying the new value for this type iterator.

Returns

This type iterator after it has been set to a copy of *itrR*.

operator==

Equality operator; tests whether this type iterator is the same as the specified type iterator.

```
int operator==(const type_iterator &other) const;
```

Parameters

other

The type iterator with which to compare this type iterator.

Returns

Nonzero if the two type iterators are equal and zero if they are different.

Discussion Two type iterators are equal if they iterate over the same iteration set and they have the same current position.

See also operator!=

operator!=

Inequality operator; tests whether this type iterator is different from the specified type iterator.

```
int operator!=(const type_iterator &moI) const;
```

Parameters *moI*

The type iterator with which to compare this type iterator.

Returns Nonzero if the two type iterators are different and zero if they are equal.

Discussion Two type iterators are different if they iterate over different iteration sets or if they are at different positions in the same iteration set.

See also operator==

Unidirectional_Relationship_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Relationship_Type ->Unidirectional_Relationship_Type**

The class `Unidirectional_Relationship_Type` represents descriptors for unidirectional relationship types that can be in the schema of the federated database. An instance of this class provides information about a particular unidirectional relationship type, called its *described type*.

You should never instantiate this class directly. Instead, you can obtain an instance of this class either from the module descriptor for the top-level module or from a relationship descriptor for a unidirectional relationship. Typically, you obtain an instance by calling the inherited `type_of` member function of a relationship descriptor.

Member Functions

is_unidirectional_relationship_type

Overrides the inherited member function. Indicates that the described type is a unidirectional relationship type.

```
virtual ooBoolean is_unidirectional_relationship_type() const;
```

Returns `oocTrue`.

VArray_Basic_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Attribute_Type**
->d_Collection_Type->VArray_Basic_Type

The class `Varray_Basic_Type` represents descriptors for numeric VArray types. An instance of this class is called a *numeric-VArray type descriptor.*

An instance of this class provides information about a particular numeric VArray type, called its *described type.* A numeric VArray type is a variable-size array type whose elements are of any fundamental character, integer, floating-point, or pointer type.

You should never instantiate this class directly. Instead, you can obtain a numeric-VArray type descriptor either from the module descriptor for the top-level module or from an attribute descriptor for a numeric VArray attribute. Typically, you obtain an instance by calling the inherited `type_of` member function of an attribute descriptor.

Reference Index

`element_base_type`	Gets the numeric type of the elements of the described type.
`is_varray_basic_type`	Overrides the inherited member function. Indicates that the described type is a variable-size array type with numeric elements.
`is_varray_type`	Overrides the inherited member function. Indicates that the described type is a variable-size array type.
`kind`	Gets the ODMG collection kind of the described type.

Member Functions

element_base_type

Gets the numeric type of the elements of the described type.

```
ooBaseType element_base_type() const;
```

Returns A code identifying the numeric element type; one of:

- ooCHAR indicates an array of 8-bit characters.
- ooINT8 indicates an array of 8-bit signed integers.
- ooINT16 indicates an array of 16-bit signed integers.
- ooINT32 indicates an array of 32-bit signed integers.
- ooINT64 indicates an array of 64-bit signed integers.
- ooUINT8 indicates an array of 8-bit unsigned integers.
- ooUINT16 indicates an array of 16-bit unsigned integers.
- ooUINT32 indicates an array of 32-bit unsigned integers.
- ooUINT64 indicates an array of 64-bit unsigned integers.
- ooFLOAT32 indicates an array of 32-bit (single-precision) floating-point numbers.
- ooFLOAT64 indicates an array of 64-bit (double-precision) floating-point numbers.
- ooPTR indicates a 32-bit pointer.

is_varray_basic_type

Overrides the inherited member function. Indicates that the described type is a variable-size array type with numeric elements.

```
virtual ooBoolean is_varray_basic_type() const;
```

Returns ooc True.

is_varray_type

Overrides the inherited member function. Indicates that the described type is a variable-size array type.

```
virtual ooBoolean is_varray_type() const;
```

Returns ooc True.

kind

Gets the ODMG collection kind of the described type.

```
d_Kind kind() const;
```

Returns ARRAY.

VArray_Embedded_Class_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Attribute_Type**
->d_Collection_Type->VArray_Embedded_Class_Type

The class `Varray_Embedded_Class_Type` represents descriptors for embedded-class VArray types. An instance of this class is called an *embedded-class-VArray type descriptor.*

An instance of this class provides information about a particular embedded-class VArray type, called its *described type.* An embedded-class VArray type is a variable-size array type whose elements are embedded instances of a particular non-persistence-capable embedded class.

You should never instantiate this class directly. Instead, you can obtain an embedded-class-VArray type descriptor either from the module descriptor for the top-level module or from an attribute descriptor for an embedded-class VArray attribute. Typically, you obtain an instance by calling the inherited `type_of` member function of an attribute descriptor.

Reference Index

`element_class_type`	Gets the class of the elements of the described type.
`is_varray_embedded_class_type`	Overrides the inherited member function. Indicates that the described type is a variable-size array type whose elements are instances of some non-persistence-capable class.

is_varray_type	Overrides the inherited member function. Indicates that the described type is a variable-size array type.
kind	Gets the ODMG collection kind of the described type.

Member Functions

element_class_type

Gets the class of the elements of the described type.

```
const d_Class  &element_class_type() const;
```

Returns A class descriptor for the class of the elements of the described array type.

is_varray_embedded_class_type

Overrides the inherited member function. Indicates that the described type is a variable-size array type whose elements are instances of some non-persistence-capable class.

```
virtual ooBoolean is_varray_embedded_class_type() const;
```

Returns `oocTrue`.

is_varray_type

Overrides the inherited member function. Indicates that the described type is a variable-size array type.

```
ooBoolean is_varray_type() const;
```

Returns `oocTrue`.

kind

Gets the ODMG collection kind of the described type.

```
d_Kind kind() const;
```

Returns ARRAY.

VArray_Object Class

Inheritance: **`Persistent_Data_Object->Collection_Object->VArray_Object`**

The class `VArray_Object` is a self-describing data type for variable-size arrays (VArrays). An instance of this class is called a *VArray object*.

See:

- "Reference Summary" on page 444 for an overview of member functions
- "Reference Index" on page 444 for a list of member functions

About VArray Objects

A VArray object provides access to a VArray (called its *associated VArray*) embedded in the data of some persistent object. You obtain a VArray object for a particular VArray attribute of a particular persistent object from a class object for that persistent object. To obtain the VArray object, you call the class object's `get_varray` member function, specifying the VArray attribute of interest.

Member functions of a VArray object enable you to get information about the VArray Object and its associated VArray, to change the size of the VArray, to get an individual element, and to set an individual element. The member functions for getting information about the VArray and changing its size are similar to the member functions of the `ooVArrayT<element_type>` class.

Chapter 3, "Examining Persistent Data," contains additional information about VArray objects.

Reference Summary

Copying VArray Objects	VArray_Object operator=
Getting Information About the VArray Object	contained_in type_of
Getting Information About the Associated VArray	size cardinality (*ODMG*) upper_bound (*ODMG*)
Testing the Associated VArray	is_empty (*ODMG*)
Getting Elements	get get_class_obj get_ooref get_string create_iterator (*ODMG*)
Setting Elements	set set_ooref replace_element_at (*ODMG*)
Changing the Size of the VArray	extend resize insert_element (*ODMG*) remove_all (*ODMG*)
Locking the Containing Persistent Object	update

Reference Index

get	Gets the data for the specified element of the associated numeric VArray.
get_class_obj	Gets the data for the specified element of the associated embedded-class or object-reference VArray.
get_ooref	Gets the data for the specified element of the associated object-reference VArray.
get_string	Gets the data for the specified element of the associated string VArray.
insert_element	(ODMG) Adds the specified element at the end of the associated numeric or object-reference VArray, increasing the size of the array.
is_empty	(ODMG) Tests whether the associated VArray is empty.
is_varray_object	Overrides the inherited member function. Indicates that this is a VArray object.
operator=	Assignment operator; sets this VArray object to a copy of the specified VArray object.
remove_all	(ODMG) Removes all the elements from the associated VArray, changing its size to 0.
replace_element_at	(ODMG) Replaces the specified element of the associated numeric or object-reference VArray with the specified value.
resize	Extends or truncates the associated VArray to the specified number of elements.
set	Sets the specified element of the associated numeric VArray.
set_ooref	Sets the specified element of the associated object-reference VArray.
size	Gets the current number of elements in the associated VArray.
type_of	Gets the element type of the associated VArray.
update	Explicitly opens the containing persistent object for update.
upper_bound	(ODMG) Gets the current number of elements in the associated VArray.
VArray_Object	Constructs a VArray that is a copy of the specified VArray object.

Constructors

VArray_Object

Constructs a VArray that is a copy of the specified VArray object.

```
VArray_Object(const VArray_Object &otherVOR);
```

Parameters *otherVOR*

The VArray object to be copied.

Discussion The copy constructor creates a new VArray object for the same attribute and persistent array data as the specified VArray object. Both copies access the same persistent data. Any change to the VArray made with one VArray object will be seen by the other VArray object.

Operators

operator=

Assignment operator; sets this VArray object to a copy of the specified VArray object.

```
VArray_Object &operator=(const VArray_Object &otherVOR);
```

Parameters *otherVOR*

The VArray object to be copied.

Returns This VArray object after it has been updated to be a copy of *otherVOR*..

Discussion Both copies access the same persistent data. Any change to the VArray made with one VArray object will be seen by the other VArray object.

Member Functions

cardinality

(*ODMG*) Gets the current number of elements in the associated VArray.

```
uint32 cardinality() const;
```

Returns Number of elements in the associated VArray.

Discussion This member function is equivalent to `size`.

contained_in

Gets this VArray object's containing class object.

```
Class_Object &contained_in() const;
```

Returns The class object for the persistent object whose data this VArray object accesses.

create_iterator

(*ODMG*) Creates a VArray iterator for the elements of the associated VArray.

```
d_Iterator<ooObj> create_iterator() const;
```

Returns A VArray iterator that finds elements of the associated object-reference VArray.

Discussion You must cast the returned VArray iterator to an the appropriate class for the element type of the associated VArray. For example, if the associated VArray is a `VArray(float64)`, you should cast the VArray iterator to `d_Iterator<float64>`.

Remember that the returned iterator is an Objectivity / C++ VArray iterator, not an Objectivity / C++ Active Schema iterator.

See also `replace_element_at`

extend

Adds the specified element at the end of the associated numeric or object-reference VArray, increasing the size of the array.

```
1.  ooStatus extend(const Numeric_Value newElem);
2.  ooStatus extend(const ooRef(ooObj) newElem);
```

Parameters *newElem*

The new element to be added at the end of the VArray.

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion The application must be able to obtain an update lock for the containing persistent object, and the lock on its container is upgraded, if necessary.

Extending a VArray implicitly resizes it, which is a potentially expensive operation. You should therefore use `extend` as a convenient way to add only a single element to a VArray. If you need to add multiple elements in a single transaction, you should instead use `resize` to allocate all the elements in one operation.

The first variant throws a `BadVArrayType` exception if the VArray's element type is not a numeric type; the second variant throws the same exception if the element type is not an object-reference type.

See also `resize`

get

Gets the data for the specified element of the associated numeric VArray.

```
Numeric_Value get(size_t index) const;
```

Parameters *index*

The zero-based index of the desired element.

Returns The numeric value at the specified index of the associated VArray.

Discussion This member function throws exceptions:

- `BadVArrayType` if the VArray's element type is not a numeric type
- `VArrayBoundsError` if *index* exceeds the VArray's upper bound

See also `set`

get_class_obj

Gets the data for the specified element of the associated embedded-class or object-reference VArray.

```
Class_Object get_class_obj(size_t index) const;
```

Parameters *index*

The zero-based index of the desired element.

Returns For an embedded-class VArray, a class object for the specified element; for an object-reference VArray, a class object for the persistent object referenced by the specified element.

Discussion To obtain an element of an object reference VArray without opening a handle for the referenced object, call `get_ooref` instead of this member function.

This member function throws exceptions:

- `BadVArrayType` if the VArray's element type is not an embedded-class type or an object-reference type
- `VArrayBoundsError` if *`index`* exceeds the VArray's upper bound

get_ooref

Gets the data for the specified element of the associated object-reference VArray.

```
ooRef(ooObj) get_ooref(size_t index) const;
```

Parameters *`index`*

The zero-based index of the desired element.

Returns The object reference at the specified index of the associated VArray.

Discussion To open a handle for the element and obtain a class object for it, call `get_class_obj` instead of this member function.

This member function throws exceptions:

- `BadVArrayType` if the VArray's element type is not an object-reference type
- `VArrayBoundsError` if *`index`* exceeds the VArray's upper bound

See also `set_ooref`

get_string

Gets the data for the specified element of the associated string VArray.

```
String_Value get_string(size_t index) const;
```

Parameters *`index`*

The zero-based index of the desired element.

Discussion This member function throws exceptions:

- `BadVArrayType` if the VArray's element type is not a string type
- `VArrayBoundsError` if *`index`* exceeds the VArray's upper bound

insert_element

(*ODMG*) Adds the specified element at the end of the associated numeric or object-reference VArray, increasing the size of the array.

```
1.  void insert_element(const Numeric_Value val);
2.  void insert_element(const ooRef(ooObj) objR);
```

Parameters

val

The numeric value to be added as a new element.

objR

The object reference to be added as a new element.

Discussion

This member function is equivalent to `extend`.

The application must be able to obtain an update lock for the containing persistent object, and the lock on its container is upgraded, if necessary.

Extending a VArray implicitly resizes it, which is a potentially expensive operation. You should therefore use `extend` as a convenient way to add only a single element to a VArray. If you need to add multiple elements in a single transaction, you should instead use `resize` to allocate all the elements in one operation.

The first variant throws a `BadVArrayType` exception if the VArray's element type is not a numeric type; the second variant throws the same exception if the element type is not an object-reference type.

is_empty

(*ODMG*) Tests whether the associated VArray is empty.

```
int is_empty() const;
```

Returns

Nonzero if the VArray has no elements; otherwise, zero.

is_varray_object

Overrides the inherited member function. Indicates that this is a VArray object.

```
virtual ooBoolean is_varray_object() const;
```

Returns

`oocTrue`.

remove_all

(*ODMG*) Removes all the elements from the associated VArray, changing its size to 0.

```
void remove_all();
```

Discussion The application must be able to obtain an update lock for the containing persistent object, and the lock on its container is upgraded, if necessary.

replace_element_at

(*ODMG*) Replaces the specified element of the associated numeric or object-reference VArray with the specified value.

```
1.  void replace_element_at(
    const Numeric_Value val,
    const d_Iterator<ooObj> &iterator);
2.  void replace_element_at(
    const Numeric_Value val,
    uint32 index);
3.  void replace_element_at(
    const ooRef(ooObj) objR,
    const d_Iterator<ooObj> &iterator);
4.  void replace_element_at(
    const ooRef(ooObj) objR,
    uint32 index);
```

Parameters

val

The new numeric value for the specified element.

iterator

A VArray iterator whose current position is the index of the desired element. The VArray iterator should be one that was obtained by calling this VArray object's `create_iterator` member function.

index

The zero-based index of the desired element.

objR

The new object reference for the specified element.

Discussion Variants 1 and 2 are similar to `set`; variants 3 and 4 are similar to `set_ooref`. The difference is that this member function does not return a status code to indicate whether the modification was successful.

The application must be able to obtain an update lock for the containing persistent object, and the lock on its container is upgraded, if necessary.

Variants 1 and 2 throw a BadVArrayType exception if the VArray's element type is not a numeric type; variants 3 and 4 throw the same exception if the element type is not an object-reference type.

All variants throw a VArrayBoundsError exception if the specified index exceeds the VArray's upper bound.

resize

Extends or truncates the associated VArray to the specified number of elements.

```
1.  ooStatus resize(size_t newSize);
2.  ooStatus resize(uint32 newSize);
```

Parameters

newSize

Total number of elements that the associated VArray is to have. Specify 0 to remove all the elements, freeing the storage allocated for the element vector.

Returns

oocSuccess if successful; otherwise oocError.

Discussion

The application must be able to obtain an update lock for the containing persistent object, and the lock on its container is upgraded, if necessary.

If the new size is larger than the current size, resize allocates storage for the additional elements, creating new, empty elements.

If the new size is smaller than the current size, resize frees the elements from index *newSize* + 1 to the end and then truncates the VArray to the new size.

set

Sets the specified element of the associated numeric VArray.

```
ooStatus set(size_t index, Numeric_Value val);
```

Parameters

index

The zero-based index of the desired element.

val

The new numeric value for the specified element.

Returns

oocSuccess if successful; otherwise oocError.

Discussion The application must be able to obtain an update lock for the containing persistent object, and the lock on its container is upgraded, if necessary.

This member function throws exceptions:

- BadVArrayType if the VArray's element type is not a numeric type
- VArrayBoundsError if *index* exceeds the VArray's upper bound

See also get

set_ooref

Sets the specified element of the associated object-reference VArray.

```
ooStatus set_ooref(size_t index, const ooRef(ooObj) objR);
```

Parameters *index*

The zero-based index of the desired element.

objR

The new object reference for the specified element.

Returns oocSuccess if successful; otherwise oocError.

Discussion The application must be able to obtain an update lock for the containing persistent object, and the lock on its container is upgraded, if necessary.

This member function throws exceptions:

- BadVArrayType if the VArray's element type is not an object-reference type
- VArrayBoundsError if *index* exceeds the VArray's upper bound

See also get_ooref

size

Gets the current number of elements in the associated VArray.

```
uint32 size() const;
```

Returns Number of elements in the associated VArray.

type_of

Gets the element type of the associated VArray.

```
const d_Type &type_of() const;
```

Returns A type descriptor for the VArray's element type.

update

Explicitly opens the containing persistent object for update.

```
ooStatus update();
```

Returns `oocSuccess` if successful; otherwise `oocError`.

Discussion When you explicitly open the persistent object for update, its container is locked for update; when the transaction commits, the entire VArray is be written to disk. You should use `update` primarily if you intend to change a large number of elements in a single transaction.

upper_bound

(*ODMG*) Gets the current number of elements in the associated VArray.

```
uint32 upper_bound() const;
```

Returns Number of elements in the associated VArray.

Discussion This member function is equivalent to `size`.

VArray_Ref_Type Class

Inheritance: **d_Meta_Object->d_Type->Property_Type->Attribute_Type**
->d_Collection_Type->VArray_Ref_Type

The class `Varray_Ref_Type` represents descriptors for object-reference VArray types. An instance of this class is called an *object-reference-VArray type descriptor.*

An instance of this class provides information about a particular object-reference VArray type, called its *described type.* An object-reference VArray type is a variable-size array type whose elements are object-references to instances of a particular persistence-capable referenced class.

You should never instantiate this class directly. Instead, you can obtain an object-reference-VArray type descriptor either from the module descriptor for the top-level module or from an attribute descriptor for an object-reference VArray attribute. Typically, you obtain an instance by calling the inherited `type_of` member function of an attribute descriptor.

Reference Index

`element_ref_type`	Gets the object-reference type of the elements of the described type.
`is_varray_ref_type`	Overrides the inherited member function. Indicates that the described type is a variable-size array type whose elements are object-references.
`is_varray_type`	Overrides the inherited member function. Indicates that the described type is a variable-size array type.
`kind`	Gets the ODMG collection kind of the described type.

Member Functions

element_ref_type

Gets the object-reference type of the elements of the described type.

```
const d_Ref_Type  &element_ref_type() const;
```

Returns A reference-type descriptor for the object-reference element type.

is_varray_ref_type

Overrides the inherited member function. Indicates that the described type is a variable-size array type whose elements are object-references.

```
ooBoolean is_varray_ref_type() const;
```

Returns `oocTrue`.

is_varray_type

Overrides the inherited member function. Indicates that the described type is a variable-size array type.

```
ooBoolean is_varray_type() const;
```

Returns `oocTrue`.

kind

Gets the ODMG collection kind of the described type.

```
d_Kind kind() const;
```

Returns ARRAY.

Error and Exception Classes

This chapter describes the Active Schema error and exception classes; the classes are listed in alphabetical order.

AccessDeletedAttribute Class

Inheritance: **`asException->AccessDeletedAttribute`**

Signals an attempt to access a deleted attribute (data member or association) of a class object.

The following member functions provide details about the exception.

attribute_of

Gets a descriptor of the attribute that was accessed.

```
const d_Attribute  &attribute_of() const;
```

Returns An attribute descriptor for the data member or association that was accessed inappropriately.

class_object

Gets the class object whose deleted attribute was accessed.

```
Class_Object  &class_object() const;
```

Returns The class object whose deleted attribute was accessed inappropriately.

AccessDenied Class

Inheritance: **asError->AccessDenied**

Signals an attempt to access a locked schema without the appropriate key.

AddAssocError Class

Inheritance: **asException->AddAssocError**

Signals a failed attempt to perform an add operation on a relationship object.

The following member function provides details about the exception.

relationship_object

Gets the relationship object on which the add operation failed.

```
Relationship_Object  &relationship_object();
```

Returns The relationship object on which the add operation failed.

AddProposedBaseClassError Class

Inheritance: **asException->AddProposedBaseClassError**

Signals an attempt to add a base class to a proposed class at an illegal position. This exception can arise when the specified position for the new base class is greater than the position of an existing or proposed attribute of the proposed class. All base classes of a class must be located before all attributes.

The following member functions provide details about the exception.

position

Gets the illegal position for the base class.

```
size_t position() const;
```

Returns The illegal position specified for the new base class.

proposed_base_class_of

Gets the new proposed base class that was added at an illegal position.

```
Proposed_Base_Class  &proposed_base_class_of() const;
```

Returns The proposed base class that was added illegally.

proposed_derived_class_of

Gets the proposed class to which the base class was added illegally.

```
Proposed_Class  &proposed_derived_class_of() const;
```

Returns The proposed class to which the base class was added illegally.

AddProposedPropertyErrorHi Class

Inheritance: **asException->AddProposedPropertyErrorHi**

Signals an attempt to add a property to a proposed class at an illegally high position. This exception can arise when the specified position for the new property is more than one greater than the position of the last existing or proposed property of the proposed class.

The following member functions provide details about the exception.

position

Gets the illegally high position at which a property was added.

```
size_t position() const;
```

Returns The illegally high position at which a property was added.

proposed_embedding_class_of

Gets the proposed class to which a property was added inappropriately.

```
Proposed_Class  &proposed_embedding_class_of() const;
```

Returns The proposed class to which a property was added inappropriately.

proposed_property_of

Gets the proposed property that was added inappropriately.

```
Proposed_Property  &proposed_property_of() const;
```

Returns The proposed property that was added inappropriately.

AddProposedPropertyErrorLo Class

Inheritance: **asException->AddProposedPropertyErrorLo**

Signals an attempt to add a property to a proposed class at an illegally low position. This exception can arise when the specified position for the new property is less than zero or lower than the position of an existing or proposed base class of the proposed class.

The following member functions provide details about the exception.

position

Gets the illegally low position at which a property was added.

```
size_t position() const;
```

Returns The illegally low position at which a property was added.

proposed_embedding_class_of

Gets the proposed class to which a property was added inappropriately.

```
Proposed_Class  &proposed_embedding_class_of() const;
```

Returns The proposed class to which a property was added inappropriately.

proposed_property_of

Gets the proposed property that was added inappropriately.

```
Proposed_Property  &proposed_property_of() const;
```

Returns The proposed property that was added inappropriately.

ArrayBoundsError Class

Inheritance: **asException->ArrayBoundsError**

Signals an attempt to access a non-existent element of the fixed-size-array in an attribute of a class object.

The following member functions provide details about the exception.

attribute_of

Gets a descriptor of the attribute that was accessed inappropriately.

```
const d_Attribute  &attribute_of() const;
```

Returns An attribute description for the attribute that was accessed inappropriately.

class_object

Gets the class object whose attribute was accessed inappropriately.

```
Class_Object  &class_object();
```

Returns The class object whose attribute was accessed inappropriately.

asError Class

Inheritance: **asError**

Abstract base class for all Active Schema error classes and exception classes.

The following operator and member functions provide details about the error.

operator const char *

Conversion operator that returns this error's message string.

```
operator const char *() const;
```

Returns This error's message string.

code

Gets this error's identifying code number.

```
uint32 code() const;
```

Returns The Objectivity / DB error code corresponding to this error.

is_system_error

Tests whether this error is a system-level error.

```
virtual ooBoolean is_system_error();
```

Returns `oocTrue` if this error is a system-level error and `oocFalse` if it is a user-level error.

Discussion The default implementation returns `oocTrue`; any derived class that represents user-level error overrides this member function.

---

asException Class

Inheritance: **`asError->asException`**

Abstract base class for all Active Schema exception classes, which represent user-level errors.

The static member functions control whether exceptions are enabled; the overridden `is_system_error` member function differentiate exception objects from error objects.

disable_exceptions

Disables exceptions.

```
static void disable_exceptions();
```

Discussion After this member function is called, when a user-level error condition occurs, Active Schema signals a standard Objectivity / DB error instead of throwing an exception.

See also `enable_exceptions`, `exceptions_are_enabled`

enable_exceptions

Enables exceptions.

```
static void enable_exceptions();
```

Discussion After this member function is called, when a user-level error condition occurs, Active Schema throws an exception. Exceptions are enabled by default.

See also disable_exceptions, exceptions_are_enabled

exceptions_are_enabled

Test whether exceptions are enabled.

```
static ooBoolean exceptions_are_enabled();
```

Returns oocTrue if exceptions are enabled; otherwise, oocFalse.

See also disable_exceptions, enable_exceptions

is_system_error

Overrides the inherited member function; indicates that this is a user-level error.

```
virtual ooBoolean is_system_error();
```

Returns oocFalse.

AssignToMO Class

Inheritance: **asException->AssignToMO**

Signals an attempt to assign to a non-null descriptor. Descriptors are read-only objects, so any attempt to use one as the left operand of an assignment operation causes an exception.

The following member function provides details about the exception.

meta_object_of

Gets the descriptor that was used in an assignment operation.

```
const d_Meta_Object  &meta_object_of() const;
```

Returns The descriptor that was used illegally as the left operand of an assignment operation.

AssignToNullMO Class

Inheritance: **asException->AssignToNullMO**

Signals an attempt to assign to a null descriptor object. Descriptors are read-only objects, so any attempt to use one as the left operand of an assignment operation causes an exception.

AttributeOutOfRange Class

Inheritance: **asException->AttributeOutOfRange**

Signals an attempt to obtain an attribute descriptor for the attribute at a non-existent attribute position in some class. This exception can arise when the specified position is larger than the number of attributes defined in the class.

The following member functions provide details about the exception.

class_of

Gets the class descriptor for the class in which the illegal access occurred.

```
const d_Class  &class_of() const;
```

Returns The class descriptor for the class.

position_of

Gets the illegal attribute position that was accessed.

```
size_t position_of() const;
```

Returns The attribute position of the non-existent attribute.

AttributeTypeError Class

Inheritance: **asException->AttributeTypeError**

Signals an attempt to use an attribute in a manner that is inappropriate for the attribute's type (basic, object reference, embedded class, VArray, and so on).

The following member functions provide details about the exception.

attribute_of

Gets a descriptor of the attribute that was accessed inappropriately.

```
const d_Attribute  &attribute_of() const;
```

Returns The attribute descriptor that was accessed inappropriately for its type.

class_of

Gets a descriptor of the class whose attribute was accessed inappropriately.

```
const d_Class  &class_of();
```

Returns The class descriptor.

formal_type

Gets the required attribute type for the operation that caused the exception.

```
ooAsType formal_type() const;
```

Returns The Active Schema type code for the required attribute type.

BadProposedVArrayElementType Class

Inheritance: **asException->BadProposedVArrayElementType**

Signals an attempt to add an object-reference VArray attribute to a proposed class, specifying an illegal object-reference type.

The following member functions provide details about the exception.

array_size

Gets the array size of the proposed attribute.

```
size_t array_size() const;
```

Returns The array size of the proposed attribute, that is, the number of VArray elements in the fixed-size array (or 1 for a single VArray).

other_class_name

Gets the referenced class.

```
const char  *other_class_name() const;
```

Returns The name of the class referenced by elements of the VArray.

proposed_attribute_name

Gets the name of the proposed VArray attribute.

```
const char  *proposed_attribute_name() const;
```

Returns The name of the proposed VArray attribute.

proposed_type

Gets the proposed element type for the VArray.

```
ooAsType proposed_type() const;
```

Returns The illegal type proposed for elements of the VArray (the only legal reference types are `d_Ref_Type_t` for object references and `Short_Ref_Type_t` for short object references).

visibility

Gets the proposed visibility for the VArray attribute.

```
d_Access_Kind visibility() const;
```

Returns The proposed visibility for the attribute.

BadVArrayIterator Class

Inheritance: **asException->BadVArrayIterator**

Signals an attempt to replace an element of a VArray object using a VArray iterator that is positioned either before the first element of the VArray or after its last element.

The following member functions provide details about the exception.

iterator_of

Gets the VArray iterator used to specify the element to be replaced.

```
d_Iterator<ooObj>  &iterator_of();
```

Returns The VArray iterator used to specify the element to be replaced.

varray_object

Gets the VArray object for which the illegal element replacement occurred.

```
VArray_Object  &varray_object();
```

Returns The VArray object.

BadVArrayType Class

Inheritance: **asException->BadVArrayType**

Signals an attempt to access an element of a VArray object in a manner that is inappropriate for the element's type (numeric, object reference, or embedded class).

The following member functions provide details about the exception.

formal_type

Gets the required element type for the operation that caused the exception.

```
ooAsType formal_type() const;
```

Returns The Active Schema type code for the required element type.

varray_object

Gets the VArray object whose element was accessed inappropriately.

```
VArray_Object  &varray_object() const;
```

Returns The VArray object on which the illegal operation occurred.

BasicModifyError Class

Inheritance: **asException->BasicModifyError**

Signals an attempt to replace a value of a non-basic type in a data member with a basic type. This exception can arise when a basic type is used as the new value for a data member that is not a basic type (or for the element of a VArray whose element type is not a basic type).

The following member functions provide details about the exception.

attribute_of

Gets the attribute in which the modification attempt occurred.

```
const d_Attribute  &attribute_of() const;
```

Returns An attribute descriptor for the attribute.

class_object

Gets the class object in which the modification attempt occurred.

```
Class_Object  &class_object();
```

Returns The class object.

CantAddModule Class

Inheritance: **asException->CantAddModule**

Signals that an attempt to add a named module to the federated-database schema failed. This exception can arise when the specified name for the new module is either null or the name of an existing module.

The following member functions provide details about the exception.

error_code

Gets the error code indicating the reason for failure.

```
ooAsAddModuleErrorCode error_code() const;
```

Returns The error code indicating the reason for failure; one of the following:

- NULL_NAME if a null name was specified for the new module.
- NAME_ALREADY_USED if the specified name is the name of an existing module.
- CREATE_FAILED if the module name was legal, but the request to create the new module failed.

module_name

Gets the proposed name for the new module.

```
const char  *module_name() const;
```

Returns The proposed name for the new module.

module_number

Gets the proposed module number for the new module.

```
uint32 module_number() const;
```

Returns The proposed module number (or zero if a new module number was to be allocated).

CantFindModule Class

Inheritance: **asException->CantFindModule**

Signals a failed attempt to obtain a module descriptor.

The following member function provides details about the exception.

module_name

Gets the name of the module whose descriptor was requested.

```
const char  *module_name() const;
```

Returns The name of the module.

CantFindRelInverse Class

Inheritance: **asError->CantRelInverse**

Signals a failure to find the inverse of a bidirectional relationship.

The following member function provides details about the error.

relationship

Gets the relationship whose inverse was requested.

```
d_Relationship  &relationship() const;
```

Returns The relationship whose inverse could not be found.

CantOpenModule Class

Inheritance: **asException->CantOpenModule**

Indicates a failed attempt to open a module.

The following member function provides details about the exception.

module_name

Gets the name of the module that could not be opened.

```
const char  *module_name() const;
```

Returns The name of the module.

ConstructNumericValueError Class

Inheritance: **asException->ConstructNumericValueError**

Indicates a failed attempt to construct a numeric value.

The following member function provides details about the exception.

actual_type

Gets the numeric type of the data from which a numeric value was being constructed.

```
ooBaseType actual_type() const;
```

Returns The numeric type of the data from which a numeric value was being constructed; one of:

- ooCHAR indicates 8-bit character.
- ooINT8 indicates 8-bit signed integer.
- ooINT16 indicates 18-bit signed integer.
- ooINT32 indicates 32-bit signed integer.
- ooINT64 indicates 64-bit signed integer.
- ooUINT8 indicates 8-bit unsigned integer.
- ooUINT16 indicates 18-bit unsigned integer.
- ooUINT32 indicates 32-bit unsigned integer.
- ooUINT64 indicates 64-bit unsigned integer.
- ooFLOAT32 indicates 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates 64-bit (double-precision) floating-point number.
- ooPTR indicates 32-bit pointer.
- ooNONE indicates an invalid or unrecognized numeric type.

base_type

Gets the numeric type that was being constructed.

```
ooBaseType base_type() const;
```

Returns The basic numeric that was being constructed; one of:

- ooCHAR indicates 8-bit character.
- ooINT8 indicates 8-bit signed integer.
- ooINT16 indicates 18-bit signed integer.
- ooINT32 indicates 32-bit signed integer.
- ooINT64 indicates 64-bit signed integer.
- ooUINT8 indicates 8-bit unsigned integer.
- ooUINT16 indicates 18-bit unsigned integer.
- ooUINT32 indicates 32-bit unsigned integer.
- ooUINT64 indicates 64-bit unsigned integer.
- ooFLOAT32 indicates 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates 64-bit (double-precision) floating-point number.
- ooPTR indicates 32-bit pointer.
- ooNONE indicates an invalid or unrecognized numeric type.

ConvertDeepPositionToInt Class

Inheritance: **asException->ConvertDeepPositionToInt**

Signals an illegal attempt to convert the class position for an inherited attribute to an integer.

DefaultValueForUnevolvedClass Class

Inheritance: **asException->DefaultValueForUnevolvedClass**

Signals an attempt to provide a default value for a proposed numeric attribute of a new proposed class.

The following member functions provide details about the exception.

attribute_name

Gets the name of the attribute that was given a default value.

```
const char  *attribute_name() const;
```

Returns The name of the attribute that was given a default value.

proposed_class_of

Gets the proposed class whose attribute was given a default value.

```
Proposed_Class  &proposed_class_of() const;
```

Returns The proposed class whose attribute was given a default value.

value

Gets the default value that was specified for the attribute.

```
Numeric_Value  &value() const;
```

Returns The default value that was specified for the attribute.

DelAssocError Class

Inheritance: **asException->DelAssocError**

Signals a failed attempt to perform a delete operation on a relationship object.

The following member function provides details about the exception.

relationship_object

Gets the relationship object on which the delete operation failed.

```
Relationship_Object  &relationship_object();
```

Returns The relationship object on which the delete operation failed.

DeletedClassObjectDependency Class

Inheritance: **asException->DeletedClassObjectDependency**

Signals and attempt to use a persistent-data object to access some property of a class that was defined at one time, but that has since been deleted by an Active Schema application.

The following member function provides details about the exception.

persistent_data_object_of

Gets the persistent-data object that was used inappropriately.

```
Persistent_Data_Object &persistent_data_object_of() const;
```

Returns The persistent-data object.

DynRelAccessError Class

Inheritance: **asException->DynRelAccessError**

Signals an attempt to perform an operation on a relationship object in a way that is inconsistent with the cardinality (to-one vs. to-many) of the described relationship.

This exception can arise when an object is reopened after a change has been made to its class in the schema.

The following member function provides details about the exception.

relationship_object

Gets the relationship object on which modification was attempted.

```
Relationship_Object  &relationship_object();
```

Returns The relationship object on which an inappropriate access attempt was made.

EvolutionError Class

Inheritance: **asException->EvolutionError**

Signals an attempt to activate an invalid or inconsistent set of proposed modifications to the schema.

FailedToFindClassByNameError Class

Inheritance: **asException->FailedToFindClassByNameError**

Signals a failure to find a class name in its module.

The following member functions provide details about the exception.

class_name

Gets the name of the class that couldn't be found.

```
const char  *class_name() const;
```

Returns The name of the class that couldn't be found.

module

Gets the module in which the class name was looked up.

```
d_Module  &module() const;
```

Returns The module in which the class name was looked up.

FailedToFindClassByNumberError Class

Inheritance: **asException->FailedToFindClassByNumberError**

Signals a failure to find a class with a given type number.

The following member function provides details about the exception.

type_number

Gets the type number of the class that couldn't be found.

```
ooTypeNumber type_number() const;
```

Returns The type number of the class that couldn't be found.

FailedToOpenObject Class

Inheritance: **asException->FailedToOpenObject**

Indicates a failed attempt to open a persistent object.

The following member function provides details about the exception.

class_object

Gets a class object for the persistent object that could not be opened.

```
Class_Object  &class_object() const;
```

Returns Class object for the persistent object that could not be opened.

mode

Gets the open mode in which the persistent object was being opened.

```
ooMode mode() const;
```

Returns One of the following constants:

- `oocRead`—the object was being opened for read.
- `oocUpdate`—the object was being opened for update.

FailedToReopenFD Class

Inheritance: **asException->FailedToReopenFD**

Indicates a failure to reopen the federated database, for example, in order to activate proposed modifications to the schema.

The following member function provides details about the exception.

fd_name

Gets the name of the federated database that could not be reopened.

```
const char  *fd_name() const;
```

Returns The name of the federated database that could not be reopened.

mode

Gets the open mode in which the persistent object was being opened.

```
ooMode mode() const;
```

Returns One of the following constants:

- `oocRead`—the object was being opened for read.
- `oocUpdate`—the object was being opened for update.

FailedToRestartTransaction Class

Inheritance: **`asException->FailedToRestartTransaction`**

Signals a failure to restart a transaction that was committed before activating proposed schema changes.

GetAssocError Class

Inheritance: **`asException->GetAssocError`**

Signals a failed attempt to perform a get operation on a relationship object.

The following member function provides details about the exception.

relationship_object

Gets the relationship object on which the get operation failed.

```
Relationship_Object  &relationship_object();
```

Returns The relationship object on which the get operation failed.

IllegalNumericCompare Class

Inheritance: **asException->IllegalNumericCompare**

Signals an illegal comparison of two numeric values of the Numeric_Value class. This exception occurs when a floating-point number is compared with a 64-bit integer or when an unsigned 64-bit integer is compared with a signed 64-bit integer.

The following member functions provide details about the exception.

value0

Gets the first of the two numeric values that were compared.

```
Numeric_Value  &value0() const;
```

Returns The numeric value used as the left operand of the illegal comparison.

value1

Gets the second of the two numeric values that were compared.

```
Numeric_Value  &value1() const;
```

Returns The numeric value used as the right operand of the illegal comparison.

IllegalNumericConvert Class

Inheritance: **asException->IllegalNumericConvert**

Signals an attempt to convert a numeric value to a type for which conversion is not supported. Some platforms do support conversion of unsigned 64-bit integers to floating-point numbers.

The following member functions provide details about the exception.

destination_type

Gets the numeric type to which conversion was attempted.

```
ooBaseType destination_type();
```

Returns The numeric type to which conversion was attempted; one of the following:

- ooCHAR indicates 8-bit character.
- ooINT8 indicates 8-bit signed integer.
- ooINT16 indicates 18-bit signed integer.
- ooINT32 indicates 32-bit signed integer.
- ooINT64 indicates 64-bit signed integer.
- ooUINT8 indicates 8-bit unsigned integer.
- ooUINT16 indicates 18-bit unsigned integer.
- ooUINT32 indicates 32-bit unsigned integer.
- ooUINT64 indicates 64-bit unsigned integer.
- ooFLOAT32 indicates 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates 64-bit (double-precision) floating-point number.
- ooPTR indicates 32-bit pointer.
- ooNONE indicates an invalid or unrecognized numeric type.

value

Gets the numeric values on which the conversion operation was attempted.

```
Numeric_Value  &value() const;
```

Returns The numeric value on which the conversion operation was attempted.

InactiveTransactionOpen Class

Inheritance: **asException->InactiveTransactionOpen**

Signals an attempt to open a persistent object when no transaction is active.

The following member functions provide details about the exception.

object_id

Gets the object ID for the object that could not be opened.

```
ooId  &object_id() const;
```

Returns The object identifier for the object that could not be opened.

InheritsFromSelfError Class

Inheritance: **asException->InheritsFromSelfError**

Indicates that a class or a proposed class inherits from itself.

The following member functions provide details about the exception.

class_of

Gets the class that inherits from itself.

```
const d_Class  &class_of() const;
```

Returns A class descriptor for the class that inherits from itself.

proposed_class_of

Gets the proposed class that inherits from itself.

```
Proposed_Class  &proposed_class_of();
```

Returns The proposed class that inherits from itself.

InitItrError Class

Inheritance: **asException->InitItrError**

Indicates a failure to initiate an object iterator to find the destination objects for a particular source object by a particular to-many relationship.

The following member function provides details about the exception.

relationship

Gets the relationship for which an object iterator could not be opened.

```
d_Relationship  &relationship();
```

Returns A relationship descriptor for the relationship whose object iterator could not be opened.

InvalidHandle Class

Inheritance: **asException->InvalidHandle**

Signals an attempt to create a class object using an invalid object handle.

The following member function provides details about the exception.

reference_object_of

Gets the invalid handle, converted to an object reference.

```
ooRef(ooObj)  &reference_object_of() const;
```

Returns The invalid handle converted to an object reference.

InvalidShape Class

Inheritance: **asException->InvalidShape**

Signals an attempt to create a class object from a persistent object that is not an instance of the specified class.

The following member functions provide details about the exception.

class_of

Gets the class specified for the class object.

```
const d_Class  &class_of() const;
```

Returns A class descriptor for the class that was specified for the class object.

object_id

Gets the object ID of the persistent object specified for the class object.

```
ooId  &object_id() const;
```

Returns The object ID for the data object that was specified for the class object.

shape_number

Gets the type number for the class and shape of which the persistent object is an instance.

```
ooTypeNumber shape_number() const;
```

Returns The type number for the class and shape of which the persistent object is an instance.

LostNameOfEvolvedClass Class

Inheritance: **asError->LostNameOfEvolvedClass**

Signals an internal error during activation of schema changes such that Active Schema can no longer find a class description that should exist.

ModuleInitError Class

Inheritance: **asException->ModuleInitError**

Signals an internal error while initializing a module. The exception may occur when retrieving the module description from the schema, when adding the module to the schema, or when activating proposals to the module.

The following member function provides details about the error.

module_name

Gets the name of the module that could not be initialized.

```
const char  *module_name() const;
```

Returns The name of the module that could not be initialized.

NameAlreadyInModule Class

Inheritance: **asException->NameAlreadyInModule**

Signals an attempt to propose a new class with a name that already exists in the module in which the proposal was attempted.

The following member functions provide details about the exception.

class_name

Gets the name of the class that was proposed inappropriately.

```
const char  *class_name() const;
```

Returns The name of the class.

module_name

Gets the name of the module in which the proposal was attempted.

```
const char  *module_name() const;
```

Returns The name of the module.

NameAlreadyProposedInModule Class

Inheritance: **asException->NameAlreadyProposedInModule**

Signals an attempt to propose a new class with a name that has already been proposed in the module in which the proposal was attempted.

The following member functions provide details about the exception.

class_name

Gets the name of the class that was proposed inappropriately.

```
const char  *class_name() const;
```

Returns The name of the class.

module_name

Gets the name of the module in which the proposal was attempted.

```
const char  *module_name() const;
```

Returns The name of the module.

NameNotInModule Class

Inheritance: **`asException->NameNotInModule`**

Signals an attempt to propose an evolved definition of a class that does not exist in the module in which the proposal was attempted.

The following member functions provide details about the exception.

class_name

Gets the name of the nonexistent class.

```
const char  *class_name() const;
```

Returns The name of the class.

module_name

Gets the name of the module in which the proposal was attempted.

```
const char  *module_name() const;
```

Returns The name of the module.

NewFail Class

Inheritance: **asError->NewFail**

Signals failure of the new operator to allocate the requested memory.

NonHandleClassObject Class

Inheritance: **asException->NonHandleClassObject**

Signals an attempt to get an object reference or handle from a class object that describes an embedded class.

The following member function provides details about the exception.

class_object_of

Gets the class object that was used inappropriately.

```
Class_Object  &class_object_of() const;
```

Returns The class object.

NonPersistentClassObject Class

Inheritance: **asException->NonPersistentClassObject**

Signals an attempt to create a class object for a new persistent object using a non-persistence-capable class.

NotOptimizedStringType Class

Inheritance: **asException->NotOptimizedStringType**

Signals an attempt to perform an operation that applies to optimized strings for some type other than an optimized string class.

The following member function provides details about the exception.

type_of

Gets the type for which the illegal operation was attempted.

```
const d_Type  &type_of() const;
```

Returns A type descriptor describing the type for which the illegal operation was attempted.

ProposeBadRel Class

Inheritance: **asException->ProposeBadRel**

Signals an attempt to activate a proposed bidirectional relationship whose inverse relationship does not exist and is not proposed.

ProposedBasicAttributeTypeError Class

Inheritance: **asException->ProposedBasicAttributeTypeError**

Signals an attempt to propose a numeric attribute with an invalid numeric type.

The following member functions provide details about the exception.

access_kind

Gets the access kind of the proposed attribute.

```
d_Access_Kind access_kind() const;
```

Returns The visibility or access kind for the attribute; one of the following:

- d_PUBLIC indicates public access.
- d_PROTECTED indicates protected access.
- d_PRIVATE indicates private access.

array_size

Gets the size of the fixed-size array of elements in the proposed attribute.

```
size_t array_size() const;
```

Returns The number of elements in the fixed-size array of elements in the proposed attribute, or 1 for an attribute that stores a single numeric value.

attribute_name

Gets the name of the proposed attribute.

```
const char  *attribute_name() const;
```

Returns The name of the proposed attribute.

base_type

Gets the numeric type specified for the proposed attribute.

```
ooBaseType base_type() const;
```

Returns The numeric type specified for the proposed attribute; one of the following:

- ooCHAR indicates 8-bit character.
- ooINT8 indicates 8-bit signed integer.
- ooINT16 indicates 18-bit signed integer.
- ooINT32 indicates 32-bit signed integer.
- ooINT64 indicates 64-bit signed integer.
- ooUINT8 indicates 8-bit unsigned integer.
- ooUINT16 indicates 18-bit unsigned integer.
- ooUINT32 indicates 32-bit unsigned integer.
- ooUINT64 indicates 64-bit unsigned integer.
- ooFLOAT32 indicates 32-bit (single-precision) floating-point number.
- ooFLOAT64 indicates 64-bit (double-precision) floating-point number.
- ooPTR indicates 32-bit pointer.
- ooNONE indicates an invalid or unrecognized numeric type.

position

Gets the position of the proposed attribute within its proposed class.

```
size_t position() const;
```

Returns The zero-base position of the proposed attribute within its proposed class.

proposed_class

Gets the proposed class in which the attribute was proposed.

```
Proposed_Class  &proposed_class();
```

Returns The proposed class in which the attribute was proposed.

ProposeEvolAndVers Class

Inheritance: **asException->ProposeEvolAndVers**

Signals an attempt to propose an evolved definition of some class and a new version of some class in the same transaction.

The following member function provides details about the exception.

class_name

Gets the name of the class on which evolution or versioning was proposed.

```
const char  *class_name() const;
```

Returns The name of the class on which evolution or versioning was proposed.

Discussion The proposed changes to existing class definitions in a given transaction must either be all proposed evolution or all proposed versioning:

- If class evolution has been proposed in a transaction, the first attempt to propose a new version of some class in the same transaction raises this exception. In that case, this function returns the name of the class on which versioning was proposed.
- If new versions have been proposed in a transaction, the first attempt to propose an evolved class in the same transaction raises this exception. In that case, this function returns the name of the class on which evolution was proposed.

ProposeEvolutionOfInternal Class

Inheritance: **asException->ProposeEvolutionOfInternal**

Signals an attempt to propose an evolved definition of an internal Objectivity / DB class.

The following member function provides details about the exception.

class_name

Gets the name of the class on which evolution was proposed.

```
const char  *class_name() const;
```

Returns The name of the internal class on which evolution was proposed.

ProposeVArrayPersistentError Class

Inheritance: **asException->ProposeVArrayPersistentError**

Signals an attempt to propose an embedded-class VArray attribute using a persistence-capable class as the embedded class.

The following member function provides details about the exception.

proposed_attribute_of

Gets the proposed attribute with the illegal embedded class.

```
Proposed_VArray_Attribute  &proposed_attribute_of();
```

Returns The proposed VArray attribute whose embedded class is a persistence-capable class.

SetAssocError Class

Inheritance: **asException->SetAssocError**

Signals a failed attempt to perform a set operation on a relationship object.

The following member function provides details about the exception.

relationship_object

Gets the relationship object on which the set operation failed.

```
Relationship_Object  &relationship_object();
```

Returns The relationship object on which the set operation failed.

StringBoundsError Class

Inheritance: **asException->StringBoundsError**

Signals an attempt to access a non-existent character of the string in an optimized string value.

The following member functions provide details about the exception.

actual_index

Gets the invalid character index.

```
size_t actual_index() const;
```

Returns The invalid zero-based character index.

optimized_string_of

Gets the optimized string value in which the invalid access occurred.

```
Optimized_String_Value  &optimized_string_of() const;
```

Returns The optimized string value in which the invalid access occurred.

string_length

Gets the length of the string.

```
size_t string_length() const;
```

Returns The number of characters in the string, including the terminating null character.

Discussion Valid character indexes range from 0 to one less than the returned string length.

SubAssocError Class

Inheritance: **asException->SubAssocError**

Signals a failed attempt to perform a subtract operation on a relationship object.

The following member function provides details about the exception.

relationship_object

Gets the relationship object on which the subtract operation failed.

```
Relationship_Object  &relationship_object();
```

Returns The relationship object on which the subtract operation failed.

UnnamedObjectError Class

Inheritance: **asException->UnnamedObjectError**

Signals a failure to supply a name to an operation that requires a name. This exception can arise if a proposed addition to the schema is given a null name.

The following member function provides details about the exception.

context_of

Gets the context in which a required name was not supplied.

```
const char  *context_of() const;
```

Returns The context in which a required name was not supplied, specified in the form *ClassName*::*MemberFunctionName*. For example, the context "d_Module::propose_new_class" indicates that no name was specified when a new class was proposed.

VArrayBoundsError Class

Inheritance: **asException->VArrayBoundsError**

Signals an attempt to access a non-existent element in a VArray object's associated VArray.

The following member functions provide details about the exception.

actual_index

Gets the invalid array index.

```
size_t actual_index() const;
```

Returns The invalid zero-based array index.

attribute_of

Gets the attribute containing the VArray that was accessed inappropriately.

```
const d_Attribute  &attribute_of() const;
```

Returns An attribute descriptor for the attribute containing the VArray.

varray_object

Gets the VArray object in which the invalid access occurred.

```
VArray_Object  &varray_object();
```

Returns The VArray object in which the invalid access occurred.

varray_size

Gets the size of the VArray.

```
size_t varray_size() const;
```

Returns The number of elements in the VArray.

Discussion Valid array indexes range from 0 to one less than the returned VArray size.

WrongCategoryOfNewObject

Inheritance: **asException->WrongCategoryOfNewObject**

Signals an attempt to create a new class object of a class whose category (container or non-container) is incompatible with the function creating the class object.

The following member functions provide details about the exception.

actual_category

Gets the class category for which the operation was attempted.

```
const char *actual_category() const;
```

Returns The class category for which the operation was attempted; one of the following:

- "ooObj" indicates the category of persistence-capable classes for basic objects.
- "ooContObj" indicates the category of container classes.

formal_category

Gets the class category required by the operation.

```
const char *formal_category() const;
```

Returns The required class category; one of the following:

- "ooObj" indicates the category of persistence-capable classes for basic objects.
- "ooContObj" indicates the category of container classes.

WrongStringType Class

Inheritance: **asException->WrongStringType**

Signals an attempt to perform an operation on a string value that contains string data of the wrong string class (for example, and attempt to convert a string value containing an optimized string to an ASCII string).

The following member functions provide details about the exception.

formal_type

Gets the type of string data required by the operation that failed.

```
ooAsStringType formal_type() const;
```

Returns The type of string data required by the operation that failed; one of the following:

- `ooAsStringOPTIMIZED` indicates an optimized string of class `ooString(`*`N`*`)`.
- `ooAsStringST` indicates a Smalltalk string of class `ooSTString`.
- `ooAsStringUTF8` indicates a Unicode string of class `ooUTF8String`.
- `ooAsStringST` indicates an ASCII string of class `ooVString`.

string_value

Gets the string value on which the operation was attempted.

```
String_Value  &string_value() const;
```

Returns The string value on which the operation was attempted.

Discussion You can call the `type` member function of the returned string value to find out what type of string data it contains.

A

Internal Classes

This appendix lists the Objectivity / DB internal classes that may appear in the schema description of a class.

In This Appendix

Persistence-Capable Classes

Non-Persistence-Capable Classes

Persistence-Capable Classes

The following table lists the Objectivity / DB persistence-capable classes that may appear in the schema description of an application-defined class. Typically, this occurs because some attribute of the class contains object-references to the internal class (possibly within a VArray or fixed-size array). In addition, the classes `ooObj`, `ooContObj`, and `ooGCContObj` appear in the schema description of any application-defined class that uses one of those internal class as a base class.

Classification	Internal Class	Description
Persistent-object base classes	`ooObj`	Abstract base class for all persistence-capable classes
	`ooContObj`	Non-garbage-collectible container class
	`ooGCContObj`	Garbage-collectible container class

Classification	Internal Class	Description
Collection classes	`ooTreeList`	List of persistent objects
	`ooHashSet`	Unordered set of persistent objects
	`ooTreeSet`	Sorted set of persistent objects
	`ooMap`	Name map
	`ooHashMap`	Unordered object map
	`ooTreeMap`	Sorted object map
String classes	`oojString`	String of Unicode characters
	`OoSTString`	Smalltalk string
Java date and time classes	`oojDate`	Calendar date
	`oojTime`	Clock time
	`oojTimestamp`	Point in time to the nearest nanosecond
Java array classes	`oojArrayOfInt8`	Variable-size array of 8-bit signed integers
	`oojArrayOfInt16`	Variable-size array of 16-bit signed integers
	`oojArrayOfInt32`	Variable-size array of 32-bit signed integers
	`oojArrayOfInt64`	Variable-size array of 64-bit signed integers
	`oojArrayOfBoolean`	Variable-size array of 8-bit unsigned integers
	`oojArrayOfCharacter`	Variable-size array of characters
	`oojArrayOfFloat`	Variable-size array of single-precision floating-point numbers
	`oojArrayOfDouble`	Variable-size array of double-precision floating-point numbers
	`oojArrayOfObject`	Variable-size array of object references
Smalltalk array class	`OoSTUInt16VArray`	Variable-size array of 16-bit signed integers

Non-Persistence-Capable Classes

The following table lists the Objectivity / DB non-persistence-capable classes and structures that may appear in the schema description of an application-defined class. A concrete internal class appears in a schema description when some attribute of the class contains the internal class as an embedded-class (possibly within a VArray or fixed-size array). The abstract class `ooSQLnull` appears as an embedded class in the schema description of any of its derived classes.

Classification	Internal Class	Description
String classes	`ooVString`	String of ASCII characters
	`ooUTF8String`	String of Unicode characters
SQL numeric classes	`ooSQLnull`	Abstract base class for all SQL data types that can represent null
	`ooSQLnull_int16`	16-bit signed integer; can represent null
	`ooSQLnull_int32`	32-bit signed integer; can represent null
	`ooSQLnumeric`	Number with a specific precision and scale; no representation of null
	`ooSQLnull_numeric`	Number with a specific precision and scale; can represent null
	`ooSQLnull_float32`	Single-precision floating-point number; can represent null
	`ooSQLnull_float64`	Double-precision floating-point number; can represent null
	`ooSQLmoney`	Number with two decimal places; no representation of null
	`ooSQLnull_money`	Number with two decimal places; can represent null

Classification	Internal Class	Description
SQL date and time classes	`ooSQLdate`	Calendar date; no representation of null
	`ooSQLnull_date`	Calendar date; can represent null
	`ooSQLtime`	Clock time; no representation of null
	`ooSQLnull_time`	Clock time; can represent null
	`ooSQLtimestamp`	Point in time to the nearest millisecond; no representation of null
	`ooSQLnull_timestamp`	Point in time to the nearest millisecond; can represent null
ODMG date and time classes	`d_Date`	Calendar date; no representation of null
	`d_Time`	Clock time; no representation of null
	`d_Timestamp`	Point in time to the nearest millisecond; no representation of null
	`d_Interval`	The duration of elapsed time between two points in time

B

Programming Examples

This appendix contains the source code for programming examples. Excerpts from some of these examples appear in Part 1.

In This Appendix

Examining the Schema

The following functions can be used to examine the schema of a federated database.

`base_type_to_text`	Converts a code of type ooBaseType to a text description of the corresponding numeric type.
`showInheritance`	Prints the parent classes and child classes of a particular class.
`showProperties`	Prints a brief description of every property of a class, including information about the property's type.
`showUses`	Lists all properties that use a particular class.

base_type_to_text

global function

Converts a code of type ooBaseType to a text description of the corresponding numeric type.

```
char *base_type_to_text(ooBaseType bt, char *textbuf) {
   switch (bt) {
      case ooCHAR: {
         sprintf(textbuf, "8-bit character");
         break;
      }
      case ooINT8: {
         sprintf(textbuf, "8-bit signed integer");
         break;
      }
      case ooUINT8: {
         sprintf(textbuf, "8-bit unsigned integer");
         break;
      }
      case ooINT16: {
         sprintf(textbuf, "16-bit signed integer");
         break;
      }
      case ooUINT16: {
         sprintf(textbuf, "16-bit unsigned integer");
         break;
      }
      case ooINT32: {
         sprintf(textbuf, "32-bit signed integer");
         break;
      }
      case ooUINT32: {
         sprintf(textbuf, "32-bit unsigned integer");
         break;
      }
      case ooINT64: {
         sprintf(textbuf, "64-bit signed integer");
         break;
      }
      case ooUINT64: {
         sprintf(textbuf, "64-bit unsigned integer");
         break;
      }
```

```
      case ooFLOAT32: {
         sprintf(textbuf,
                  "single-precision floating-point number");
         break;
      }
      case ooFLOAT64: {
         sprintf(textbuf,
                  "double-precision floating-point number");
         break;
      }
      case ooPTR: {
         sprintf(textbuf, "32-bit pointer");
         break;
      }
      default: {
         sprintf(textbuf, "unrecognized numeric type");
         break;
      }
   } // End switch
   return textbuf;
} // End base_type_to_text
```

---

showInheritance

global function

Prints the parent classes and child classes of a particular class.

```
void showInheritance(const d_Class &aClass) {

   // Print parent classes, if any
   inheritance_iterator itr = aClass.base_class_list_begin();
   if (itr != aClass.base_class_list_end()) {
      cout << "Parent classes of " << aClass.name();
      cout << ":" << endl;
      while (itr != aClass.base_class_list_end()) {
         const d_Inheritance &curInh = *itr;
         const d_Class &curParent = curInh.derives_from();
         cout << curParent.name();
         d_Access_Kind access = curInh.access_kind();
         if (access == d_PROTECTED) {
            cout << " (protected)";
         }
         else if (access == d_PRIVATE) {
            cout << " (private)";
         }
```

```
            cout << endl;
            ++itr;
         } // End while more parents
         cout << endl;
      } // End if any parents
      else {
         cout << aClass.name() << " has no parent classes";
         cout << endl << endl;
      } // End else no parents

      // Print child classes, if any
      itr = aClass.sub_class_list_begin();
      if (itr != aClass.sub_class_list_end()) {
         cout << "Child Classes of " << aClass.name();
         cout << ":" << endl;
         while (itr != aClass.sub_class_list_end()) {
            const d_Inheritance &curInh = *itr;
            const d_Class &curChild = curInh.inherits_to();
            cout << curChild.name() << endl;
            ++itr;
         } // End while more child classes
         cout << endl;
      } // End if any child classes
      else {
         cout << aClass.name() << " has no subclasses";
         cout << endl << endl;
      } // End else no child classes
   } // End showInheritance
```

---

showProperties

global function

Prints a brief description of every property of a class, including information about the property's type.

```
void showProperties(const d_Class &aClass) {
   char textbuf[64];

   cout << aClass.name() << " Properties:" << endl;

   // Iterate through all properties (defined and inherited)
   attribute_plus_inherited_iterator itr =
      aClass.attributes_plus_inherited_begin();
   while (itr != aClass.attributes_plus_inherited_end()) {
      // Get descriptor for current property
      const d_Attribute &curAttr = *itr;
```

```
// Print property name
cout << endl << curAttr.name();

// Test whether property is inherited
const Class_Position pos =
   aClass.position_in_class(curAttr);
if (! pos.is_convertible_to_uint()) {
   cout << " (inherited)";
}

// Describe the property

if (curAttr.is_relationship()) {
   // Property is a relationship
   const d_Relationship &rel =
      (const d_Relationship &)curAttr;

   // Test whether the relationship is to-many
   if (rel.is_to_many()) {
      cout << " [to-many]";
   }
   cout << ":" << endl;

   if (rel.is_bidirectional()) {
      cout << "  bidirectional relationship to ";
      cout << rel.otherClass().name() << "; inverse: ";
      cout << rel.inverse().name();
   } // End bidirectional

   else {
      cout << "  unidirectional relationship to ";
      cout << rel.otherClass().name();
   } // End unidirectional
} // End if property is a relationship

else {
   // Property is an attribute

   // Test whether the attribute has multiple values
   size_t nVals = curAttr.array_size();
   if (nVals > 1) {
      cout << " [" << nVals << " values]";
   }
   cout << ":" << endl;

   // Describe the type of the attribute
   const d_Type &curType = curAttr.type_of();
```

```
if (curType.is_basic_type()) {
   // Property is a numeric attribute
   const Basic_Type &bt =
      (const Basic_Type &)curType;
   cout << "  " << base_type_to_text(bt.base_type(),
                                     textbuf);
} // End numeric attribute

else if (curType.is_ref_type()) {
   // Property is an object-reference attribute
   const d_Ref_Type &ref =
      (const d_Ref_Type &)curType;
   cout << "  reference to ";
   cout << ref.referenced_type().name();
} // End object-reference attribute

else if (curType.is_class()) {
// Property is an embedded attribute or a base class

   if (curAttr.is_base_class()) {
      cout << "  base class";
   } // End base class
   else {
      cout << "  embedded instance of ";
      cout << curType.name();
   } // End embedded class
} // End embedded or base class

else if (curType.is_varray_type()) {
   // Property is a VArray attribute
   cout << "  Varray of ";

   // Describe the element type
   if (curType.is_varray_basic_type()) {
      // Property is a numeric VArray attribute
      const VArray_Basic_Type &vbt =
         (const VArray_Basic_Type &)curType;
      cout << base_type_to_text(
                  vbt.element_base_type(),
                  textbuf);
   } // End numeric VArray
   else if (curType.is_varray_ref_type()) {
      // Property is an object-reference VArray
      // attribute
      const VArray_Ref_Type &vref =
         (const VArray_Ref_Type &)curType;
```

```
                    const d_Ref_Type &ref = vref.element_ref_type();
                    cout << "reference to ";
                    cout << ref.referenced_type().name();
                 } // End object-reference VArray
                 else if (curType.is_varray_embedded_class_type()){
                    // Property is embedded-class VArray attribute
                    const VArray_Embedded_Class_Type &vembd =
                       (const VArray_Embedded_Class_Type &)curType;
                    cout << "embedded instance of ";
                    cout << vembd.element_class_type().name();
                 } // End embedded-class VArray
                 else {
                    cout << "  unrecognized VArray type";
                 }
              } // End VArray attribute
              else {
                 cout << "  unrecognized attribute type";
              }
           } // End else property is an attribute
           cout << endl;
           ++itr;
        } // End while more properties
        cout << endl;
     } // End showProperties
```

---

showUses

global function

Lists all properties that use a particular class.

```
void showUses(const d_Class &aClass) {
   const char *name = aClass.name();
   cout << "Properties that use the class ";
   cout << name << endl;

   // Iterate through all properties that use the class
   property_iterator itr = aClass.used_in_property_begin();
   while (itr != aClass.used_in_property_end()) {
      const d_Property &curProp = *itr;
      cout << "  " << curProp.name() << " of ";
      cout << curProp.defined_in_class().name();
      const d_Type &curType = curProp.type_of();
```

```
if (aClass.persistent_capable()) {
   // Every property that uses a persistence-capable
   // class should be either an object-reference
   // attribute or an object-reference VArray attribute

   if (curType.is_ref_type()) {
      // Property is an object-reference attribute
      // Check whether it uses short or standard
      // references
      const d_Ref_Type &ref =
         (const d_Ref_Type &)curType;
      if (ref.is_short()) {
         cout << " ooShortRef(";
      }
      else {
         cout << " ooRef(";
      }
      cout << name << ")" << endl;
   } // End object-reference type

   else if (curType.is_varray_ref_type()) {
      // Property is an object-reference VArray
      // attribute
      // Check whether it uses short or standard
      // references
      const VArray_Ref_Type &vref =
         (const VArray_Ref_Type &)curType;
      const d_Ref_Type &ref = vref.element_ref_type();
      if (ref.is_short()) {
         cout << " ooVArray(ooShortRef(";
      }
      else {
         cout << " ooVArray(ooRef(";
      }
      cout << name << "))" << endl;
   } // End object-reference VArray

   else {
      cout << " unexpected type" << endl;
   }
} // End if persistence-capable

else {
   // Every property that uses a non-persistence-capable
   // class should be either an embedded-class
   // attribute or an embedded-class VArray attribute
```

```
            if (curType.is_class()) {
               // Property is an embedded-class attribute
               // Check whether it uses short or standard
               // references
               cout << " " << name << endl;
            } // End embedded-class type

            else if (curType.is_varray_embedded_class_type()) {
               // Property is an embedded-class VArray attribute
               cout << " ooVArray(";
               }
               cout << name << ")" << endl;
            } // End embedded-class VArray

            else {
               cout << " unexpected type" << endl;
            }
         } // End else non-persistence-capable
         ++itr;
      } // End while more properties
} // End showUses
```

Examining Persistent Data

The following functions can be used to examine the persistent data in a federated database.

showData	Prints the data for a class object.
showNumeric	Prints the data for a numeric value.
showRef	Prints an object reference, giving the class name and object ID of the referenced persistent object.
showRefVArray	Iterates through the elements of a VArray object for an object-reference VArray, printing each element.
showRelationship	Prints the data for a relationship object.
showString	Prints the data for a string value.
showVArray	Prints the data for a VArray object.

showData

global function

Prints the data for a class object.

```
ooStatus showData (
   Class_Object &CO, // Class object to display
   ooBoolean mbd = oocFalse, // True if base or embedded class
   char *prefix = "") // Prefix for attribute of embedded class
{
   ooStatus rc;

   // Check for null class object
   if (! CO) {
      cout << "(null)" << endl;
      return oocSuccess;
   }

   const d_Class &classOfObj = CO.type_of();

   if (! classOfObj) {
      cerr << "Can't find class of object" << endl;
      return oocError;
   }
   else if (! mbd) {
      cout << endl << "Object of class " << classOfObj.name();
      cout << endl;
   }

   // Iterate through components, showing data for each
   size_t nComponents = classOfObj.number_of_attributes();
   for (size_t pos = 0; pos < nComponents; ++pos) {

      // Get an attribute descriptor for the component
      const d_Attribute &curAttr =
         classOfObj.attribute_at_position(pos);

      // Get the type of the component
      const d_Type &curType = curAttr.type_of();

      // Determine the kind of component
      if (curAttr.is_base_class()) {
         // Ignore internal base classes like ooObj
         if (! ((const d_Class &)curType).is_internal()) {
            // Recursively show properties of base class
            rc = showData(CO.get_class_obj(pos),
                          oocTrue,
                          prefix);
```

```
            if (rc != oocSuccess) {
               return rc;
            }
         } // End if not internal
      } // End if base class

      else if (curAttr.is_relationship()) {
         cout << prefix << curAttr.name() << ":" << endl;
         // Get the relationship object
         rc = showRelationship(CO.get_relationship(pos));
         if (rc != oocSuccess) {
            return rc;
         }
         cout << endl;
      } // End if relationship

      else {
         // Component is an attribute
         cout << prefix << curAttr.name();

         // Get the number of values in the fixed-size array
         size_t nVals = curAttr.array_size();
         if (nVals > 1) {
            cout << " (" << nVals << ")";
         }
         cout << ":" << endl;

         // Test the attribute type to determine how to
         // access the attribute's data

         if (curType.is_basic_type()) {
            // Get the number value(s)
            if (nVals == 1) {
               rc = showNumeric(CO.get(pos));
               if (rc != oocSuccess) {
                  return rc;
               }
            } // End if one value
            else {
               for (size_t n = 0; n < nVals; ++n) {
                  cout << n << ". ";
                  rc = showNumeric(CO.get(pos, n));
                  if (rc != oocSuccess) {
                     return rc;
                  }
               } // End for each value
            } // End else array of values
```

```
   cout << endl;
} // End if numeric attribute

else if (curType.is_ref_type()) {
   // Get the object reference(s)
   if (nVals == 1) {
      ooRef(ooObj) ref = CO.get_ooref(pos);
      showRef(ref);
   } // End if one value
   else {
      for (size_t n = 0; n < nVals; ++n) {
         cout << n << ". ";
         ooRef(ooObj) ref = CO.get_ooref(pos, n);
         showRef(ref);
      } // End for each value
   } // End else array of values
   cout << endl;
} // End if object-reference attribute

else if (curType.is_string_type()) {
   // Get the string value(s)
   if (nVals == 1) {
      rc = showString(CO.get_string(pos));
      if (rc != oocSuccess) {
         return rc;
      }
   } // End if one value
   else {
      for (size_t n = 0; n < nVals; ++n) {
         cout << n << ". ";
         rc = showString(CO.get_string(pos, n));
         if (rc != oocSuccess) {
            return rc;
         }
      } // End for each value
   } // End else array of values
   cout << endl;
} // End if string attribute

else if (curType.is_class()) {
   // Set attrName to prefix for attributes
   // of the embedded class
   char *attrName =
      new char[strlen(prefix) +
               strlen(curAttr.name()) + 2];
```

```
            if (prefix) {
               sprintf(attrName, "%s.%s", prefix,
                                          curAttr.name());
            }
            else {
               sprintf(attrName, "%s", curAttr.name());
            }

            // Get the embedded instance(s)
            if (nVals == 1) {
               rc = showData(CO.get_class_obj(pos),
                             oocTrue,
                             attrName);
               if (rc != oocSuccess) {
                  return rc;
               }
            } // End if one value
            else {
               for (size_t n = 0; n < nVals; ++n) {
                  cout << n << ". ";
                  rc = showData(CO.get_class_obj(pos, n),
                                oocTrue,
                                attrName);
                  if (rc != oocSuccess) {
                     return rc;
                  }
               } // End for each value
            } // End else array of values
            delete [] attrName;
            cout << endl;
         } // End if embedded non-string-class attribute

         else if (curType.is_varray_type()) {
            // Get the varray(s)
            if (nVals == 1) {
               rc = showVArray(CO.get_varray(pos));
               if (rc != oocSuccess) {
                  return rc;
               }
            } // End if one value
            else {
               for (size_t n = 0; n < nVals; ++n) {
                  cout << n << ". ";
                  rc = showVArray(CO.get_varray(pos, n));
                  if (rc != oocSuccess) {
                     return rc;
                  }
```

```
                } // End for each value
             } // End else array of values
             cout << endl;
          } // End if varray attribute

          else {
             cout << "unrecognized attribute type" << endl;
          }
       } // End else component is attribute
   } // End for all components
   return oocSuccess;
} // End showData
```

---

showNumeric

global function

Prints the data for a numeric value.

```
ooStatus showNumeric (Numeric_Value numVal) {
   // Use the kind of numeric data to determine
   // how to print the value
   ooBaseType bt = numVal.type();
   switch (bt) {
      case ooCHAR: {
         cout << (char)numVal << endl;
         break;
      }
      case ooINT8: {
         cout << (int8)numVal << endl;
         break;
      }
      case ooUINT8: {
         cout << (uint8)numVal << endl;
         break;
      }
      case ooINT16: {
         cout << (int16)numVal << endl;
         break;
      }
      case ooUINT16: {
         cout << (uint16)numVal << endl;
         break;
      }
      case ooINT32: {
         cout << (int32)numVal << endl;
         break;
      }
```

```
        case ooUINT32: {
           cout << (uint32)numVal << endl;
           break;
        }
        case ooINT64: {
           cout << (int64)numVal << endl;
           break;
        }
        case ooUINT64: {
           cout << (uint64)numVal << endl;
           break;
        }
        case ooFLOAT32: {
           cout << (float32)numVal << endl;
           break;
        }
        case ooFLOAT64: {
           cout << (float64)numVal << endl;
           break;
        }
        case ooPTR: {
           cout << "(pointer)" << endl;
           break;
        }
        default: {
           cout << "(unrecognized numeric type)" << endl;
           break;
        }
     } // End switch
} // End showNumeric
```

showRef

global function

Prints an object reference, giving the class name and object ID of the referenced persistent object.

```
void showRef(ooRef(ooObj) ref) {
   if (ref.is_null()) {
      cout << "(null)" << endl;
   }
   else {
      cout << ref.typeName() << " object ";
      cout << ref.sprint() << endl;
   }
} // End showRef
```

---

showRefVArray

global function

Iterates through the elements of a VArray object for an object-reference VArray, printing each element.

```
ooStatus showRefVArray (VArray_Object &VO) {

   // Verify that we have an object-reference VArray.
   if (! VO.type_of().is_ref_type()) {
      cerr << "Not an object-reference VArray" << endl;
      return oocError;
   }

   // Check for null VArray object
   if (! VO) {
      cout << "(null VArray)" << endl;
      return oocSuccess;
   }

   // Get the number of elements in the VArray
   uint32 nVals = VO.size();
   if (nVals == 0) {
      cout << "(empty VArray)" << endl;
      return oocSuccess;
   }

   // Get VArray iterator for elements
   d_Iterator<ooRef(ooObj)> dit =
         (d_Iterator<ooRef(ooObj)> &)VO.create_iterator();
```

```
   // Use the VArray iterator to get each element
   while (dit.not_done()) {
      ooRef(ooObj) ref = dit.get_element();
      showRef(ref);
      ++dit;
   }
} // End showRefVArray
```

showRelationship

global function

Prints the data for a relationship object.

```
ooStatus showRelationship (Relationship_Object &RO) {

   // Check for null relationship object
   if (! RO) {
      cout << "(no associated object)" << endl;
      return oocSuccess;
   }
   // Get relationship descriptor
   const d_Relationship &rel = RO.relationship();

   // Test whether relationship is to-many
   if (rel.is_to_many()) {
      ooItr(ooObj) objItr;
      // Initialize an object iterator for destination objects
      RO.get_iterator(objItr);
      // Iterate through the destination objects
      int n = 0;
      while (objItr.next()) {
         cout << n << ". ";
         // Get handle for this destination object
         ooHandle(ooObj) curObjH(objItr);
         showRef(curObjH);
         ++n;
      } // End while more destination objects
   } // End if to-many
   else { // Relationship is to-one
      // Get object reference for destination object
      ooRef(ooObj) destination = RO.get_ooref();
      showRef(destination);
   } // End else relationship is to-one
   return oocSuccess;
} // End showRelationship
```

showString

global function

Prints the data for a string value.

```
ooStatus showString (String_Value strVal) {
   // Use the kind of string to determine how
   // to print the value
   switch (strVal.type()) {
      case ooAsStringVSTRING: {
         ooVString *vStr = strVal;
         cout << (const char *)(*vStr) << endl;
         break;
      }
      case ooAsStringUTF8: {
         ooUtf8String *utf8Str = strVal;
         cout << (const char *)(*utf8Str) << endl;
         break;
      }
      case ooAsStringOPTIMIZED: {
         Optimized_String_Value optStr(strVal);
         cout << optStr.get_copy() << endl;
         break;
      }
      case ooAsStringST: {
         cout << "(Smalltalk string)" << endl;
         break;
      }
      default: {
         cout << "(unrecognized string class)" << endl;
         break;
      }
   } // End switch kind of string
} // End showString
```

showVArray

global function

Prints the data for a VArray object.

```
ooStatus showVArray (VArray_Object &VO) {

   // Check for null VArray object
   if (! VO) {
      cout << "(null VArray)" << endl;
      return oocSuccess;
   }

   // Get the number of elements in the VArray
   uint32 nVals = VO.size();
   if (nVals > 0) {
      cout << "VArray of " << nVals << "elements" << endl;
   }
   else {
      cout << "(empty VArray)" << endl;
      return oocSuccess;
   }

   // Test the element type to determine how to
   // access the elements
   const d_Type &elemType = VO.type_of();

   if (elemType.is_basic_type()) {
      // Get the numeric elements
      for (size_t n = 0; n < nVals; ++n) {
         cout << n << ". ";
         rc = showNumeric(VO.get(n));
         if (rc != oocSuccess) {
            return rc;
         }
      } // End for each element
      cout << "End of VArray" << endl << endl;
   } // End if numeric VArray

   else if (elemType.is_ref_type()) {
      // Get the object-reference elements
      for (size_t n = 0; n < nVals; ++n) {
         cout << n << ". ";
         ooRef(ooObj) ref = VO.get_ooref(n);
         showRef(ref);
      } // End for each element
      cout << "End of VArray" << endl << endl;
   } // End if object-reference VArray
```

```
   else if (elemType.is_string_type()) {
      // Get the string elements
      for (size_t n = 0; n < nVals; ++n) {
         cout << n << ". ";
         rc = showString(VO.get_string(n));
         if (rc != oocSuccess) {
            return rc;
         }
      } // End for each element
      cout << "End of VArray" << endl << endl;
   } // End if string attribute

   else if (elemType.is_class()) {
      // Get the embedded-instance elements
      const char *name = elemType.name();
      for (size_t n = 0; n < nVals; ++n) {
         cout << n << ". embedded instance of ";
         cout << name << endl;
         rc = showData(VO.get_class_obj(n),
                       oocTrue,
                       "");
         if (rc != oocSuccess) {
            return rc;
         }
      } // End for each element
      cout << "End of VArray" << endl << endl;
   } // End if embedded non-string-class attribute

   else {
      cout << "unrecognized element type" << endl << endl;
   }
   return oocSuccess;
} // End showVArray
```

Glossary

actual iterator. An iterator that steps through its iteration set. The iterator is initialized with the first element of the iteration set as its current element. See also *loop-control iterator*.

application-defined class. A class defined by some application that accesses a particular federated database. See also *internal class*.

attribute ID. A class-specific integer index that identifies a particular attribute, relationship, or base class of the class. Attribute IDs do not change as the class evolves.

attribute of a class. The component data of an instance of the class, including attributes defined by the class and attributes inherited from base classes. Attributes correspond to standard data members of a C++ class, fields of a Java class, and instance variables of a Smalltalk class.

attribute position. An integer that identifies the position of the attribute's data within the data of an instance of the class that defines the attribute. Every attribute, relationship, and base class of a class has an attribute position; a base class is considered equivalent to an embedded-class attribute.

attribute type. A data type for attributes. An attribute's type specifies the kind of data that can be stored in the attribute.

basic numeric type. An attribute type for values of a particular fundamental character, integer, floating-point, or pointer type.

bidirectional relationship. A relationship that has an inverse relationship.

cardinality of a relationship. Characteristic of a relationship that indicates how many destination objects the relationship can associate with a single source object: either to-one or to-many.

class position. A class-specific sequence of attribute positions that gives the position of an attribute's data within the data of a particular class that defines or inherits that attribute. The class position indicates nesting of attributes within data inherited from each ancestor class.

component of a class. The immediate base classes of the class, and the attributes and relationships defined by the class.

current element of an iterator. The element in the iterator's current position in its iteration set.

current position of an iterator. The iterator's position within its iteration set.

descriptor. An object containing information about some entity (a module, type, or property in the schema), or about a proposed class and its properties.

destination class of a relationship. The class to which instances of the relationship's source class are associated by the relationship.

destination object in a relationship. An instance of the destination class of a relationship that is associated by the relationship to an instance of the source class (called the source object for this destination object).

directionality of a relationship. Characteristic of a relationship that indicates whether it has an inverse relationship. A bidirectional relationship has an inverse; a unidirectional relationship does not.

embedded-class type. An attribute type for values that are instances of a specified non-persistence-capable class embedded within the data of the instance containing the attribute.

evolution of a class. Change over time in the definition of a class. Each definition in the evolution process is a different shape for the class. If the federated database contains different versions of a class, each version can evolve independently.

fixed-size array. A collection of a specified number of values of a particular attribute type that constitutes the data for an attribute.

internal class. A class defined by Objectivity / DB. See also *application-defined class*.

iteration set. The group of items through which an iterator steps.

iterator. An object that provides the ability to step through a group of items.

inverse relationships. A pair of relationships that together define a bidirectional link between classes. The source class of one relationship is the destination class of the other relationship.

loop-control iterator. An iterator that represents the termination condition for an associated actual iterator. The iterator is initialized with its current position after the last element in its iteration set. See also *actual iterator*.

module. One disjoint portion of the schema of a federated database.

named module. An application-defined non-top-level module, which is identified by its name.

non-persistence-capable class. A class whose instances cannot be saved independently in a federated database, but which can be embedded within the data of instances of a persistence-capable class. Such an embedded instance cannot be retrieved independently; it can be retrieved only as part of the data of the containing instance.

null descriptor. A descriptor with no corresponding described entity—that is, a descriptor that describes nothing.

object-reference type. An attribute type for values that are references to instances of a specified persistence-capable class.

persistence-capable class. A class whose instances can be saved independently in a federated database and retrieved independently. Within the federated database, every instance of a persistence-capable class is identified by a unique object identifier (OID).

persistent static properties of a class. Properties of a persistent object stored with a class description to represent information about the described class itself (as opposed to information about instances of the class).

property of a class. The attributes and relationships defined on a class, which represent information about the instances of the class. See also *attribute ID* and *relationship of a class*.

proposal list of a module descriptor. List of proposed classes representing new classes to be added to the described module or evolved definitions of existing application-defined classes in the module.

relationship of a class. A directional association from the defining class (or source class) to a destination class. The destination class can be any persistence-capable class, including the source class itself.

relationship type. A data type for relationships. A relationship's type specifies its directionality (unidirectional or bidirectional) and its destination class.

schema. A collection of descriptions of all types and classes used in a particular federated database.

schema evolution. Modification to the description of a class in the schema of a federated database.

scope. An entity in the federated database schema that organizes the other entities. Modules and classes serve as scopes. A module is the scope for the types and classes whose descriptions it contains; a class is the scope for the properties it defines.

shape number. A unique number that identifies a particular shape of a particular version of a particular class in the schema of a federated database.

shape of a class. The physical storage layout for an instance of a class. As the definition of a class evolves, each new definition is a different shape for the class, identified by a unique shape number. If the federated database contains different versions of a class, each version can evolve independently, so each version can have its own shape.

source class of a relationship. The class that defines the relationship. Instances of the source class are associated by the relationship to instances of the relationship's destination class.

source object in a relationship. An instance of the source class of a relationship that is associated by the relationship to one or more instances of the destination class (called the destination objects for this source object).

top-level module. The initial module of a federated database schema, which contains descriptions of all non-class types and all internal classes.

type number. A unique number that identifies a particular type, class, or version of a class in the schema of a federated database.

unidirectional relationship. A relationship that does not have an inverse relationship.

variable-size array. See *VArray*.

variable-size array type. See *VArray type*.

VArray. A one-dimensional variable-size array of elements of the same type. The element type can be a basic numeric type, an object-reference type, or an embedded class type.

VArray type. An attribute type for values that are VArrays of a specified element type.

version of a class. A distinct type created by the Objectivity / C++ class-versioning feature from the original definition of the class. The federated database may contain instances of each version of a given class.

Topic Index

This index lists topics that are discussed in this book. For a list of classes, see "Classes Index" on page 537; for a list of functions, including member functions, see "Functions Index" on page 547; for a list of non-class types and constants, see "Types and Constants Index" on page 559.

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

Classes Index

This index contains an alphabetical list of classes, with member functions listed under each class. For an alphabetical list of all functions, including member functions, see “Functions Index” on page 547; for a list of non-class types and constants, see “Types and Constants Index” on page 559.

A

B

C

D

O

P

R

S

T

U

V

W

Functions Index

This index contains an alphabetical list of all functions, including member functions. For an alphabetical list of classes, with member functions listed under each class, see “Classes Index” on page 537.

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

Types and Constants Index

This index lists the non-class types and constants in the Active Schema programming interface. For a list of classes, see “Classes Index” on page 537.

O

P

R

S

U

V