ORB Interface 4

Contents

This chapter contains the following sections.

4.1 Overview

The ORB interface is the interface to those ORB functions that do not depend on which object adapter is used. These operations are the same for all ORBs and all object implementations, and can be performed either by clients of the objects or implementations. Some of these operations appear to be on the ORB, others appear to be on the object reference. Because the operations in this section are implemented by the ORB itself, they are not in fact operations on objects, although they may be described that way and the language binding will, for consistency, make them appear

that way. The ORB interface also defines operations for creating lists and determining the default context used in the Dynamic Invocation Interface. Those operations are described in the Dynamic Invocation Interface chapter.

```
module CORBA {
typedef unsigned short ServiceType;
typedef unsigned long ServiceOption;
typedef unsigned long ServiceDetailType;

const ServiceType Security = 1;

struct ServiceDetail {
ServiceDetailType  service_detail_type;
sequence <octet>  service_detail;
};

struct ServiceInformation {
sequence <ServiceOption>  service_options;
sequence <ServiceDetail>  service_details;
};

interface ORB {                                                    // PIDL
    string object_to_string (in Object obj);
    Object  string_to_object (in string str);

    Status  create_list (
    in long              count,
    out NVList           new_list
);
    Status   create_operation_list (
    in OperationDef    oper,
    out NVList           new_list
);

Status get_default_context (out Context ctx);
boolean get_service_information (
in ServiceType  service_type;
out ServiceInformation  service_information;
);
// get_current deprecated operation - should not be used by new code
// new code should use resolve_initial_reference operation instead
Current get_current();
    };
};
```

All types defined in this chapter are part of the CORBA module. When referenced in OMG IDL, the type names must be prefixed by “CORBA::”.

The **get_current** operation is described in “Thread-related operations” on page 4-19.

4.1.1 Converting Object References to Strings

Because an object reference is opaque and may differ from ORB to ORB, the object reference itself is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. Two problems must be solved: allowing an object reference to be turned into a value that a client can store in some other medium, and ensuring that the value can subsequently be turned into the appropriate object reference.

An object reference may be translated into a string by the operation **object_to_string**. The value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the **string_to_object** operation will accept a string produced by **object_to_string** and return the corresponding object reference.

To guarantee that an ORB will understand the string form of an object reference, that ORB’s **object_to_string** operation must be used to produce the string. For all conforming ORBs, if obj is a valid reference to an object, then **`string_to_object(object_to_string(obj))`** will return a valid reference to the same object, if the two operations are performed on the same ORB. For all conforming ORB's supporting IOP, this remains true even if the two operations are performed on different ORBs.

For a description of the **create_list** and **create_operation_list** operations, see “List Operations” on page 5-11. The **get_default_context** operation is described in the section “get_default_context” on page 5-15.

4.1.2 Getting Service Information

get_service_information

```
boolean get_service_information (
    in ServiceType  service_type;
    out ServiceInformation  service_information;
);
```

The **get_service_information** operation is used to obtain information about CORBA facilities and services that are supported by this ORB. The service type for which information is being requested is passed in as the in parameter **service_type**, the values defined by constants in the **CORBA** module. If service information is available for that type, that is returned in the out parameter **service_information**, and the operation returns the value **TRUE**. If no information for the requested services type is available, the operation returns **FALSE** (i.e., the service is not supported by this ORB).

4.2 Object Reference Operations

There are some operations that can be done on any object. These are not operations in the normal sense, in that they are implemented directly by the ORB, not passed on to the object implementation. We will describe these as being operations on the object reference, although the interfaces actually depend on the language binding. As above, where we used interface Object to represent the object reference, we will define an interface for Object:

```
module CORBA {

interface Object {                                          // PIDL
    ImplementationDef get_implementation (); //deprecated as of 2.2
    InterfaceDef get_interface ();
    boolean is_nil();
    Object duplicate ();
    void            release ();
    boolean             is_a (in string logical_type_id);
    boolean             non_existent();
    boolean             is_equivalent (in Object other_object);
    unsigned            long hash(in unsigned long maximum);

    Status create_request (
        in Context          ctx,
        in Identifier       operation,
        in NVList           arg_list,
        inout NamedValue    result,
        out Request         request,
        in Flags            req_flags
    );
    Policy get_policy (
    in PolicyType       policy_type
    );
    DomainManagersList get_domain_managers ();
    };
};
```

The **create_request** operation is part of the Object interface because it creates a pseudo-object (a Request) for an object. It is described with the other Request operations in the section Section 5.2, “Request Operations,” on page 5-5.

4.2.1 Determining the Object Interface

Note – The **get_implementation** operation is deprecated in this version of the CORBA specification. No new code should make use of this interface and operation, since they will be eliminated in a future version of the CORBA specification.

An operation on the object reference, **get_interface**, returns an object in the Interface Repository, which provides type information that may be useful to a program. See the Interface Repository chapter for a definition of operations on the Interface Repository. An operation on the Object called **get_implementation** will return an object in an implementation repository that describes the implementation of the object.

```
InterfaceDef get_interface ();                    // PIDL
    ImplementationDef get_implementation ();
```

4.2.2 Duplicating and Releasing Copies of Object References

Because object references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them. Therefore, there are operations defined to copy or release an object reference.

```
Object duplicate ();                              // PIDL
    void release ();
```

If more than one copy of an object reference is needed, the client may create a **duplicate**. Note that the object implementation is not involved in creating the duplicate, and that the implementation cannot distinguish whether the original or a duplicate was used in a particular request.

When an object reference is no longer needed by a program, its storage may be reclaimed by use of the **release** operation. Note that the object implementation is not involved, and that neither the object itself nor any other references to it are affected by the **release** operation.

4.2.3 Nil Object References

An object reference whose value is OBJECT_NIL denotes no object. An object reference can be tested for this value by the **is_nil** operation. The object implementation is not involved in the nil test.

```
boolean is_nil ();                                // PIDL
```

4.2.4 Equivalence Checking Operation

An operation is defined to facilitate maintaining type-safety for object references over the scope of an ORB.

```
boolean is_a(in RepositoryID logical_type_id);    // PIDL
```

The **logical_type_id** is a string denoting a shared type identifier (RepositoryId). The operation returns true if the object is really an instance of that type, including if that type is an ancestor of the “most derived” type of that object.

This operation exposes to application programmers functionality that must already exist in ORBs which support “type safe narrow” and allows programmers working in environments that do not have compile time type checking to explicitly maintain type safety.

4.2.5 Probing for Object Non-Existence

```
boolean                 non_existent ();                    // PIDL
```

The **non_existent** operation may be used to test whether an object (e.g., a proxy object) has been destroyed. It does this without invoking any application level operation on the object, and so will never affect the object itself. It returns true (rather than raising CORBA::OBJECT_NOT_EXIST) if the ORB knows authoritatively that the object does not exist; otherwise, it returns false.

Services that maintain state that includes object references, such as bridges, event channels, and base relationship services, might use this operation in their “idle time” to sift through object tables for objects that no longer exist, deleting them as they go, as a form of garbage collection. In the case of proxies, this kind of activity can cascade, such that cleaning up one table allows others then to be cleaned up.

4.2.6 Object Reference Identity

In order to efficiently manage state that include large numbers of object references, services need to support a notion of object reference identity. Such services include not just bridges, but relationship services and other layered facilities.

```
unsigned long hash(in unsigned long maximum);          // PIDL
    boolean is_equivalent(in Object other_object);
```

Two identity-related operations are provided. One maps object references into disjoint groups of potentially equivalent references, and the other supports more expensive pairwise equivalence testing. Together, these operations support efficient maintenance and search of tables keyed by object references.

Hashing: Object Identifiers

Object references are associated with ORB-internal identifiers which may indirectly be accessed by applications using the **hash()** operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are *not* identical.

The **maximum** parameter to the **hash** operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision chained hash table of object references, the more randomly distributed the values are within that range, and the cheaper those values are to compute, the better.

For bridge construction, note that proxy objects are themselves objects, so there could be many proxy objects representing a given "real" object. Those proxies would not necessarily hash to the same value.

Equivalence Testing

The **is_equivalent()** operation is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns TRUE if the target object reference is known to be equivalent to the other object reference passed as its parameter, and FALSE otherwise.

If two object references are identical, they are equivalent. Two different object references which in fact refer to the same object are also equivalent.

ORBs are allowed, but not required, to attempt determination of whether two distinct object references refer to the same object. In general, the existence of reference translation and encapsulation, in the absence of an omniscient topology service, can make such determination impractically expensive. This means that a FALSE return from **is_equivalent()** should be viewed as only indicating that the object references are distinct, and not necessarily an indication that the references indicate distinct objects.

A typical application use of this operation is to match object references in a hash table. Bridges could use it to shorten the lengths of chains of proxy object references. Externalization services could use it to "flatten" graphs that represent cyclical relationships between objects. Some might do this as they construct the table, others during idle time.

4.2.7 Getting Policy Associated with the Object

The **get_policy** operation returns the policy object of the specified type (see "Policy Object" on page 4-12), which applies to this object.

```
Policy get_policy (
    in PolicyType       policy_type
);
```

Parameters

policy_type	The type of policy to be obtained.

Return Value

policy	A policy object of the type specified by the **policy_type** parameter.

Exceptions

CORBA::BAD_PARAM	raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

4.2.8 Getting the Domain Managers Associated with the Object

The **get_domain_managers** allows administration services (and applications) to retrieve the domain managers (see "Management of Policy Domains" on page 4-14), and hence the security and other policies applicable to individual objects that are members of the domain.

```
DomainManagersList get_domain_managers ();
```

Return Value

The list of immediately enclosing domain managers of this object. At least one domain manager is always returned in the list since by default each object is associated with at least one domain manager at creation.

4.3 ORB and OA Initialization and Initial References

Before an application can enter the CORBA environment, it must first:

- Be initialized into the ORB and possibly the object adapter environments.
- Get references to ORB pseudo-object (for use in future ORB operations) and perhaps other objects (including some Object Adapter objects).

CORBA V2.2 provides operations, specified in PIDL, to initialize applications and obtain the appropriate object references. The following is provided:

- Operations providing access to the ORB. These operations reside in the CORBA module, but not in the ORB interface and are described in "ORB Initialization" on page 4-8.
- Operations providing access to Object Adapters, Interface Repository, Naming Service, and other Object Services. These operations reside in the ORB interface and are described in "Obtaining Initial Object References" on page 4-10.

In addition, this manual provides a mapping of the PIDL initialization and object reference operations to several languages.

4.4 ORB Initialization

When an application requires a CORBA environment it needs a mechanism to get the ORB pseudo-object reference and possibly an OA object reference. This serves two purposes. First, it initializes an application into the ORB and OA environments. Second, it returns the ORB pseudo-object reference and the OA object reference to the application for use in future ORB and OA operations.

The ORB and OA initialization operations must be ordered with ORB occurring before OA: an application cannot call OA initialization routines until ORB initialization routines have been called for the given ORB. The operation to initialize an application in the ORB and get its pseudo-object reference is not performed on an object. This is because applications do not initially have an object on which to invoke operations. The ORB initialization operation is an application's bootstrap call into the CORBA world. The PIDL for the call (Figure 7-1) shows that the ORB_init call is part of the CORBA module but not part of the ORB interface.

Applications can be initialized in one or more ORBs. When an ORB initialization is complete, its pseudo reference is returned and can be used to obtain other references for that ORB.

In order to obtain an ORB pseudo-object reference, applications call the **ORB_init** operation. The parameters to the call comprise an identifier for the ORB for which the pseudo-object reference is required, and an **arg_list**, which is used to allow environment-specific data to be passed into the call. PIDL for the ORB initialization is as follows:

```
// PIDL
module CORBA {
    typedef string ORBid;
    typedef sequence <string> arg_list;
    ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

Figure 7-1

The identifier for the ORB will be a name of type CORBA::ORBid. All ORBid strings other than the empty string are allocated by ORB administrators and are not managed by the OMG. ORBid strings other than the empty string are intended to be used to uniquely identify each ORB used within the same address space in a multi-ORB application. These special ORBid strings are specific to each ORB implementation and the ORB administrator is responsible for ensuring that the names are unambiguous.

If an empty ORBid string is passed to ORB_init, then the arg_list arguments shall be examined to determine if they indicate an ORB reference that should be returned. This is achieved by searching the arg_list parameters for one preceded by "-ORBid," for example, "-ORBid example_orb" (the whitespace after the "-ORBid" tag is ignored) or "-ORBidMyFavoriteORB" (with no whitespace following the "-ORBid" tag). Alternatively, two sequential parameters with the first being the string "-ORBid" indicates that the second is to be treated as an ORBid parameter. If an empty string is passed and no arg_list parameters indicate the ORB reference to be returned, the default ORB for the environment will be returned.

Other parameters of significance to the ORB can also be identified in arg_list, for example, "Hostname," "SpawnedServer," and so forth. To allow for other parameters to be specified without causing applications to be re-written, it is necessary to specify the parameter format that ORB parameters may take. In general, parameters shall be formatted as either one single arg_list parameter:

–ORB<suffix><optional whitespace> <value>

or as two sequential arg_list parameters:

-ORB<suffix>

<value>

Regardless of whether an empty or non-empty ORBid string is passed to ORB_init, the arg_list arguments are examined to determine if any ORB parameters are given. If a non-empty ORBid string is passed to ORB_init, all ORBid parameters in the arg_list are ignored. All other -ORB<suffix> parameters in the arg_list may be of significance during the ORB initialization process.

The ORB_init operation may be called any number of times and shall return the same ORB reference when the same ORBid string is passed, either explicitly as an argument to ORB_init or through the arg_list. All other -ORB<suffix> parameters in the arg_list may be considered on subsequent calls to ORB_init.

4.5 Obtaining Initial Object References

Applications require a portable means by which to obtain their initial object references. References are required for the root POA, POA Current, Interface Repository and various Object Services instances. (The POA is described in Chapter 9 of this manual; The Interface Repository is described in Chapter 8 of this manual; Object Services are described in *CORBAservices: Common Object Services Specification*.) The functionality required by the application is similar to that provided by the Naming Service. However, the OMG does not want to mandate that the Naming Service be made available to all applications in order that they may be portably initialized. Consequently, the operations shown in this section provide a simplified, local version of the Naming Service that applications can use to obtain a small, defined set of object references which are essential to its operation. Because only a small well defined set of objects are expected with this mechanism, the naming context can be flattened to be a single-level name space. This simplification results in only two operations being defined to achieve the functionality required.

Initial references are obtained via operations on the ORB pseudo-object interface, providing facilities to list and resolve initial object references. The PIDL for these operations is shown below.

```
// PIDL interface for getting initial object references

module CORBA {
    interface ORB {
    typedef string ObjectId;
    typedef sequence <ObjectId> ObjectIdList;

    exception InvalidName {};
```

```
    ObjectIdList list_initial_services ();

    Object resolve_initial_references (in ObjectId identifier)
        raises (InvalidName);
  }

}
```

The **resolve_initial_references** operation is an operation on the ORB rather than the Naming Service's NamingContext. The interface differs from the Naming Service's resolve in that **ObjectId** (a string) replaces the more complex Naming Service construct (a sequence of structures containing string pairs for the components of the name). This simplification reduces the name space to one context.

ObjectIds are strings that identify the object whose reference is required. To maintain the simplicity of the interface for obtaining initial references, only a limited set of objects are expected to have their references found via this route. Unlike the ORB identifiers, the **ObjectId** name space requires careful management. To achieve this. the OMG may, in the future, define which services are required by applications through this interface and specify names for those services.

Currently, reserved **ObjectIds** for CORBA Core are **RootPOA**, **POACurrent**, and **InterfaceRepository**; for CORBA Services, they are **NameService, TradingService**, **SecurityCurrent**, and **TransactionCurrent.**

To allow an application to determine which objects have references available via the initial references mechanism, the **list_initial_services** operation (also a call on the ORB) is provided. It returns an **ObjectIdList,** which is a sequence of **ObjectIds**. **ObjectIds** are typed as strings. Each object, which may need to be made available at initialization time, is allocated a string value to represent it. In addition to defining the id, the type of object being returned must be defined, i.e. "InterfaceRepository" returns a object of type Repository, and "NameService" returns a CosNamingContext object.

The application is responsible for narrowing the object reference returned from **resolve_initial_references** to the type which was requested in the ObjectId. For example, for InterfaceRepository the object returned would be narrowed to **Repository** type.

In the future, specifications for Object Services (in *CORBAservices: Common Object Services Specification*) will state whether it is expected that a service's initial reference be made available via the **resolve_initial_references** operation or not (i.e., whether the service is necessary or desirable for bootstrap purposes).

4.6 Current Object

ORB and CORBA services may wish to provide access to information (context) associated with the thread of execution in which they are running. This information is accessed in a structured manner using interfaces derived from the **Current** interface defined in the **CORBA** module.

Each ORB or CORBA service that needs its own context derives an interface from the **CORBA** module's **Current**. Users of the service can obtain an instance of the appropriate **Current** interface by invoking **ORB::resolve_initial_references**. For example the Security service obtains the **Current** relevant to it by invoking

```
ORB::resolve_initial_references("SecurityCurrent")
```

A CORBA service does not have to use this method of keeping context but may choose to do so.

```
module CORBA {
    // interface for the Current object
    interface Current {
    };
};
```

Operations on interfaces derived from **Current** access state associated with the thread in which they are invoked, not state associated with the thread from which the **Current** was obtained. This prevents one thread from manipulating another thread's state, and avoids the need to obtain and narrow a new **Current** in each method's thread context.

Current objects must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a **MARSHAL** system exception. **Current**s are per-process singleton objects, so no destroy operation is needed.

4.7 Policy Object

An ORB or CORBA service may choose to allow access to certain choices that affect its operation. This information is accessed in a structured manner using interfaces derived from the **Policy** interface defined in the **CORBA** module. A CORBA service does not have to use this method of accessing operating options, but may choose to do so. As examples, in CORBA Core the PortableServer module uses this technique to specify how the POA operates and The *Security Service* uses this technique for associating *Security Policy* with objects in the system.

```
module CORBA {
    typedef unsigned long PolicyType;

    // Basic IDL definition
    interface Policy
    {
        readonly attribute PolicyType policy_type;
        Policy copy();
        void destroy();
    };

    typedef sequence <Policy> PolicyList;
};
```

PolicyType defines the type of **Policy** object. The values of **PolicyType**s are allocated by OMG. New values for PolicyType should be obtained from OMG by sending mail to request@omg.org. In general the constant values that are allocated are defined in conjunction with the definition of the corresponding **Policy** object.

Copy

Policy copy();

Return Value

This operation copies the policy object. The copy does not retain any relationships that the policy had with any domain, or object.

Destroy

void destroy();

This operation destroys the policy object. It is the responsibility of the policy object to determine whether it can be destroyed.

Exceptions

CORBA::NO_PERMISSION	raised when the policy object determines that it cannot be destroyed.

Policy_type

readonly attribute policy_type

Return Value

This readonly attribute returns the constant value of type PolicyType that corresponds to the type of the Policy object.

4.8 Management of Policy Domains

4.8.1 Basic Concepts

This section describes how policies, such as security policies, are associated with objects that are managed by an ORB. The interfaces and operations that facilitate this aspect of management is described in this section together with the section describing Policy Objects.

Policy Domain

A **policy domain** is a set of objects to which the policy(ies) associated with that domain applies. The objects are the domain members. The policy(ies) represent(s) the rules and criteria that constrain activities of the objects which belong to the domain. On object creation, the ORB implicitly associates the object with one or more policy domains. Policy domains provide leverage for dealing with the problem of scale in policy management by allowing application of policy at a domain granularity rather than at an individual object instance granularity.

Policy Domain Manager

A policy domain includes a unique object, one per policy domain, called the **domain manager**, which has associated with it the policy objects for that domain. The domain manager also records the membership of the domain and provides the means to add and remove members. The domain manager is itself a member of a domain, possibly the domain it manages.

Policy Objects

A policy object encapsulates a policy of a specific type. The policy encapsulated in a policy object is associated with the domain by associating the policy object with the domain manager of the policy domain.

There may be several policies associated with a domain, with a policy object for each. There is at most one policy of each type associated with a policy domain. The policy objects are thus shared between objects in the domain, rather than being associated with individual objects. Consequently, if an object needs to have an individual policy, then must be a singleton member of a domain.

Object Membership of Policy Domains

An object can simultaneously be a member of more than one policy domain. In that case the object is governed by all policies of its enclosing domains. The reference model allows an object to be a member of multiple domains, which may overlap for the same type of policy (for example, be subject to overlapping access policies). This would require conflicts among policies defined by the multiple overlapping domains to

be resolved. The specification does not include explicit support for such overlapping domains and, therefore, the use of policy composition rules required to resolve conflicts at policy enforcement time.

Policy domain managers and policy objects have two types of interfaces:

- The operational interfaces used when enforcing the policies. These are the interfaces used by the ORB during an object invocation. Some policy objects may also be used by applications, which enforce their own policies.

  The caller asks for the policy of a particular type (e.g., the delegation policy), and then uses the policy object returned to enforce the policy. The caller finding a policy and then enforcing it does not see the domain manager objects and the domain structure.

- The administrative interfaces used to set policies (e.g., specifying which events to audit or who can access objects of a specified type in this domain). The administrator sees and navigates the domain structure, so is aware of the scope of what he is administering.

Note that this specification does not include any explicit interfaces for managing the policy domains themselves: creating and deleting them; moving objects between them; changing the domain structure and adding, changing and removing policies applied to the domains. Such interfaces are expected to be the province of other object services and facilities such as Management Facilities and/or Collection Service in the future.

Domains Association at Object Creation

When a new object is created, the ORB implicitly associates the object with the following elements forming its environment:

- One or more *Policy Domains*, defining all the policies to which the object is subject.
- The *Technology Domains*, characterizing the particular variants of mechanisms (including security) available in the ORB.

The ORB will establish these associations when the creating object calls **`CORBA::BOA::create`** or an equivalent. Some or all of these associations may subsequently be explicitly referenced and modified by administrative or application activity, which might be specifically security-related but could also occur as a side-effect of some other activity, such as moving an object to another host machine.

In some cases, when a new object is created, it needs to be created in a new domain. Within a given domain a construction policy can be associated with a specific object type thus causing a new domain (i.e., a domain manager object) to be created whenever an object of that type is created and the new object associated with the new domain manager. This construction policy is enforced at the same time as the domain membership (i.e., by **`BOA::create`** or equivalent).

Implementor's View of Object Creation

For policy domains, the construction policy of the application or factory creating the object proceeds as follows. The application (which may be a generic factory) object calls **`BOA::create`** or equivalent to create the new object reference. The ORB obtains the construction policy associated with the creating object, or the default domain absent a creating object.

By default, the new object that is created is made a member of the domain to which the parent object belongs. Non object applications on the client side are associated with a default, per process policy domain by the ORB. Thus, when they create objects the new objects are by default associated with the default domain associated with them.

Each domain manager has a construction policy associated with it, which controls whether, in addition to creating the specified new object, a new domain manager is created with it. This object provides a single operation **make_domain_manager** which can be invoked with the **constr_policy** parameter set to **TRUE** to indicate to the ORB that new objects of the specified type are to be created within their own separate domains. Once such a construction policy is set, it can be reversed by invoking **make_domain_manager** again with the constr_policy parameter set to **FALSE**.

When creating an object of the type specified in the **make_domain_manager** call with **constr_policy** set to **TRUE**, the ORB must also create a new domain for the newly created object. If a new domain is needed, the ORB creates both the requested object and a domain manager object. A reference to this domain manager can be found by calling **get_domain_managers** on the newly created object's reference.

While the management interface to the construction policy object is standardized, the interface from the ORB to the policy object is assumed to be a private one, which may be optimized for different implementations.

If a new domain is created, the policies initially applicable to it are the policies of the enclosing domain. The ORB will always arrange to provide a default enclosing domain with default ORB policies associated with it, in those cases where there would be no such domain as in the case of a non-object client invoking object creation operations.

The calling application, or an administrative application later, can change the domains to which this object belongs, using the domain management interfaces, which will be defined in the future.

4.8.2 Domain Management Operations

This section defines the interfaces and operations needed to find domain managers and find the policies associated with these. However, it does not include operations to manage domain membership, structure of domains, and manage which policies are associated with domains, as these are expected to be developed in a future Management Facility specification (for example, one based on the X/Open Systems Management Preliminary Specification); the Collection Service is also relevant here.

This section also includes the interface to the construction policy object, as that is relevant to domains. The basic definitions of the interfaces and operations related to these are part of the **CORBA** module, since other definitions in the **CORBA** module depend on these.

```
module CORBA
{
    interface DomainManager {
        Policy get_domain_policy (
            in PolicyType policy_type
        );
    };

    const PolicyType SecConstruction = 11;

    interface ConstructionPolicy:  Policy{
        void make_domain_manager(
            in CORBA::InterfaceDef object_type,
            in boolean constr_policy
        );
    };

    typedef sequence <DomainManager> DomainManagerList;
};
```

Domain Manager

The domain manager provides mechanisms for:

- Establishing and navigating relationships to superior and subordinate domains.
- Creating and accessing policies.

There should be no unnecessary constraints on the ordering of these activities, for example, it must be possible to add new policies to a domain with a preexisting membership. In this case, some means of determining the members that do not conform to a policy that may be imposed is required.

All domain managers provide the **get_domain_policy** operation. By virtue of being an object, the Domain Managers also have the **get_policy** and **get_domain_managers** operations, which is available on all objects (see “Getting Policy Associated with the Object” on page 4-7 and “Getting the Domain Managers Associated with the Object” on page 4-8).

CORBA::DomainManager::get_domain_policy

This returns the policy of the specified type for objects in this domain.

```
Policy get_domain_policy (
    in PolicyType policy_type
);
```

Parameters

policy_type	The type of policy for objects in the domain which the application wants to administer. For security, the possible policy types are described in *CORBAservices: Common Object Services Specification*, Security chapter, Security Policies Introduction section.

Return Value

A reference to the policy object for the specified type of policy in this domain.

Exceptions

CORBA::BAD_PARAM	raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

Construction Policy

The construction policy object allows callers to specify that when instances of a particular interface are created, they should be automatically assigned membership in a newly created domain at creation time.

CORBA::ConstructionPolicy::make_domain_manager

This operation enables the invoker to set the construction policy that is to be in effect in the domain with which this **ConstructionPolicy** object is associated. Construction Policy can either be set so that when an instance of the interface specified by the input parameter is created, a new domain manager will be created and the newly created object will respond to **get_domain_managers** by returning a reference to this domain manager. Alternatively the policy can be set to associate the newly created object with the domain associated with the creator. This policy is implemented by the ORB during execution of **BOA::create** (or equivalent) and results in the construction of the application-specified object and a Domain Manager object if so dictated by the policy in effect at the time of the creation of the object.

```
void make_domain_manager (
    in InterfaceDef object_type,
    in boolean constr_policy
);
```

Parameters

object_type	The type of the objects for which Domain Managers will be created. If this is nil, the policy applies to all objects in the domain.

constr_policy	If **TRUE** the construction policy is set to create a new domain manager associated with the newly created object of this type in this domain. If **FALSE** construction policy is set to associate the newly created object with the domain of the creator or a default domain as described above.

4.9 Thread-related operations

To support single-threaded ORBs, as well as multi-threaded ORBs that run multi-thread-unaware code, several operations are included in the ORB interface. These operations can be used by single-threaded and multi-threaded applications. An application that is a pure ORB client would not need to use these operations. Both the ORB::run() and ORB::shutdown() are useful in fully multi-threaded programs.

Note – These operations are defined on the ORB rather than on an object adapter to allow the main thread to be used for all kinds of asynchronous processing by the ORB. Defining these operations on the ORB also allows the ORB to support multiple object adapters, without requiring the application main to know about all the object adapters. The interface between the ORB and an object adapter is not standardized.

```
module CORBA
{
        ...
    interface ORB {
        ...
        boolean work_pending( );
        void perform_work();
        void shutdown( in boolean wait_for_completion );
        void run();
```

4.9.1 work_pending

boolean work_pending( );

This operation returns an indication of whether the ORB needs the main thread to perform some work.

A result of TRUE indicates that the ORB needs the main thread to perform some work and a result of FALSE indicates that the ORB does not need the main thread.

4.9.2 perform_work

void perform_work();

If called by the main thread, this operation performs an implementation-defined unit of work. Otherwise, it does nothing.

It is platform specific how the application and ORB arrange to use compatible threading primitives.

The **work_pending()** and **perform_work()** operations can be used to write a simple polling loop that multiplexes the main thread among the ORB and other activities. Such a loop would most likely be needed in a single-threaded server. A multi-threaded server would need a polling loop only if there were both ORB and other code that required use of the main thread.

Here is an example of such a polling loop:

```
// C++
for (;;) {
   if (orb->work_pending()) {
      orb->perform_work();
   }
   // do other things
   // sleep?
}
```

4.9.3 run

void run();

This operation returns when the ORB has shut down. If called by the main thread, it enables the ORB to perform work using the main thread. Otherwise, it simply waits until the ORB has shut down.

This operation can be used instead of **perform_work()** to give the main thread to the ORB if there are no other activities that need to share the main thread. Even in a pure multi-threaded server, calling **run()** in the main thread is useful to ensure that the process does not exit until the ORB has been shut down.

4.9.4 shutdown

void shutdown(in boolean wait_for_completion);

This operation instructs the ORB to shut down. Shutting down the ORB causes all object adapters to be shut down. If the **wait_for_completion** parameter is TRUE, this operation blocks until all ORB processing (including request processing and object deactivation or other operations associated with object adapters) has completed.