Naming Service Specification 3

3.1 Service Description

3.1.1 Overview

A name-to-object association is called a *name binding*. A name binding is always defined relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different contexts at the same time. There is no requirement, however, that all objects must be named.

To *resolve a name* is to determine the object associated with the name in a given context. To *bind a name* is to create a name binding in a given context. A name is always resolved relative to a context — there are no absolute names.

Because a context is like any other object, it can also be bound to a name in a naming context. Binding contexts in other contexts creates a *naming graph* — a directed graph with nodes and labeled edges where the nodes are contexts. A naming graph allows more complex names to reference an object. Given a context in a naming graph, a sequence of names can reference an object. This sequence of names (called a *compound name*) defines a path in the naming graph to navigate the resolution process. Figure 3-1 shows an example of a naming graph.

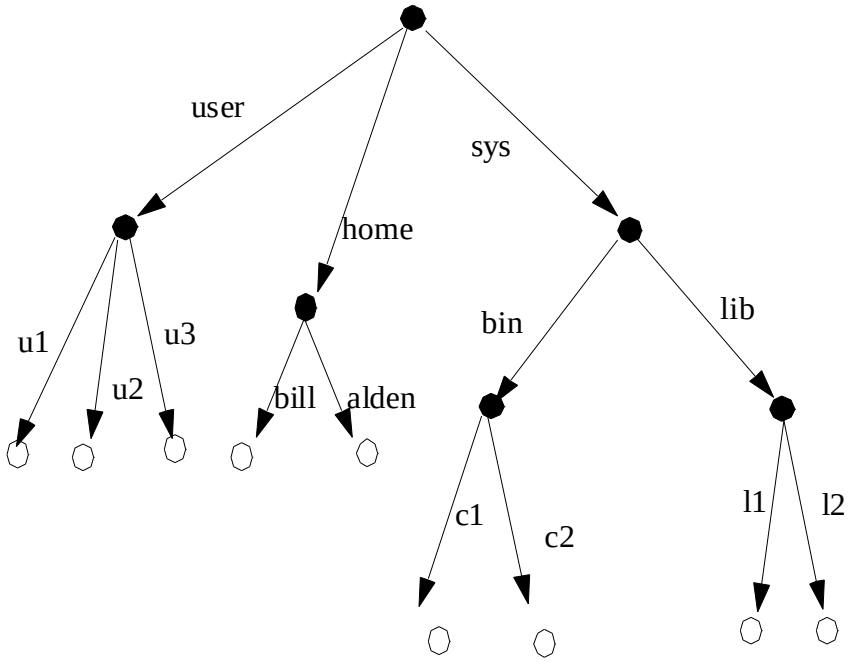

Figure 3-1 A Naming Graph

3.1.2 Names

Many of the operations defined on a naming context take names as parameters. Names have structure. A name is an ordered sequence of *components*.

A name with a single component is called a *simple name*; a name with multiple components is called a *compound name*. Each component except the last is used to name a context; the last component denotes the bound object. The notation:

```
< component1 ; component2 ; component3 >
```

indicates the sequences of components.

Note – The semicolon (;) characters are simply the notation used in this document and are not intended to imply that names are sequences of characters separated by semicolon.

A name component consists of two attributes: the *identifier attribute* and the *kind attribute*. Both the `identifier` attribute and the `kind` attribute are represented as IDL strings.

The `kind` attribute adds descriptive power to names in a syntax-independent way. Examples of the value of the `kind` attribute include *c_source*, *object_code*, *executable*, *postscript*, or " ". The naming system does not interpret, assign, or manage

these values in any way. Higher levels of software may make policies about the use and management of these values. This feature addresses the needs of applications that use syntactic naming conventions to distinguish related objects. For example Unix uses suffixes such as `.c` and `.o`. Applications (such as the C compiler) depend on these syntactic convention to make name transformations (for example, to transform `foo.c` to `foo.o`).

The lack of name syntax is especially important when considering internationalization issues. Software that does not depend on the syntactic conventions for names does not have to be changed when it is localized for a natural language that has different syntactic conventions —unlike software that does depend on the syntactic conventions (which must be changed to adopt to new conventions).

To avoid issues of differing name syntax, the Naming Service always deals with names in their structural form (that is, there are no canonical syntaxes or distinguished meta characters). It is assumed that various programs and system services will map names from the representation into the structural form in a manner that is convenient to them.

3.1.3 Names Library

To allow the representation of names to evolve without affecting existing clients, it is desirable to hide the representation from client code. Ideally, names themselves would be OMG IDL objects; however, names must be lightweight entities that can be very efficiently created and manipulated in memory and passed as parameters in requests by value. In order to simplify name manipulation and provide representation independence, names can be presented to programs through the names library. Note, however, it is not necessary to use the names library to use the basic operations of the naming service.

The names library implements names as pseudo-objects. A client makes calls on a pseudo-object in the same way it makes calls on an ordinary object. Library names are described in pseudo-IDL. The names library supports two pseudo-IDL interfaces: the *LNameComponent* interface and the *LName* interface. The *LNameComponent* interface defines the get and set operations associated with name component `identifier` and the `kind` attributes.The *LName* Interface includes operations for manipulating library name and library name component pseudo objects and producing and translating a structure that can be passed as a parameter to a normal object request.

3.1.4 Example Scenarios

This section provides two short scenarios that illustrate how the naming service specification can be used by two fairly different kinds of systems -- systems that differ in the kind of implementations used to build the Naming Service and that differ in models of how clients might use the Naming Service with other object services to locate objects.

In one system, the Naming Service is implemented using an underlying enterprise-wide naming server such as DCE CDS. The Naming Service is used to construct large, enterprise-wide naming graphs where NamingContexts model "directories" or "folders" and other names identify "document" or "file" kinds of objects. In other words, the

naming service is used as the backbone of an enterprise-wide filing system. In such a system, non-object-based access to the naming service may well be as commonplace as object-based access to the naming service. For example, the name of an object might be presented to the DCE directory service as a null-terminated ASCII string such as “/.../DME/nls/moa-1/ID-1”.

The Naming Service provides the principal mechanism through which most clients of an ORB-based system locate objects that they intend to use (make requests of). Given an initial naming context, clients navigate naming contexts retrieving lists of the names bound to that context. In conjunction with properties and security services, clients look for objects with certain "externally visible" characteristics, for example, for objects with recognized names or objects with a certain time-last-modified (all subject to security considerations). All objects used in such a scheme register their externally visible characteristics with other services (a name service, a properties service, and so on).

Conventions are employed in such a scheme that meaningfully partition the name space. For example, individuals are assigned naming contexts for personal use, groups of individuals may be assigned shared naming contexts while other contexts are organized in a public section of the naming graph. Similarly, conventions are used to identify contexts that list the names of services that are available in the system (e.g., that locate a translation or printing service).

In an alternative system, the Naming Service can be used in a more limited role and can have a less sophisticated implementation. In this model, naming contexts represent the types and locations of services that are available in the system and a much shallower naming graph is employed. For example, the Naming Service is used to register the object references of a mail service, an information service, a filing service.

Given a handful of references to "root objects" obtained from the Naming Service, a client uses the Relationship and Query Services to locate objects contained in or managed by the services registered with the Naming Service. In such a system, the Naming Service is used sparingly and instead clients rely on other services such as query services to navigate through large collections of objects. Also, objects in this scheme rarely register "external characteristics" with another service - instead they support the interfaces of Query or Relationship Services.

Of course, nothing precludes the Naming Service presented here from being used to provide both models of use at the same time. These two scenarios demonstrate how this specification is suitable for use in two fairly different kinds of systems with potentially quite different kinds of implementations. The service provides a basic building block on which higher-level services impose the conventions and semantics which determine how frameworks of application and facilities objects locate other objects.

3.1.5 Design Principles

Several principles have driven the design of the Naming Service:

1. The design imparts no semantics or interpretation of the names themselves; this is up to higher-level software. The naming service provides only a structural convention for names, e.g. compound names.

2. The design supports distributed, heterogeneous implementation and administration of names and name contexts.

3. Names are structures, not just character strings. A `struct` is necessary to avoid encoding information syntactically in the name string (e.g., separating the human-meaningful name and its type with a ".", and the type and version with a "!"), which is a bad idea with respect to the generality, extensibility, and internationalization of the name service. The structure define includes a human-chosen string plus a kind field.

4. Naming service clients need not be aware of the physical site of name servers in a distributed environment, or which server interprets what portion of a compound name, or of the way that servers are implemented.

5. The Naming Service is a fundamental object service, with no dependencies on other interfaces.

6. Name contexts of arbitrary and unknown implementation may be utilized together as nested graphs of nodes that cooperate in resolving names for a client. No "universal" root is needed for a name hierarchy.

7. Existing name and directory services employed in different network computing environments can be transparently encapsulated using name contexts. All of the above features contribute to making this possible.

8. The design does not address name security since there is currently no OMG security model. The Naming Service can be evolved to provide name security when an object security service is standardized.

9. The design does not address namespace administration. It is the responsibility of higher-level software to administer the namespace.

3.1.6 Resolution of Technical Issues

This specification addresses the issues identified for a name service in the OMG *Object Services Architecture* document[1] as follows:

- *Naming standards*: Encapsulation of existing naming standards and protocols is allowed using naming contexts. Transparent encapsulation is made possible by the design features outlined above.

1. *Object Services Architecture*, Document Number 92-8-4, Object Managment Group, Framingham, MA, 1992.

- *Federation of namespaces*: The specification supports distributed federation of namespaces; no assumptions are made about centralized or universal functions. Namespaces may be nested in a graph in any fashion, independent of the implementation of each namespace. There need be no distinguished root context, and existing graphs may be joined at any point.
- *Scope of names*: Name contexts define name scope. Names must be unique within a context. Objects may have multiple names, and may exist in multiple name contexts. Name contexts may be named objects nested within another name context, and cycles are permitted. The name itself is not a full-fledged ORB object, but does support structure, so it may have multiple components. No requirements are placed on naming conventions, in order to support a wide variety of conventions and existing standards.
- *Operations*: The specification supports bind, unbind, lookup, and sequence operations on a name context. It does not support a rename operation, because we do not see how to implement this correctly in a distributed environment without transactions.

3.2 The CosNaming Module

The CosNaming Module is a collection of interfaces that together define the naming service. This module contains two interfaces:

- The *NamingContext* interface
- The *BindingIterator* interface

This section describes these interfaces and their operations in detail.

The CosNaming Module is shown in Figure 3-2. Note that `Istring` is a placeholder for a future IDL internationalized string data type.

```
module CosNaming
{
    typedef string Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };

    typedef sequence <NameComponent> Name;

    enum BindingType {nobject, ncontext};

    struct Binding {
        Name binding_name;
        BindingType binding_type;
```

Figure 3-2 The CosNaming Module

```
};

typedef sequence <Binding> BindingList;


interface BindingIterator;

interface NamingContext {

    enum NotFoundReason { missing_node, not_context, not_object};

    exception NotFound {
            NotFoundReason why;
            Name rest_of_name;
    };

    exception CannotProceed {
            NamingContext cxt;
            Name rest_of_name;
    };

    exception InvalidName{};
    exception AlreadyBound {};
    exception NotEmpty{};

    void bind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    void rebind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName);
    void bind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    void rebind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName);
    Object resolve (in Name n)
         raises(NotFound, CannotProceed, InvalidName);
    void unbind(in Name n)
         raises(NotFound, CannotProceed, InvalidName);
    NamingContext new_context();
    NamingContext bind_new_context(in Name n)
         raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
    void destroy( )
         raises(NotEmpty);
    void list (in unsigned long how_many,
               out BindingList bl, out BindingIterator bi);
};

interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
```

Figure 3-2 The CosNaming Module *(Continued)*

```
                    out BindingList bl);
        void destroy();
    };
};
```

Figure 3-2 The CosNaming Module *(Continued)*

The following sections describe the operations of the *NamingContext* interface:

- binding objects
- name resolution
- unbinding
- creating naming contexts
- deleting contexts
- listing a naming context

3.2.1 Binding Objects

The binding operations name an object in a naming context. Once an object is bound, it can be found with the `resolve` operation. The Naming Service supports four operations to create bindings: *bind*, *rebind*, *bind_context* and *rebind_context*.

```
void bind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName);
void bind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName);
```

bind

Creates a binding of a name and an object in the naming context. Naming contexts that are bound using `bind` do not participate in name resolution when compound names are passed to be resolved.

A `bind` operation that is passed a compound name is defined as follows:

ctx->bind(< c1 ; c2 ; ... ; cn >, obj) ≡
(ctx->resolve(< c1 ; c2 ; ... ; cn-1 >))->bind(< cn >, obj)

rebind

Creates a binding of a name and an object in the naming context even if the name is already bound in the context. Naming contexts that are bound using `rebind` do not participate in name resolution when compound names are passed to be resolved.

bind_context
: Names an object that is a naming context. Naming contexts that are bound using `bind_context`() participate in name resolution when compound names are passed to be resolved.

  A `bind_context` operation that is passed a compound name is defined as follows:

  ctx->bind_context(< c1 ; c2 ; ... ; cn >, nc) ≡
  (ctx->resolve(< c1 ; c2 ; ... ; cn-1 >))->bind_context(< cn >, nc)

rebind_context
: Creates a binding of a name and a naming context in the naming context even if the name is already bound in the context. Naming contexts that are bound using `rebind_context`() participate in name resolution when compound names are passed to be resolved.

Table 3-1 describes the exceptions raised by the binding operations.

Table 3-1 Exceptions Raised by Binding Operations

Exception Raised	Description
NotFound	Indicates the name does not identify a binding.
CannotProceed	Indicates that the implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.
InvalidName	Indicates the name is invalid. (A name of length 0 is invalid; implementations may place other restrictions on names.)
AlreadyBound	Indicates an object is already bound to the specified name. Only one object can be bound to a particular name in a context. The `bind` and the `bind_context` operations raise the `AlreadyBound` exception if the name is bound in the context; the `rebind` and rebind_context operations unbind the name and rebind the name to the object passed as an argument.

3.2.2 Resolving Names

The `resolve` operation is the process of retrieving an object bound to a name in a given context. The given name must exactly match the bound name. The naming service does not return the type of the object. Clients are responsible for "narrowing" the object to the appropriate type. That is, clients typically cast the returned object from Object to a more specialized interface. The OMG IDL definition of the `resolve` operation is:

```
Object resolve (in Name n)
     raises (NotFound, CannotProceed, InvalidName);
```

Names can have multiple components; therefore, name resolution can traverse multiple contexts. A compound resolve is defined as follows:

ctx->resolve(< c1 ; c2 ; ... ; cn >) ≡
ctx->resolve(< c1 ; c2 ; ... ; cn-1 >)->resolve(< cn >)

Table 3-2 describes the exceptions raised by the `resolve` operation.

Table 3-2 Exceptions Raised by Resolve Operation

Exception Raised	Description
NotFound	Indicates the name does not identify a binding.
CannotProceed	Indicates that the implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.
InvalidName	Indicates the name is invalid. (A name of length 0 is invalid; implementations may place other restrictions on names.)

3.2.3 Unbinding Names

The `unbind` operation removes a name binding from a context. The definition of the `unbind` operation is:

```
void unbind(in Name n)
     raises (NotFound, CannotProceed, InvalidName);
```

A `unbind` operation that is passed a compound name is defined as follows:

ctx->unbind(< c1 ; c2 ; ... ; cn >) ≡
(ctx->resolve(< c1 ; c2 ; ... ; cn-1 >))->unbind(< cn >)

Table 3-3 describes the exceptions raised by the `unbind` operation.

Table 3-3 Exceptions Raised by Unbind Operation

Exception Raised	Description
NotFound	Indicates the name does not identify a binding.
CannotProceed	Indicates that the implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.
InvalidName	Indicates the name is invalid. (A name of length 0 is invalid; implementations may place other restrictions on names.)

3.2.4 Creating Naming Contexts

The Naming Service supports two operations to create new contexts: *new_context* and *bind_new_context.*

```
NamingContext new_context();

NamingContext bind_new_context(in Name n)
raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
```

new_context
: This operation returns a naming context implemented by the same naming server as the context on which the operation was invoked. The new context is not bound to any name.

bind_new_context
: This operation creates a new context and binds it to the name supplied as an argument. The newly-created context is implemented by the same naming server as the context in which it was bound (that is, the naming server that implements the context denoted by the name argument excluding the last component).

  A `bind_new_context` that is passed a compound name is defined as follows:

  ctx->bind_new_context(< c1 ; c2 ; ... ; cn >) ≡
  (ctx->resolve(< c1 ; c2 ; ... ; cn-1 >))->bind_new_context(< cn >)

Table 3-4 describes the exceptions raised when new contexts are being created.

Table 3-4 Exceptions Raised by Creating New Contexts

Exception Raised	Description
NotFound	Indicates the name does not identify a binding.
CannotProceed	Indicates that the implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.
InvalidName	Indicates the name is invalid. (A name of length 0 is invalid; implementations may place other restrictions on names.)
AlreadyBound	Indicates an object is already bound to the specified name. Only one object can be bound to a particular name in a context.

3.2.5 Deleting Contexts

The `destroy` operation deletes a naming context:.

```
void destroy()
    raises(NotEmpty);
```

If the naming context contains bindings, the `NotEmpty` exception is raised.

3.2.6 Listing a Naming Context

The `list` operation allows a client to iterate through a set of bindings in a naming context.

```
enum BindingType {object, ncontext};

struct Binding {
    Name    binding_name;
    BindingType binding_type;
};

typedef sequence <Binding> BindingList;

void list (in unsigned long how_many,
               out BindingList bl, out BindingIterator bi);
};
```

The `list` operation returns at most the requested number of bindings in `BindingList bl`.

- If the naming context contains additional bindings, the `list` operation returns a `BindingIterator` with the additional bindings.
- If the naming context does not contain additional bindings, the binding iterator is a nil object reference.

3.2.7 The BindingIterator Interface

The *BindingIterator* interface allows a client to iterate through the bindings using the next_one or next_n operations:

```
interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
               out BindingList bl);
    void destroy();
};
```

next_one
: This operation returns the next binding. If there are no more bindings, *false* is returned.

next_n
: This operation returns at most the requested number of bindings.

destroy
This operation destroys the iterator.

3.3 The Names Library

To allow the representation of names to evolve without affecting existing clients, it is desirable to hide the representation of names from client code. Ideally, names themselves would be objects; however, names must be lightweight entities that are efficient to create, manipulate, and transmit. As such, names are presented to programs through the *names library*.

The names library implements names as *pseudo-objects*. A client makes calls on a pseudo-object in the same way it makes calls on an ordinary object. Library names are described in pseudo-IDL (to suggest the appropriate language binding). C and C++ clients[2] use the same client language bindings for pseudo-IDL (PIDL) as they use for IDL.

Pseudo-object references cannot be passed across OMG IDL interfaces. As described in Section 3.2, "The CosNaming Module," the naming service supports the *NamingContext* OMG IDL interface. The names library supports an operation to convert a library name into a value that can be passed to the name service through the *NamingContext* interface.

Note – It is not a requirement to use the names library in order to use the Naming Service.

The names library consists of two pseudo-IDL interfaces: the *LNameComponent* interface and the *LName* interface, as shown in Figure 3-3.

2. As anticipated

```
interface LNameComponent {    // PIDL
    exception NotSet{};
    string get_id()
        raises(NotSet);
    void set_id(in string i);
    string get_kind()
        raises(NotSet);
    void set_kind(in string k);
    void destroy();
};

interface LName {                              // PIDL
    exception NoComponent{};
    exception OverFlow{};
    exception InvalidName{};
    LName insert_component(in unsigned long i,
                   in LNameComponent n)
            raises(NoComponent, OverFlow);
    LNameComponent get_component(in unsigned long i)
         raises(NoComponent);
    LNameComponent delete_component(in unsigned long i)
         raises(NoComponent);
    unsigned long num_components();
    boolean equal(in LName ln);
    boolean less_than(in LName ln);
    Name to_idl_form()
           raises(InvalidName);
    void from_idl_form(in Name n);
    void destroy();
};

LName create_lname();                          // C/C++
LNameComponent create_lname_component(); // C/C++
```

Figure 3-3 The Names Library Interface in PIDL

3.3.1 Creating a Library Name Component

To create a library name component pseudo-object, use the following C/C++ function:

```
LNameComponent create_lname_component();     // C/C++
```

The returned pseudo-object can then be operated on using the operations in Figure 3-3.

3.3.2 Creating a Library Name

To create a library name pseudo-object, use the following C/C++ function.

```
LName create_lname();                          // C/C++
```

The returned pseudo-object reference can then be operated on using the operations in Figure 3-3.

3.3.3 The LNameComponent Interface

A name component consists of two attributes: the `identifier` attribute and the `kind` attribute. The *LNameComponent* interface defines the operations associated with these attributes.

```
string get_id()
    raises(NotSet);
void set_id(in string k);
string get_kind()
    raises(NotSet);
void set_kind(in string k);
```

get_id
: The `get_id` operation returns the `identifier` attribute's value. If the attribute has not been set, the `NotSet` exception is raised.

set_id
: The `set_id` operation sets the `identifier` attribute to the string argument.

get_kind
: The `get_kind` operation returns the `kind` attribute's value. If the attribute has not been set, the `NotSet` exception is raised.

set_kind
: The `set_kind` operation sets the `kind` attribute to the string argument.

3.3.4 The LName Interface

The following operations are described in this section:

- destroying a library name component pseudo object
- creating a library name
- inserting a name component
- getting the i^{th} name component
- deleting a name component
- number of name components

- testing for equality
- testing for order
- producing an idl form
- translating an idl form
- destroying a library name pseudo object

Destroying a Library Name Component Pseudo Object

The `destroy` operation destroys library name component pseudo-objects.

```
void destroy();
```

Inserting a Name Component

A name has one or more components. Each component except the last is used to identify names of subcontexts. (The last component denotes the bound object.) The `insert_component` operation inserts a component after position *i*.

```
LName insert_component(in unsigned long i, in LNameComponent lnc)
    raises(NoComponent, OverFlow);
```

If component *i*-1 is undefined and component *i* is greater than 1, the `insert_component` operation raises the `NoComponent` exception.

If the library cannot allocate resources for the inserted component, the `Overflow` exception is raised.

Getting the i^{th} Name Component

The `get_component` operation returns the i^{th} component. The first component is numbered *1*.

```
LNameComponent get_component(in unsigned long i)
    raises(NoComponent);
```

If the component does not exist, the `NoComponent` exception is raised.

Deleting a Name Component

The `delete_component` operation removes and returns the i^{th} component.

```
LNameComponent delete_component(in unsigned long i)
    raises(NoComponent);
```

If the component does not exist, the `NoComponent` exception is raised.

After a `delete_component` operation has been performed, the compound name has one fewer component and components previously identified as *i+1...n* are now identified as *i...n-1*.

Number of Name Components

The `num_components` operation returns the number of components in a library name.

```
unsigned long num_components();
```

Testing for Equality

The `equal` operation tests for equality with library name ln.

```
boolean equal(in LName ln);
```

Testing for Order

The `less_than` operation tests for the order of a library name in relation to library name ln.

```
boolean less_than(in LName ln);
```

This operation returns *true* if the library name is less than the library name ln passed as an argument. The library implementation defines the ordering on names.

Producing an IDL form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo object; therefore, it cannot be passed across the IDL interface for the naming service. Several operations in the *NamingContext* interface have arguments of an IDL-defined structure, `Name`. The following PIDL operation on library names produces a structure that can be passed across the IDL request.

```
Name to_idl_form()
    raises(InvalidName);
```

If the name is of length 0, the `InvalidName` exception is returned.

Translating an IDL Form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo object; therefore, it cannot be passed across the IDL interface for the naming service. The *NamingContext* interface defines operations that return an IDL struct of type `Name`. The following PIDL operation on library names sets the components and `kind` attribute for a library name from a returned IDL defined structure, `Name`.

```
void from_idl_form(in Name n);
```

Destroying a Library Name Pseudo-Object

The `destroy` operation destroys library name pseudo-objects

```
void destroy();
```