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**

ANSA Phase III

Writing a DIMMA Application

Nicola Howarth

Abstract

This document provides all the necessary information for building a DIMMA application written to use the extended subset of the CORBA API known as Jet. It describes the Jet IDL and facilities provided by the API, the interactions with the underlying DIMMA libraries and demonstrates the use of these with a simple example program.

DIMMA also provides an alternative programming ‘personality’ based on the concepts of the ODP-RM. However, this has no associated IDL compiler and is used mainly to implement other ‘personalities’ such as Jet. This ODP API is only briefly covered in this document.

APM.2037.01 **Approved** 10th July 1997

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Writing a DIMMA Application

Writing a DIMMA Application

Nicola Howarth

APM.2037.01

10th July 1997

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

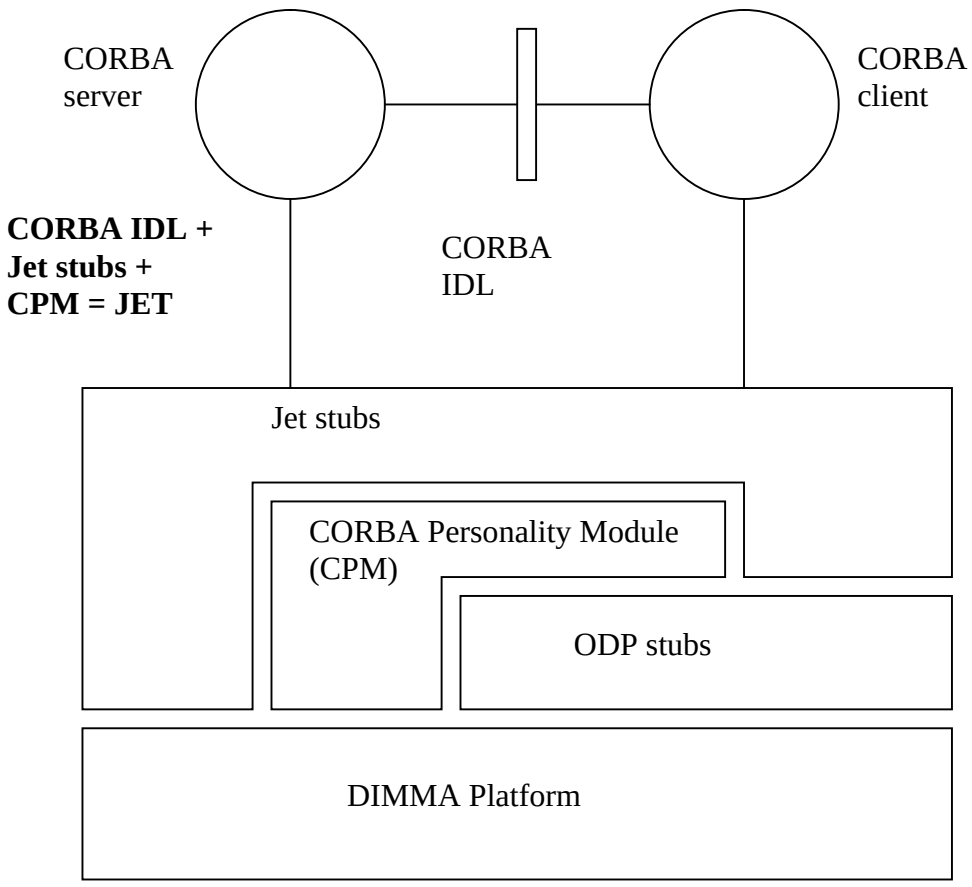
TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

Jet provides an API for DIMMA. While is it possible to write application programs for DIMMA by writing the necessary ODP stubs by hand, this is time-consuming and error-prone. By providing an extended subset of the CORBA API, Jet provides a full interface into DIMMA. CORBA is a well-known and popular standard, but on its own is unable to provide support for streams, for example, while DIMMA on its own is able to provide this sort of support, but is not CORBA-compliant. Jet forms a bridge between CORBA and DIMMA, by providing an extended subset of the CORBA API, supported on the DIMMA environment. Figure 1.1 shows how Jet provides this bridge.

Figure 1.1: Jet: CORBA and the ODP API



This document provides all the necessary information for building a Jet application.

Chapter 2 describes the Jet IDL and API. This is an extended subset of the CORBA API. The extension provides support for flows, while some components of the CORBA API are not supported.

Chapter 3 describes the use of the Jet IDL compiler, and explains the code which it generates.

Chapter 4 describes the interactions with the underlying DIMMA libraries.

Chapter 5 takes a simple example program, and explains it step by step.

Chapter 6 discusses some more advanced features of programming in Jet.

Chapter 7 describes briefly how to program for DIMMA without using Jet.

Suggested route through this document: skim Chapter 2 briefly, then if you just want to run a simple program, go directly to Chapter 5. This will redirect you where further information is necessary.

This document assumes that you have installed DIMMA and set up your environment appropriately as described in [APM.1983].

2 Jet - an extended subset

Jet supports most of the CORBA C++ mapping. It does not support any CORBA-style repositories.

2.1 Support for CORBA

Tables 2.1 and 2.1 show which aspects of the CORBA C++ mapping are supported. It is possible that some aspects currently unsupported may be supported in the future if there is sufficient demand.

Table 2.1: Supported features

Supported types	notes
Scoped names	
Modules	
Interfaces	
Object references	see section 6.1
Constants	
Basic data types	
Enums	
String types	_alloc, _free and _dup supported
Struct types	
Sequence types	see section 6.2
Typedefs	
Exception types	see section 6.4
Operations and argument passing	
Object	
Inheritance	see section 6.5

Table 2.2: Unsupported features

Unsupported types	notes
Namespaces	
Union types	
Array types	
the Any type	
Attributes	
Environment	uses exceptions instead
NamedValue	
NVList	
Request	
Contexts	
Principal	

Table 2.2: Unsupported features

Unsupported types	notes
TypeCode	
BOA	
ORB	

2.2 Extensions to CORBA IDL - flows

CORBA IDL has been extended with the “flow” keyword which may be applied to an interface to specify that it is a flow rather than a operational interface. Flow interfaces define data 'frames' (in an analogous way to operations) but may only be specified in terms of “in” parameters.

A flow interface comprises a single, uni-directional data flow. Flows may be either implicitly (receiver first, transmitter binds on first transmit) or explicitly bound. Explicit binding supports both receiver first and transmitter first scenarios: the latter being useful for multicast applications such as video conferencing.

Flows, insofar as the IDL is concerned, are very similar to interfaces, with the restriction that operations are all oneway, and can take no out or inout arguments, and return no results.

An interface is described using the following syntax:

```
interface myname
{ ......
}
```

while a flow has the following syntax:

```
flow myflow
{ .....
}
```

All operations with the flow must be void, and all arguments must be of type “in”.

Although flows appear similar to interfaces in the IDL, they are distinct entities, and a flow cannot inherit from an interface, nor can an interface inherit from a flow.

3 The Jet IDL compiler

3.1 Running the compiler

The jet_idl compiler will normally be run automatically when a makefile is executed (see section 5.2). It is possible however to run it on its own, using the command:

```
<dimma/platform>/bin/jet_idl myfile.idl
```

where `<dimma/platform>` is the top-level dimma directory and shadow tree, for example `/vobs/dimma/sun4_sos_5.5_build`.

3.2 Compiler-generated code

The jet_idl compiler generates one header file and three C++ files, plus template files: a further one header file and one C++ file for the server implementation. The files are (for IDL file xxxx.idl):

- xxxx.hh: general header file for all types and methods defined in idl file
- xxxx_N.cc: C++ code and data common to client and server
- xxxx_C.cc: C++ client stubs and client specific code
- xxxx_S.cc: C++ server stubs and server specific code
- xxxx_i.th: template header file for server implementation
- xxxx_i.tc: template implementation file for server implementation

The use of these files in the client and server is shown in Figure 3.1. The files marked * are templates, i.e. basic server implementation files, intended to be modified and renamed by the application programmer (see sections 3.2.5 and 3.2.6).

3.2.1 xxxx.hh

This is the main header file for the application. All the C++ files generated by the stub compiler will include it, as will the client code. The server implementation will include it through the implementation header file, xxxx_i.hh. The header file includes the following, also shown in Figure 3.2:

- An iref_var class for each interface reference (iref) in the IDL file.
- An iref_IH class for each iref, which includes all type definitions and methods from the IDL file.
- An iref class for each iref, which inherits from the iref_IH class (the distinction between the two classes is used for inheritance support). This includes:
  - — enumeration of exceptions declared within the interface
  - — enumeration of methods declared within the interface

Figure 3.1: Use of files in client/server

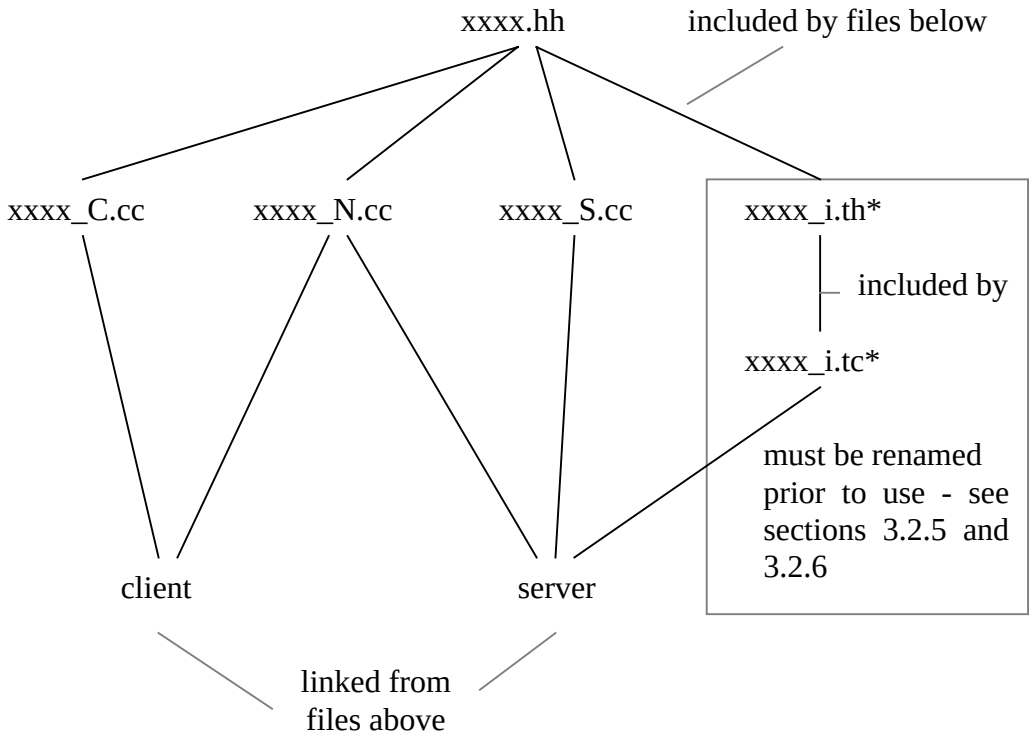
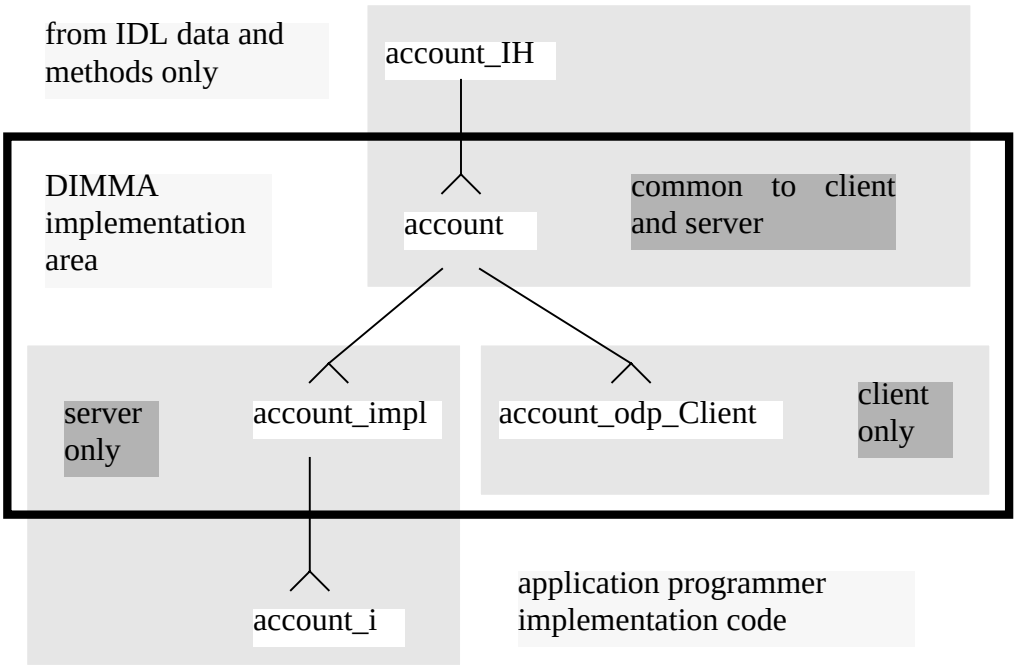


Figure 3.2: Class hierarchy for a single interface



— a dispatch table used by the distribution code

— declaration of the _bind method which returns an iref_var to the client

  - — various conversion and marshalling operators used by the distribution code
  - — declaration of iref methods _narrow, _duplicate and _object_to_string
- An iref_odp_Client class, which inherits from the iref class and from the DIMMA odp_ClientStub class - this class will have client stubs.
- an iref_impl class which inherits from the iref class - this class has the facility to generate server stubs for the interface. This class will be inherited by the implementation class used by the application programmer.

3.2.2 xxxx_N.hh

This file holds code used for distribution, used by both client and server. It includes the dispatch table with names of all interfaces, methods and exceptions, and marshalling functions for iref_vars, and for structures, if any.

3.2.3 xxxx_C.hh

This file holds the client stubs and associated methods. This includes the implementations of _bind, _narrow, and various methods used for distribution, as well as the client stubs for each method declared in the IDL.

3.2.4 xxxx_S.hh

This file holds the server stubs and associated methods. This includes dispatcher constructor and methods used for distribution, and the server stubs for each method declared in the IDL.

3.2.5 xxxx_i.th

This is the first of two files for use in the implementation of the server. This header file declares the iref_i class for each interface iref, together with default constructor and destructor, and method declarations. If additional data and/or methods are required then this file must be modified, otherwise no changes are needed (see section 6.3). In either case, this file should be copied to the location of the source files, and renamed as iref_i.hh.

3.2.6 xxxx_i.tc

This is the main server implementation file. This file should be copied to the source directory and renamed as xxxx_i.cc. The implementation of the server methods should then be added. Dummy methods for each operation are given, to reduce subsequent effort. In addition this file contains the “main” function, which may be removed to a separate file if required. The default operation of “main” is to create an instance of each interface defined in the IDL, and to export that instance to the trader facility. This action may be modified as required.

3.3 Compilation and Linking

Figure 3.1 shows how the various files are related. For straightforward linking of distinct client and servers, the Makefile can make use of config/examples.mk, as done in most of the examples/Functional makefiles. The stages of client/server production are then as follows:

1. Generate the IDL file and a Makefile similar to those in examples/Functional/... which includes examples.mk.
2. Run the IDL compiler using the command “gmake templates”. This results in the production of the files generated by the compiler.
3. Rename the _i.th file to _i.hh and add any data and methods not in the IDL.
4. Rename the _i.tc file to _i.tc and add the implementation for the methods in the IDL and any additional methods.
5. Build the client and server using the command “gmake”. This will re-run the IDL compiler, but as the _i.tc and _i.th files have been renamed, this is not a problem. If you “gmake clean”, then a subsequent “gmake” will need to re-generate these files.

4 The DIMMA Trader interface

4.1 Trading

The DIMMA trader is colocated within each application, so no additional service needs to be activated.

The trader currently supplied with DIMMA is a very simple one, which exports an interface reference by writing it to a file, and imports it by reading the file. This file must be visible to both client and server, eg. using NFS. An extra argument can be given to the trader methods to determine where this file should reside, if not in the local directory. For an interface `Echo`, the `_export` method invokes the `trader_export` function to create a file `Echo` which holds the interface reference of `Echo`, and the `trader_import` function reads the file, processes the interface reference and returns it to the invoking code in the form of an interface `_var` (`Echo_var` in this example).

4.2 Exporting an Interface Instance

The _export function is used to export an interface reference to the trader; it is found in the file `dpe/Jet-CORBA/JetDistributed/jet.cc`. It is defined as:

```
int _export( CORBA::Object_var& iref, const char *ctxt=0);
```

and may be invoked (for interface Echo) as:

```
Echo_var my_Echo = new Echo_i;
if ( _export(my_Echo) == -1 )
{
     cout << “failed to export Echo interface” << endl;
     exit(1);
}
```

Since the `_export` function takes a CORBA::Object_var, any interface reference of type `_var` can be used, with the widening being transparent.

The `_export` function creates an instance of the trader class (if it does not already exist), and invokes the `trader_export` function to generate the file which will hold the interface reference for the exported interface, this being the file which is subsequently read by the `trader_import` function during a `_bind` operation.

The second argument can be used to specify a directory into which the file generated by `trader_export` should be placed. For example:

```
Echo_var my_Echo = new Echo_i;
if ( _export(my_Echo, "/usr/users/myname/trader_files") == -1 )
{
     cout << "failed to export Echo interface" << endl;
     exit(1);
}
```

Here the file generated will be /usr/users/myname/trader_files/Echo.

4.3 Importing and binding to an Interface Instance

The _bind method imports and binds to an instance of an interface. For an interface Echo, it is defined as:

```
Echo_var Echo::_bind( const char *ctxt=0 );
```

A typical invocation might be:

```
Echo_var ev = Echo::_bind();
```

or

```
Echo_var ev = Echo::_bind( "/usr/users/myname/trader_files");
```

The ctxt argument is used to specify the directory where the file containing the interface reference can be found. In the first example the file is expected to be local, in the second, the directory is specified. The interface name itself is used to identify the server, hence the file in this case will have the name Echo.

As can be seen from the code above, the _bind method is invoked on the interface class (Echo class for interface Echo), and returns an interface _var, here Echo_var. This implies an implementation of _bind for each interface declared in the IDL, and this implementation is in the xxxx_C.cc file generated by the stub compiler.

5 A simple example

This chapter describes how to implement a very simple example, in order to demonstrate how to communicate between a client and server.

5.1 The IDL file

The first step in implementing your example is to write the IDL file. Figure 5.1shows a very simple IDL file. It contains a single interface, Echo, with one type and one operation defined.

Figure 5.1: A simple IDL file, echo.idl

```
interface Echo
{
 typedef string MyString;
 MyString Echo_1 (in MyString s1);
};
```

5.2 Building the example

The next stage is to generate the stubs and the template files for the server implementation. For simple examples, most of the work of building the example can be done using the file `config/examples.mk`. Many of the examples use this. A suitable Makefile is shown in Figure 5.2. This Makefile

Figure 5.2: A simple Makefile

```
#Makefile for examples/Strings
ROOT      := echo
CLIENT_OBJS := $(ROOT)_client.o
SERVER_OBJS := $(ROOT)_i.o
include $(DIMMA)/config/examples.mk
```

assumes that the server and client and not colocated, and that the server does not invoke any server methods from within an invocation. If it does this, then the server will need to have client stubs, and the Makefile will require the following change:

```
SERVER_OBJS := $(ROOT)_C.o $(ROOT)_i.o
```

Any additional server or client files can be added to these lines as required. For colocation, see the Makefile for the examples/Functional/Colocate example.

To generate the stubs and template files, having generated this Makefile, and assuming you have GNU make installed as “gmake”, simply type

```
$ gmake templates
```

This will run the IDL stub compiler and generate the stub files.

Full details of building DIMMA and the .mk files are given in APM.1983, “Building DIMMA 2.0”.

If you build within the DIMMA directory struction, you will now find the files listed below in the shadow tree, in the equivalent directory to your source directory. Otherwise you will find the files local to your IDL file, or according to any changes you have made in the Makefile.

- echo.hh - main header file
- echo_C.cc - client stubs
- echo_S.cc - server stubs
- echo_N.cc - distribution code
- echo_i.th - template for implementation header file
- echo_i.tc - template for implementation code

5.3 A simple server

Once you have used the IDL compiler to generate the stub files and templates, you can code the server. The IDL compiler generates two templates which are of use to the server, the_i.th and _i.tc files; in our example these are:

- echo_i.th
- echo_i.tc

These should be copied to your source directory, and renamed as:

- echo_i.th becomes echo_i.hh
- echo_i.tc becomes echo_i.cc

If you need to include data within the interface class, or add methods which are not in the IDL (and so will be local to the server, since no stubs are generated), then you will need to modify the _i.hh file. In our simple example we are doing neither of these things, so the _i.hh file can be left alone. All it contains is a default constructor and destructor for the interface, and a declaration of the method.

The _i.cc file has two parts. The first is the “main” function, which creates an instance of the interface, and exports it to the trader. This is shown in Figure 5.3. The significant part of the method is the line marked as “// **** 1” and the line following it:

Figure 5.3: The “main” function

```
/*
 * Server implementation generated by
 * /vobs/dimma/sun4_sos_5.5_build/bin/jet_idl,
 * version 1.3.0
 * Backend version ANSA 2.0
 * on Fri Apr 25 10:03:08 1997
 */
/* echo_i.tc */

#include <echo_i.hh>
//
// Default main program. Creates one new instance of each
// interface, and exports it to the trader.
//
int main( int argc, char **argv)
{
  extern void capsule_ready(void);
  TRACE_ENTRYEXIT(“main(server)”);

  Echo_var my_Echo = new Echo_i;// ****** 1
  if(Echo_i::_export(my_Echo) == 1)
  {
       cout << “failed to export Echo interface” << endl;
       exit(1);
  }
  capsule_ready(); // required for single threaded case.
  return 0 ;
}
```

```
Echo_var my_Echo = new Echo_i;// ****** 1
if(Echo_i::_export(my_Echo, const char*) “/DIMMA”) == 1)
{
     cout << “failed to export Echo interface” << endl;
     exit(1);
}
```

Here a new instance of the Echo interface is created, then exported to the trader, with a check to ensure that this took place successfully.

The second part of the _i.cc file holds a “template”, or dummy, for each method declared in the IDL. This is shown in Figure 5.4. This is where the implementation for the method should be added. Figure 5.5 shows the template with a simple implementation: the server simply returns a copy of the input string.

To summarise, take the _i.th and _i.tc files generated by the compiler, rename them to .hh and .cc files, add any data or methods, then add the implementations of the methods.

Figure 5.4: The method template

```
Echo::MyString Echo_i::Echo_1 (
     const Echo::MyString s1 )// in
     throw ( CORBA::SystemException )
{
     TRACE_METHOD("Echo::Echo_1");
}
```

Figure 5.5: The method implementation

```
Echo::MyString Echo_i::Echo_1 (
     const Echo::MyString s1 )// in
     throw ( CORBA::SystemException )
{
     TRACE_METHOD("Echo::Echo_1");
     return CORBA::string_dup( s1 );
}
```

5.4 A simple client

Prior to making invocations on a server, the client will need to bind to that server. A very simple client is shown in Figure 5.6. This shows the client "main" program, which binds to the Echo server by invoking the _bind method on the Echo interface, invokes the Echo_1 method, and prints out the result..

Figure 5.6: A simple client

```
#include "echo.hh"
main()
{
  Echo_var ev;
  try {
       ev = Echo::_bind();
  }
  catch(CORBA::SystemException& se) {
       cout << "Unexpected failure whilst binding: ";
       cout << &se << endl;
  }
  MyString str1 = ev->Echo_1("hello world");
  cout << str1 << endl;
}
```

5.5 And finally

You are now in a position to run gmake again to build the client and server, which for our example will be called echo_client and echo_server. You can now run the server, and in a different window, run the client. Once you have done this you will see a new file in the same directory as the client and server

(in the shadow tree) with the name of the interface, here `Echo`. This is the interface reference which is created on invoking `_export`, and read by `_bind`. It will look something like:

```
IOR:000000000000000054563686f000000000000000100000000000000200000
1000000000007737069646572000cfe400000000000800000000000000001
```

You have now successfully built and run a simple Jet application program.

The next chapter explores some more advanced features of programming in Jet.

6 Advanced Jet

6.1 Interfaces and Objects

For each interface declared in the IDL file, there will be a class declared in the .hh file for the interface, and another class for the _var for that interface. The _ptr class is typedef'ed to the _var class. The _var is constructed using C++ templates, and so appears in the .hh file as a typedef statement, e.g. for an interface Ping:

```
#ifndef Ping_var_type       // don't define the following twice
#define Ping_var_type
class Ping;                 // forward reference
typedef corba_InvocationRef<Ping> Ping_var;// the _var
typedef Ping_var Ping_ptr;  // and the _ptr
#endif
```

The normal use of an interface is with the _var class. It is not necessary to use the _ptr form, but if it is used, it is treated identically to _var. The _bind method returns an _var, and invocations are made on this:

```
Echo_var evar;                  // declare the _var
try {
     evar = Echo::_bind(); // get it from _bind
}
catch(CORBA::SystemException& se) {
     cout << "Unexpected sys ex.: " << &se << endl;
}
Ping_var p1, p2, p3;            // some more _vars
p1 = evar->Echo_1(p2);          // invocation on _var
```

6.1.1 Methods on an object

The _var form of interface reference can be copied using either of two forms of duplicate. The _duplicate method is defined (here within class Ping) as:

```
static Ping_var _duplicate( Ping_var ref )
```

which takes as an argument the reference to be copied, and returns a duplicate. This would be invoked as:

```
Ping_var obj2 = Ping::_duplicate( obj1 );
```

Since the _var form is reference counted, it can be duplicated without using a result:

```
Ping::_duplicate(obj1);
```

The _duplicate method can also be invoked directly on the interface:

```
obj1._duplicate();
```

Examples of the use of duplicate can be found in `examples/Functional/Refrences/echo_i.cc`.

The reference can also be converted to a string using the `_object_to_string` method:

```
char *myref = evar->_object_to_string();
```

6.1.2 Use of CORBA::Object

Where interface references of this sort are passed between client and server, the sender willcreate server stubs, and the receiver will create client stubs. When interface references are being passed via a third party, for example a trader, it may be both unnecessary and unhelpful to create server and client stubs. In this case the CORBA::Object should be used. A CORBA::Object is an untyped interface reference, which can be passed around, but on which no invocations may be directly invoked. In this implementation all interface _var's inherit from CORBA::Object, and can be widened to CORBA::Object directly, as in this method in the References example which converts from a Ping_var to a CORBA::Object:

```
CORBA::Object_var Echo_i::make_object (
     Ping_var p )// in
throw ( CORBA::SystemException )
{
     return p;          // convert from Ping_var to CORBA::Object
}
```

The reverse operation makes use of the _narrow method provided for each interface _var. This is also demonstrated in the References example:

```
Ping_var Echo_i::make_ping (
     CORBA::Object_var p )// in
throw ( CORBA::SystemException )
{
     return Ping::_narrow( p );  // narrow to Ping_var
}
```

Two additional methods are provided for CORBA::Objects:

```
static void release( Object_var obj )
```

frees the Object and

```
static Boolean is_nil( Object_var obj )
```

returns true if the Object is null.

6.2 Sequence types

Sequences are all declared using C++ templates, and are managed, in that each sequence has associated with it a reference count for managing multiple copies. The standard constructors and copy operators and other methods are supported as defined in the CORBA specification for sequences in the C++ mapping, along with the allocbuf and freebuf methods.

Many of the features of sequences are demonstrated in the following examples:

- Sequence
- SeqStrings
- StructSeq
- BdSequence

6.3 Adding to the Interface Class

The IDL compiler generates two template, or dummy, files for the implementation:

- file_i.th
- file_i.tc

where file.idl is the name of the IDL file. These should be copied to the source directory, and renamed as .hh and .cc files. The implementation code is then added to the .cc file.

These two files initially reflect the information in the IDL file only, and so all methods in the generated files have stubs provided for them for both client and server. In some cases, however, it may be required to add data and methods to the interface. Data types and static data should be added to the .hh file, along with declarations of any new methods. The implementation of the new methods should be added to the .cc file.

An example of such additions can be found in the References example. Here IDL for interface Ping is simply:

```
interface Ping
{
     long Id();
};
```

This would normally generate the Ping_i class as follows:

```
class Ping_i : public Ping_impl
{
public:
     Ping_i () { TRACE_CONSTRUCTOR(“Ping_i”); };
     ~Ping_i () { TRACE_DESTRUCTOR(“Ping_i”); };
protected:
     CORBA::Long Id ( )throw ( CORBA::SystemException );
}; // end of Ping_i class
```

which consists of a default constructor and destructor, and a declaration of the method from the IDL file.

In the Reference example, this has been modified:

```
class Ping_i : public Ping_impl
{
private:
     CORBA::Long id;         // data
public:
     Ping_i () : id(0) { TRACE_CREATE("Ping_i"); };
     Ping_i (CORBA::Long n):id(n) { TRACE_CREATE("Ping_i(n)"); };
     void factorySettings (int val) {
          TRACE_ENTRYEXIT("factorySettings");
          id = val;
     }
protected:
CORBA::Long Ping_i::Id ( )throw ( CORBA::SystemException );
}; // end of Ping_i class
```

Here we have some data, `CORBA::Long id`, and two constructors. The first of these takes no arguments and clears the data, the second takes an argument and initialises the data. In addition we have a new method, `factorySettings`. The .cc file holds the implementation of this new method, which can be invoked from the server, but cannot be invoked from the client, since it has neither client nor server stubs (unless client and server are in the same capsule, when the stubs are not required).

6.4 Exceptions

Support for exceptions is entirely through the C++`try` and `catch` mechanism; no support is provided for the `CORBA::Environment` variable. Examples of the use of exception handling can be found in the Exceptions example.

6.5 Inheritance

Inheritance of interfaces has been implemented and an example is given in the Inheritance example. Multiple inheritance is not supported. Operations may be overridden within the _i file. Figure 6.1 indicates the structure of the classes declared for two interfaces: `account`, and `currentAccount` which inherits from account, with IDL:

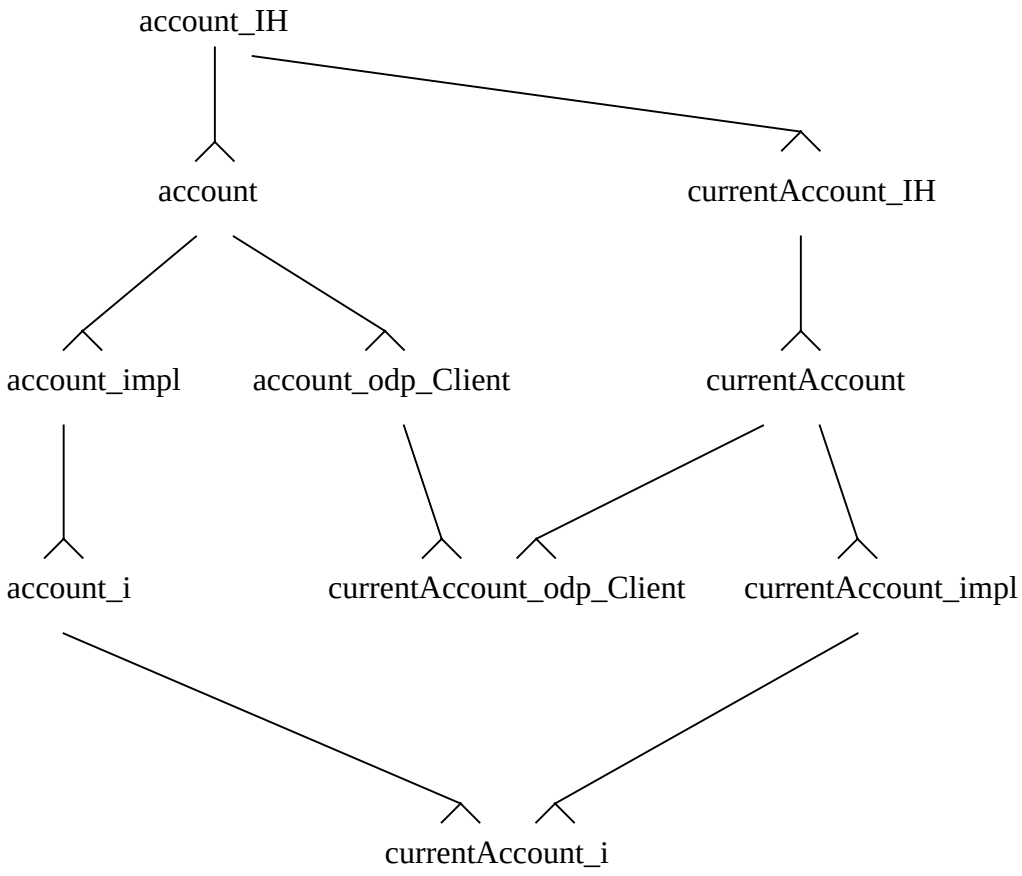
```
interface account {
     .....
};
interface currentAccount : account {
     ......
};
```

The various groups of classes signify the following:

- xxxx_IH: types and methods declared in the IDL
- xxxx: the main class for the interface, includes _bind
- xxxx_odp_Client: the class which includes the client stubs
- xxxx_impl: the class which includes the server stubs
- xxxx_i: the implementation class

It can clearly be seen here that since the implementations inherit directly from each other, only a single implementation is required which is then

Figure 6.1: Inheritance of interface classes



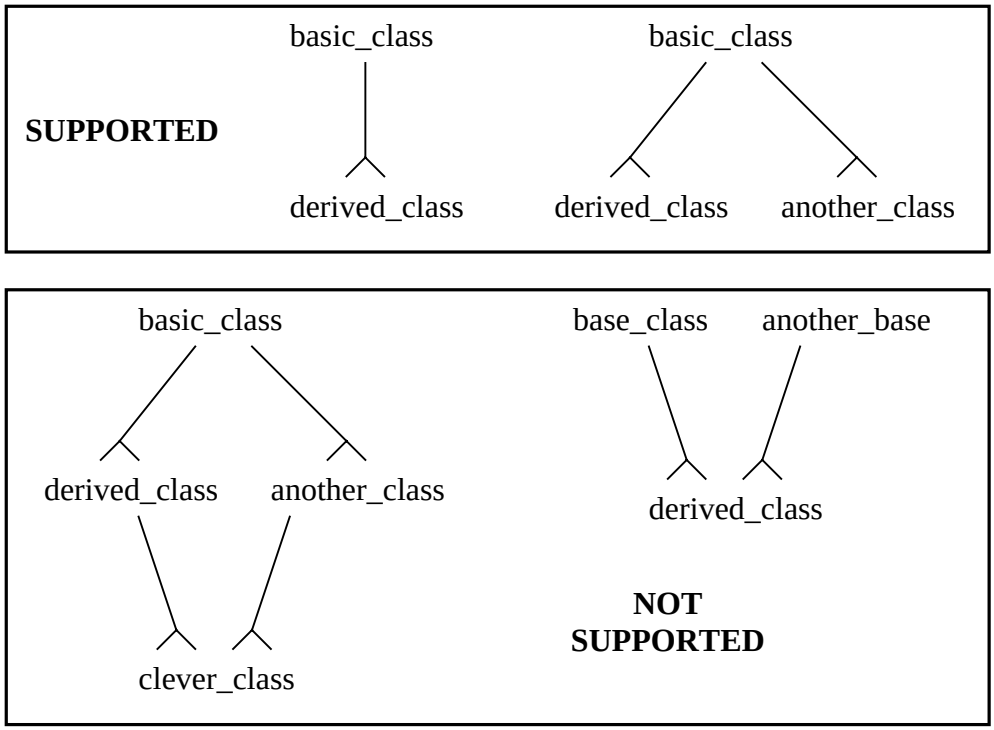
inherited by the appropriate interfaces. It is equally possible to override any methods in the implementation as required.

The various types of inheritance which are supported are shown in Figure 6.2.

6.6 Collocation

Where client and server are to exist in the same capsule, then only one of client and server executables needs to be built. The standard Makefile used in the examples makes both, so a different approach is required. An example is given in the Makefile for the Collocate example, where only a server is built.

Figure 6.2: Supported inheritance



7 Building ODP applications (without Jet)

It is possible to build ODP applications without using CORBA. The main disadvantage in doing this, is that all marshalling stubs must be created by hand. The advantages are that such an application results in a cleaner model, without the aspects of CORBA which may not be required. The ODP model also provides greater flexibility, allowing multiple interfaces within an object.

There are examples of non-CORBA applications in examples/ODP. The main points to note are as follows:

- the class for the interface signature inherits from odp_Signature (see tiny_bank/bank.hh)
- the class for the client stubs inherits both from the interface signature and from odp_ClientStub (see tiny_bank/bank_C.cc)
- the class for the server stubs inherits from odp_ServerStub and has a method Dispatach which switches on each operation (see tiny_bank/bank_S.cc)
- lists of interfaces, operations and exceptions must be set up in odp_names - note that string lengths are given in octal (see tiny_bank/bank_N.cc)

Note that for flows, odp_ServerStub is replaced by odp_ReceiverStub, and odp_ClientStub is replaced by odp_TransmitterStub.

If there is a requirement to build an application without Jet, then it is recommended that the tiny_bank example is examined and understood in detail first.

References

[APM.1995]

Macmillan, I. A., *An Introduction to DIMMA*; **APM.1995**, APM Ltd., Cambridge U.K., May 1997.

[APM.1983]

Howarth, N. J., *Building DIMMA 2.0*; **APM.1983**, APM Ltd., Cambridge U.K., April 1997.

[APM.1980]

Hayton, R. J., *DIMMA Tracing*; **APM.1980**, APM Ltd., Cambridge U.K., April 1997.

[APM.1994]

Macmillan, I. A., *DIMMA Design and Implementation;* **APM.1994**, APM Ltd., Cambridge U.K., June 1997.