`tibet` DRAFT OMG IDL to Perl Mapping Specification

Bionic Buffalo Corporation
2004.07.19

Table of Contents

1. Document Information and Copyright

1.1. Document Context

This document is a portion of the documentation for Bionic Buffalo's `tibet` product. Information about the product and other, related documentation can be found at *http://www.tatanka.com/prod/tibet.html*

Bionic Buffalo offers a family of products implementing CORBA and related standards and protocols. For more information, see *http://www.tatanka.com/prod/corba.html*

1.2. Copyright, License, and Trademarks

Copyright 2004 by Bionic Buffalo Corporation. All rights reserved, including moral rights.

Tatanka™ and *TOAD*™ are trademarks of Bionic Buffalo Corporation.

This document may be reproduced and distributed (including by means of the Internet) without payment of fees or without notification to Bionic Buffalo, as long as it is not changed, altered, or edited in any way. Any distribution or copy must include the entire document, including the original title, front matter, contents, appendices, and back matter.

1.3. Author and Publisher

Bionic Buffalo Corporation
502 North Division Street
Carson City, Nevada 89703
USA

telephone +1 775 882 1842
fax +1 775 882 6047
e-mail *query@tatanka.com*
web *http://www.tatanka.com*

PGP/GnuPG key fingerprint:
`a836 e7b0 24ad 3259 7c38 b384 8804 5520 2c74 1e5a`

1.4. Document History

Project name: *tibet*
File name: *tbt_perl_mapping*
Previous revision date: Monday 2004.06.07
Formal revision date: Monday 2004.07.19
Last modified: 2004-07-19 14:14:55

For the latest version of this document, or for other forms of this document, see
http://www.tatanka.com/doc/man/tibet/index.html

Updates and revisions of this document are announced on Bionic Buffalo'sCORBA e-mail announcement list. For more information, see
http://www.tatanka.com/bbc/lists.html

2. Overview and Basic Principles

2.1. CORBA and Perl

The Object Management Group (OMG) specified the Common Object Request Broker Architecture (CORBA). As part of that architecture, OMG specified the Interface Definition Language (IDL).

OMG IDL is used describe interfaces, and includes mechanisms to describe the parameters, exceptions, and other entities used by those interfaces. It is a declarative language, and is not used to describe the implementations of those interfaces. However, IDL is not by itself used to create CORBA-compatible programs.

Clients and implementations are constructed in various programming languages (C, C++, Perl, Ada, Java, and others), and the architecture allows them to interoperate. Toward this end, a mapping is defined from IDL to each of these programming languages, in such a way that these programming languages can work together.

Some programming languages have officially-approved mappings from IDL, but OMG does not yet define officially a mapping from IDL to Perl. However, there are several unofficial mappings in use. The unofficial mappings include those used by ORBit, MICO, COPE and ILU/Perl. When Bionic Buffalo set out to implement the complete CORBA specification for Perl programs, it was decided that the existing de facto mappings had shortcomings of various kinds. As a consequence, the mapping described by this document was created. Although this specification owes a great deal to previous work, programs written to comply with it will be incompatible in greater or lesser ways with programs written to the previous specifications.

Although Perl has built-in object-oriented facilities, they are not so extensive as the concepts defined by CORBA. For example, CORBA allows object references which refer to objects not yet created, and object references which survive the destruction of any program. Perl objects are more limited than this, but can be used as building blocks to construct a complete CORBA system. In fact, IDL maps easily to Perl. However, one must be careful to distinguish CORBA Objects from Perl objects, and to distinguish other CORBA entities (such as strings) from their Perl counterparts.

This document assumes a basic familiarity with CORBA and IDL, as well as with Perl. It is a specification first, and an introduction second. However, it contains explanations and examples, and sometimes offers the rationale behind some of the design choices made when this mapping was created. It also contains some explanation of CORBA concepts to distinguish them from analogous or similarly-named Perl concepts, to make clear and emphatic the differences.

2.2. Language Mapping in Practice

A CORBA-compliant system includes a description, in IDL, of the interfaces available to clients. This usually is a file with a suffix `.idl`. The IDL source file is passed through an IDL compiler to generate definitions, header files, and other code in the target language (such as Perl). Often, more than one output file is generated, either as variations for different purposes, or to contain different parts of the resulting output.

Each statement in the IDL source generates an equivalent or contributes to an equivalent in the target file. For example, the IDL definition of an operation causes the creation of a stub program, which can be used by an application to invoke that operation. (The stub program has a proprietary structure, since it communicates with the proprietary interfaces of the object request broker, or ORB.)

Neither the IDL nor the resulting compiler output is a complete definition of an application. As is explained below, IDL describes only the interfaces to objects, and does not specify the internals of an object, or what the methods should do. Therefore, the programmer must add to the compiler output to construct a complete application.

2.3. Origin of a Language Mapping

This Perl mapping, as other official and unofficial language mappings, is made in three layers.

The first layer consists of fundamental equivalents, such as the Perl equivalent to an IDL `interface` or `sequence`. These are defined for each language, depending on the language itself and the way it relates to the underlying semantics of the IDL.

The second layer is an automatic mapping from a standard collection of IDL definitions, based on the first layer. For example, the standard IDL definitions define a `CORBA::Container` interface, and this will be mapped to each language using the rules for interfaces established in that language's first mapping layer.

The third and final layer consists of missing bits and exceptions from the first two layers. For instance, although the mapping for `CORBA::Container` is unremarkable, the mapping for the `CORBA::ORB` initialization functions usually is tailored and modified for each programming language.

This document, as do other language mapping specifications, mostly describes the first and third layers. The automatic mappings of specific, unremarkable interfaces such as `CORBA::Container` are mostly left out, except where they serve as examples. For a complete picture of the mapping, therefore, one should supplement the contents of this specification by reference to the header files used by the applications, and refer to the CORBA specifications themselves.

2.4. Contrasting CORBA and Perl Objects

(*Caution!* This section is not meant to be rigourous. It is only an introduction. If you want rigour, then go to the specifications.)

In general object technology, a class describes "a set of objects that share the same attributes, operations, methods, relationships, and behavior." (Rumbaugh, Jacobson, and Booch, *The Unified Modeling Language Reference Manual*) This description comprises the external (attributes and operations), internal (methods and behaviour) and contextual (relationships) characteristics of the objects.

In Perl, object classes are basically defined by packages, which describe the internal and external characteristics, and perhaps some inherent relationships, of objects. CORBA, however, doesn't much use the word "class". IDL describes the external characteristics of objects (interfaces, with attributes and operations), but is silent on the implementations, states, methods, and other private details.

This is the first great divide between the CORBA and Perl models of objects. The second big difference is in CORBA's use of a broker to mediate the interactions between clients and objects. An object request broker (ORB) intercepts all calls to object methods, locating the implementation, performing services, enforcing policies, and doing other tasks. Brokers also communicate with other brokers, and can pass requests among each other, so objects can be distributed on a network, or move from place to place. All of this is (or can be) transparent to clients: an application doesn't necessarily know where an object's implementation is located.

In the Perl model, method invocations are simply subroutine calls. In the CORBA model, they are not: they might even involve network protocols if the implementation is remote from the client application.

In the Perl model, an application can look into an object implementation (package), and might even alter the behaviour of the methods. In CORBA, the client application has no visibility into the implementation of the object. CORBA object implementation are opaque to their customers.

In Perl, an object reference refers to a collection of data describing the state of the object as defined in the appropriate package. In CORBA, an object reference, though opaque to the application, might exist before an object exists, or after it is destroyed, may contain complex locator data (such as network addresses), and may (in fault-tolerant systems) reference several alternative objects (so that one may take over if another fails).

In Perl, constructor and destructor routines are used to create and destroy instances of classes. Perl objects go away when there are no more references to them. In CORBA, not all objects can be created or destroyed, and even when they can, there are no standard methods invoked to accomplish this. Two examples illustrate the possibilities:

1. A disk file on a server might be considered a CORBA object, with read and write methods. but without any way to create or destroy the file explicitly. The object reference might be created or destroyed, but the object itself might live on forever, or be destroyed at any time.
2. A implementation might create an object reference for a disk file which doesn't yet exist. If no one ever calls the object's write method, then the object (file) might never exist, but the object reference might live on forever.

Many interfaces include destructor methods (to destroy the object to which the method belongs), and many interfaces are factories (which have operations to create other objects). On the other hand, many interfaces don't have destructors, and many objects are created spontaneously, are destroyed spontaneously, or are immortal.

The IDL to Perl mapping in this document uses Perl objects to allow applications to employ or implement CORBA objects, even when the client and object are on different machines or when they are written in different programming languages.

3. Repository Definitions and TypeCodes

The process of compiling the definitions in an IDL source file result in the creation of repository definition objects and `CORBA::TypeCode` objects. These objects capture the essential details of the definitions from the IDL. These objects have interfaces which allow an application to learn about the definitions, and can even be used to reconstruct the IDL from which they were created. (The reconstructed IDL will likely be different in form, and devoid of contents, but will have the important bits accurate.)

The generated repository definition and `TypeCode` objects are included in the output files created by the IDL compiler.

Repository definition objects have various interfaces, depending on they type of IDL object defined. Some of these allow modification of the definition, but the compiler-generated repository definition objects do not implement the write interfaces which can change the definitions. This is partly to conserve resources, but mostly because it doesn't make sense to modify the definition when the original IDL source file is not available to applications to change, and when such a modification would likely require a change in the application source code itself.

By definition, `CORBA::TypeCode` objects are constants, and cannot change in any case.

The names of the generated objects are based on the names of the IDL definition. For IDL definition `ABC`, any generated repository definition object is named `ABC__reposdef`, and any generated `TypeCode` object is named `ABC__typecode`. (Note the double underscore in the names.)

Standard types have associated `CORBA::TypeCode` objects and repository definition objects, named the same way. For example, the `CORBA::TypeCode` for a CORBA `string` is `CORBA::string__typecode`.

4. Modules, Packages, and Names

4.1. CORBA Modules and Perl Packages

IDL modules map closely to Perl packages. In both languages, they serve to enclose namespaces, and act as containers for other entities (including nested modules or packages).

Qualified IDL names are similar to qualified Perl names, except that Perl names must be fully qualified.

There is no real equivalent in IDL to a Perl module. An arbitrary number of IDL modules can appear in an IDL source file.

4.2. Module Definitions

For module definition `ABC`, the IDL compiler generates a `CORBA::ModuleDef` repository object named `ABC__reposdef`.

No `CORBA::TypeCode` objects are defined for IDL modules.

4.3. Standard Packages

To use the Bionic Buffalo IDL mapping to Perl described in this document, a Perl program should

```
    use Tibet ;
```

to pull in the Tibet module at compile time. Tibet is Bionic Buffalo's code which includes the CORBA mappings, plus proprietary mappings.

5. Interfaces and Object References

5.1. Interfaces. Definitions, and TypeCodes

For each interface `ABC` defined in IDL, the compiler generates a Perl package also known as `ABC`. These generated Perl packages are known as object proxy classes. Their use is explained in the next section.

For each interface defined in IDL, the compiler generates an appropriate `CORBA::TypeCode` and repository definition object.

For IDL interface definition `ABC`, the generated `TypeCode` object is `ABC__typecode`.

For IDL interface definition `ABC`, the generated repository definition object is `ABC__reposdef`. The type of the repository definition object is one of the following:

- `CORBA::InterfaceDef`
- `CORBA::ExtInterfaceDef`
- `CORBA::AbstractInterfaceDef`
- `CORBA::ExtAbstractInterfaceDef`
- `CORBA::LocalInterfaceDef`
- `CORBA::ExtLocalInterfaceDef`

depending on the type of IDL interface.

5.2. CORBA Object References and Perl Proxy Objects

An application does not work directly with CORBA object references. Instead, it employs Perl proxy objects, each of which encapsulates a CORBA object reference as private data. The form of the private CORBA object reference within the Perl proxy object is defined by the implementation.

There are three ways an application can get a proxy object:

1. The `CORBA::ORB::init()` operation returns a proxy object for the object request broker (ORB).

2. Operations on some objects, starting with operations on the ORB itself, may return yet more proxy objects.
3. An ORB can convert certain strings, called “stringified object references”, into proxy objects. These stringified references can be created from “normal” object references by the ORB, or they can be constructed manually from certain kinds of URLs.

It is important to note that the destruction of a proxy object does not equate to the destruction of the related CORBA object. Furthermore, there may be zero or many proxy objects for a given CORBA object.

For each operation and attribute defined on the CORBA object, there is an equivalent operation on the proxy object. The mapping from IDL to these proxy operations is described in a later chapter.

5.3. The `CORBA::Object` Interface; Duplication of References

As specified by CORBA, all objects inherit the `CORBA::Object` interface. Any operation which can be invoked on a `CORBA::Object`, can be invoked on any object. These operations include `get_interface()`, `is_nil()`, `duplicate()`, and others.

Object references, including references to proxy objects, must not simply be copied. When a new copy of a reference is required, the application should call `CORBA::Object::duplicate()`. When a reference is no longer needed, `CORBA::Object::release()` should be invoked. These routines allow ORBs to manage resources, coordinate activities with other ORBs, and perform other tasks. *It is important not to rely on Perl garbage collection* when using CORBA object references and proxy objects.

6. Values

(TO BE SUPPLIED)

value
  regular
  boxed
  abstract
  forward
  inheritance

1.13 Mapping for Value Type

1.13.1 Java Interfaces Used For Value Types
1.13.1.1 ValueBase Interface
1.13.1.2 StreamableValue Interface
1.13.1.3 CustomMarshal Interface
1.13.1.4 CustomValue Interface
1.13.1.5 ValueFactory Interface

1.13.2 Basics For Stateful Value Types
1.13.2.1 Inheritance From Value Types
1.13.3 Abstract Value Types
1.13.4 CORBA::ValueBase
1.13.5 Example A
1.13.6 Example B
1.13.7 Parameter Passing Modes
1.13.7.1 Example
1.13.8 Value Factory and Marshaling

1.14 Value Box Types

1.14.1 Generic BoxedValueHelper Interface

1.14.2 Boxed Primitive Types
  1.14.2.1 Primitive Type Example

1.14.3 Complex Types

1.14.3.1 Complex Type Example

1.5 Helpers

1.5.1 Helpers for Boxed Values

1.5.2 Helper Classes (except Boxed Values)

1.5.2.1 Value type Factory Convenience Methods

1.5.3 Examples

7. Constants

Constants in IDL map to a Perl subroutine which returns the value of the constant.

For example, the IDL

```
const float pi = 3.1416 ;
```

maps to

```
sub pi ( ) { return 3.1416 ; }
```

In addition, for each constant `ABC`, the IDL compiler generates a `CORBA::TypeCode` object named `ABC__typecode`, and a `CORBA::ConstantDef` object named `ABC__reposdef`.

(Unlike simple variables, subroutines are not easily overwritten.)

8. Types

8.1. Repository Definitions and TypeCodes

For each named type `ABC`, the IDL compiler generates `CORBA::TypeCode` and repository definition objects named `ABC__typecode` and `ABC__reposdef`. For unnamed types (as are found in operation parameters, structure members, and so on), an application can use standard operations on the enclosing, named `CORBA::TypeCode` or repository definition object to obtain anonymous `TypeCode` or repository definition objects for parameters, return values, or members, as needed.

8.2. Basic Types

8.2.1. Interoperability and Conversions

IDL defines the interfaces, but not the operations, of objects. It is up to the application (client) or method (server) to perform appropriate conversions. The goal of the mapping is to define a form for passing data among clients and methods. In some cases, additional conversion may be required of the client or method, but that is outside the scope of the mapping. For example, the numeric mappings may pass very large numbers as strings of digits, and the Perl program must convert them to numeric format in any reasonable manner if computation is necessary. It is up to the Perl program to know what conversions must be done. When necessary, a program can use associated `CORBA::TypeCode` and repository definition objects to find the type of conversion needed.

8.2.2. Boolean

`CORBA::TRUE` is 1, `CORBA::FALSE` is the empty string.

8.2.3. Characters and Wide Characters

In CORBA, characters (and strings) are associated with specified character sets and encodings. When the client's and method's character set and encoding don't match, the ORB or ORBs mediating the request are expected to perform appropriate translations. The difference between

CORBA `char` and `wchar` is that the former is limited to 8 bits, while the latter can be any multiple of 8 bits.

Perl makes no distinction between narrow and wide characters, but may store wide characters as sequences of bytes. Accordingly, both CORBA `char` and `CORBA::wchar` map to a Perl string of length one. It is up to the implementation (when mediating requests) to perform the appropriate conversions.

8.2.4. Octet

CORBA `octet` maps to a Perl scalar. The value is viewed as a simple number; no character set conversions are performed by the ORB when mediating requests.

8.2.5. Integer Types

The ORB and compiler will convert CORBA integers to Perl scalars. Numbers too large for the underlying implementation will be passed as numeric strings.

8.2.6. Floating Point Types

The ORB and compiler will convert CORBA floating point numbers to Perl scalars. Numbers with more precision than supported by the underlying implementation will be passed as numeric strings.

8.2.7. Any

CORBA `any` is mapped to Perl array with two elements. The first element is the `CORBA::TypeCode` object describing the second element, which is the value of the data itself.

For example,

```
    (CORBA::string__typecode, ”abc”) ;
```

defines a CORBA `any` encapsulating a CORBA string.

When the value member is not a scalar, a reference should be used instead of the value itself.

8.3. Constructed Types

8.3.1. Enum

The value of an IDL `enum` maps to a string containing the unscoped name of its identifier.

8.3.2. Struct

An IDL `struct` maps to a Perl hash. The keys are the member names, and the values are the values of the `struct` members. When the members are not scalars, references are used for the member values.

8.3.3. Union

An IDL `union` maps to a Perl array with two members. The first member is the value of the discriminator, and the second is the value associated with that discriminator's value. If the first value is not a legal discriminator value, then the second value is undefined. When the members are not scalars, references are used for the member values.

8.4. Template Types

8.4.1. Sequence

IDL sequences of `char` and `wchar` map to Perl strings. All other kinds of sequences map to Perl arrays.

8.4.2. String and Wide String

IDL `string`s and `wstring`s map to Perl strings.

8.4.3. Fixed

IDL `fixed` maps to a two-element Perl array. The first element is the digits, and the second is the scale.

8.5. Complex Types

8.5.1. Arrays

IDL arrays map to Perl arrays, with elements of the appropriate type. (Note that arrays of IDL `char` and IDL `wchar` map to Perl arrays of strings, with one character for each string.)

8.5.2. Native

IDL `native` variables are mapped to Perl objects.

9. Exceptions

IDL `exception`s are similar to `struct`s, except that they may have no members. They are mapped the same as `struct`s (see above).

In Perl, exceptions are stacked in frames. There is an exception stack for each thread. Initially, the stack contains one frame, which has no exception. Each frame contains a `CORBA::any` representing the value of the exception at that level of the stack. When there is no exception, the frame contains `(CORBA::any__typecode, undef)`.

The following operations work on the frame at the top of the exception stack:

> To raise an exception, an implementation calls `CORBA::Exception::raise()`. The only argument is a `CORBA::any`, constructed from the `CORBA::TypeCode` of the exception and from the value of the exception structure.
>
> To test for the presence of an exception, `CORBA::Exception::test()` can be called. It takes no arguments, and returns `CORBA::TRUE` if the top frame of the exception stack is not empty.
>
> To catch an exception, a client calls `CORBA::Exception::value()`, which returns a copy of the `CORBA::any` used to raise the exception. This routine may be called repeatedly, each time returning a copy of the `CORBA::any`.
>
> To clear the exception from the frame at the top of the exception stack, the operation `CORBA::Exception::free()` is called. It takes no arguments.

To push another frame onto the exception stack, call `CORBA::Exception::push()`. An empty frame (with no exception) is pushed onto the stack.

To pop the top frame from the stack, `CORBA::Exception::pop()` is called. This routine also returns a copy of the exception value in the popped frame.

Note that, when an application calls CORBA::Exception::value, it receives the value of any exception present. Unlike catch in some programming models, there is no limitation to any one class of exception.

Exceptions need not be caught by the immediate caller, but will persist until caught by any program which tests for them. However, invoking a CORBA object method will clear the top frame of the exception stack, and return with the new method's exception value (if any).

10. Operations

10.1. Proxy Stub Subroutines

Each IDL operation is mapped to an operation on the interface's Perl proxy object. To invoke the CORBA object's operation, an application invokes the equivalent proxy object's operation. The proxy operation, known as a *stub*, finds the actual CORBA object, and invokes the CORBA object's operation. The process involves the ORB, and (if the object is remote) may involve network protocols. Each ORB implementation may do this differently, but the mapping of the proxy operation is the same for all ORBs.

10.2. Parameters and Return Values

The proxy (stub) operation has, as its arguments, all of the `in` and `inout` parameters defined for the IDL operation, in the order defined in the IDL. Any `out` parameters are omitted. In addition, an optional `CORBA::Context` proxy object may be passed as an additional, final parameter.

If a parameter is not a scalar, then a reference is passed instead of the value itself. If the parameter is `inout`, then it is passed as a reference. This means that if a parameter is not a scalar, and `inout`, then it is passed as a double reference.

The return value, and any out parameters, are returned in a list. The return value (if any) is first in the list, the other values follow in the order defined in the IDL. Values which are not scalars are returned as references.

10.3. Repository Definitions

The IDL compiler generates a repository definition object of type `CORBA::OperationDef`, named by suffixing `__reposdef` to the name of the operation.

10.4. Operation of a Stub

(TO BE SUPPLIED)

11. Attributes

Attributes are mapped similarly to operations. For all attributes, an operation named by prefixing `_get_` to the attribute name is generated. For attributes which are not `readonly`, another operation named by prefixing `_set_` is also generated.

12. Events

(TO BE SUPPLIED)

event
- regular
- abstract
- eventtype inheritance

13. Components

(TO BE SUPPLIED)

components
- event sources
- event sinks
- basic and extended components

14. Home

(TO BE SUPPLIED)

15. The Common `CORBA::ORB` Interface

15.1. Requirements for a Common CORBA::ORB Interface

The CORBA specifications describe the interface of the ORB available to applications, but do not extend the definition to include functions necessary to implement stubs and the dynamic skeleton interface. The necessary extensions are implicitly left to the implementor. Most language mapping specifications are also silent on the matter, and as a result the stubs from one vendor are almost always incompatible with another vendor's ORB. The Java mapping is a notable exception: it extends the definition of the ORB interface with its “Java ORB Portability Interface” to allow better interoperability among vendors' products.

Bionic Buffalo produces several different ORB products, and arrived at yet another reason to more completely define the ORB interface used by stubs: there was sometimes a need to use two or more different ORBs with an application at the same time. In such an environment, the same stub often must be shared among different ORB implementations, which must therefore have compatible interfaces.

For these reasons, the IDL to Perl mapping specification follows the example of the Java mapping specification, in providing this extended ORB interface specification.

It should be noted that there is a very good reason that the extended interface definitions do not belong in the base specification: it is clear that different programming languages, for reasons of efficiency, require different interfaces.

15.2. CORBA::ORB and its Subclasses

The Perl CORBA::ORB package has three subclasses:

- `Object` defines operations on object references owned by the ORB
- `Valuebase` defines operations on valuetype references owned by the ORB
- `Auxiliary` defines additional operations, including ones used by stubs

These subclasses are defined below.

15.3. CORBA::ORB::Object

The operations of the `CORBA::ORB::Object` subclass are the same operations defined on the generic `CORBA::Object` interface. This allows each ORB to use different implementations of objects. When a `CORBA::Object` operation is invoked, the `CORBA::Object` class method uses the appropriate method, depending on the ORB to which the object belongs.

15.4. CORBA::ORB::ValueBase

The operations of the `CORBA::ORB::Valuebase` subclass are the same operations defined on the `CORBA::ValueBase` interface. This subclass does for valuetypes what the `Object` subclass does for objects.

15.5. CORBA::ORB::Auxiliary

The operations of the `CORBA::ORB::Auxiliary` subclass are not defined in the base CORBA specifications. They are extensions used mostly by the stubs to communicate with the ORB.

(DETAILS TO BE SUPPLIED)

15.5.1. `identity`

The `identity` operation returns a string identifying the ORB.

15.5.2. `orb_initializer`

This operation initializes the ORB. It is invoked by the `CORBA::ORB::init()` routine defined by the base specification.

15.5.3. `objref_validator`

This operation is used to verify the correct format of an object reference.

15.5.4. `count_incrementor`

This operation is used to increment the use count for an object reference. It is called when an object reference is created or duplicated.

15.5.5. `count_decrementor`

This operation is used to decrement the use count for an object reference. It is called when an object reference is destroyed.

15.5.6. `target_locator`

This operation locates an object implementation. It is used by the stub to decide how to invoke an object's methods. The object might be implemented locally in Perl or in some other language, it may or may not be associated with a servant and POA, or it might be remotely located and require network communication.

15.5.7. `remote_caller`

This operation is used by a stub when an object is remote, and must be accessed over the network.

15.5.8. `adapter_pusher`

When an object is implemented using the POA and an adapter, the semantics of adapter use require that the context be pushed onto a stack of adapters. This operation accomplishes that purpose.

15.5.9. `adapter_popper`

This is the converse of the `adapter_pusher` operation.

15.5.10. `local_object_to_string`

The base specification defines a `CORBA::Object::object_to_string` operation, which produces a stringified object reference. However, that stringified reference is not very useful to humans who just want a quick way tell one object's reference from that of another. This operation provides a simple, shorter string, in a format proprietary to the ORB implementation.

15.5.11. `object_repository_id`

This operation returns the `CORBA::RepositoryId` string associated with an object.

15.5.12. `object_data_set`

This operation allows an arbitrary data item to be associated with an object.

15.5.13. `object_data_get`

This operation retrieves the data item associated with an object, as set by `object_data_set`.

15.5.14. `object_data_destructor_set`

This operation associates a subroutine with an object. That subroutine is invoked when the object is destroyed, and allows destruction (with side effects) of the data defined by `object_data_set`.

15.5.15. `orb_object_table_list`

This operation lists the objects belonging to an ORB. It is intended for use in debugging and development.

15.5.16. `orb_poa_tree_list`

This operation lists the POAs belonging to an ORB. It is intended for use in debugging and development.

15.5.17. `orb_statistics_list`

This operation provides various statistics associated with an ORB. It is intended for use in debugging and development.

16. Proxy Object Implementation

The basic requirements for a Perl proxy object are:

- the stub operations of the related CORBA interface should be defined
- an additional operation, _orb, also must be defined; it returns the proxy for the ORB instance which owns the object

These rules also apply to pseudo objects and local objects, such as `NVList`, `TypeCode`, `Request`, and others.

Otherwise, the implementation of the proxy object is defined by the implementor. However, the repertoire of ORB functions available to a stub is limited to those defined in this specification.

17. Server-Side Mapping

17.1. Mapping for `PortableServer::Servant`

`PortableServer::Servant` is implemented as a Perl object. For each IDL interface ABC, the compiler generates a class `ABC__servant`. Such a class has three methods, and a subclass.

17.1.1. class method `new()`

The new operation creates a new instance of the Servant. It takes three parameters:
- a reference to a hash defining the methods to be used by the servant; the keys are the operation names, and the values are references to the methods
- a reference to a hash defining the servants for the base interfaces; the keys are names of the base interfaces, and the values are the servants for those base interfaces
- a reference to a procedure to be invoked when the `destroy()` instance method is invoked

17.1.2. instance method `base_interfaces()`

This operation returns a list of the base interface servants.

17.1.3. instance method `destroy()`

This operation is used to destroy a servant. After invoking the method provided as the third argument to the constructor, any necessary cleanup of the instance itself is performed.

17.1.4. `methods` subclass

The methods subclass contains a method (operation) for each operation defined in the IDL. The operation names are the same as described in the IDL, but the signatures include two extra parameters at the beginning: a reference to the servant, and a reference to an object of type `CORBA::Current`.

17.1.5. selection of method

Once the POA has selected a servant for the invocation, it looks first for the method in the `methods` subclass. If the method is not found, then it looks for the method in the base interface servants. Therefore: the `methods` subclass can override definitions in the base servants, and no base servants are necessary if all methods are defined in the `methods` subclass.

17.2. Mapping for `PortableServer::Cookie`

`PortableServer::Cookie` is mapped to a Perl object reference.

18. The Dynamic Skeleton Interface

There are no special considerations for the dynamic skeleton interface. It is mapped as expected. Pseudo-objects and local objects are mapped compatibly with ordinary objects.

19. ORB Initialization

This mapping supports the simultaneous use of multiple ORB implementations in an application.

The global variable `tbt_orb_implementation_list` is a list of the `CORBA::ORB` implementations available in the environment. When an application calls `CORBA::ORB::init()`, that subroutine looks through the list of ORB implementation to find one whose identifier matches the ORB identifier specified in the subroutine call. The first ORB implementation that matches is selected. Then, that ORB's `orb_initialization` method is called, and the result returned to the caller.