Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

Training

ANSAwise - Using Advanced CORBA IDL Features

Chris Mayers

Abstract

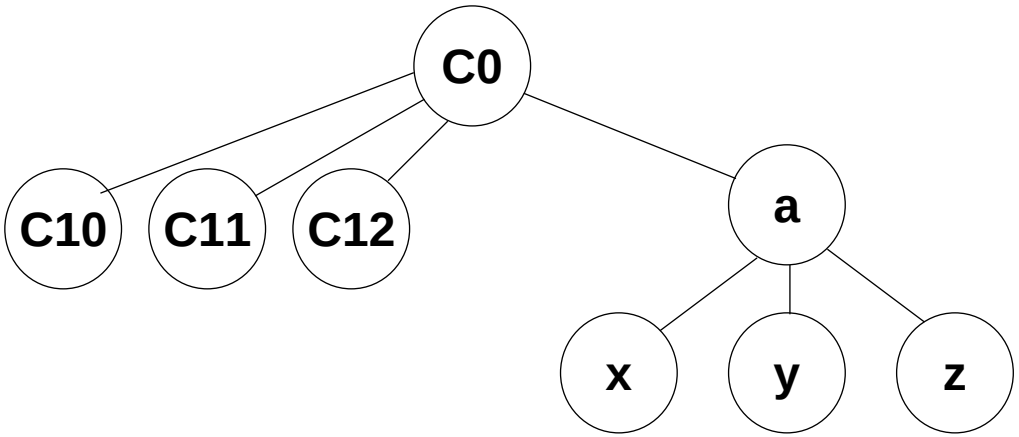
Organizations expecting make sophisticated use of CORBA may wish to have a complete grasp of the features of CORBA IDL.

This module of the ANSAwise training programme covers the features of CORBA IDL that are not covered in other modules, and the circumstances in which you might wish to use them.

APM.1542.01 **Approved** 31st July 1995
Briefing Note

Distribution:
Supersedes:
Superseded by:

Using Advanced CORBA IDL Features



In this session

- ***Explain some advanced features of CORBA IDL***
  - **type any, typecodes, and contexts**
  - **and when you might need to use them**

Type any

- *Specify type any when an argument is to be of any type*
  - not just any object...
  - but also could be a basic type: integer, string,...
  - or a compound type: struct, union,...
- *Example from the Interface Repository*

```
struct Description{
   Contained          contained_object;
   Identifier         name;
   any                value
};
```


When to use type any

- *Don't use type any when type Object would do*
- *Better still, use a more specific object type than Object if you can*
  - exploit polymorphism
- *Use type any if you need dynamic typing*
  - see the Persistence Object Service for another example

The need for typecodes

- *You may need to*
  - construct an any of a given actual type
  - find out the actual type of any (when you don't know it)
  - convert an any to a value of its actual type (when you do know it)
- *This means that the type must be held somewhere*
  - unlike C void*, the actual type can always be determined...
  - ...this involves using typecodes

Handling unknown type anys

- *This is done in different ways in different language mappings*
  - in the C mapping, the representation of an any is exposed

```
typedef struct CORBA_any
   { CORBA_ TypeCode _type; void *_value}
any;
```

  - in the Ada mapping, there is a function in the CORBA package

```
function TypeOf (The_Any: in Any) return TypeCode.Ref;
```


Typecodes

```
interface TypeCode {
   enum TCKind {
      tk_null, tk_void,
      tk_short, tk_long, tk_ushort, tk_ulong,
      tk_float, tk_double, tk_boolean, tk_char,
      tk_octet, tk_any, tk_TypeCode, tk_Principal,tk_objref
      tk_struct, tk_union, tk_enum, tk_string,
      tk_sequence, tk_array
   };

   exception Bounds {};
   boolean equal (in TypeCode tc);
   TCKind kind ();
   long param_count ();
   any parameter (in long index) raises Bounds;
    // The index'th parameter in a compound type
};
```


The TypeCode interface

- *The param_count and parameter operations allow you to pick apart a compound type*
  - refer to *The Common Object Request Broker: Architecture and Specification* for details
- *The enum will need to change when and if new IDL types are defined*
  - for example 64-bit (hyper) integer
- *Note that TypeCode itself uses type any!*

known type any

- *Again, this is done in different ways, depending on the language mapping*
- *In the C++ language mapping, heavy use is made of overloading*
  - unfortunately, mapped C++ types are not always distinct...
  - ...and in C++ (as in C) arrays as arguments decay into pointers to their first elements
  - ... so special cases are needed

Language mappings for type any

- *The language mappings for type any are bound to be clumsy*
  - there is no near equivalent in most programming languages
- *Memory management for values of type any needs particular care*

Contexts

- ***Some operations need additional information to control their behaviour***
  - **the information may be somewhat unrelated to the operation itself...**
  - **... the information should be passed separately from the operation's arguments**

Specifying contexts

- *Specify a list of the context property names after the operation's arguments*

- *For example*

```
interface printer {

   void printFile (in string fileName)
      context ("default_printer","quality");
};
```


Where do context values come from?

- *If a context clause is specified, the operation will have an extra parameter*
  - as determined by the language mapping
- *The default context is set up in an implementation-dependent way*
- *ORB implementations may also fall back to other sources*
  - for example, environment variables may be used
- *To manipulate contexts themselves, use the Context interface*

Context interface -1

```
interface Context {
   Status set_one_value (
      in Identifier prop_name,    // property name to add
      in string     value         // property value to add
   );

   Status set_values (
      in NVList     values        // property values to be changed
   );

   Status get_values (
      in Identifier search_scope, // search scope
      in Flags      op_flags,     // operation flags
      in Identifier prop_name,    // name(s) to retrieve
      out NVList    values        // requested property(s)
   );
```

Context interface - 2

```
   Status delete_values (
      in Identifier   prop_name        // property(s) to delete
    );

   Status create_child (
      in Identifier   ctx_name          // name of context object
      out Context     child_ctx        // context object created
   );


   Status delete (
      in Flags        del_flags        // flags to control delete
   );

};
```

Manipulating contexts

- ***The ORB interface operation get_default_context allows you to manipulate the default context using the Context operations***
  - **this may be useful if the caller does not wish to specify the context on every call...**
  - **...to modify the behaviour for all calls within an object implementation**
- ***Because contexts are objects, you can similarly modify a context, and pass its object reference for use by any caller***
  - **but note Context objects are pseudo-objects...**
  - **...they cannot be created via a factory, and cannot be used as general object references in IDL**

Context object trees and searching

- *Context objects form a tree*
  - the get_values operation allows you to control the search space
  - scope names are implementation-specific, for example

    "_USER"
    "_SYSTEM"

- *This tree is entirely unrelated to the Naming object service*
- *Wildcards (trailing "*") are also supported for context property names*

Contexts for ORB request control

- *"The ORB and/or object is free to use information in this request context during request resolution and performance"*
  - that is, the ORB may supply and use context values in requests
- *Object implementations can peek at these values*

When to use contexts

- *May be appropriate for interfacing to legacy systems and protocols*
- *If you already know the type of information required, do not use contexts*
  - contexts should not be treated as an escape hatch
  - contexts are effectively untyped (strings only)
- *Contexts cannot be used to select an object to target*
  - for this, use the Trading service with the Properties object service
  - ...but note that context properties are entirely unrelated to the Properties object service

Recursive types

- ***Recursive types can be written down, but are only legal via sequences***
  - **this is a special rule**

```
struct foo {
   long value;
   sequence <foo> chain;
}
```

- ***Remember that defining complicated interfaces are a bad idea!***

Summary

- *Type any, and contexts may be useful in some circumstances*
  - but are generally best avoided
- *To find out more*
  - see *The Common Object Request Broker: Architecture and Specification (OMG and X/Open)*
  - see also the various language mapping documents