

Práctica de ejemplo.

Quinto de Ingeniería Informática
Programación Concurrente y Distribuida
Curso 2001/02

Resumen

Esta práctica muestra el proceso de desarrollo de un servidor CORBA empleando Visibroker 3.3 o 3.4 para Java, desde su enunciado hasta la implementación final. Por último, se realiza también una aplicación cliente en modo texto que emplea las operaciones y atributos de los objetos CORBA creados.

Todos los textos fuentes tanto Java como IDL están disponibles en formato electrónico en la página web <http://murillo.fie.us.es/~so/pcd> junto con este enunciado.

Requisitos: Para la comprensión sin problemas de todos los pasos que se van a seguir, se requiere que el alumno haya asimilado mínimamente los conceptos sobre programación en CORBA que se han expuesto en clase. Así mismo, se le supone también una mínima cultura en el lenguaje de programación Java (con los conceptos explicados en clase, debe ser más que suficiente). Por último, y aunque en absoluto de forma imprescindible, contar con un libro de apoyo puede ser de ayuda. Al final de este documento se citan algunas referencias bibliográficas.

Sugerencias: El movimiento se demuestra andando. No sería mala idea que el alumno consiga los fuentes de este ejemplo, los compile y pruebe a ejecutarlos y a realizar modificaciones sobre los mismos. Recuérdese que se tiene una versión de evaluación de Visibroker 3.4 instalado en murillo.

1 Enunciado

Se pretende crear una clase de objetos¹ **Banco** que encapsule un subconjunto mínimo de los servicios que proporciona un banco a sus clientes. Los objetos de dicha clase deben ser objetos CORBA accesibles en un entorno de red por clientes que cumplan también dicha norma. Cada objeto Banco será localizado por los clientes mediante su nombre.

La funcionalidad mínima exigida a los objetos de la clase **Bancoes**:

1. Posibilidad de mantener información sobre sus clientes, estando cada cliente identificado por su NIF. Se debe permitir dar clientes de alta y de baja, así como consultar y modificar sus datos. En cualquier caso, se deberá tener en cuenta las siguientes normas de coherencia:
 - No pueden existir dos clientes con el mismo NIF
 - No se puede dar de baja a un cliente mientras que tenga abierta alguna cuenta.
2. Se debe permitir la apertura de cuentas a nombre de un cliente del banco. Se recomienda que las cuentas se implementen también como objetos CORBA para que puedan ser accedidas

¹Conforme a estándar CORBA 2.0 de OMA

por clientes² CORBA remotos. La funcionalidad exigida para las cuentas se expondrá mas adelante. En cualquier caso, todo objeto **Banco** deberá mantener las siguientes reglas de integridad con respecto a las cuentas:

- Una cuenta debe estar siempre a nombre de un cliente del banco.
- Una cuenta debe estar a nombre de un sólo cliente (¡toma simplificación!)
- Cada cuenta estará identificada por un número de cuenta que le asignará el banco en el momento de ser creada, y que permanece invariable durante su vida.

Por supuesto, la única forma de abrir una cuenta debe ser a través de un objeto **Banco**.

3. Un objeto **Banco** debe ser capaz de proporcionar todas las cuentas abiertas a nombre de uno de sus clientes.
4. Se debe permitir la cancelación de una cuenta. En cualquier caso, se debe tener en cuenta la siguiente restricción de integridad:
 - No se debe permitir la cancelación de una cuenta que tenga fondos (cuyo saldo sea mayor que cero).
5. Se debe poder consultar fácilmente el nombre de un objeto **Banco**.

Como se ha indicado anteriormente, se recomienda que los objetos **Cuenta** se implementen también como objetos CORBA para que puedan ser accedidos de forma remota por cualquier cliente CORBA a través de la red. La funcionalidad que se le exige a los objetos de dicha clase es:

1. Posibilidad de anotar movimientos sobre la cuenta. Los movimientos a realizar podrán ser cargos o abonos, por una cantidad determinada. Se deberá tener en cuenta la siguiente restricción:
 - No se debe permitir efectuar un cargo por un importe superior al saldo actual de la cuenta.
2. Se debe poder consultar fácilmente la siguiente información sobre una cuenta:
 - Número de cuenta
 - NIF del cliente al que pertenece
 - Saldo actual

2 Trabajo a realizar

Dado el anterior enunciado, se pretende obtener:

1. Sendas interfaces IDL **Banco** y **Cuenta** que permitan encapsular los comportamientos descritos en el apartado anterior.
2. Sendas implementaciones en Java para ambas interfaces.
3. Un programa Java que funcione como servidor de un objeto **Banco**.
4. Un programa cliente de ejemplo, que demuestre el uso de todas las operaciones implementadas.

²Cada vez que lea la palabra "cliente", cuide no confundir "cliente CORBA" con "cliente del banco". Aunque la palabra sea la misma, su significado es totalmente distinto. Tenga en cuenta el contexto en que la lee.

3 Solución

Antes de comenzar, conviene aclarar que este ejercicio, junto con su solución, no pretende ser ni mucho menos un ejemplo de una aplicación "real", ni siquiera mostrar la forma en que determinadas cosas deben hacerse. Por ejemplo, en una aplicación real, la información sobre cuentas, clientes, etc., debería ser almacenada en bases de datos. Ahora bien, exponer aquí el mecanismo de acceso a bases de datos desde Java (JDBC) hubiese sido inviable.

Este es simplemente un ejemplo cuyo enunciado y solución han sido "forzados" para mostrar el mayor número posible de aspectos de la programación CORBA.

Para la implementación del sistema propuesto, seguiremos los pasos habituales que ya conocemos:

1. Escribir la(s) especificación(es) IDL
2. Compilar dicha(s) especificación(es) con un compilador de IDL que genere código en nuestro lenguaje de programación favorito.
3. Escribir la(s) implementación(es) para las interfaces, por el método que más nos convenga (herencia o delegación).
4. Escribir un proceso servidor, o insertar nuestra(s) implementación(es) en un servidor que ya tengamos creado.
5. Escribir el(los) cliente(s).

Recuérdese que dado que el cliente está totalmente aislado de la implementación del servidor, en la vida real el punto 5 es independiente de los puntos 3 y 4. Si nosotros lo ponemos en último lugar, es sólo por conveniencia.

Este ejercicio lo implementaremos enteramente sobre Visibroker 3.3 para Java. No obstante, no es complicado de portar a otro entorno, como pudiera ser ORBAcus 3.0.

3.1 Especificaciones IDL

Primero tenemos que elegir tipos IDL para representar los distintos tipos de datos que se deducen del enunciado, o bien construir nuestros propios tipos IDL.

Del enunciado, parece deducirse claramente los siguientes tipos de datos:

- NIF
- Información sobre el cliente
- Número de cuenta

Otro posible candidato podría ser el saldo. En cualquier caso, dado que el saldo no es más que una cantidad, lo representaremos como un entero, sin necesidad de construir un tipo para él. El número de cuenta, aunque lo representemos como un número entero, sí es buena idea que cuente con su propio tipo, para dar libertad a posibles cambios de implementación.

Respecto a la información sobre el cliente, dado que el enunciado no concreta nada al respecto, supondremos que basta con el nombre completo y la dirección.

Escribiremos dos archivos `.idl`, uno para la interfaz `Banco` y otro para la interfaz `Cuenta`. Aunque nada impide escribir ambas en una misma interfaz, ganamos facilidad de reutilización escribiendo dos archivos separados.

Dado que los tipos NIF y datos del cliente se necesitarán en ambos ficheros, escribiremos un tercer fichero `.idl` con dichos tipos.

El NIF lo representaremos mediante un tipo `TNIF` que será simplemente una cadena de texto. Los datos del cliente los representaremos mediante un `struct TInfoCliente` que contendrá dos cadenas, una para el nombre y otra para la dirección. Por tanto, escribiremos un fichero `tipos.idl` que contendrá:

```
#ifndef __tipos_idl__
#define __tipos_idl__

typedef string TNIF;
struct TInfoCliente
{
    string nombre;
    string direccion;
};
#endif
```

Las directivas `#ifndef ... #endif` se utilizan para prevenir la recompilación accidental, de igual forma que se usan en C. Si en cualquier caso no está acostumbrado a su uso, ignórelas.

A continuación, escribamos el fichero `cuenta.idl` con la interfaz `Cuenta`. Por los requisitos del enunciado, dicha interfaz contendrá:

- Atributos `NumCuenta`, `NIF` y `Saldo`. Dado que no tiene sentido modificar estos atributos directamente, los declararemos como de sólo lectura. El atributo `NumCuenta` será de un tipo `TNumCuenta` que definiremos previamente (como entero).
- Operación `RealizarMovimiento`. Por conveniencia, esta operación permitirá realizar tanto un abono como un cargo. Para ello, recibirá un parámetro de tipo `TTipoMovimiento` que definiremos también previamente.
- Dado que la operación `RealizarMovimiento` debe detectar un caso de error (sacar mas dinero del disponible ;-), parece ser buena idea declarar que dicha operación pueda elevar una excepción. Para ello, declararemos una excepción `FondosInsuficientesError`, e indicaremos que la operación `RealizarMovimiento` puede elevarla.

Además, el fichero `cuenta.idl` debe incluir el anterior fichero `tipos.idl`, ya que emplea tipos en él declarados.

Con esto, el fichero `cuenta.idl` queda:

```
#ifndef __cuenta_idl__
#define __cuenta_idl__
#include "tipos.idl"

enum TTipoMovimiento {abono, cargo};
typedef long TNumCuenta;

exception FondosInsuficientesError {};

interface Cuenta
{
    readonly attribute TNIF NIFCliente;
```

```

readonly attribute TNumCuenta NumCuenta;
readonly attribute long Saldo;

void RealizarMovimiento (in TTipoMovimiento tipo, in long cantidad)
    raises (FondosInsuficientesError);
};
#endif

```

Por último, queda escribir la interfaz **Banco**. Ésta va a contener:

- Un atributo de sólo lectura **Nombre**, que permita consultar desde un cliente CORBA el nombre del banco³.
- Operación **AltaCliente**, que recibe un NIF e información sobre un cliente, y lo da de alta en el banco. Dado que esta operación debe informar del error que se produce al intentar dar de alta a un cliente con un NIF que ya está asociado a otro cliente (posiblemente, un alta duplicada), esta operación declara que puede elevar una excepción **NIFDuplicadoError** que previamente hay que declarar.
- Operación **BajaCliente**, que recibe un NIF y da de baja al cliente con dicho NIF. Dado que esta operación debe informar del error que se produce al intentar dar de baja a un cliente que tenga abierta una cuenta, esta operación declara que puede elevar una excepción **BajaNoValidaError**, que previamente hay que declarar. Y dado que también ha de informar del error producido al intentar dar de baja a un cliente que no ha sido dado de alta, también declara poder elevar una excepción **NIFNoEncontradoError**, que a tales efectos hay que declarar.
- Operación **ModificaCliente**, que recibe un NIF y los datos de un cliente, y cambia los datos asociados al mismo por los nuevos datos recibidos. Dado que el cliente cuyo NIF se pasa debe existir, esta operación también declara que puede elevar la excepción **NIFNoEncontradoError**.
- Operación **ConsultaCliente**, que recibe un NIF y devuelve los datos asociados a dicho cliente. Dado que el cliente cuyo NIF se pasa debe existir, esta operación también declara que puede elevar la excepción **NIFNoEncontradoError**.
- Operación **NuevaCuenta**, que recibe un NIF, y crea una nueva cuenta a nombre del cliente identificado por el NIF. Esta operación devuelve una referencia al objeto **Cuenta** creado. Esto es lo que en terminología CORBA se denomina un "*Factory Object*", es decir, un objeto que "fabrica" otros objetos. El objeto Cuenta creado será otro objeto CORBA que se puede emplear como tal. Nótese que de esta forma, obtenemos una referencia a un objeto CORBA, sin tener que usar ni el servicio de nombres, ni el servicio de trading, ni el método **bind**. Dado que el cliente cuyo NIF se pasa debe existir, esta operación también declara que puede elevar la excepción **NIFNoEncontradoError**.
- Operación **CancelarCuenta**, que recibe una referencia a un objeto CORBA **Cuenta**, y borra dicha cuenta del banco. Así como la operación anterior era un ejemplo de cómo una operación puede devolver una interfaz (lo que en la implementación se traducirá a una referencia a un objeto CORBA) esta operación es un ejemplo de cómo una operación puede también recibir como parámetro una interfaz. Dado que esta operación debe informar del error que se produce al intentar cancelar una cuenta cuyo saldo no sea cero, esta operación declara que puede elevar una excepción **CancelaciónNoValidaError**, que previamente hay que declarar.
- Operación **ObtenerCuentas**, que recibe un NIF y devuelve una secuencia (lista) de todas las cuentas abiertas a nombre del cliente identificado por dicho NIF. Dado que una operación no

³Este nombre será establecido por el servidor a la hora de crear el objeto

se puede declarar directamente que retorne un tipo `sequence<>`, se define primero mediante `typedef` un tipo `ListaCuentas` como una secuencia de `Cuenta`, y se declara la operación de este tipo. Nótese que esta operación devuelve una lista de interfaces. Dado que el cliente cuyo NIF se pasa debe existir, esta operación también declara que puede elevar la excepción `NIFNoEncontradoError`.

Nótese que dado que vamos a utilizar tipos declarados dentro de `cuenta.idl` y de `tipos.idl`, debemos incluir estos ficheros.

Con esto, el fichero `banco.idl` queda:

```
#ifndef __banco_idl__
#define __banco_idl__

#include "tipos.idl "
#include "cuenta.idl"

exception NIFDuplicadoError {};
exception NIFNoEncontradoError {};
exception BajaNoValidaError {};
exception CancelacionNoValidaError {};

interface Banco
{
    readonly attribute string Nombre;

    void AltaCliente (in TNIF NIF, in TInfoCliente cliente)
        raises(NIFDuplicadoError);
    void BajaCliente (in TNIF NIF)
        raises (NIFNoEncontradoError, BajaNoValidaError);
    void ModificaCliente (in TNIF NIF, in TInfoCliente cliente)
        raises (NIFNoEncontradoError);
    TInfoCliente ConsultaCliente (in TNIF NIF)
        raises (NIFNoEncontradoError);
    Cuenta NuevaCuenta (in TNIF NIF)
        raises (NIFNoEncontradoError);
    void CancelarCuenta (in Cuenta cuenta)
        raises (CancelacionNoValidaError);

    typedef sequence<Cuenta> ListaCuentas;
    ListaCuentas ObtenerCuentas (in TNIF NIF) raises (NIFNoEncontradoError);
};
#endif
```

3.2 Compilación de especificaciones IDL

Bueno, esto es lo más difícil :):

```
$ vbjc -no_comments banco.idl
```

Dado que `banco.idl` incluye a `cuenta.idl` y `tipos.idl`, no es necesario compilar por separado estos ficheros.

En cualquier caso, no es mala idea curiosear un poco el código generado. Analícese por ejemplo la clase `TTipoMovimiento` generada para el enumerado definido con el mismo nombre en la especificación IDL. También es buena idea darle un vistazo a los ficheros `Cuenta.java` y `Banco.java` (interfaz Java resultante de la traducción directa de las interfaces IDL de mismo nombre) y a los *Skeletons* `_CuentaImplBase.java` y `_BancoImplBase.java`. Nótese que ambos implementan respectivamente las interfaces `Cuenta` y `Banco`. Por último, démosle un vistazo a los *Stubs* `_st_Cuenta` y `_st_Banco`, y compruébese como ambos también implementan su respectiva interfaz.

Aunque el número de ficheros generados es considerable, no debe cundir el pánico. Recuérdese que no es necesario editar ninguno de estos ficheros. Es más, al nivel que nos vamos a mover, incluso podemos ignorar el contenido de la mayoría de ellos.

3.3 Implementación de las Interfaces

Ahora ha llegado el momento de escribir una clase Java para implementar el comportamiento de las interfaces IDL que hemos escrito. Esto lo podemos hacer tanto por el método de herencia como por el método de delegación. Por escoger un método, emplearemos el de herencia.

Implementación de la interfaz `Cuenta` Para implementar la interfaz `Cuenta`, escribiremos una clase que (por convenio) se llamará `CuentaImpl` y que debe heredar de su *Skeleton*, `_CuentaImplBase`:

```
public class CuentaImpl extends _CuentaImplBase
{
```

Declaramos los siguientes atributos privados para guardar respectivamente el saldo, el NIF del cliente al que pertenece la cuenta y el número asignado a la cuenta:

```
private int Saldo;
private String NIFCliente;
private int NumCuenta;
```

Y escribimos al menos un constructor de clase. Esta clase `CuentaImpl` va a ser una excepción a la regla general de "suministrar al menos dos constructores, uno no-arg y otro que admita como parámetro el nombre del objeto", ya que los objetos de esta clase no van a ser localizados nunca a través del servicio de nombres, ni por el método `bind`, sino que será un objeto de la clase `Banco` a quien le vamos a pedir que nos localice (o produzca) los objetos `Cuenta`. Por conveniencia, hacemos que nuestro constructor admita como parámetro el número asignado a la cuenta y el NIF del cliente a que pertenece (ambos permanecen invariables durante toda la vida del objeto):

```
CuentaImpl (int NumCuenta, String NIFCliente)
{
    super ();
    Saldo= 0;
    this.NIFCliente= NIFCliente;
    this.NumCuenta= NumCuenta;
}
```

Acto seguido, escribiremos los métodos de lectura de los atributos de la interfaz (recuérdese que los atributos IDL se mapean a dos métodos Java, uno de lectura, y otro de escritura). Casualmente, al ser todos los atributos de sólo lectura, no existe ningún método de escritura:

```

public String NIFCliente()
{
    return NIFCliente;
}

public int NumCuenta()
{
    return NumCuenta;
}

public int Saldo()
{
    return Saldo;
}

```

Por último, vamos a implementar la única operación que contiene el interfaz `Cuenta`: la operación `RealizarMovimiento`. Aquí es interesante destacar el tratamiento que se ha hecho del tipo enumerado `TTipoMovimiento`. Para ello, vamos a dar un vistazo al contenido de `TTipoMovimiento.java`, que contiene la clase del mismo nombre:

```

final public class TTipoMovimiento
{
    final public static int _abono = 0;
    final public static int _cargo = 1;
    final public static TTipoMovimiento abono = new TTipoMovimiento(_abono);
    final public static TTipoMovimiento cargo = new TTipoMovimiento(_cargo);
    private int __value;
    private TTipoMovimiento(int value)
    {
        this.__value = value;
    }
    public int value()
    {
        return __value;
    }

    [...más cosas que no nos importan ahora...]
}

```

Nótese que el tipo enumerado se traduce en una clase con el mismo nombre, la cual tiene como constantes de clase (atributos `final static`) de tipo entero a los posibles valores del enumerado (atributos `_abono` y `_cargo`). Démonos cuenta también que tiene un atributo `_value` que contiene el valor real del objeto (es decir, el valor que toma del enumerado) y un constructor, que toma un valor entero como argumento, y almacena este valor en el atributo `_value`. Por último, nótese que esta clase tiene también como constantes de clase a dos referencias a objetos de sí misma, cada uno de ellos inicializado con su correspondiente valor en el constructor. ¡Qué casualidad, que estos dos atributos tienen los mismos nombres que los literales del enumerado!

Estos dos atributos serán los que habitualmente empleemos para manejar la clase con la que se modela el enumerado. Es decir, para declarar una variable del tipo "enumerado" `TTipoMovimiento`, haremos:

```

TTipoMovimiento t;

```

```

t= TTipoMovimiento.abono;
[...]
if (t == TTipoMovimiento.cargo)
[...]

```

Esta clase `TTipoMovimiento` nos proporciona dos objetos que representan los dos posibles valores que puede tomar el enumerado y que son apuntados por las referencias `TTipoMovimiento.abono` y `TTipoMovimiento.cargo`. Lo que nosotros haremos será poner cualquier referencia a este "enumerado" apuntando a un objeto o a otro, y después, comprobar a cual de los dos objetos apunta.

La operación `RealizarMovimiento`, sólo queda tener en cuenta que esta operación puede elevar una excepción (`FondosInsuficientesError`) en el caso de que la operación sea un cargo, y no haya saldo suficiente para atenderlo. Esto es muy simple, ya que el compilador de IDL se ha encargado de generar una excepción Java con el mismo nombre (véase fichero `FondosInsuficientesError.java`) y se ha encargado también de declarar en la cabecera del método `RealizarMovimiento` que éste puede "lanzar" dicha excepción. Por tanto, nosotros sólo hemos de preocuparnos de lanzar la excepción cuando sea necesario.

Ya tenemos todo lo necesario para escribir la implementación del método `RealizarMovimiento`:

```

public void RealizarMovimiento (TTipoMovimiento tipo, int cantidad)
    throws FondosInsuficientesError
{
if (tipo.equals (TTipoMovimiento.abono))
    Saldo+= cantidad;
else if (tipo.equals (TTipoMovimiento.cargo))
    if (Saldo < cantidad)
        throw new FondosInsuficientesError ();
    else
        Saldo-= cantidad;
}

```

Con esto ya tenemos implementación para el interfaz `Cuenta`. Si lo deseamos, podemos probar a compilarla con el comando:

```
$ vbjc CuentaImpl.java
```

Obteniendo el fichero `CuentaImpl.class`, si todo ha funcionado correctamente.

Implementación de la interfaz Banco Vamos a aplicar ahora la misma metodología para implementar la interfaz `Banco`. Emplearemos también el método de herencia.

Para implementar la interfaz `Banco`, escribiremos una clase que (por convenio) se llamará `BancoImpl` y que debe heredar de su *Skeleton*, `_BancoImplBase`. Previamente, importaremos el paquete `java.util` completo para poder emplear las clases predefinidas `Vector` y `Hashtable`:

```

import java.util.*;
public class BancoImpl extends _BancoImplBase
{

```

A continuación, declaramos algunos atributos privados que nos servirán para almacenar información de utilidad:

- Nombre del objeto `Banco`, suministrando además un nombre por defecto:

```
private String Nombre= "BSN (Banco Sin Nombre)";
```

- Número de cuenta que se le asignó a la última cuenta que se abrió:

```
private int numero_ultima_cuenta= 0;
```

- Tabla Hash de clientes. Los clientes que se den de alta en el banco, se almacenarán (a modo de ejemplo) en una tabla Hash(tipo `Hashtable` de `java.util`). En esta tabla se almacenarán objetos de la clase `TInfoCliente`, utilizando el NIF como clave:

```
private Hashtable clientes;
```

- Vector de cuentas. Todos los objetos `Cuenta` creados por el banco, se almacenarán en un vector (tipo `Vector`, de `java.util` es básicamente un `array` de tamaño variable). No hay ninguna forma de indexación, por lo que la búsqueda es siempre secuencial:

```
private Vector cuentas;
```

A continuación, escribimos como suele ser buena costumbre dos constructores de clase, uno no-arg, y otro que toma como parámetro el nombre del objeto. Ambos constructores crean la tabla Hash de clientes y el vector de cuentas:

```
BancoImpl ()
{
    super ();
    clientes= new Hashtable ();
    cuentas= new Vector ();
}

BancoImpl (String Nombre)
{
    super (Nombre);
    this.Nombre= Nombre;
    clientes= new Hashtable ();
    cuentas= new Vector ();
}
```

Acto seguido, implementamos el método de lectura del único atributo que tiene la interfaz Banco. Por ser este atributo de sólo lectura, no hay que escribir el método de escritura:

```
public String Nombre ()
{
    return Nombre;
}
```

Y ahora, comenzamos a escribir los métodos Java que implementan las operaciones de la interfaz.

Los métodos `AltaCliente`, `BajaCliente`, `ModificaCliente`, `ConsultaCliente` no aportan nada nuevo, excepto tal vez el uso de la clase `Vector` y `Hashtable`. El uso de ambas clases es simple, y el ejemplo es fácil de entender. Si se desea mayor información sobre cualquiera de ellos, se puede consultar cualquier libro o tutorial sobre Java, incluso la propia documentación del JDK:

```

public void AltaCliente (String NIF, TInfoCliente cliente)
    throws NIFDuplicadoError
{
    // La información relativa al cliente se inserta en la tabla hash de
    // clientes, identificada por su NIF. Previamente, se comprueba que dicho
    // NIF no exista ya como clave.

    if (clientes.get (NIF) != null)
        throw new NIFDuplicadoError ();
    else
        clientes.put (NIF, cliente);
}

public void BajaCliente (String NIF)
    throws NIFNoEncontradoError, BajaNoValidaError
{
    // Se comprueba que exista un cliente en tabla hash con dicho NIF:
    if (clientes.get (NIF) == null)
        throw new NIFNoEncontradoError ();
    else
    {
        boolean bajalegal= true;
        CuentaImpl cuenta;
        // y nos aseguramos que no haya ninguna cuenta de dicho cliente en el
        // vector de cuentas:
        for (int i= 0; i < cuentas.size () && bajalegal; i++)
        {
            cuenta= (CuentaImpl) cuentas.elementAt (i);
            if (NIF.equals (cuenta.NIFCliente ()))
                bajalegal= false;
        }
        // con lo que actuamos en consecuencia:
        if (bajalegal)
            clientes.remove (NIF);
        else
            throw new BajaNoValidaError ();
    }
}

public void ModificaCliente (String NIF, TInfoCliente cliente)
    throws NIFNoEncontradoError
{
    // Se comprueba que exista un cliente en tabla hash con dicho NIF:
    if (clientes.get (NIF) == null)
        throw new NIFNoEncontradoError ();
    else
    {
        clientes.remove (NIF);          // Se elimina anterior información
        clientes.put (NIF, cliente);    // Y se coloca nueva información
    }
}

public TInfoCliente ConsultaCliente (String NIF)

```

```

    throws NIFNoEncontradoError
    {
    // Se obtiene la informacion asociada al NIF en la tabla hash de clientes,
    // o null en caso contrario.
    TInfoCliente resultado= (TInfoCliente) clientes.get (NIF);
    if (resultado==null)
        throw new NIFNoEncontradoError ();

    return resultado;
    }

```

El método `NuevaCuenta` es un método especial, ya que es un objeto que crea y retorna un nuevo objeto CORBA (nótese que su tipo de retorno en la especificación IDL es una interfaz). En los objetos vistos en clase, se ha usado un proceso servidor escrito *ad hoc* para crear objetos CORBA. Ahora bien, un objeto CORBA se puede crear y poner a disposición de los clientes en cualquier parte de la aplicación. Recordemos que para crear un objeto CORBA y hacer éste accesible hay que:

1. Declarar una referencia a la clase implementación (en este caso `CuentaImpl`), y crear un objeto de dicha clase, empleando para ello cualquiera de sus constructores. Si se hubiese empleado el mecanismo de delegación para implementar los objetos `Cuenta`, el objeto a registrar habría sido de la clase delegadora (`_tie.Cuenta`).
2. Registrar dicho objeto en el BOA.

Lo primero lo podemos hacer sin problema en cualquier parte de la aplicación, pero, ¿y lo segundo? Recordemos que para registrar un objeto en el BOA, teníamos que obtener una referencia al BOA (lo hacíamos al principio de la aplicación, cuando se inicializaba el BOA) e invocar al método `obj_is_ready`:

```

BOA boa= orb.BOA _init();
boa.impl_is_ready (mi_objeto);

```

Ahora bien, ¿dónde está esa referencia al BOA? Aunque también existen otras soluciones, lo más cómodo es usar el método `_boa()` (que toda implementación de un objeto CORBA realizada por herencia hereda de su `Skeleton`), el cual devuelve una referencia al BOA, siempre y cuando este haya sido inicializado. Aunque no nos haga falta en este caso, bueno es saber que también se hereda un método `_orb()` que devuelve una referencia al ORB.

Por el hecho de que los objetos `Banco` sean capaces de crear y proporcionar referencias a otros objetos CORBA, se dice que son "*Factory Objects*".

Con esto, ya sabemos todo lo necesario para escribir el método `NuevaCuenta`:

```

public Cuenta NuevaCuenta(String NIF) throws NIFNoEncontradoError
{
// Se comprueba que exista un cliente en tabla hash con dicho NIF:
if (clientes.get (NIF) == null)
    throw new NIFNoEncontradoError ();

// Se calcula el numero que se le va a dar a la cuenta, y se crea un
// nuevo objeto CuentaImpl (implementacion de la interfaz cuenta), inser-
// tandolo en el vector de cuentas.
numero_ultima_cuenta++;

```

```

CuentaImpl nueva_cuenta= new CuentaImpl (numero_ultima_cuenta, NIF);
cuentas.addElement (nueva_cuenta);

// Dado que este objeto va a ser accedido por clientes CORBA, lo registra-
// mos en el BOA, para que lo puedan ver los clientes. Empleamos para ello
// el método boa () heredado de CORBA.Object que nos permite acceder al BOA
_boa().obj_is_ready (nueva_cuenta);

// Retornamos cuenta. Nótese que esto no plantea conflicto entre el tipo
// Cuenta (devuelto por este metodo) y el tipo CuentaImpl (al que perte-
// nece nueva_cuenta) ya que la clase CuentaImpl implementa Cuenta
return nueva_cuenta;
}

```

En el método `CancelarCuenta`, nos encontraremos justo con el problema contrario. Es decir, ahora tenemos que hacer que un objeto que el BOA tiene registrado y accesible a (en principio) cualquier cliente CORBA, deje de estarlo. Esto se realiza mediante el método del BOA `deactivate_obj`, método que podemos considerar simétrico a `obj_is_ready`. Nótese que esto además tiene como consecuencia que el BOA deja de apuntar mediante cualquier referencia al objeto. Esto implica que si nuestro programa también deja de apuntar al objeto en cuestión, dicho objeto podrá ser eliminado por el recolector de basura.

Con esto, podemos ya escribir el método `CancelarCuenta`:

```

public void CancelarCuenta (Cuenta cuenta) throws CancelacionNoValidaError
{
// Se comprueba saldo de la cuenta
if (cuenta.Saldo () != 0)
    throw new CancelacionNoValidaError ();

// Se localiza la cuenta en el vector de cuentas:
boolean encontrado= false;
for (int i=0; i<cuentas.size () && !encontrado; i++)
    if (cuenta == cuentas.elementAt (i)) // Esto no es un error!
    {
        encontrado= true;
        cuentas.removeElementAt (i);        // Se borra cuenta del vector
        _boa ().deactivate_obj (cuenta);    // Se da de baja en el BOA
    }

// Esto nunca debería ocurrir, luego si ocurre, reconozcamos nuestro error
if (!encontrado)
    System.err.println ("Error interno!");
}

```

Por último, sólo queda implementar el método `ObtenerCuentas`. Este método sólo tiene de particular el hecho de que su especificación IDL indica que retorna una secuencia de `Cuenta`. El *mapping* de IDL a Java establece (curiosamente) que las secuencias IDL se traducen a *arrays* Java. Por tanto, nuestra aplicación debe crear un *array* con el resultado y retornar el mismo de la siguiente forma:

```

public Cuenta[] ObtenerCuentas (String NIF) throws NIFNoEncontradoError
{

```

```

// Se comprueba que exista un cliente en tabla hash con dicho NIF:
if (clientes.get (NIF) == null)
    throw new NIFNoEncontradoError ();

// Se crea un vector temporal en el que se introducen las cuentas del
// cliente (de momento, no sabemos cuantas tiene)
Vector cuentas_cliente= new Vector ();
for (int i= 0; i<cuentas.size (); i++)
{
    CuentaImpl cuenta= (CuentaImpl)cuentas.elementAt (i);
    if (NIF.equals (cuenta.NIFCliente ()))
        cuentas_cliente.addElement (cuenta);
}

// Ahora que ya se tiene en el vector las cuentas del cliente, y ya se
// sabe cuantas son, se pasan al array que se devuelve como resultado.
Cuenta resultado []= new Cuenta [cuentas_cliente.size ()];
for (int i=0; i<cuentas_cliente.size (); i++)
    resultado [i]= (Cuenta) cuentas_cliente.elementAt (i);

return resultado;
}

```

Con esto ya tenemos implementación para la interfaz **Banco**. Si lo deseamos, podemos probar a compilarla con el comando:

```
$ vbjc BancoImpl.java
```

Obteniendo el fichero `BancoImpl.class`, si todo ha funcionado correctamente.

3.4 Creación del servidor

Una vez que ya tenemos escritas las implementaciones de todas las interfaces de nuestra aplicación, sólo queda que algún programa cree objetos de dichas implementaciones, los registre en el BOA, y a falta de otra cosa que hacer, quede en espera de que lleguen peticiones de los clientes sobre estos objetos (lo que se llama "bucle de servicio", o "dispatch loop"). Recordemos que los pasos a seguir por dicho programa son:

1. Inicializar ORB.
2. Inicializar BOA.
3. Crear los objetos que se van a poner a disposición de los clientes.
4. Registrar dichos objetos en el BOA.

Escribamos pues una pequeña aplicación Java que siga estos pasos. Dado que el objeto **Banco** se localiza por su nombre, nuestra aplicación recibirá en la línea de comandos el nombre con el que se registra dicho objeto:

```
import org.omg.CORBA.*;
class servidor
```

```

{
public static void main (String args [])
{
    // Se comprueba que se ha pasado el nombre del objeto que se registra
    if (args.length != 1)
    {
        System.err.println ("Uso: servidor \"Nombre del banco\");
        System.exit (1);
    }

    ORB orb= ORB.init (args, null);    // Inicialización ORB
    BOA boa= orb.BOA_init ();        // Inicialización BOA

    // Creación y registro del objeto Banco
    System.out.println ("Creando objeto Banco:");
    BancoImpl banco= new BancoImpl (args [0]);

    System.out.println ("Activando banco \"\" + args [0] + \"\");
    boa.obj_is_ready (banco);

    // Bucle de servicio
    System.out.println ("Entrada en bucle de servicio.");
    boa.impl_is_ready ();
}
}

```

Ya sólo queda compilar éste programa:

```
$ vbjc servidor.java
```

Obteniendo el fichero `servidor.class` si todo ha funcionado correctamente. En dicho caso, ya podemos incluso ejecutar el servidor, mediante el comando:

```
$ vbj servidor "Banco Pitufo"
```

Lo cual producirá la siguiente salida:

```

Creando objeto Banco:
Activando banco "Banco Pitufo"
Entrada en bucle de servicio.

```

3.5 Creación del cliente

Creernos que nuestro servidor CORBA funcione correctamente es hasta ahora un asunto de fe. Para probarlo, necesitamos al menos un cliente CORBA que obtenga una referencia al objeto `Banco`, y que utilice sus servicios.

La implementación de un cliente es mucho mas simple que la del servidor, ya que nos basta con obtener una referencia al objeto CORBA que deseamos emplear, y una vez obtenida dicha referencia, manejamos dicho objeto como cualquier otro objeto Java, siendo totalmente transparente a nuestra aplicación el hecho de que el objeto sea remoto.

Para ello, recordemos que los pasos a seguir en la construcción de un cliente son:

1. Inicializar el ORB.
2. Declarar una referencia a la clase del objeto que queremos manejar. ¡Ojo! No de la clase implementación, sino de la aquella interfaz Java a la que se traducía la interfaz IDL (**Banco**, en este caso)
3. Obtener una referencia al objeto mediante cualquiera de estos métodos:
 - Servicio de nombres o servicio de trading
 - Mediante un "Factory Object". De esta forma, por ejemplo, se obtendrán los objetos **Cuenta**, a través del objeto **Banco**.
 - Mediante el método `bind`⁴ declarado en la clase **Helper** del objeto que queremos localizar.

Para probar nuestro servidor CORBA, hemos implementado también un cliente que utiliza todos y cada unos de los atributos y métodos que hemos declarado en la interfaz IDL. Dicho cliente está implementado también en Java, y se ejecuta exclusivamente en modo texto, para que pueda ser utilizado desde cualquier terminal.

Incluir aquí el código completo de dicho ejemplo sería un desgaste innecesario de papel, ya que además de que dicho código está disponible en su totalidad en la dirección web que aparece en la introducción, el hecho de que los objetos que manipule sean objetos CORBA es totalmente transparente. Es decir, sólo "se nota" que emplea CORBA en las primeras líneas, que son las que vamos a comentar a continuación.

Nuestro cliente intentará obtener una referencia a un objeto **Banco**, el cual localizará a través de su nombre mediante el método `bind()`. El nombre del objeto que intenta localizar lo recibe en la línea de comandos. Por tanto, el código que inicializa el ORB y obtiene la referencia al **Banco** es:

```
import org.omg.CORBA.*;
import java.io.*;
class cliente
{
    [...]

    public static void main (String args [])
    {
        [...]
        // Se comprueba linea de comandos:
        if (args.length != 1)
        {
            System.err.println ("Uso: cliente \"Nombre del banco\");
            System.exit (1);
        }

        // Inicialización ORB
        ORB orb= ORB.init (args, null);

        // Se obtiene referencia a banco
        Banco banco= BancoHelper.bind (orb, args [0]);
        [...]
```

⁴Se recuerda que dado que este método no está estandarizado, no está implementado en todos los entornos CORBA.

De igual modo que el servidor, compilamos el cliente haciendo:

```
$ vbjc cliente.java
```

Según la versión de JDK con que se compile, puede dar un aviso sobre el uso de un API obsoleto. Podemos ignorar tranquilamente este aviso. Finalmente, podemos ejecutar el servidor haciendo:

```
$ vbj cliente "Banco Pitufo"
```

Si todo funciona correctamente, debemos obtener una salida textual tal como:

```
MENU PRINCIPAL
-----
1- Alta cliente
2- Baja cliente
3- Modificacion cliente
4- Consulta de un cliente
5- Apertura de cuenta
6- Realizar abono en cuenta
7- Realizar cargo a cuenta
8- Cancelar cuenta
9- Salir

Seleccione opcion:_
```

A partir de aquí, sólo hay que experimentar.

4 Referencias Bibliográficas

1. **Programming with Visibroker.** *Doug Pedrik, Jonathan Weedon, Jon Goldberg y Erik Bleifield.* Wiley. ISBN: 0-471-23901-1.
2. **Java programming with CORBA.** *Andreas Vogel y Keith Duddy.* Wiley. ISBN:0-471-24765-0.
3. **Teach Your Self Corba in 14 days.** *Jeremy L. Rosemberg.* Sams Publishing. ISBN: 0-672-31208-5.