



**Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom**

TELEPHONE:
INTERNATIONAL:
FAX:
E-MAIL:

**Cambridge (01223) 515010
+44 1223 515010
+44 1223 359779
apm@ansa.co.uk**

Training

ANSAwise - Specifying Services in CORBA IDL

Chris Mayers

Abstract

CORBA specifies an interface definition language (IDL) as the basis of a 'contract' between client and server.

This module of the ANSAwise training programme gives an overview of CORBA IDL and points out some of its pitfalls. This module also shows a few fragments of the C++ Language Mapping to show how IDL is used in a typical build process

[This module is a variant of APM.1348. This module covers only CORBA IDL, and does not relate it to DCE IDL or ANSAware IDL.]

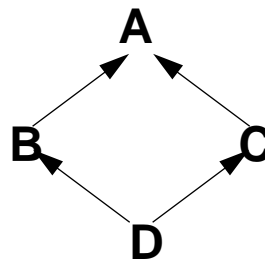
APM.1534.01

Approved
Briefing Note

28th July 1995

Distribution:
Supersedes:
Superseded by:

Specifying Services in CORBA IDL





In this session

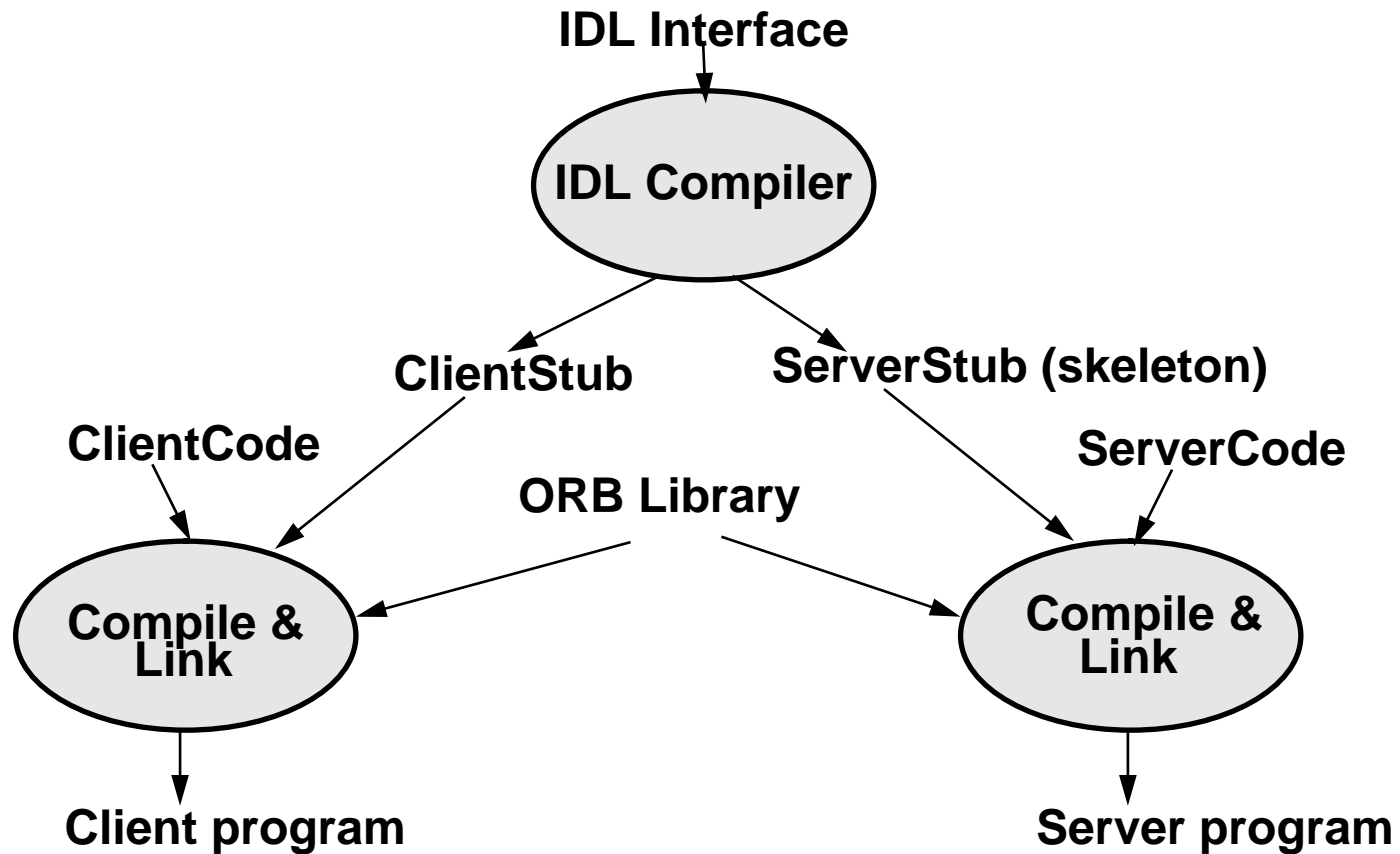
- *Explain how to write service specifications in CORBA IDL (Interface Definition Language)*
- *Explore the basic constructs of CORBA IDL*
- *Explain the significance of interface inheritance in CORBA*



Interface Definitions

- *Are a 'contract' between service provider and service user*
- *Contain only definitions and declarations*
 - no statements
 - ...they cannot be executed
 - ... they are 'sophisticated header files'
 - ... they are compiled
- *Support many programming languages (C, C++, Smalltalk, Ada,...) from one interface definition*
 - any programming language for which there is a *language mapping*

CORBA IDL in a simple build process





CORBA applications and interfaces

- ***One CORBA application can***
 - use many IDL interfaces (as clients)
 - implement many IDL interfaces (as a server - an *object implementation*)
- ***One CORBA interface can***
 - be used by many applications
 - be implemented by many applications



A simple service in CORBA IDL

- *This looks rather like C++*

```
interface Echo {  
  
    // Comment lines start with two slashes  
  
    string Echo (in string Src);  
  
    string Reverse (in string Src);  
  
};
```




Example C++ Interface Mapping

- *This mapping is typical*

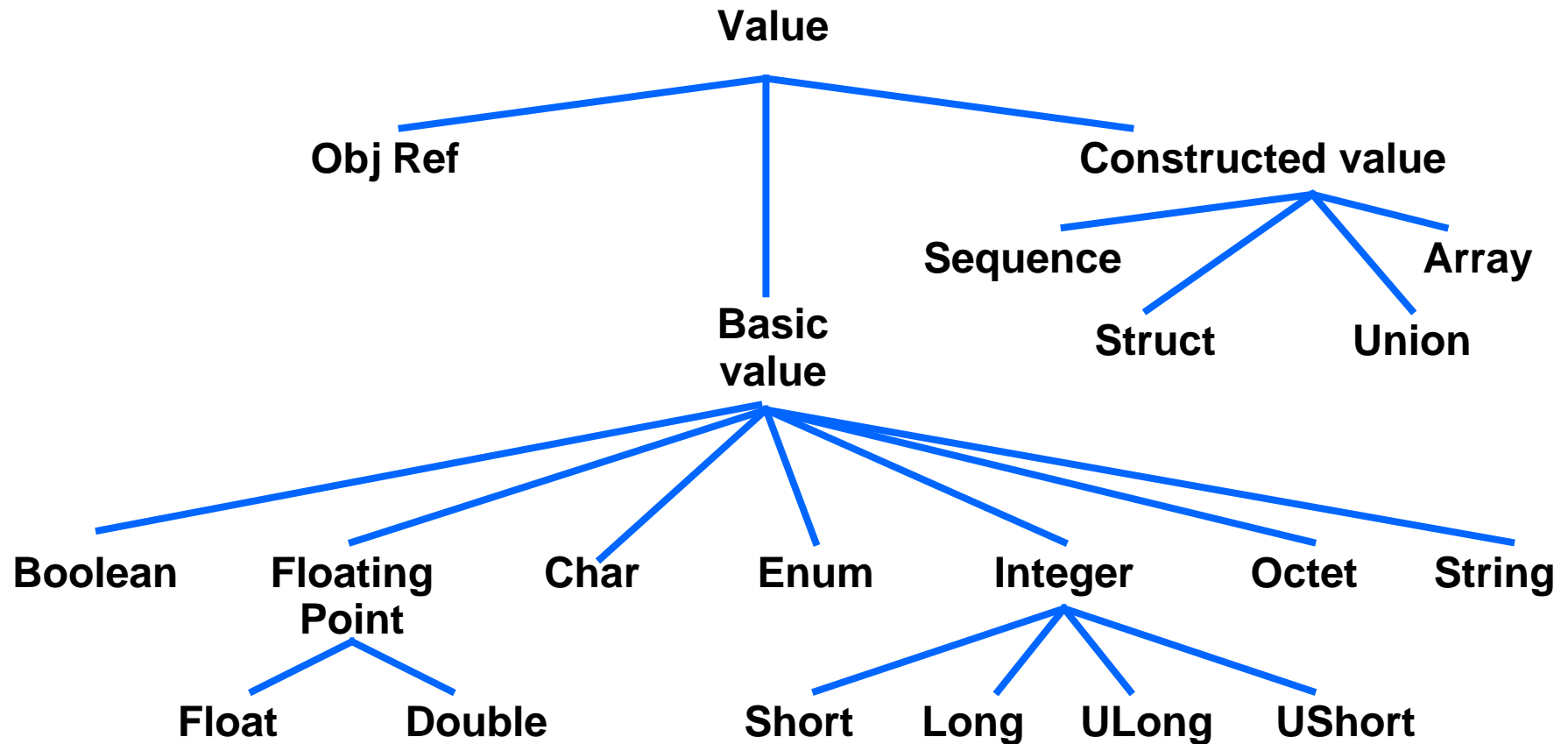
```
class Echo;
typedef Echo *Echo_ptr;
typedef Echo_ptr Echoref;
class A : public virtual Object
{
    // Inherits from the C++ class Object defined by CORBA
    public:
    ...
    virtual char *Echo (const char *Src) = 0;
    virtual char *Reverse (const char *Src) = 0;
    ...
};
...
```



CORBA IDL for specification

- *CORBA IDL is close to a pure specification language*
 - special cases are included for efficiency of language mappings
 - ... there are few arbitrary restrictions

CORBA Data Types





CORBA Types Have Specified Values

- *Unlike some programming languages, CORBA types have a specified set of values*
 - for example, short has exactly the range $-2^{15} .. 2^{15}-1$
 - ... no more, no less
- *What about 64-bit integers?*
 - Sorry, no 64-bit integers - long is the longest: $-2^{32} .. 2^{32}-1$
 - ... and there's no 8-bit integer either (although there is an octet type)

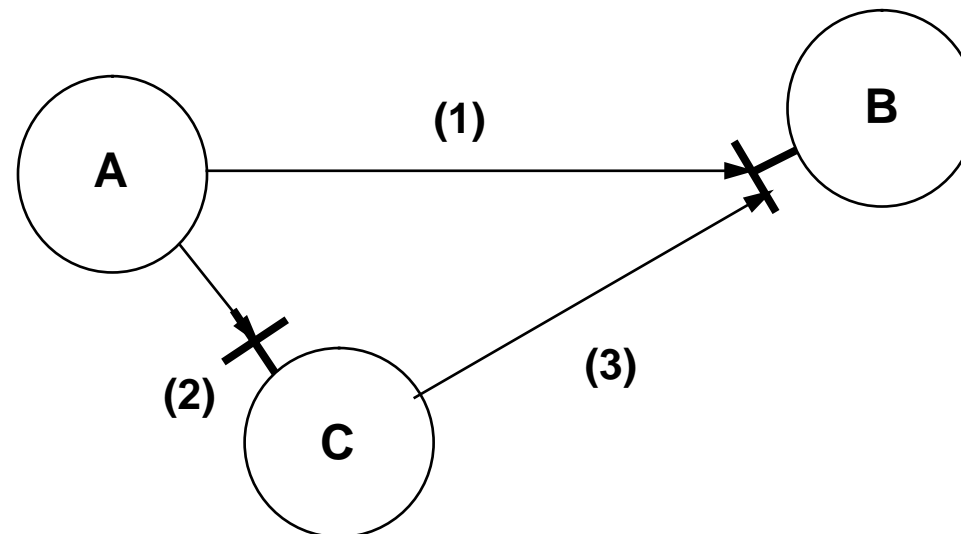


CORBA Types Do Not Have Specified Representations

- *The representation of the values is not specified*
 - *it will differ between machines (e.g byte ordering)*
 - *... the stubs resolve this (when marshalling/unmarshalling)*
 - *... the application programmer does not have to worry*

Transfer of Object References

- *Recall that in a distributed system, we need to be able to transfer object references*
 - Object A is using Object B
 - Suppose it needs to tell C to use the same interface



- It must be possible to pass a reference to B's interface between A and C



CORBA Object References

- *For an object reference, just use the name of the interface*

```
interface B {  
    ...  
};  
interface C {  
    ...  
    void Op (in B my_B);  
    ...  
};
```

- A calls C's Op operation passing a B parameter; the implementation of C can now call operations of B
- *Object references can be used freely in CORBA IDL*
 - not just as parameters - also in structs, sequences, unions, arrays



CORBA Enumerations

- *Enumerations are ordered values*
 - ...for those languages that support ordered enums
 - ...so declare them in ascending order

```
enum boolean {FALSE, TRUE}
```

- *Enumerations can have up to 2^{32} enumerators*



CORBA Structures

- *Structures are records with fields*
- *For example*

```
struct position_t {  
    float x, y;  
};
```



CORBA Unions

- **CORBA unions are ‘discriminated’**
 - they must have an explicit typed discriminator (tag field)
 - you’ll probably want to use an enum (or perhaps boolean) for the discriminator, but you can use integer types or even char
- **For example**

```
union Example switch (long) {
    case 1: long x;
    case 2: float y;
    default: char z;
};
```
- **Note that the discriminator has a type (here, it is long), but no name**
 - language mappings give it a standard name



CORBA Sequences

- *Sequences are one-dimensional, with*
 - a maximum size (fixed at compile time)
 - a length (determined at run time)
- *Sequences can be specified as bounded or unbounded (with or without a maximum size)*
- *For example, a bounded sequence of longs...*
`sequence<long,10>`
- *or an unbounded sequence of sequences*
`sequence<sequence<long> >`



CORBA Strings

- ***Strings are sequences of char***
 - **except the NUL character, of course**
- ***Strings can be specified as bounded or unbounded***
`string<10>`
 - **the bound is the maximum length...**
 - **... in some language mappings, you'll have to allocate extra storage space for a NUL terminator (11 bytes for C)**
- ***Strings are not that special***
 - **they are provided because their language mappings may be more convenient and efficient**



CORBA Arrays

- *Arrays are multi-dimensional, and fixed size*
 - indexed by integers
 - language mapping determines how the array indices are used (for example, 0..size-1 for C)



The absence of pointers

- *CORBA IDL deliberately does not support pointers*
 - they are not meaningful in a distributed system
- *Why do we normally use pointers?*
 -
 -
 -
 -
 -



Sensible alternatives to pointers

- *How should these pointers be described in CORBA IDL instead?*

-

-

-

-

-

Get ready to discuss this



CORBA Constants

- *CORBA interfaces can declare constants of basic types*
- *For example*

```
const long L =4;
```

- *This is handy for declaring common bounds for arrays, sequences, and strings*
- *Constant expressions are also allowed, using C-style operators*



CORBA Operations

- *Recall that operations are much like function prototypes...*

```
Status create_request (  
    in Context          ctx,  
    in Identifier      operation,  
    in NVList          arg-list,  
    inout NamedValue  result,  
    out Request        request,  
    in Flags           req_flags  
);
```

- *... note the use of modes in, out, and inout*
 - *and a result (here, of type Status)*



CORBA Exceptions

- *An exception is like a named 'error status'*
 - but it can have additional information...
 - ... it is like a struct
- *An operation must declare which exceptions it raises*
- *You can declare your own exceptions*

```
interface A {  
    exception E {  
        long L;  
    };  
    void f() raises(E);  
};
```



CORBA standard exceptions

- *There is a standard set of exceptions that can occur during any request*
 - these are predefined in module StExcept
- *CORBA exceptions are not necessarily implemented as programming language exceptions*
 - this depends on the language mapping



Inheritance in CORBA IDL

- *Inheritance in CORBA IDL is solely about extension of specifications*
 - it is nothing to do with object implementations
- *An interface can be derived from another interface*

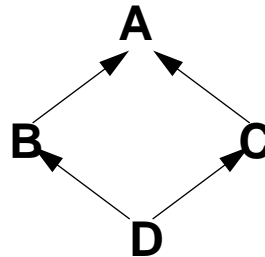
```
interface A {...};  
interface B: A {...};
```

- here, A is the base interface of B
 - all the definitions of A implicitly appear in B
- *Redefinitions are possible too*

Inheritance rules

- *Multiple inheritance is possible*

```
interface A {...};  
interface B: A {...};  
interface C: A {...};  
interface D: B, C {...};
```



- *There are various rules to avoid strange interactions when names are redefined or names clash due to inheritance*
 - *ambiguities can be resolved by explicit qualification*



The effect of inheritance in CORBA

- *Inheritance simply avoids the effort of writing down definitions a second time...*

```
interface A {void f (in float x)}  
interface B {long g (in long x)}  
interface C: B, A {void h (in long x)}
```

- *... interface C is completely equivalent to:*

```
interface C:{void f {in float x}  
            long g {in long x}  
            void h {in long x}}
```

- *The important thing is that an object with interface C may be substituted wherever clients require A or B*
 - *it makes no difference which way you write down C*



Is IDL really necessary?

- *“Why can’t I just write in my ordinary programming language?”*
 - if we can convert between IDLs, why not directly from C++ headers?
- *One day, this may be practical*
 - ...perhaps using special C++ pragmas embedded in header files
 - ...but for the foreseeable future, IDL is going to be the way to specify interfaces



General guidance for using CORBA IDL

- *Interfaces don't have to contain any operations*
 - you can use them to collect together type definitions, constants,...
- *Watch out for incomplete CORBA products*
 - some products don't yet support full CORBA IDL
- *Beware of mode inout*
 - when an unbounded string or sequence is passed as an inout parameter, the returned value cannot be longer than the input value...
 - if an exception is raised, the value is undefined (it is not necessarily the input value)
- *Think carefully before declaring new exceptions*



Summary

- ***IDL defines interfaces, not implementations***
 - ***clients and servers can be implemented in any supported programming language***

- ***For more on CORBA IDL***
 - ***see [The Common Object Request Broker: Architecture and Specification \(OMG and X/Open\)](#)***