APM

POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE • CB3 0RD UNITED KINGDOM
+44 1223 515010 • Fax: +44 1223 359779 • Email: apm@ansa.co.uk • URL: http://www.ansa.co.uk

ANSA Phase III

DIMMA WORKSHOP - CORBA Personality (JET)

Nicola Howarth

Abstract

This presentation covers the CORBA Personality work as at November 1996.

APM.1903.00.01 **Draft** 6th December 1996

Briefing Note

Distribution:
Supersedes:
Superseded by:

DIMMA WORKSHOP - CORBA Personality (JET)

DIMMA WORKSHOP - CORBA Personality (JET)

Nicola Howarth

APM.1903.00.01

6th December 1996

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by APM Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

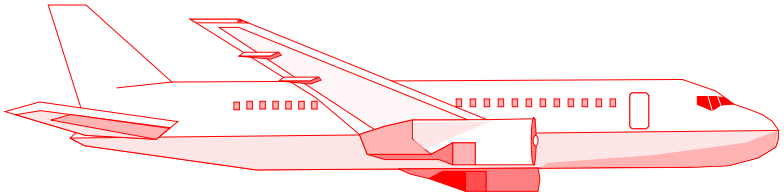
APM Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

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

DIMMA WORKSHOP

CORBA Personality (JET)



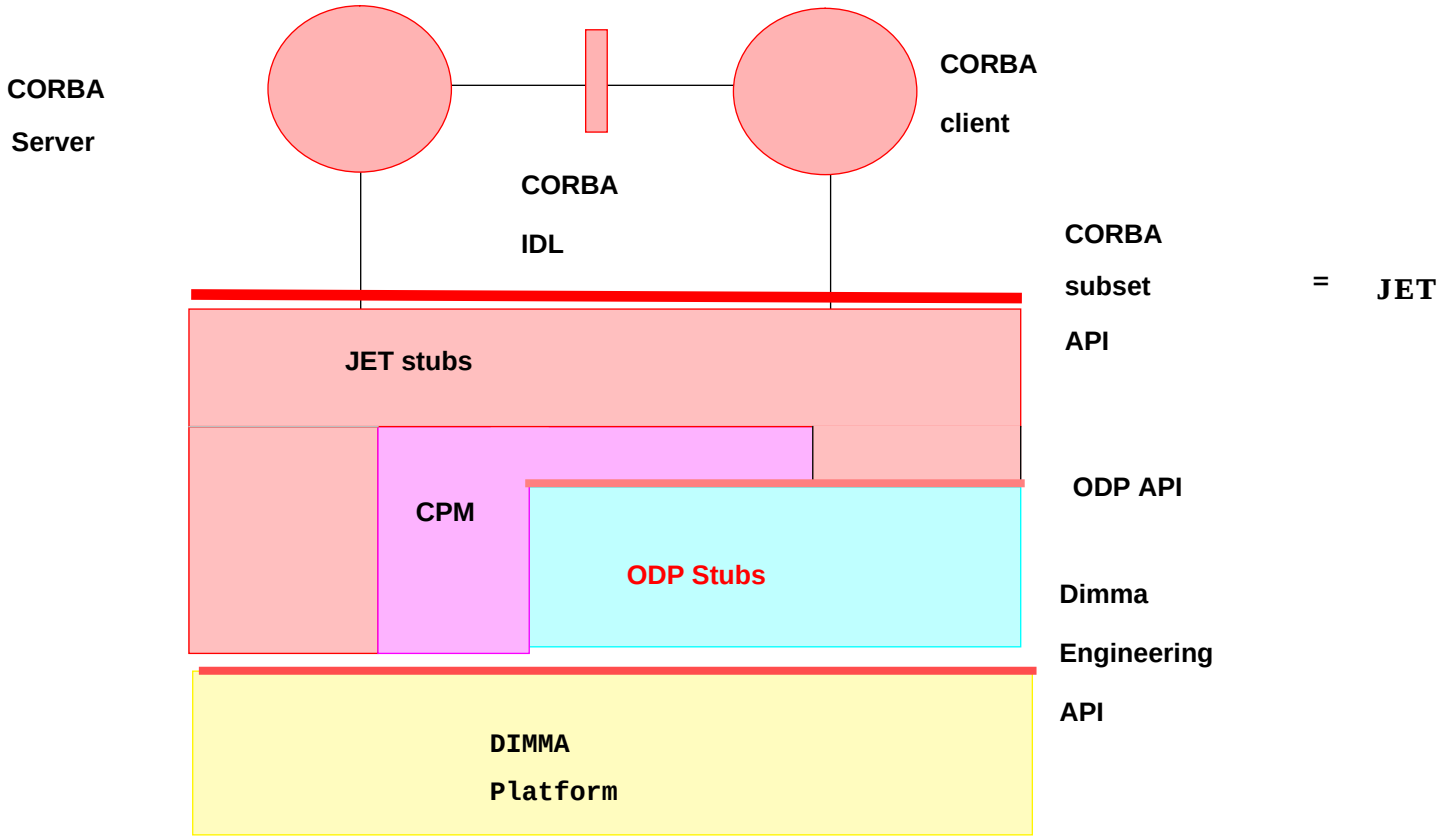
Nicola Howarth

<njh@ansa.co.uk>

Agenda

- How Jet fits into DIMMA
- Jet = CORBA - an extended subset?
- Jet components
- A Simple Example
- Building a Jet application
- Advanced Jet

How JET works - CORBA and the ODP API



Jet = CORBA - an extended subset?

- not included
  - context parameter
  - oneway invocations - but are provided by streams
  - BOA object adapter
  - concrete any
  - dumb pointers T* - impinges on memory management
- in this release - CORBA::Object, arrays, unions
- maybe included in future releases
  - attributes
  - CORBA::Object
  - arrays, unions

Memory Management - T_var v. T*

- A CORBA API can be implemented using either T_var or T*
- you can do everything with one or the other
- you don’t need both

- T_var uses smart pointers with a reference count
- easy to implement garbage collection

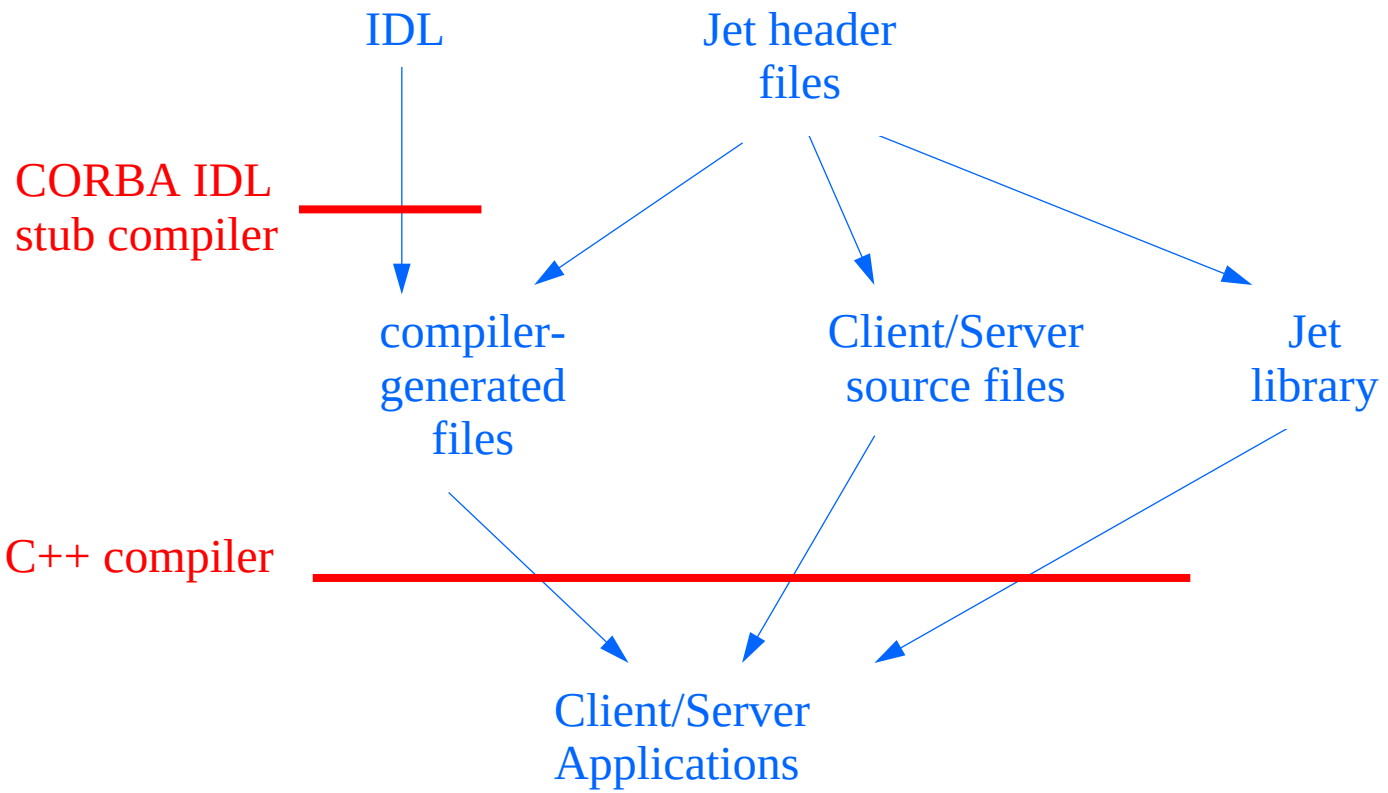
- T* needs release/duplicate
- garbage collection harder

- support is provided for T_var only

Major Features

- support for simple types, structures, sequences and interfaces
- invocations (with in/inout/out)
- top level marshalling (underlying marshalling done in ODP API)
- _bind (trader import for client)
- exceptions

Jet Components



Header Files - 1

- corba.hh - definition of CORBA class
  - standard type definitions (e.g. CORBA::Long)
  - methods within the CORBA namespace (e.g. CORBA::string_alloc(CORBA::ULong n))
- corba_exceptions.hh
  - base class for all exceptions
  - base class for all system exceptions
  - base class for all user exceptions
  - definitions of system exceptions

Header Files - 2

- corba_structured.hh - class definitions and templates for:
  - String_var
  - struct_var
  - sequences
  - sequence_vars
- jet.hh
  - Jet class - the Jet personality module (as opposed to CORBA)

Jet Library

- part of the interface between CORBA and ODP
- maps ODP system exceptions onto CORBA system exceptions
- provides methods used for trading

CORBA IDL Compiler

- **Uses SunSoft's implementation of the compiler front end for OMG Interface Definition Language. This consists of:**
  - **main program for driving the compilation process**
  - **parser and attendant utilities**
  - **back end for taking in the processed input and producing output as required**
- **Modifications to SunSoft's implementation consist of:**
  - **changes to the parser for streams**
  - **the back end to generate Jet internals and stubs appropriate for ODP**

Files Generated - 1

- for a given idl, e.g. echo.idl:
- echo.hh - defines the class for each interface. This includes:
  - interface declarations
  - operation declarations
  - declarations for Jet internals
  - _bind declaration
- echo_Server.hh contains:
  - application programmer extendable EchoObject class which contains the Echo interface
  - application programmer extendable Echo INTERFACE class - ties the Echo i/f with its originating object
- echo_i.hh contains:
  - factory class for generating a new instantiation of the object
  - the _export method for the object
  - method templates for each operation

Files Generated - 2

- **echo_C.cc - client stubs and associated methods**
- **echo_M.cc - any IDL specific marshalling (e.g. for structures)**
- **echo_N.cc - the dispatch table, used by the server to identify operations**
- **echo_S.cc - server stubs and associated methods**
- **echo_Srv_Main.cc - a typical main program - can be used directly or replaced by a user-defined version.**
  - creates a single Echo object
  - exports the object reference to the trader
  - invokes capsule_ready() to put the server into a state where it is awaiting invocations
- **echo_orbC.cc - import and bind methods for the client**

Jet Internals - Server

Object_factory

- Echo_i EchoObject_factory() { return (new EchoObject()->body(); }
  // creates an EchoObject object - normally invoked from main program (Echo_Srv_Main.cc)
- Echo_i* EchoObject::body() { return new Echo_i(this); }
  // creates an Echo interface

_export

- converts interface reference to odp_Any (used as generic interface reference) and passes it as an argument to the trader->export() method

Jet Internals - Client

_bind

- the _bind function builds a set of trader properties based on the interface name, and passes this to the Echo_import function
- Echo_import invokes the _import method on the trader to obtain the required interface reference, which can subsequently be used for invoking operations

Server Stubs

- Dispatcher uses switch statement to execute code specific to required operation
- Declarations are made for all arguments
- In and Inout arguments are unmarshalled
  - any exceptions are dealt with
- The invocation is made
- The results are unmarshalled
- Any locally allocated memory is deleted
- Any exceptions are dealt with

Client Stubs

- The Client stub is invoked directly from the Client
- In and Inout arguments are marshalled
- The operation is invoked
- The stub executes a switch on the response from the invocation:
  - 0 - successful response
  - 1 - system exception
  - >1 - user exceptions
- The results (or exceptions) are unmarshalled and returned to the Client
- Any exceptions occurring within the client stub are dealt with

A Simple Example

```
interface Echo                                          // echo.idl
{
  unsigned long Echo_1(in short mysh, out long mylng,
        inout float myfl );
};

#include “echo_i.hh”                                    // echo_i.cc

CORBA::ULong
Echo_i:: Echo_1
(
    CORBA::Short src,
    CORBA::Long& resL,
    CORBA::Float& resF
)
{
  resL = src;
  resF = src;
  return src;
}
```

```
#include “iostream.h”                                    // Client.cc
#include “echo.hh”

main()
{
  Echo_var evar = Echo::_bind(“:IntSrv”);
  CORBA::Short aSh;
  CORBA::Long aL;
  CORBA::ULong auL;
  CORBA::Float aF;

  aSh = 100;
  auL = evar->Echo_1(aSh, aL, aF);
  cout << auL << “ ” << aL << “ ” << aF << endl;

}
```

```
class Echo;                                                    // echo.hh
typedef odp_InvocationRef<Echo> Echo_var;
extern Echo_var Echo_var_client (odp_Reference * ref) ;
extern void operator <<= (odp_Any&, const Echo_var&);
extern Boolean operator >>= (const odp_Any&, Echo_var&);

class Echo : public odp_Signature {
public:
  virtual CORBA::ULong Echo_1 (
    CORBA::Short mysh,
    CORBA::Long &mylng,
    CORBA::Float &myfl) = 0 ;

  static const char * const odp_names [] ;

  static odp_InvocationRef<Echo> _bind(
    const char* IT_markerServer = "",
    const char* host = "");

  static Echo_var _duplicate( Echo_var ref )
    { ref->odp_increfs(); return ref; };
} ;
```

```
/
/* echo_Srv_Main.cc */

#include <echo_Server.hh>

extern void capsule_ready(void);

int main( int argc, char **argv)
{
  Echo_var myEcho = EchoObject_factory(); // create Echo object

  if (EchoObject::_export(myEcho, (const char*) “/DIMMA”) == -1)
{
    cout << “failed to export Echo interface” << endl; exit(1); }

#ifdef ODP_DON
       capsule_ready(); // required for the single threaded case.
#endif
         return 0 ;
}
```

```
odp_Transmitter &                              // echo_S.cc - extract
  odp_Dispatcher<Echo_var>::dispatch(odp_Receiver*b)
  throw (){
  switch (_request(b)) {
  case 1: {
    CORBA::Short a1;
    CORBA::Long r2;
    CORBA::Float i1;
    try {
      *b >> a1 >> i1 ;
    } catch (odp_Termination& t) {
      //failed to de-marshall arguments
      return response( b, 1 ) << Jet::asException(t);
    }
    try {
      CORBA::ULong r1 = ir->Echo_1( a1, r2, i1 );
      odp_Transmitter &x = response(b) << r1
        << r2
        << i1;
      return x;
    }
    // deal with exceptions here
```

```
CORBA::ULong odp_Echo_Client::Echo_1 (    // echo_C.cc - extract
  CORBA::Short a1,
  CORBA::Long &r2,
  CORBA::Float &i1 ) {
    try {
      odp_Transmitter *const out = request(1) ;
      *((odp_Transmitter *)out)
         << a1
         << i1 ;
      const odp_Releaser in = invoke( out ) ;
      switch ( _response( *in ) ) {
        case 0: {
          CORBA::ULong r1;
          *in >> r1 >> r2 >> i1;
          return r1 ;
        } break;
        case 1: {
          CORBA::SystemException se;
          *in >> se;
          throw se;
        } break; } }
// handle exceptions here
```

```
// echo_orbC.cc - extracts
Echo_var Echo_import(char *type, char *ctxt, char *prop)
{
  Echo_var objRef;
  ansa_Trader trader;
  try {
    trader = capsule_manager->ansa_trader();
  }
  catch (odp_Termination& t) {
    throw Jet::asException (t);
  }
  catch (...) {
    throw CORBA::UNKNOWN ();
  }

  if (!(trader->import(type, ctxt, prop) >>= objRef)) {
    throw CORBA::NO_IMPLEMENT ();
    // Cannot find implementation of interface
  }

  return ( objRef );
}
```

```
// echo_OrbC.cc contd.

Echo_var Echo::_bind( const char* IT_markerServer, const char
*host)
{
  char ctxt_name[256];
  char type_name[256];
  char prop[1024];
  strcpy( type_name, “Echo” );
  strcpy( ctxt_name, “/DIMMA” );
  strcpy( prop, “” );

  Jet::build_trader_properties(IT_markerServer, host, prop);
  return Echo_import( type_name, ctxt_name, prop );
}
```

Building a Jet Application within the DIMMA source tree

```
ROOT = echo
BINS := $(ROOT)_server $(ROOT)_client
LIBRARIES       += -ljet
ifeq ($(ODP_DPE),ansa)
        DEPS    += $(SHADOW)/lib/libodp_ansa.a
endif
ifeq ($(ODP_DPE),don)
        DEPS    += $(SHADOW)/lib/libodp_don.a
endif
IDL_OUTPUT    :=    $(ROOT).hh    $(ROOT)_orbC.cc    $(ROOT)_C.cc
$(ROOT)_S.cc $(ROOT)_N.cc
$(ROOT)_Server.hh $(ROOT)_i.hh
all: $(BINS)
CLIENT_OBJS     := $(ROOT)_orbC.o $(ROOT)_C.o $(ROOT)_N.o
$(ROOT)_client: Client.o $(CLIENT_OBJS)
SERVER_OBJS := $(ROOT)_S.o $(ROOT)_N.o
$(ROOT)_server: $(ROOT)_Srv_Main.o $(ROOT)_i.o $(SERVER_OBJS)
$(dependencies): $(IDL_OUTPUT)
$(IDL_OUTPUT): $(ROOT).idl
        cd  $(<D); \\p        $(IDL) $(<F)
```

Advanced Jet

- Sequences
- Structures
- Interface references
- Streams
- Storage

Sequences - 1

- implementation uses templates
- for an idl definition:

```
interface Echo{
typedef sequence <long> BuffType;
}
```

- echo.hh would hold:

```
typedef corba_ManagedSequence<corba_Sequence<CORBA::Long,0>,
  CORBA::Long,0> BuffType ;
typedef corba_Pointer<BuffType, CORBA::Long> BuffType_var ;
```

- three templates used:
  - **corba_Sequence** - template <class ELEMENT, int TYPE>
  - **corba_ManagedSequence** - template <class SEQUENCE, class ELEMENT, int TYPE>
  - **corba_Pointer** - template <class SEQUENCE, class ELEMENT>

Sequences - 2

- TYPE - of historic value, no longer used, will disappear in next release
- corba_Sequence - the generic sequence
- corba_ManagedSequence - inherits from SEQUENCE, also from odp_RefCounter, to enable smart pointer support
- odp_Pointer is a friend of corba_ManagedSequence, and provides the smart pointer support for sequence_vars

Sequences - 3

- additional templates are used in the implementation of sequences
  - corba_BoundedSequence - template <class ELEMENT, const CORBA::ULong MAXIMUM, int TYPE> - similar to corba_Sequence, but has a fixed maximum size.

```
interface Echo{
typedef sequence<long, 15> BuffType;
}
```

generates

```
typedef
corba_ManagedSequence<corba_BoundedSequence<CORBA::Long,
   15,0>,CORBA::Long,0> BuffType ;
typedef corba_Pointer<BuffType,CORBA::Long> BuffType_var ;
```

Sequences - 4

- Other sequence classes (templates)
  - class corba_SeqSeq - supports nested sequences
  - class corba_BoundedSeqSeq - supports nested bounded sequences
- Problems with sequences - primarily memory allocation
  - vague and ambiguous specification
  - improved specification arrived too late
  - problems only occur when sharing data between sequences
- Implementation of sequences completely revised for next release
  - TYPE has gone
  - sequences of strings handled correctly
  - memory allocation and use of _release handled correctly

Structures - 1

- Structs differ from sequences for marshalling purposes
  - sequences need to marshall a length, and that number of elements
  - structs may have any number of varied element types
- Templates cannot be used
- Specific marshalling functions must be created for each struct
- These are generated by the IDL compiler into echo_M.cc

Structures - 2

```
struct VLstruct
{
  string str;
  long l;
};

odp_Transmitter &operator<< (odp_Transmitter &buffer,
  const Echo::VLstruct &s)
{
  buffer << s.str << s.l;
  return buffer;
}

odp_Receiver &operator>> (odp_Receiver &buffer,
  Echo::VLstruct &s)
{
  buffer >> s.str >> s.l;
  return buffer;
}
```

- Similar marshalling functions generated for struct_var's

Interface References - 1

- are marshalled as odp_References - a generic interface reference

```
// idl
interface Ping {};
interface Echo {
  Ping Echo_1( in Ping a, out Ping b );
};

// server code - exception handling omitted for clarity

Ping_var a1;
odp_Reference *a1_ref;
Ping_var r2;
*b >> a1_ref;
a1 = Ping_var_client(a1_ref);
Ping_var r1 = ir->Echo_1( a1, r2 );
odp_Transmitter &x = response(b)
  << odp_Dispatcher<Ping_var>::ref(r1)
  << odp_Dispatcher<Ping_var>::ref(r2);
return x;
```

Interface References - 2

```
// client code - exception handling omitted for clarity
Ping_var odp_Echo_Client::Echo_1 (
  Ping_var a1,
  Ping_var &r2 ) {
      odp_Transmitter *const out = request(1) ;
      *((odp_Transmitter *)out)
         << a1->odp_stub()->ref() ;
      const odp_Releaser in = invoke( out ) ;
      switch ( _response( *in ) ) {

        case 0: {
          odp_Reference *r1_ref ;
          odp_Reference *r2_ref;
          *in >> r1_ref >> r2_ref;
          r2 = Ping_var_client(r2_ref);
          return Ping_var_client(r1_ref) ;
        } break;
...
```

Streams

- in the idl:
  - use stream instead of interface
  - arguments must all be in
  - function type must be void

```
stream Flow {
        void frame1( in string src, in long no);
};
```

- some minor differences in the generated code
  - different dispatch function
  - different marshalling functions used
  - server cannot return exceptions, so throws them
  - client does not attempt to catch exceptions

Storage of Data within an Interface - 1

- examples/References/echo.idl:

```
interface Ping {
  long Id();
};
interface Echo { // returns different instances of Ping
  Ping Echo_1(out Ping p);
};
interface Bounce {
  void Bounce_1(out long l1, out long l2, in Ping p1,
     inout Ping p2);
};
```

- modifications to echo_Srv_Main.cc - default main program
  - includes “echoStore.hh” instead of “echo_Server.hh” (more later)
  - normally creates a single instance of each interface and exports this to the trader
  - here we only want to create instances of Echo and Bounce - Ping is created only on request - creation of Ping in echo_Srv_Main is removed

Storage of Data within an Interface - 2

- echoStore.hh
  - includes echo_Server.hh, which defines xObject, x_i classes for each interface
  - defines classes PingStore (inherits from PingObject)
    PingImpl (inherits from Ping_i)
    EchoStore (inherits from EchoObject)
  - these classes may include data and additional methods
- echo_i.cc (user written server)
  - defines PingOBJIMPL and PingINTFIMPL as PingStore and PingImpl respectively, prior to including echo_i.hh
- echo_i.hh
  - defines xOBJIMPL and xINTFIMPL to be xObject and x_i for each interface, provided these are not already defined
  - here they are already defined for Ping, so PingObject_factory will create an instance of the PingStore object and PingImpl interface, rather than PingObject and Ping_i classes

Storage of Data within an Interface - 3

- complications due to possibility of odp objects having multiple interfaces, while in CORBA the object and the interface is virtually indistinguishable
- for Ping, we now have an object PingStore, and an interface PingImpl
- in the References example:
  - PingStore has data: CORBA::Long id
  - PingImpl has a method factorySettings by which id can be initialised
- We can now write our server methods - the idl was:

```
interface Ping {
     long Id();
   };
   interface Echo { // returns different instances of Ping
     Ping Echo_1(out Ping p);
   };
   interface Bounce {
     void Bounce_1(out long l1, out long l2, in Ping p1,
        inout Ping p2); };
```

Storage of Data within an Interface - 4

- class Ping_i has a method which returns the value of id

```
CORBA::Long
Ping_i:: Id ()
{
  return ((PingStore*) object)->id;
}
```

- The id is stored in the PingStore object, so must be retrieved from there

Storage of Data within an Interface - 5

- class Echo_i has a method which returns two different Ping_var's, each with a different id

```
Ping_var Echo_i:: Echo_1 (Ping_var& p) {
  PingImpl* obj1 = (PingImpl*)PingObject_factory();
  obj1->factorySettings(5);
  PingImpl* obj2 = (PingImpl*)PingObject_factory();
  obj2->factorySettings(10);
  p = obj2;
  return obj1;
}
```

- the PingObject_factory (which creates PingStore/PingImpl classes) is invoked to create a new Ping instance, and the factorySettings method invoked to set the id. The factorySettings method in the PingImpl class is specified as:

```
void factorySettings (int val)
              { ((PingStore*)object)->id = val; }
```

Summary

- Overview of current release
- CORBA component of future releases
  - upgraded support for sequences
  - support for CORBA::Object for passing around generic interface references
  - a simpler method for storing and accessing data
  - What else?