CORBA Quick Start

Using TAO with C++Builder 6

Christopher Kohlhoff (chris@kohlhoff.com)

Contents

Chapter 1

Introduction

This *CORBA Quick Start* provides an overview of using C++Builder 6 with TAO to get you started developing CORBA applications. It also suggests where to look for further details about CORBA development in C++Builder.

- **Chapter** 2 looks at the development of CORBA applications using C++Builder 6 and TAO, including creating a project, precompiled headers, and using the IDL compiler.
- **Chapter** 3 takes you through the creation of a simple CORBA application to illustrate the basic steps involved in using TAO with C++Builder 6. It shows the use of CORBA in console programs.
- **Chapter** 4 takes you through the creation of a VCL-based CORBA application to show how to use TAO in programs with graphical user interfaces.
- **Appendix** A explains how to build and install ACE+TAO from source code using C++Builder 6.

1.1 What is CORBA?

Common Object Request Broker Architecture (CORBA) is an open, vendor-independent specification for an architecture and infrastructure that allows applications to communicate over networks.

The core features of CORBA are:

- A high-level Interface Definition Language (IDL), allowing applications to specify their distributed communication in an object-oriented fashion.

- Standardised protocols, GIOP and IIOP, for on-the-wire CORBA communication.
- A set of programming APIs to address the middleware needs of client to server connectivity.

These features allow all CORBA-based programs to interoperate, even though they may be written in almost any programming language, and running on almost any operating system or network.

1.2 What is TAO?

TAO is a standard-compliant implementation of CORBA that is designed for applications with high-performance and real-time requirements. TAO is freely available, open source software, and has been developed by research groups at Washington University and University of California at Irvine. TAO's development has been funded by various industrial sponsors, and it is being actively used and enhanced by a large development community.

1.3 Finding Information

The following web sites, newsgroups, mailing lists and books may provide useful further information about the use of CORBA and TAO.

- The TAO project home page at `http://www.cs.wustl.edu/~schmidt/TAO.html`. Includes links to download the latest releases and documentation.
- The OMG's CORBA web site at `http://www.corba.org` provides an overview of CORBA and lists CORBA case studies and success stories.
- The ACE+TAO mailing lists and newsgroup (`http://www.cs.wustl.edu/~schmidt/TAO-mail.html`) are relatively high volume discussion forums that provide peer support for users of ACE and TAO.
- OCI (`http://www.ociweb.com`) provides commercial support for TAO.
- The book:

  Henning, Michi and Steve Vinoski (1999). *Advanced CORBA Programming with C++*. Addison-Wesley.

contains comprehensive information on the use of CORBA with C++, and is virtually essential reading for anyone doing CORBA development using C++.

- The TAO with C++Builder page at `http://www.tenermerx.com/tao_bcb` includes information on building and using ACE and TAO with C++Builder.

1.4 Typographical Conventions

This manual uses the typographical conventions outlined below to indicate special text.

- IDL and C++ source code appears in `fixed width type`.
- Environment variables and any commands you must type in appear in `fixed width type`.
- File names appear as ***bold italic type***.
- Menu items, buttons and other user interface elements appear in sans serif type.

Chapter 2

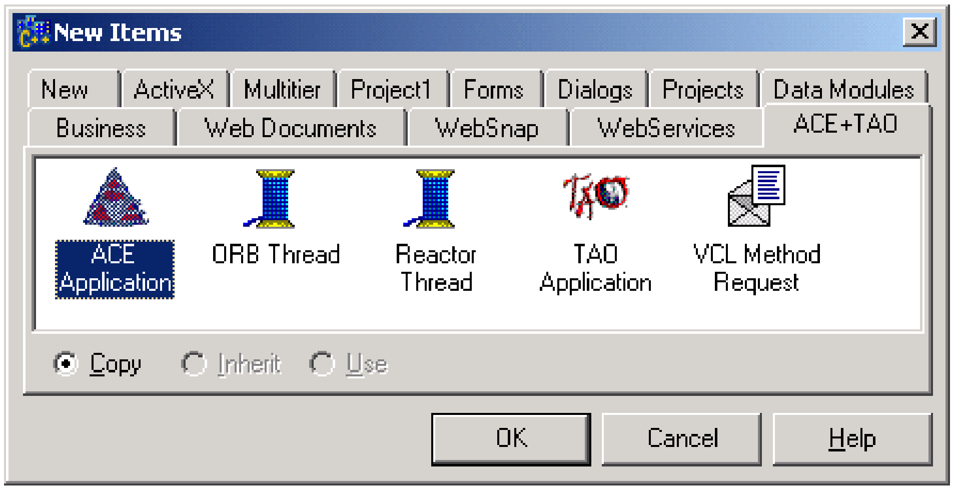
Programming with C++Builder and TAO

This chapter provides an overview of the development of CORBA applications using C++Builder 6 and TAO, including creating a project, precompiled headers, and using the IDL compiler.

2.1 Creating a Project Using the Wizard

The ACE+TAO C++Builder project wizards can be used to generate C++Builder IDE projects that use ACE or TAO. They also include support for generating some useful helper classes.

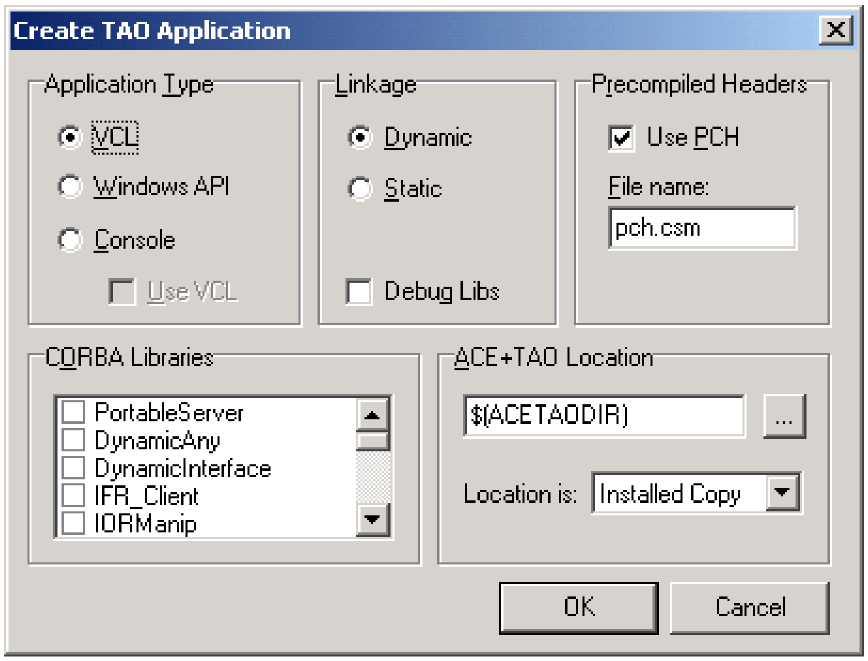
If installed correctly, the wizards are found in the New Items dialog on a tab named ACE+TAO:



The items shown on the ACE+TAO tab are:

- **ACE Application:** Wizard for creating VCL, console or Windows API applications that use ACE only.
- **TAO Application:** Wizard for creating VCL, console or Windows API applications that use TAO.
- **ORB Thread:** Wizard for generating a class descended from TThread that runs the ORB event loop in a background thread.
- **Reactor Thread:** Wizard for generating a class that runs the default ACE reactor in a background thread.
- **VCL Method Request:** Wizard for generating a class that provides access to the VCL `Synchronize` functionality from anywhere in a program.

To create C++Builder IDE project that uses TAO, select TAO Application. The Create TAO Application dialog that appears presents the following options:



- **Application Type:** Select whether to create a VCL, Windows API or console application.

- **Linkage:** Choose whether to use dynamic or static versions of the ACE+TAO libraries, and whether to use libraries that contain debugging information. Note that ACE+TAO prebuilt for C++Builder 6 only includes the dynamic libraries without debug information.
- **Precompiled Headers:** Enable the use of precompiled headers, and set the name of the file to use to store the precompiled header information.
- **CORBA Libraries:** Select the additional CORBA libraries to be linked into the application.
- **ACE+TAO Location:** Specify the directory containing an installed version of ACE+TAO or an ACE+TAO source tree. The wizard can generate projects that use the ACE+TAO libraries and header files from either location.

The code generated by the ACE+TAO wizards can be customised by editing the template files located in the same directory as the wizard package (the ***.bpl*** file).

2.2 Creating a Project Manually

This section outlines how to manually create a C++Builder IDE project that uses TAO.

2.2.1 Creating a Dynamically-Linked Console Project

Note: These instructions assume that you have either:

- installed a pre-built copy of the dynamically-linked ACE+TAO libraries, executables and header files; or
- built and installed the dynamically-linked version of ACE+TAO according to the instructions in appendix A.

1. Create a new console application by going File→New→Other... and using the Console Wizard. Make sure that the source type is C++, and that both Multi Threaded and Console Application *are* checked.
2. TAO is built on top of the ACE library, so you need to initialise ACE from your program. This will be done automatically provided your `main` function has the `argc` and `argv` parameters:

```
#include <tao/corba.h>

int main(int argc, char* argv[])
{
  // ...
}
```

3. Go to Project→Options..., then to the Linker page and make sure that Use dynamic RTL *is* checked.

4. Still in the Project Options dialog, go to the Directories/Conditionals page.

5. Add the following directory to the include path:

   ```
   $(ACETAODIR)\include
   ```

Note: See section A.2.5 if you have not set up the ACETAODIR environment variable.

6. Add this directory to the library path:

   ```
   $(ACETAODIR)\lib
   ```

7. Close the Project Options dialog.

8. Go to Project→Edit Option Source and add the following libraries to the end of the SPARELIBS value:

   ```
   ACE_b.lib TAO_b.lib
   ```

   You may add other TAO libraries as required for the CORBA Services and other TAO features. For example, to use the Naming Service you would add:

   ```
   TAO_CosNaming_b.lib
   ```

9. Save and close the project's option source.

10. Build your application.

2.2.2 Creating a Statically-Linked Console Project

Note: These instructions assume that you have built and installed the statically-linked version of ACE+TAO according to the instructions in appendix A. A pre-built statically-linked version is not available.

1. Create a new console application by going File→New→Other... and using the Console Wizard. Make sure that the source type is C++, and that both Multi Threaded and Console Application *are* checked.

2. TAO is built on top of the ACE library, so you need to initialise ACE from your program. This will be done automatically provided your `main` function has the `argc` and `argv` parameters:

```
#include <tao/corba.h>

int main(int argc, char* argv[])
{
  // ...
}
```

3. Go to Project→Options..., then to the Linker page and make sure that Use dynamic RTL *is not* checked.

4. Still in the Project Options dialog, go to the Directories/Conditionals page.

5. Add the following directory to the include path:

```
$(ACETAODIR)\include
```

Note: See section A.2.5 if you have not set up the `ACETAODIR` environment variable.

6. Add this directory to the library path:

```
$(ACETAODIR)\lib
```

7. Add these conditional defines:

```
ACE_AS_STATIC_LIBS=1
TAO_AS_STATIC_LIBS=1
```

8. Close the Project Options dialog.

9. Go to Project→Edit Option Source and add the following libraries to the end of the SPARELIBS value:

   ```
   ACE_bs.lib TAO_bs.lib
   ```

   You may add other TAO libraries as required for the CORBA Services and other TAO features. To statically link your application you will need to specify a list of libraries that completely satisfies the symbols required for linking. For example, to use the Naming Service may have to add:

   ```
   TAO_CosNaming_bs.lib
   TAO_Svc_Utils_bs.lib
   TAO_PortableServer_bs.lib
   ```

10. Add the following library to the end of the ALLLIB value:

    ```
    ws2_32.lib
    ```

11. Save and close the project's option source.

12. Build your application.

2.2.3 Creating a Dynamically-Linked VCL Project

Note: These instructions assume that you have either:

- installed a pre-built copy of the dynamically-linked ACE+TAO libraries, executables and header files; or
- built and installed the dynamically-linked version of ACE+TAO according to the instructions in appendix A.

1. Create a new VCL application by going File→New→Application.

2. In all *.cpp* files in the project, replace occurrences of

   ```
   #include <vcl.h>
   #pragma hdrstop
   ```

   with

```
#include <tao/corba.h>
#pragma hdrstop
```

3. TAO is built on top of the ACE library, so you need to initialise ACE from your program. To do this you must add calls to `ACE::init` and `ACE::fini` from startup and cleanup functions:

    ```
    #pragma package(smart_init)

    void ace_init(void)
    {
    #pragma startup ace_init
      ACE::init();
    }

    void ace_fini(void)
    {
    #pragma exit ace_fini
      ACE::fini();
    }

    WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
    {
      // ...
    }
    ```

    For VCL applications it is important that you have the line

    ```
    #pragma package(smart_init)
    ```

    so that the `ACE::init` and `ACE::fini` calls occur before and after the VCL cleanup and initialisation respectively.

4. Go to Project→Options..., then to the Linker page and make sure that Use dynamic RTL *is* checked.

5. Still in the Project Options dialog, go to the Directories/Conditionals page.

6. Add the following directory to the include path:

    ```
    $(ACETAODIR)\include
    ```

Note: See section A.2.5 if you have not set up the ACETAODIR environment variable.

7. Add this directory to the library path:

   ```
   $(ACETAODIR)\lib
   ```

8. Add the following conditional define:

   ```
   ACE_HAS_VCL=1
   ```

   This define causes the TAO header files to include *vcl.h* for you automatically. This is important as ACE+TAO, for reasons of portability, needs to control the order in which system header files are included.

9. Close the Project Options dialog.

10. Go to Project→Edit Option Source and add the following libraries to the end of the SPARELIBS value:

    ```
    ACE_b.lib TAO_b.lib
    ```

    You may add other TAO libraries as required for the CORBA Services and other TAO features. For example, to use the Naming Service you would add:

    ```
    TAO_CosNaming_b.lib
    ```

11. Save and close the project's option source.

12. Build your application.

2.2.4 Creating a Statically-Linked VCL Project

Note: These instructions assume that you have built and installed the statically-linked version of ACE+TAO according to the instructions in appendix A. A pre-built statically-linked version is not available.

1. Create a new VCL application by going File→New→Application.

2. In all *.cpp* files in the project, replace occurrences of

   ```
   #include <vcl.h>
   #pragma hdrstop
   ```

with

```
#include <tao/corba.h>
#pragma hdrstop
```

3. TAO is built on top of the ACE library, so you need to initialise ACE from your program. To do this you must add calls to `ACE::init` and `ACE::fini` from startup and cleanup functions:

   ```
   #pragma package(smart_init)

   void ace_init(void)
   {
   #pragma startup ace_init
     ACE::init();
   }

   void ace_fini(void)
   {
   #pragma exit ace_fini
     ACE::fini();
   }

   WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
   {
     // ...
   }
   ```

   For VCL applications it is important that you have the line

   ```
   #pragma package(smart_init)
   ```

   so that the `ACE::init` and `ACE::fini` calls occur before and after the VCL cleanup and initialisation respectively.

4. Go to Project→Options..., then to the Linker page and make sure that Use dynamic RTL *is not* checked.

5. Still in the Project Options dialog, go to the Directories/Conditionals page.

6. Add the following directory to the include path:

   ```
   $(ACETAODIR)\include
   ```

Note: See section A.2.5 if you have not set up the `ACETAODIR` environment variable.

7. Add this directory to the library path:

   ```
   $(ACETAODIR)\lib
   ```

8. Add the following conditional defines:

   ```
   ACE_AS_STATIC_LIBS=1
   TAO_AS_STATIC_LIBS=1
   ACE_HAS_VCL=1
   ```

   The `ACE_HAS_VCL` define causes the TAO header files to include ***vcl.h*** for you automatically. This is important as ACE+TAO, for reasons of portability, needs to control the order in which system header files are included.

9. Close the Project Options dialog.

10. Go to Project→Edit Option Source and add the following libraries to the end of the `SPARELIBS` value:

    ```
    ACE_bs.lib TAO_bs.lib
    ```

    You may add other TAO libraries as required for the CORBA Services and other TAO features. To statically link your application you will need to specify a list of libraries that completely satisfies the symbols required for linking. For example, to use the Naming Service may have to add:

    ```
    TAO_CosNaming_bs.lib
    TAO_Svc_Utils_bs.lib
    TAO_PortableServer_bs.lib
    ```

11. Add the following library to the end of the `ALLLIB` value:

    ```
    ws2_32.lib
    ```

12. Save and close the project's option source.

13. Build your application.

2.3 Using Precompiled Headers

If you include IDL-generated files in your project, or if you want to create an application of any great size, your compile speed will be improved if you create your own header file to use for precompiled header generation.

The simplest example of such a file would be something like:

```
#ifndef PchH
#define PchH

#include <tao/corba.h>

#endif
```

Further improvements in compile speed can be made by adding additional CORBA header files that you use:

```
#ifndef PchH
#define PchH

#include <tao/corba.h>
#include <orbsvcs/CosNamingC.h>
#include <orbsvcs/CosEventChannelAdminC.h>

#endif
```

Unfortunately, some of the TAO header files contain source code that is not compatible with C++Builder's precompiled header mechanism. You may want to experiment with the different header files to see what results you get.

2.4 Using the IDL Compiler

The simplest way to use the IDL compiler from the C++Builder IDE is to use a batch file project.

To run the IDL compiler with the default options, use a batch file project with commands like the following:

```
set TAO_ROOT=%ACETAODIR%\include
tao_idl myfile.idl
```

To generate source files which include a precompiled-header file, use commands like the following:

```
set TAO_ROOT=%ACETAODIR%\include
tao_idl -Wb,pch_include=pch.h myfile.idl
```

Given an input file called *name***.idl**, the IDL compiler will generate the following files:

Name	**Description**
*name***C.h**	Client-side declarations.
*name***C.i**	Client-side inline implementation.
*name***C.cpp**	Client-side implementation. Add this source file to your project for both client and server programs.
*name***S.h**	Server-side declarations.
*name***S.i**	Server-side inline implementation.
*name***S.cpp**	Server-side implementation. Add this source file to your project for server programs.
*name***S_T.h**	Server-side template declarations.
*name***S_T.i**	Server-side template inline implementation.
*name***S_T.cpp**	Server-side template implementation. There is no need to add this file to any project, it is included automatically.

Chapter 3

Hello World - A Tutorial

This tutorial takes you through the creation of a simple CORBA application to illustrate the basic steps involved in using TAO with C++Builder 6. It shows the use of CORBA in a console application.

Note: These instructions assume that you have either:

- installed a pre-built copy of the ACE+TAO libraries, executables and header files; or
- built and installed ACE+TAO according to the instructions in appendix A.

The instructions also make use of the *ACE+TAO Project Wizard* to simplify the creation of CORBA applications. Please refer to section 2.1 to check that you have this installed in the C++Builder 6 IDE.

3.1 Writing the IDL

The first step in developing this simple distributed application is to write the IDL. The IDL defines the operations that may be executed remotely, and is used to generate the C++ source code that we will use in our client and server programs.

For Hello World, define the following IDL in a file called ***hello.idl***:

```
#ifndef HELLO_IDL
#define HELLO_IDL

interface Hello
{
  void say_hello(in string name);
};

#endif
```

As you can see, we have defined an interface called `Hello` which contains a single operation `say_hello`. This operation takes one input parameter, a string containing the name of a person.

To compile this IDL file, do the following:

1. Go to File→New→Other... and create a new Batch File.
2. Right-click the batch file in the Project Manager, select Edit/Options..., and add the following commands:

   ```
   set TAO_ROOT=%ACETAODIR%\include
   tao_idl hello.idl
   ```

3. Save the batch file project into the same directory as the IDL file.
4. Right-click the batch file in the Project Manager and select Execute. You will see that a number of source files have now been generated.

3.2 Writing the Client

We will begin development of the application itself with the console-based client program.

1. Go to File→New→Other..., switch to the ACE+TAO tab and choose to create a new TAO Application.
2. Set the Application Type to Console and make sure that Use VCL *is not* checked.
3. Set the Linkage to Dynamic and ensure that Debug Libs *is not* checked.
4. Under Precompiled Headers, uncheck Use PCH since we will not be using a precompiled header for this console project.
5. The ACE+TAO Location should be set to `$(ACETAODIR)` and the Location is an Installed Copy.

6. Press OK to generate a new TAO console application, and use File→Save All to save the files as ***HelloClient.cpp*** and ***HelloClient.bpr*** in the same directory as the IDL file created above.

7. Add the IDL-generated source file ***helloC.cpp*** to the project. This file contains the `Hello` interface's *stub* code. The stub code is a group of classes that a client application can use to communicate with any server objects that implement the `Hello` interface.

8. Returning to ***HelloClient.cpp***, we must include the header file containing the client-side declarations, ***helloC.h***, so that the IDL definitions for the `Hello` interface are available to the client program.

```
#include <tao/corba.h>
#include <iostream>
#include <fstream>
#include <string>
#include "helloC.h"
```

9. Next the ORB object must be created and initialised using the command line arguments. The `ORB_init` function strips away any command line arguments that it recognises as ORB-related, so that you are left only with your application's arguments to process yourself.

```
int main(int argc, char* argv[])
{
  try
  {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

10. The client expects that `argv[1]` specifies the name of a file that contains what is called an Interoperable Object Reference (IOR). This is a string containing all the necessary information for a client to contact the server object, and is somewhat like a distributed pointer.

```
    // Check command line arguments.
    if (argc != 2)
    {
      std::cerr << "Usage: HelloClient <filename>"
                << std::endl;
      return 1;
    }
```

11. The `string_to_object` function takes the IOR and converts it into an in-memory object reference. However, before we can use it as a `Hello` object, we must narrow it to the correct interface type. This `_narrow` has a similar purpose to a C++ `dynamic_cast`, and will return nil if it cannot cast to the specified type.

    ```
        // Get an object reference to the hello object.
        std::string ior;
        std::ifstream is(argv[1]);
        std::getline(is, ior);
        CORBA::Object_var obj
          = orb->string_to_object(ior.c_str());
        Hello_var hello = Hello::_narrow(obj);
        if (CORBA::is_nil(hello))
        {
          std::cerr << "Unable to get hello reference"
                    << std::endl;
          return 1;
        }
    ```

12. Now the client has an object reference for the remote `Hello` object. This can be used this to invoke operations on the `Hello`, much as you would on a local object:

    ```
        // Send a message to the hello object.
        hello->say_hello("World");
    ```

13. Finally, before the client exits we clean up the ORB object.

    ```
        orb->destroy();
    ```

14. We must have exception handlers to catch any CORBA exceptions that may be thrown when an error occurs. For example, an exception may be caught if the client is unable to communicate with the server, say if the server is not running.

```
      }
      catch (CORBA::Exception& e)
      {
        std::cerr << "CORBA Exception: " << e << std::endl;
      }

      return 0;
    }
```

15. The ***HelloClient.cpp*** source file should now look like this:

```
#include <tao/corba.h>
#include <iostream>
#include <fstream>
#include <string>
#include "helloC.h"

int main(int argc, char* argv[])
{
  try
  {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Check command line arguments.
    if (argc != 2)
    {
      std::cerr << "Usage: HelloClient <filename>"
                << std::endl;
      return 1;
    }

    // Get an object reference to the hello object.
    std::string ior;
    std::ifstream is(argv[1]);
    std::getline(is, ior);
    CORBA::Object_var obj
      = orb->string_to_object(ior.c_str());
    Hello_var hello = Hello::_narrow(obj);
    if (CORBA::is_nil(hello))
    {
      std::cerr << "Unable to get hello reference"
                << std::endl;
      return 1;
    }

    // Send a message to the hello object.
    hello->say_hello("World");

    orb->destroy();
  }
```

```
        catch (CORBA::Exception& e)
        {
          std::cerr << "CORBA Exception: " << e << std::endl;
        }
        return 0;
      }
```

16. Build the project. However, it cannot be run until we have written a corresponding server program.

3.3 Writing the Server

Now we will develop a server program that implements the `Hello` interface we defined in the IDL.

1. Go to File→New→Other. . . , switch to the ACE+TAO tab and choose to create a new TAO Application.

2. Set the Application Type to Console and make sure that Use VCL *is not* checked.

3. Set the Linkage to Dynamic and ensure that Debug Libs *is not* checked.

4. Under Precompiled Headers, uncheck Use PCH since we will not be using a precompiled header.

5. In the CORBA Libraries list, check the entry called PortableServer.

6. The ACE+TAO Location should be set to `$(ACETAODIR)` and the Location is an Installed Copy.

7. Press OK to generate a new TAO console application, and use File→Save All to save the files as ***HelloServer.cpp*** and ***HelloServer.bpr*** in the same directory as the IDL file created above.

8. Add the IDL-generated source files ***helloC.cpp*** and ***helloS.cpp*** to the project. These files contain the `Hello` interface's *stub* code and *skeleton* code respectively. The skeleton code is a group of classes that a server application can use to receive requests from clients through the `Hello` interface.

9. Returning to ***HelloServer.cpp***, we must include the header file containing the server-side declarations, ***helloS.h***, so that the IDL definitions required to implement the `Hello` interface are available to the server program.

```
#include <tao/corba.h>
#include <iostream>
#include <fstream>
#include "helloS.h"
```

10. The `Hello` interface defined in the IDL must be implemented in the C++ code by what is called a *servant*. To write a servant we must derive a class from `POA_Hello`, and provide an implementation of the `say_hello` function. The `POA_Hello` class is an abstract base class generated by the IDL compiler.

```
class HelloImpl : public virtual POA_Hello
{
public:
  virtual void say_hello(const char* name)
    throw(CORBA::SystemException);
};
```

11. Write the definition of the `say_hello` function:

```
void HelloImpl::say_hello(const char* name)
  throw(CORBA::SystemException)
{
  std::cout << "Hello " << name << std::endl;
}
```

12. Next we move to the `main` fuction, where we declare an instance of the above servant. Since we are using a stack-based servant it is important that it have a longer lifetime than the ORB (i.e. it is declared first). This is necessary to make sure that the servant is not destroyed while the ORB is still using it. In the second tutorial we see how to use a reference-counted heap-based servant which takes care of these lifetime issues automatically.

```
int main(int argc, char* argv[])
{
  try
  {
    HelloImpl hello_servant;
```

13. The ORB object must be created and initialised using the command line arguments. The `ORB_init` function strips away any command line arguments that it recognises as ORB-related, so that you are left only with your application's arguments to process yourself.

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

14. The server expects that `argv[1]` contains the name of a file. Later, the Interoperable Object Reference (IOR) will be written to this file.

```
// Check command line arguments.
if (argc != 2)
{
  std::cerr << "Usage: HelloServer <filename>"
            << std::endl;
  return 1;
}
```

15. All servants must be registered with a Portable Object Adapter (POA). A POA is responsible for delivering incoming requests to the correct servant. In this example we will simply use the so-called *Root POA*, which has been created for you automatically by the ORB. POA objects in the ORB are arranged hierarchically, and have a range of configurable options. In larger applications you would typically create custom POA objects according to your specific needs.

```
// Get a reference to the Root POA.
CORBA::Object_var poa_obj
  = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa
  = PortableServer::POA::_narrow(poa_obj);
```

16. Before a POA can process incoming requests it must be activated. This is done via its associated manager object.

```
// Activate the POA manager.
PortableServer::POAManager_var poa_mgr
  = poa->the_POAManager();
poa_mgr->activate();
```

17. Next we will activate the servant and write its IOR to the file. First, the calling the `_this` function on the servant automatically registers it with the Root POA, and returns the corresponding CORBA object that may be used by remote clients. The `object_to_string` function takes the object and converts it into an IOR string. This IOR can be made available to client programs by some means (in this example we write it to a file).

```
        // Activate the hello object and write out its IOR.
        Hello_var hello_obj = hello_servant._this();
        CORBA::String_var str
          = orb->object_to_string(hello_obj);
        std::ofstream os(argv[1]);
        os << str << std::endl;
        os.close();
```

18. Run the ORB event loop to allow it to process incoming requests from clients.

```
        orb->run();
```

19. Finally, we must have exception handlers to catch any CORBA exceptions that may be thrown when an error occurs.

```
      }
      catch (CORBA::Exception& e)
      {
        std::cerr << "CORBA Exception: " << e << std::endl;
      }

      return 0;
    }
```

20. The ***HelloServer.cpp*** source file should now look like this:

```
    #include <tao/corba.h>
    #include <iostream>
    #include <fstream>
    #include "helloS.h"

    class HelloImpl : public virtual POA_Hello
    {
    public:
      virtual void say_hello(const char* name)
        throw(CORBA::SystemException);
    };

    void HelloImpl::say_hello(const char* name)
      throw(CORBA::SystemException)
    {
```

```
  std::cout << "Hello " << name << std::endl;
}

int main(int argc, char* argv[])
{
  try
  {
    HelloImpl hello_servant;

    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Check command line arguments.
    if (argc != 2)
    {
      std::cerr << "Usage: HelloServer <filename>"
                << std::endl;
      return 1;
    }

    // Get a reference to the Root POA.
    CORBA::Object_var poa_obj
      = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa
      = PortableServer::POA::_narrow(poa_obj);

    // Activate the POA manager.
    PortableServer::POAManager_var poa_mgr
      = poa->the_POAManager();
    poa_mgr->activate();

    // Activate the hello object and write out its IOR.
    Hello_var hello_obj = hello_servant._this();
    CORBA::String_var str
      = orb->object_to_string(hello_obj);
    std::ofstream os(argv[1]);
    os << str << std::endl;
    os.close();

    orb->run();
  }
```

```
        catch (CORBA::Exception& e)
        {
          std::cerr << "CORBA Exception: " << e << std::endl;
        }

        return 0;
      }
```

21. Build the project.

3.4 Running the Application

To run the application, follow these steps:

1. Open a Command Prompt (DOS Box) and change to the directory containing the client and server executables.

2. Run the server program, specifying that the hello object's IOR should be written to a file called ***hello.ior***.

   ```
   HelloServer hello.ior
   ```

3. Open a second Command Prompt (DOS Box) and again change to the directory containing the client and server executables.

4. Run the client program, specifying that the file containing the IOR is called ***hello.ior***.

   ```
   HelloClient hello.ior
   ```

5. You should see the text `Hello World` appear on the server program's output.

Chapter 4

VCL Hello World - A Tutorial

This tutorial takes you through the creation of VCL-based CORBA application to show how to use TAO in an application with a graphical user interface.

Note: These instructions assume that you have either:

- installed a pre-built copy of the ACE+TAO libraries, executables and header files; or
- built and installed ACE+TAO according to the instructions in appendix A.

The instructions also make use of the *ACE+TAO Project Wizard* to simplify the creation of CORBA applications. Please refer to section 2.1 to check that you have this installed in the C++Builder 6 IDE.

4.1 Writing the IDL

As with the console-based tutorial, define the following IDL in a file called ***hello.idl***:

```
#ifndef HELLO_IDL
#define HELLO_IDL

interface Hello
{
  void say_hello(in string name);
};

#endif
```

To compile this IDL file, do the following:

1. Go to File→New→Other... and create a new Batch File.
2. Right-click the batch file in the Project Manager, select Edit/Options..., and add the following commands:

   ```
   set TAO_ROOT=%ACETAODIR%\include
   tao_idl -Wb,pch_include=pch.h hello.idl
   ```

   In this example we will be using a precompiled header file. The `-Wb,pch_include` command-line argument is used to specify the name of this file so that it is included in the IDL-generated source code.
3. Save the batch file project into the same directory as the IDL file.
4. Right-click the batch file in the Project Manager and select Execute. You will see that a number of source files have now been generated.

4.2 Writing the Client

We will begin coding the application itself by starting with the console-based client program.

1. Go to File→New→Other..., switch to the ACE+TAO tab and choose to create a new TAO Application.
2. Set the Application Type to VCL.
3. Set the Linkage to Dynamic and ensure that Debug Libs *is not* checked.
4. Under Precompiled Headers, make sure that Use PCH *is* checked.
5. The ACE+TAO Location should be set to `$(ACETAODIR)` and the Location is an Installed Copy.
6. Press OK to generate a new TAO VCL application, and use File→Save All to save the files as ***HelloClientWnd.cpp*** and ***HelloClient.bpr*** in the same directory as the IDL file created above.
7. Create a new header file called ***pch.h*** with the following contents:

```
#ifndef PchH
#define PchH

#include <tao/corba.h>

#endif
```

8. Go to Project→Options. . . , switch to the Compiler tab, and change the value for Stop after in the Pre-compiled Headers group to be `pch.h`.

9. For every *.cpp* file currently in the project, replace the lines:

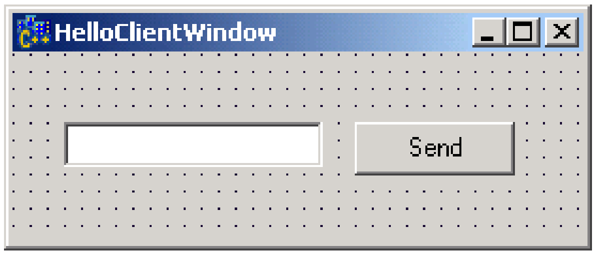
   ```
   #include <tao/corba.h>
   #pragma hdrstop
   ```

   with:

   ```
   #include "pch.h"
   #pragma hdrstop
   ```

10. Add the IDL-generated source file ***helloC.cpp*** to the project.

11. Switch to the design view of the form and, using the Object Inspector change its name property to `HelloClientWindow`. Then add an edit box and a button so that the form looks like this:



12. In the ***HelloClientWnd.h***, we must include the header file containing the client-side declarations, ***helloC.h***, so that the IDL definitions for the `Hello` interface are available to the client program. This must be included from the form's header file as we will be using some IDL-generated classes as data members.

    ```
    #include <Classes.hpp>
    #include <Controls.hpp>
    #include <StdCtrls.hpp>
    #include <Forms.hpp>
    #include "helloC.h"
    ```

13. Add data members to hold a reference to the ORB object and to a remote `Hello` object.

    ```
    private:  // User declarations
      CORBA::ORB_var orb_;
      Hello_var hello_;
    ```

14. Next, in the file ***HelloClientWnd.cpp*** the ORB object is created and initialised from the form's constructor. We pass it the global command-line argument variables `_argc` and `_argv`.

    ```
    __fastcall THelloClientWindow::THelloClientWindow(
          TComponent* Owner)
      : TForm(Owner)
    {
      // Initialise the ORB.
      orb_ = CORBA::ORB_init(_argc, _argv);
    ```

15. Still in the constructor, we will use the `string_to_object` function to obtain a reference to the remote `Hello` object. Here we use a URL-style IOR to simplify the extraction of the reference from a file called ***hello.ior***.

    ```
      // Get a reference to the hello object.
      CORBA::Object_var obj
        = orb_->string_to_object("file://hello.ior");
      hello_ = Hello::_narrow(obj);
      if (CORBA::is_nil(hello_))
        throw Exception("Unable to get hello object");
    }
    ```

16. Now the client has an object reference for the `Hello` object which can be used to make remote calls. Add a handler for the button's `OnClick` event which sends the contents of the edit box using the `say_hello` function:

    ```
    void __fastcall THelloClientWindow::Button1Click(
          TObject *Sender)
    {
      hello_->say_hello(Edit1->Text.c_str());
    }
    ```

17. The project source file ***HelloClient.cpp*** should now look like this:

```
//--------------------------------------------------
#include "pch.h"
#pragma hdrstop
#include <ace\ACE.h>
#include <sstream>
//--------------------------------------------------
USEFORM("Unit1.cpp", HelloClientWindow);
//--------------------------------------------------
#pragma package(smart_init)
void ace_init(void)
{
#pragma startup ace_init
  ACE::init();
}
void ace_fini(void)
{
#pragma exit ace_fini
  ACE::fini();
}
//--------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  try
  {
    Application->Initialize();
    Application->CreateForm(
        __classid(THelloClientWindow),
        &HelloClientWindow);
    Application->Run();
  }
  catch (CORBA::Exception &exception)
  {
    try
    {
      std::ostringstream os;
      os << "CORBA Exception: " << exception;
      throw Exception(os.str().c_str());
    }
    catch (Exception &exception)
    {
          Application->ShowException(&exception);
        }
      }
      catch (Exception &exception)
      {
        Application->ShowException(&exception);
      }
      return 0;
    }
    //--------------------------------------------------
```

18. The form's header file ***HelloClientWnd.h*** should contain:

```
    //--------------------------------------------------
    #ifndef HelloClientWndH
    #define HelloClientWndH
    //--------------------------------------------------
    #include <Classes.hpp>
    #include <Controls.hpp>
    #include <StdCtrls.hpp>
    #include <Forms.hpp>
    #include "HelloC.h"
    //--------------------------------------------------
    class THelloClientWindow : public TForm
    {
    __published:  // IDE-managed Components
      TEdit *Edit1;
      TButton *Button1;
      void __fastcall Button1Click(TObject *Sender);
    private:  // User declarations
      CORBA::ORB_var orb_;
      Hello_var hello_;
    public:   // User declarations
      __fastcall THelloClientWindow(TComponent* Owner);
    };
    //--------------------------------------------------
    extern PACKAGE THelloClientWindow *HelloClientWindow;
    //--------------------------------------------------
    #endif
```

19. The form's source file ***HelloClientWnd.cpp*** would look like:

```
//--------------------------------------------------
#include "pch.h"
#pragma hdrstop
#include "HelloClientWnd.h"
//--------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
THelloClientWindow *HelloClientWindow;
//--------------------------------------------------
__fastcall THelloClientWindow::THelloClientWindow(
      TComponent* Owner)
  : TForm(Owner)
{
  // Initialise the ORB.
  orb_ = CORBA::ORB_init(_argc, _argv);

  // Get a reference to the hello object.
  CORBA::Object_var obj
    = orb_->string_to_object("file://hello.ior");
  hello_ = Hello::_narrow(obj);
  if (CORBA::is_nil(hello_))
    throw Exception("Unable to get hello object");
}
//--------------------------------------------------
void __fastcall THelloClientWindow::Button1Click(
      TObject *Sender)
{
  hello_->say_hello(Edit1->Text.c_str());
}
//--------------------------------------------------
```

20. Build the project.

4.3 Writing the Server

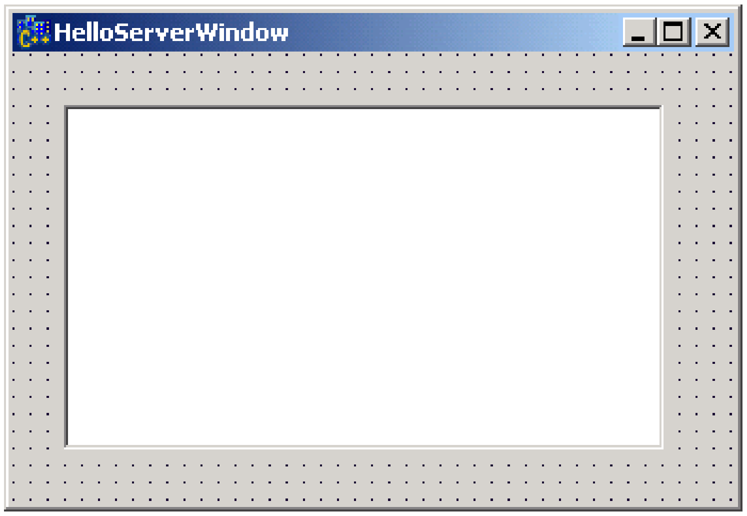
Now we will develop the VCL server program that implements the `Hello` interface we defined in the IDL.

1. Go to File→New→Other. . . , switch to the ACE+TAO tab and choose to create a new TAO Application.

2. Set the Application Type to VCL.
3. Set the Linkage to Dynamic and ensure that Debug Libs *is not* checked.
4. Under Precompiled Headers, make sure that Use PCH *is* checked.
5. In the CORBA Libraries list, check the entry called PortableServer.
6. The ACE+TAO Location should be set to `$(ACETAODIR)` and the Location is an Installed Copy.
7. Press OK to generate a new TAO VCL application, and use File→Save All to save the files as ***HelloServerWnd.cpp*** and ***HelloServer.bpr*** in the same directory as the IDL file created above.
8. Add a new unit (to contain the implementation of the `Hello` interface) and save it as ***HelloImpl.cpp***.
9. Go to Project→Options..., switch to the Compiler tab, and change the value for Stop after in the Pre-compiled Headers group to be `pch.h`.
10. Make sure every *.cpp* file currently in the project starts with the lines:

    ```
    #include "pch.h"
    #pragma hdrstop
    ```

11. Add the IDL-generated source files ***helloC.cpp*** and ***helloS.cpp*** to the project.
12. Switch to the design view of the form and, using the Object Inspector change its name property to `HelloServerWindow`. Then add a memo control so that the form looks like this:



13. In the file ***HelloImpl.h***, we must include the header file containing the server-side declarations, ***helloS.h***, so that the IDL definitions required to implement the `Hello` interface are available to the server program.

    ```
    //-------------------------------------------------
    #ifndef HelloImplH
    #define HelloImplH
    //-------------------------------------------------
    #include "helloS.h"
    ```

14. To implement the servant we derive a class from `POA_Hello`, and provide an implementation of the `say_hello` function. The class is also derived from `PortableServer::RefCountServantBase`, which is a mixin class that adds reference counting support to the servant.

    ```
    class HelloImpl : public virtual POA_Hello,
      public virtual PortableServer::RefCountServantBase
    {
    public:
      virtual void say_hello(const char* name)
        throw(CORBA::SystemException);
    };
    ```

15. In ***HelloImpl.cpp*** include the header file for the form:

    ```
    #include "HelloServerWnd.h"
    ```

16. Write the definition of the `say_hello` function so that it adds a line of output to the memo:

    ```
    void HelloImpl::say_hello(const char* name)
      throw(CORBA::SystemException)
    {
      String line = "Hello ";
      line += name;
      HelloServerWindow->Memo1->Lines->Append(line);
    }
    ```

17. The ORB event loop needs to be run so that the ORB can process incoming requests. In the previous console-based tutorial this event loop was simply run from the `main` function in the program's main thread. In a VCL application this is not possible since the VCL event loop uses the main thread.

Instead, the ORB will be run using a background thread. To do this, go to File→New→Other..., switch to the ACE+TAO tab and choose ORB Thread. Name the class `TORBThread` and save the generated unit as ***ORBThread.cpp***. You should also change ***ORBThread.cpp*** so that, like the other ***.cpp*** files, it includes ***pch.h*** first.

18. In the ***HelloServerWnd.h*** file, include the header file ***ORBThread.h*** which contains the declaration of the ORB thread class.

    ```
    #include <Classes.hpp>
    #include <Controls.hpp>
    #include <StdCtrls.hpp>
    #include <Forms.hpp>
    #include <memory>
    #include "ORBThread.h"
    ```

19. Add a data member to the `THelloServerWindow` class to point to a `TORBThread` object. We use an `auto_ptr` so that the object is deleted automatically when the form is destroyed.

    ```
    private:  // User declarations
      std::auto_ptr<TORBThread> orb_thread_;
    ```

20. In the file ***HelloServerWnd.cpp***, we include the servant's header file ***HelloImpl.h***.

    ```
    #include "pch.h"
    #pragma hdrstop
    #include <fstream>
    #include "HelloServerWnd.h"
    #include "HelloImpl.h"
    ```

21. The ORB object is created and initialised from the form's constructor. We pass it the global command-line argument variables `_argc` and `_argv`.

    ```
    __fastcall THelloServerWindow::THelloServerWindow(
          TComponent* Owner)
      : TForm(Owner)
    {
      // Initialise the ORB.
      CORBA::ORB_var orb
        = CORBA::ORB_init(_argc, _argv);
    ```

22. Still in the form constructor, we obtain a reference to the Root POA, and activate its manager so that it can process requests.

```
    // Get a reference to the Root POA.
    CORBA::Object_var poa_obj
      = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa
      = PortableServer::POA::_narrow(poa_obj);

    // Activate the POA manager.
    PortableServer::POAManager_var poa_mgr
      = poa->the_POAManager();
    poa_mgr->activate();
```

23. Next we create an instance of the servant class, and register it with the POA. Since we are now using a reference counted servant, we call `_remove_ref` to indicate that we are done with the servant. The POA will also maintain a reference to the servant as long as required, ensuring that the servant is not deleted until everything has finished using it.

```
    // Activate the hello object and write out its IOR.
    HelloImpl* hello_servant = new HelloImpl;
    Hello_var hello_obj = hello_servant->_this();
    hello_servant->_remove_ref();
    CORBA::String_var str
      = orb->object_to_string(hello_obj);
    std::ofstream os("hello.ior");
    os << str << std::endl;
    os.close();
```

Note: The explicit call to `_remove_ref` shown above is not exception safe, since if the `_this` call throws an exception, the servant will leak. There is a smart pointer class called `PortableServer::ServantBase_var` that can be used to take care of incrementing and decrementing the reference count automatically.

24. Finally, we create a `TORBThread` object to run the ORB event loop.

```
    orb_thread_.reset(new TORBThread(orb));
  }
```

25. The ***HelloServer.cpp*** source file should now look like this:

```
//-----------------------------------------------------
#include "pch.h"
#pragma hdrstop
#include <ace\ACE.h>
#include <sstream>
//-----------------------------------------------------
USEFORM("HelloServerWnd.cpp", HelloServerWindow);
//-----------------------------------------------------
#pragma package(smart_init)
void ace_init(void)
{
#pragma startup ace_init
  ACE::init();
}
void ace_fini(void)
{
#pragma exit ace_fini
  ACE::fini();
}
//-----------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  try
  {
    Application->Initialize();
    Application->CreateForm(
        __classid(THelloServerWindow),
        &HelloServerWindow);
    Application->Run();
  }
  catch (CORBA::Exception &exception)
  {
    try
    {
      std::ostringstream os;
      os << "CORBA Exception: " << exception;
      throw Exception(os.str().c_str());
    }
    catch (Exception &exception)
    {
      Application->ShowException(&exception);
    }
  }
  catch (Exception &exception)
  {
    Application->ShowException(&exception);
  }
  return 0;
}
//--------------------------------------------------
```

26. The form’s header file ***HelloServerWnd.h*** should contain:

```
//--------------------------------------------------
#ifndef HelloServerWndH
#define HelloServerWndH
//--------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <memory>
#include "ORBThread.h"
//--------------------------------------------------
class THelloServerWindow : public TForm
{
__published:  // IDE-managed Components
  TMemo *Memo1;
private:  // User declarations
  std::auto_ptr<TORBThread> orb_thread_;
public:   // User declarations
  __fastcall THelloServerWindow(TComponent* Owner);
};
//--------------------------------------------------
extern PACKAGE THelloServerWindow *HelloServerWindow;
//--------------------------------------------------
#endif
```

27. The form’s source file ***HelloServerWnd.cpp*** should contain:

```
//--------------------------------------------------
```

```
#include "pch.h"
#pragma hdrstop
#include <fstream>
#include "HelloServerWnd.h"
#include "HelloImpl.h"
//-------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
THelloServerWindow *HelloServerWindow;
//-------------------------------------------------
__fastcall THelloServerWindow::THelloServerWindow(
      TComponent* Owner)
  : TForm(Owner)
{
  // Initialise the ORB.
  CORBA::ORB_var orb
    = CORBA::ORB_init(_argc, _argv);

  // Get a reference to the Root POA.
  CORBA::Object_var poa_obj
    = orb->resolve_initial_references("RootPOA");
  PortableServer::POA_var poa
    = PortableServer::POA::_narrow(poa_obj);

  // Activate the POA manager.
  PortableServer::POAManager_var poa_mgr
    = poa->the_POAManager();
  poa_mgr->activate();

  // Activate the hello object and write out its IOR.
  HelloImpl* hello_servant = new HelloImpl;
  Hello_var hello_obj = hello_servant->_this();
  hello_servant->_remove_ref();
  CORBA::String_var str
    = orb->object_to_string(hello_obj);
  std::ofstream os("hello.ior");
  os << str << std::endl;
  os.close();

  orb_thread_.reset(new TORBThread(orb));
}
//--------------------------------------------------
```

28. The servant's header file ***HelloImpl.h*** should have the contents:

```
//--------------------------------------------------
#ifndef HelloImplH
#define HelloImplH
//--------------------------------------------------
#include "helloS.h"
//--------------------------------------------------
class HelloImpl : public virtual POA_Hello,
  public virtual PortableServer::RefCountServantBase
{
public:
  virtual void say_hello(const char* name)
    throw(CORBA::SystemException);
};
//--------------------------------------------------
#endif
```

29. The servant's source file ***HelloImpl.cpp*** would now look like:

```
//--------------------------------------------------
#include "pch.h"
#pragma hdrstop
#include "HelloImpl.h"
#include "HelloServerWnd.h"
//--------------------------------------------------
void HelloImpl::say_hello(const char* name)
  throw(CORBA::SystemException)
{
  String line = "Hello ";
  line += name;
  HelloServerWindow->Memo1->Lines->Append(line);
}
//--------------------------------------------------
```

30. Build the project.

4.4 Running the Application

To run the application, follow these steps:

1. Run the server program by double clicking it in Windows Explorer.
2. Run the client program by double clicking it in Windows Explorer.
3. Type your name into the edit box in the client program, and press the Send button.
4. You should see the text `Hello` *`Your Name`* appear on the server program's memo.

Appendix A

Building and Installing ACE+TAO

This appendix explains how to build ACE+TAO from source code using C++Builder 6. For instructions on how to build ACE+TAO using earlier versions of C++Builder, please see the *TAO with C++Builder* website located at `http://www.tenermerx.com/tao_bcb`.

A.1 Quick Build Instructions

These are the basic steps for building TAO using C++Builder 6. For more detailed information, or if you run into problems, please see the additional sections below.

1. Open a Command Prompt (DOS Box).
2. Set the `ACE_ROOT` environment variable to the path where you unpacked the source kit. The source kit is typically in a directory called ***ACE_wrappers***. For example:

   ```
   set ACE_ROOT=C:\ACETAO\src\ACE_wrappers
   ```

3. Let the build process know what version of C++Builder you are using:

   ```
   set BCBVER=6
   ```

4. If you are building on Windows 9*x*, download an additional directory creation utility from `http://www.tenermerx.com/programming/corba/tao_bcb/build/mkdirtree.zip` and put the ***.exe*** on your path (e.g. in ***C:\Windows***). Then type:

   ```
   set MKDIR=mkdirtree
   ```

before continuing.

5. Change to the ace source directory:

```
cd C:\ACETAO\src\ACE_wrappers\ace
```

6. Create the configuration header file. If you are building for Windows NT, 2000 or XP, create a config header file like this:

```
echo #include "ace/config-win32.h" > config.h
```

   If you are building for Windows 95, 98 or Me, you should create a config header file like this:

```
echo #define ACE_HAS_WINNT4 0 > config.h
echo #include "ace/config-win32.h" >> config.h
```

7. Change to the TAO source directory:

```
cd C:\ACETAO\src\ACE_wrappers\TAO
```

8. Build the ACE+TAO libraries and executables:

```
make -f Makefile.bor
```

9. Install the ACE+TAO header files, libraries and executables for use in your applications. Here we are installing them into ***C:\ACETAO***:

```
make -f Makefile.bor -DINSTALL_DIR=C:\ACETAO install
```

10. Add the directory containing the installed libraries and executables (in the example above it is ***C:\ACETAO\bin***) to your path. On Windows NT, 2000 and XP you can do this using the System Control Panel and updating the path environment variable.

11. To make it simpler to use TAO from inside the C++Builder IDE, set up an environment variable called `ACETAODIR` that points to the install directory that you specified above (e.g. ***C:\ACETAO***). This can be done from inside the C++Builder 6 IDE by going to the Tools menu, selecting Environment Options, then going to the Environment Variables tab and adding a new User override.

A.2 Detailed Build Instructions

The following sections provided detailed instructions on how to build and install TAO using C++Builder 6.

A.2.1 Checking System Requirements

Disk Space

You should make sure you have lots of disk space available before you build. As a rough guide it will take up to 250MB of space to build the dynamically linked release versions of the TAO libraries and executables. To build debug and/or statically linked versions will obviously consume more space. To build all configurations will probably require approximately 1GB of drive space, however most of the time you will not want or need to build all configurations.

C++Builder Command Line Tools

To build TAO the Borland C++Builder command line tools need to be on the path. These are usually installed in ***C:\Program Files\Borland\CBuilder6\bin***. This directory should have already been added to you path when you installed C++Builder 6, but if not you can add it to your path using the System Control Panel, or by executing the command:

```
set PATH=C:\Progra~1\Borland\CBuilder6\bin;%PATH%
```

A.2.2 Creating a Configuration Header File

Before building TAO you must create a file called ***config.h***, which needs to be located in the ***ACE_wrappers\ace*** directory. This file is used to tell the TAO build process what platform is being targeted, and also allows you to configure certain ACE and TAO features.

Typically this file will simply contain:

```
#include "ace/config-win32.h"
```

If building for Windows 95, 98 or Me, the file should contain the lines:

```
#define ACE_HAS_WINNT4 0
#include "ace/config-win32.h"
```

A.2.3 Build Configurations

TAO may be built in several different configurations. You should choose the configuration most appropriate to your needs (there may be more than one).

Option	Description	Suffix	Default
`DEBUG`	If enabled causes the TAO binaries to be built with debug information. You may want to build debug libraries if you want to track down a problem in TAO.	d	Off
`STATIC`	Determines whether to use statically linked libraries instead of DLLs. When using DLLs you must also ship your application with the C++Builder run-time library and TAO DLLs. Using static libraries will remove this need, but the static libraries themselves can be very large (especially when built with debug information).	s	Off
`CODEGUARD`	If enabled causes TAO binaries to be built with Codeguard support. Should be used only when `DEBUG` is also on. Does not have any affect on the output directory.	*n/a*	Off

These options may be switched on by using environment variables, for example:

```
set DEBUG=1
```

and turned off again:

```
set DEBUG=
```

You may also enable the options by passing them as command line arguments to make, for example:

```
make -f Makefile.bor -DDEBUG -DCODEGUARD
```

In addition to modifying the compiler options used to build TAO, these configuration options determine the output directory of the binaries. When building the dynamically-linked release version the core executables and utilities are put into ***ACE_wrappers\bin\Dynamic\Release***, dynamically-linked versions with debug information into ***ACE_wrappers\bin\Dynamic\Debug***, and so on.

The filenames of the libraries are also affected by the configuration options. Every TAO library is given a *_b* suffix to indicate that it is a library for a Borland compiler. Other suffixes are added according to the table above. For example, the dynamically-linked debug version of the TAO DLL is called ***tao_bd.dll***.

A.2.4 Build Steps

1. Open a Command Prompt (DOS Box).

2. Set the `ACE_ROOT` environment variable to the path where you unpacked the source kit. The source kit is typically in a directory called ***ACE_wrappers***. For example:

    ```
    set ACE_ROOT=C:\ACETAO\src\ACE_wrappers
    ```

3. Let the build process know what version of C++Builder you are using:

    ```
    set BCBVER=6
    ```

4. If you are building on Windows 9x the `mkdir` command may not support the ability to create a directory tree in a single invocation. This ability is required by the TAO makefiles. Download an additional directory creation utility from `http://www.tenermerx.com/programming/corba/tao_bcb/build/mkdirtree.zip` and put the ***.exe*** on your path (e.g. in ***C:\Windows***). Then type:

    ```
    set MKDIR=mkdirtree
    ```

    before continuing.

5. Change to the TAO source directory:

    ```
    cd C:\ACETAO\src\ACE_wrappers\TAO
    ```

6. If, in addition to the core libraries and executables, you want to build all of the tests and example programs, set the following environment variable:

    ```
    set BUILD=all
    ```

    Please note that this is not required to use TAO in your own applications, and setting this option will use lots of additional disk space.

7. Start the build:

```
make -f Makefile.bor
```

Note: On some systems you may get an error from make saying the command line arguments are too long. If this happens, try using the `-l` option when you run make, or failing that, reducing the length of your `PATH` environment variable.

8. Install the ACE+TAO header files, libraries and executables for use in your applications. Here we are installing them into ***C:\ACETAO***:

    ```
    make -f Makefile.bor -DINSTALL_DIR=C:\ACETAO install
    ```

    This command will copy the DLLs and executables into ***C:\ACETAO\bin***, the include files into ***C:\ACETAO\include***, and the ***.lib*** files required to link your applications into ***C:\ACETAO\lib***.

A.2.5 Using TAO from C++Builder

Before you can use TAO from inside the C++Builder IDE, you should make the following changes to your system:

1. Add the directory containing the installed libraries and executables (in the example in the build steps above it is ***C:\ACETAO\bin***) to your path. On Windows NT, 2000 and XP you can do this using the System Control Panel and updating the path environment variable.

2. To make it simpler to use TAO from inside the C++Builder IDE, set up an environment variable called `ACETAODIR` that points to the install directory that you specified above (e.g. ***C:\ACETAO***). This can be done from inside the C++Builder 6 IDE by going to the Tools menu, selecting Environment Options, then going to the Environment Variables tab and adding a new User override.

    Please note that you should not have any spaces in the value for `ACETAODIR`. For example, if TAO is installed in ***C:\Program Files\ACETAO*** use the short pathname ***C:\Progra∼1\ACETAO*** instead.