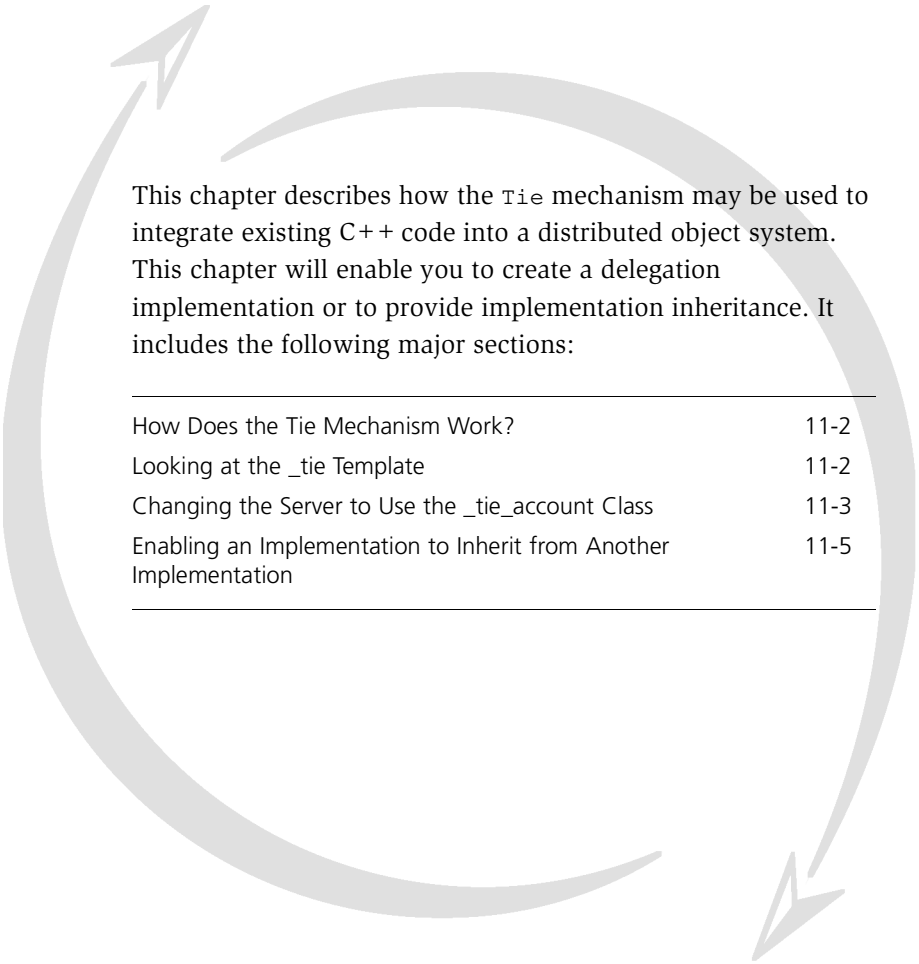
CHAPTER 11

USING THE TIE MECHANISM: AN ALTERNATIVE TO INHERITANCE



This chapter describes how the `Tie` mechanism may be used to integrate existing C++ code into a distributed object system. This chapter will enable you to create a delegation implementation or to provide implementation inheritance. It includes the following major sections:

How Does the Tie Mechanism Work?	11-2
Looking at the _tie Template	11-2
Changing the Server to Use the _tie_account Class	11-3
Enabling an Implementation to Inherit from Another Implementation	11-5

HOW DOES THE TIE MECHANISM WORK?

Object implementation classes normally inherit from a skeleton class generated by the `idl2cpp` compiler. The skeleton class, in turn, inherits from `CORBA::Object`. When it is not convenient or possible to change existing classes to inherit from the VisiBroker skeleton class, the *tie* mechanism offers an attractive alternative.

The tie mechanism provides object servers with a *delegator implementation* class that inherits from `CORBA::Object`. The delegator implementation does not provide any semantics of its own. It simply delegates every request it receives to the real implementation class, which can be implemented separately. The real implementation class is not required to inherit from `CORBA::Object`.

Steps for Modifying the Server to Use the Tie Mechanism

To use the tie mechanism, your server must perform the following actions:

1. Initialize the ORB and BOA.
2. Instantiate an `AccountImpl`.
3. Instantiate a `_tie_Account` and initialize it with the `AccountImpl`.
4. Export the `_tie_Account` object.
5. Wait for requests.

Location of an Example Program Using the Tie Mechanism

A version of the `Account` example server that uses the tie mechanism can be found in the VisiBroker for C++ distribution under examples/account in the directory where the VisiBroker distribution was installed. The only difference from the simple `Account` example introduced in Chapter 3, "Quick Start for Development with VisiBroker for C++" is in the account_srvrtie.C file.

LOOKING AT THE _TIE TEMPLATE

The `idl2cpp` compiler will automatically generate a `_tie_Account` template class, as shown in Code Sample 11-1. The `_tie_Account` class is instantiated by the object server and initialized with an instance of `AccountImpl`. The `_tie_Account` class delegates every operation request it receives to `AccountImpl`, the real implementation class. In this example, the class `AccountImpl` does not inherit from the `_sk_Account` class.

CODE
```
...
template <class T>
class _tie_Account : public Account {
    private:
        short _rel_flag;
        T& _ref;
    public:
        _tie_Account(T& t, const char *obj_name=(char*)NULL, short _r_f=0) :
                Account(obj_name), _ref(t) {
            _rel_flag=_r_f;
            _object_name(obj_name);
        }
        _tie_Account(T& t, const char *service_name,
                const CORBA::ReferenceData& id,
                short _r_f=0)
            :_ref(t) {
                _rel_flag=r_f;
                _service(service_name, id);
        }
        ~_tie_Account() { if(_rel_flag) delete &_ref; }
        short rel_flag() { return _rel_flag; }
        void rel_flag(short _r_f) { _rel_flag=_r_f; }

        CORBA::Float balance() {
            return _ref.balance();
        }
};
```

Code Sample 11-1 Looking at the `_tie` template.

CHANGING THE SERVER TO USE THE _TIE_ACCOUNT CLASS

Code Sample 11-2 shows the modifications to the account_srvr.C file required to use the `_tie_account` class. In the code sample, an `AccountImpl` is created and then attached to the `_tie_Account` object which implements the `Account` interface. This tie object is then declared ready to the ORB.

CODE
```
int main(int argc, char* const* argv)
{
     try {
         // Initialize the ORB and BOA
         CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
         CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
```

CONTINUES ON NEXT PAGE

CODE CONTINUED FROM PREVIOUS PAGE

```
            // Create a new account object.
            AccountImpl account;

            // Create the _tie object, initialize with the account object.
            _tie_Account<AccountImpl> tieServer(account, "Jack B. Quick");

            // Export the newly created _tie object.
            boa->obj_is_ready(&tieServer);
            cout << "Account object is ready." << endl;

            // Wait for incoming requests
            boa->impl_is_ready();

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

Code Sample 11-2 Example of a server using the `_tie` class.

Changing the Object Implementation to No Longer Inherit from _sk_account

The only changes that are made to the `AccountImpl` class are that it no longer inherits from `_sk_account`. Instead, it is a free-standing object that implements the `balance()` method.

CODE

```
class AccountImpl
{
    public:
        AccountImpl()
        {
            // Set a random balance between 0 and 10000
            _balance = abs(rand()) % 100000 / 100.0;
        }

        CORBA::Float balance()
        {
            return _balance;
        }

    protected:
        CORBA::Float _balance;
};
```

Code Sample 11-3 Changing the inheritance of the object implementation.

ENABLING AN IMPLEMENTATION TO INHERIT FROM ANOTHER IMPLEMENTATION

The tie mechanism can also be used for *implementation inheritance*, where one implementation inherits from another. Assume that after implementing the `AccountImpl` class, you decide to create a new interface, named `Checking`, that inherits from the `Account` interface. The required changes to the IDL file are shown in IDL Sample 11-1.

IDL
```
interface Account {
      float balance();
};

interface Checking : Account {
      void deposit(in float amount);
};
```

IDL Sample 11-1 Deriving a new interface from `Account`.

When you run the `idl2cpp` compiler, the account_s.hh file will contain a `_sk_checking` class, shown in Code Sample 11-4.

CODE
```
class _sk_Checking : public Checking
{
      public:
          ...

          // The following operations need to be implemented by the server.
          virtual CORBA::Float balance() = 0;
          virtual void deposit(CORBA::Float amount) = 0;
          ...
};
```

Code Sample 11-4 The `_sk_checking` class generated by the `idl2cpp` compiler.

If you implement `CheckingImpl` in the usual way, by deriving from the `_sk_Checking` class, the implementation of the `AccountImpl::balance()` method cannot be used. Using multiple inheritance will not work because both interfaces define a `balance()` method. Therefore, we need to inherit `AccountImpl`'s implementation of the `balance()` method, as well as export the new `Checking` implementation. The `tie` mechanism allows us to do this as shown in Code Sample 11-5 and Code Sample 11-6.

NOTE *Implementation inheritance can also be accomplished by compiling your IDL with the `_virtual_impl_inh` flag of the `idl2cpp` compiler.*

Looking at the CheckingImpl Class

If you derive the `CheckingImpl` class from the `AccountImpl` class, you will be able to use the `AccountImpl::balance()` method that you implemented earlier. You must use the `_tie_checking` template class to allow the `CheckingImpl` object to handle client requests for the `checking::deposit()`.

The `CheckingImpl` class inherits from `AccountImpl` and provides an implementation for the new `deposit()` method.

CODE
```
class CheckingImpl : public AccountImpl
{
      public:

          void deposit(CORBA::Float amount)
          {
              _balance += amount;
          }
};
```

Code Sample 11-5 The `CheckingImpl` class.

Changing the Server to Enable Implementation Inheritance

Code Sample 11-6 shows the modifications to the account_srvr.C file required to use the `_tie_checking` class. Here are the steps the server implements.

1. Initialize the ORB and BOA.
2. Instantiate a `CheckingImpl`.
3. Instantiate a `_tie_checking` class and initialize it with `CheckingImpl`.
4. Export the `_tie_checking` object.
5. Wait for requests.

CODE

```
int main(int argc, char* const* argv)
{
      try {
          // Initialize the ORB and BOA
          CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
          CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

          // Create a new checking object.
          CheckingImpl account;

          // Create the checking _tie object, initialized with the
          // checking object.
          _tie_checking<CheckingImpl> tieCheckingServer( account,
                                                         "Jack B. Quick");

          // Export the newly created _tie object.
          boa->obj_is_ready(&tieCheckingServer);
          cout << "Checking object is ready." << endl;

          // Wait for incoming requests
          boa->impl_is_ready();

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

Code Sample 11-6 The new main routine that enables implementation inheritance.