The Bank example using David

April 12, 2002

This document describes step by step how the Bank application has been developped with Jonathan.

1 Step 1: write an IDL specification

Like with the Hello World example, the first step is to describe the interfaces to be accessed remotely using the CORBA *Interface Definition Language* (IDL). In the following, we'll use this IDL specification[1] as an example:

```
 1 exception NoCashException {
 2    string reason;
 3 };
 4
 5 interface Account {
 6    readonly attribute double balance;
 7    readonly attribute string owner;
 8    void add(in double amount);
 9    void remove(in double amount) raises (NoCashException);
10 };
13 module bank {
14    exception OpenException {
15       string reason;
16    };
17
18    interface Position {
19       void open(in string name,in double init) raises (OpenException);
20       double close(in ::Account account);
21       ::Account get(in string name);
22    };
23 };
```

2 Step 2: compile your IDL specification

The following command starts the compilation of `bank.idl`:

```
java org.objectweb.david.tools.idlcompiler.Idl2Java -p idl test.idl
```

[1] contained in `examples/david/bank/bank.idl`

Calling the compiler this way assumes that the Jonathan classes are in your classpath. This operation is performed automatically if you use the provided `Makefile`[2].

The compilation results in Java source files. In our example, these files are generated in two directories:

- `idl`
- `idl/bank`[3]

The `idl` directory contains the files corresponding to the unscoped IDL types; `idl/bank` contains the files corresponding to the IDL types specified within the `bank` scope (`module bank {...};`). These files have been generated below the `ti` package because the `-p` option of the IDL compiler has been used. For more information about the IDL compiler, have a look at its HTML documentation.

3 Step 3: write servers

The file Server.java[4] contains implementations for the two interfaces `Account` and `Position` specified in `bank.idl`, and a `main` method to run the server. Let's go through the code:

```
25
26 import java.util.Hashtable;
27 import org.omg.CORBA.ORB;
28 import org.objectweb.david.apis.services.naming.NameServer;
29 import org.objectweb.david.apis.services.naming.NameServerHelper;
30 import idl.*;
31 import idl.bank.*;
```

Like in the Hello World example, `AccountImpl` is the class implementing the `Account` interface. It *extends* `_AccountImplBase`.

```
35 class AccountImpl extends _AccountImplBase {
36    double balance;
37    String owner;
```

The constructor of the `AccountImpl` class initializes the two instance variables `balance` and `owner`, and then "connects" the instance to an ORB.

```
42    AccountImpl(String owner, double balance) {
43       this.balance = balance;
44       this.owner = owner;
45       Server.orb.connect(this);
46    }
```

The following code implements in a straightforward way the methods defined in interface `Account`.

[2] contained in `examples/david/bank/Makefile`

[3] We assume that the directory separator is `/`; If you are working on a Windows platform, you should use `\`; On a Macintosh, `::`.

[4] contained in `examples/david/bank/Server.java`

```
 50    public synchronized double balance() {
 51       return balance;
 52    }
 53
 54    public synchronized String owner() {
 55       return owner;
 56    }
 57
 58    public synchronized void add(double amount) {
 59       balance += amount;
 60    }
 61
 62    public synchronized void remove(double amount)
 63       throws NoCashException {
 64       if (amount > balance) {
 65          throw new NoCashException("Not enough money in account." +
 66                                    " Operation refused.");
 67       } else {
 68          balance -= amount;
 69       }
 70    }
 71 }
```

As we have just seen, extending an skeleton is the first possible method to tell the system that an interface may be used in remote invocations. The problem with that method, especially when the implementation language doesn't allow multiple inheritance – like Java –, is that it constraints the server implementation.

To avoid this, it is also possible to implement an interface generated by the IDL compiler, that simply maps the methods specified in the IDL specification for the interface type. In our example, to the IDL interface `Position` corresponds the Java interface `PositionOperations`.

`PositionImpl` simply implements the `PositionOperations` interface. The `PositionImpl` code thus contains no CORBA-specific code, but simply an implementation of the methods defined in `PositionOperations`.

```
 88 class PositionImpl implements PositionOperations {
 89    Hashtable accounts;
 90
 91    PositionImpl() {
 92       accounts = new Hashtable();
 93    }
 94
 95    public synchronized void open(String name,double init)
 96       throws OpenException {
 97       if (init <= 0) {
 98          throw new OpenException("Account must be opened with a positive" +
 99                                  " sum of money. Operation refused.");
100       } else {
101          AccountImpl account = (AccountImpl) accounts.get(name);
102          if (account == null) {
103             try {
104                accounts.put(name,new AccountImpl(name,init));
```

```
105             } catch (org.omg.CORBA.SystemException e) {
106                throw new OpenException("Technical problems in opening account." +
107                                        " Operation abandoned.");
108             }
109          } else {
110             throw new OpenException("Account in name of " + name +
111                                     " already exists. Operation refused.");
112          }
113       }
114    }
115
116    public synchronized Account get(String name) {
117       return (Account) accounts.get(name);
118    }
119
120    public synchronized double close(Account account) {
121       accounts.remove(account.owner());
122       return account.balance();
123    }
124 }
```

The `Server` class is the main class. It contains a reference to an `ORB` instance (used by the `AccountImpl` constructor) , and a `main` method.

```
129 public class Server {
130
131    static ORB orb;
132
133    public static void main (String[] args){
134       try {
```

ORB initialization is performed like in the Hello World example.

```
138          orb = ORB.init(args,null);
```

The `PositionImpl` class does not really implement the `Position` interface, but simply the non-CORBA subset of its methods. To create an implementation of the `Position` interface, we use the `_PositionImplBase` class. This is a non-standard way to implement the CORBA TIE mechanism.

```
145          Position position =
146             new _PositionImplBase(new PositionImpl());
```

Like in the `AccountImpl` case, the implementation must be “connected” to the ORB instance.

```
150          orb.connect(position);
```

David provides a very simple name server. Like the COS naming service, it may be obtained through the orb instance, using the ”NameServer” name.

```
155          NameServer name_server =
156             NameServerHelper.narrow(orb.resolve_initial_references("NameServer"))
```

This invocation registers `position` in the name server under the name `"Jonathan Position"`, replacing any older interface registered under the same name (`true` flag) (see the `NameServerOperations` HTML documentation).

```
164          name_server.put("Jonathan Position",position,true);
165          System.out.println("Bank Server is ready.");
```

This call blocks the calling thread until the `shutdown` method is called on `orb`.

```
168          orb.run();
169       } catch(Exception e) {
170          System.out.println("Bank Server exception:");
171          e.printStackTrace();
172       }
173    }
174 }
```

4 Step 4 write a client

The file `Client.java`[5] contains a client for our server. Here is the code:

```
25
26 import org.omg.CORBA.ORB;
27 import org.objectweb.david.apis.services.naming.NameServer;
28 import org.objectweb.david.apis.services.naming.NameServerHelper;
29 import idl.*;
30 import idl.bank.*;
```

4.1 Class Client

The Client code only consists in a `main` method.

```
35 public class Client {
36    public static void main(String args[]) {
37       try {
```

The ORB initialization, and the retrieval of the name server reference are preformed exactly like in the Server case.

```
40          ORB orb = ORB.init(args,null);
41          org.omg.CORBA.Object ns_ref =
42             orb.resolve_initial_references("NameServer");
43          NameServer name_server =
44             NameServerHelper.narrow(ns_ref);
```

After having retrieved a reference to the name server, the client retrieves a reference to a server registered in the name server.

```
48          Position position =
49             PositionHelper.narrow(name_server.get("Jonathan Position"));
```

[5] contained in `examples/david/bank/Client.java`

The rest of the code is distribution independent.

```
53          System.out.println("Opening Johnny Doe's account with $123.45");
54          try {
55             position.open("Johnny Doe", 123.45);
56          } catch (OpenException e) {
57             System.out.println(e.reason);
58          }
59          Account account = position.get("Johnny Doe");
60          System.out.println("Crediting $345.35 to Johnny's account");
61          account.add(345.35);
62          System.out.println("Debiting $635.23 from Johnny's account");
63          try {
64             account.remove(635.23);
65          } catch (NoCashException e) {
66             System.out.println(e.reason);
67          }
68          System.out.println
69             ("The balance in Johnny's account is $" + account.balance());
70       } catch(Exception e) {
71          System.err.println("Bank Client exception: ");
72          e.printStackTrace();
73       }
74    }
75 }
```

5 Step 5: run your client and server

The provided `Makefile` automates all the compilation phases. To compile the IDL file and the Java files, simply type `make` at a shell prompt in the example directory.

Once compiled, you may start the different applications:

- `make naming` starts the name server;
- `make server` starts the bank server;
- `make client` starts the client.