

COBRA

Concise Object Relational Architecture

Kimble Consultancy Services Ltd

14th June 1998

David Harvey-George

Revision 1.3, 1st March 2000

COBRA	1
INTRODUCTION.....	4
Design Goals	4
THE DATA DICTIONARY	6
Data_Dictionary table	6
Database table	6
Class_Dictionary table	6
Data Dictionary bootstrap	6
Priming the data dictionary.....	7
2-TIER APPLICATION.....	8
Initialisation	8
Creating Persistent Objects.....	9
Persistent Object Meta Data	9
Adding Retrieval Restrictions	10
Saving Objects	10
Deleting Objects	10
Deleting with Criteria.....	10
Retrieving Objects.....	11
Retrieving with Criteria.....	11
Transactions.....	11
REMOTE METHOD INVOCATION.....	13
Initialising the RMI Interface	14
Proxy Objects	15
JAVA SERVER PAGES	16
CORBA	19
STATEMENT CACHING.....	20
CALLBACK	21

OBJECT LOCKING	22
KNOWN BUGS	23
PERFORMANCE	24
Loopback Test	24
Remote Test.....	24
Local (no RMI).....	24
Serialized Objects.....	25

Introduction

Object Oriented programming (OOP) has been touted as the silver bullet to solve today's programming challenges. At the same time many systems will involve the manipulation of data stored in a Relational Database Management System (RDBMS). The dichotomy presented by interfacing the Object and Data models is often referred to as an 'impedance mismatch'.

OO programmers are used to thinking in terms of the relations between classes that encapsulate both data and methods, database gurus think about the relations between data encapsulated in tables.

COBRA is an object persistence layer written in the Java programming language. It uses relational database technology to provide the persistent storage mechanism; however the store is fully encapsulated shielding programmers from the details of relational database access.

Design Goals

The COBRA persistence layer fulfils the following requirements:

- 1) Fully encapsulates the underlying persistence mechanism. Only the methods save, retrieve and delete need to be performed on an object in order to create, retrieve, update or delete its attributes. There is no need for programmers to hard code SQL into their objects.
- 2) A data dictionary to map objects and their attributes onto their respective database, table and columns. Changing the database does not imply recompiling the application.
- 3) Supports concurrent access by the same or different users to the persistence mechanism.
- 4) Supports the grouping of actions on objects into discreet transactions
- 5) Supports Object (or Persistence) Identifiers as a primary mechanism for identifying objects.
- 6) Support legacy database schemas.
- 7) Support database cursors. Prior to version 2.0 Java Database Connectivity (JDBC) does not directly support cursors, this omission can result in large amounts of data being transferred over the network where potentially only the first few rows of the query are of interest.
- 8) Support for object inheritance and composition.
- 9) Provide raw access to the persistence mechanism.
- 10) Supports multiple architectures, e.g. 2 – n tier.
- 11) Native support for distributed object architectures including Common Object Request Broker (CORBA) and Remote Method Invocation (RMI)

Concise **Object Relational Architecture**

- 12) Support Multiple databases simultaneously.
- 13) Support for Java Database Connectivity (JDBC)
- 14) Provided access to RDBMS table meta data.
- 15) Permit retrieval/update/deletion of single or sets of objects based on sophisticated selection criteria
- 16) Support pre-compiled SQL for efficiency

Not all of these objectives will be fulfilled in earlier releases.

The Data Dictionary

The data dictionary maps Java persistent objects onto the corresponding RDBMS tables. A class is mapped onto a single instance of an RDBMS table. The data dictionary comprises 3 tables held in the primary database.

Data_Dictionary table

class_name (PK)	table_name	database_name
String	String	String

Database table

database (PK)	Driver	url	username	passwd	connections	vendor
String	String	String	String	String	integer	String

Class_Dictionary table

table_name (PK)	class_attribute	table_column	is_primary
String	String	String	char

Data Dictionary bootstrap

The data dictionary is stored in relational database tables accessible using JDBC. These may be part of the main database or on an entirely separate database or RDBMS. In order to initialise the data dictionary a bootstrap properties file must be defined, this is called connection.properties and contains the following variables:-

```

driver=JDBC driver name
url=JDBC URL of database
username=Database username
password=Database password
data_dictionary=table where main data dictionary is stored
database_table=table name where database information is stored
    
```

An example would be:-

```

driver=postgresql.Driver
url=jdbc:postgresql://sun2.kimble.co.uk:5432/orderdb
username=zardoz
    
```

```
password=wizard  
data_dictionary=Data_Dictionary  
database_table=Databases
```

Priming the data dictionary

The databases table must be primed with entries for the database containing the data dictionary and for any additional databases. The data dictionary is held in a database called *primary*, this is shared with the applications tables.

```
INSERT INTO databases VALUES ('primary', 'postgresql.Driver',  
'jdbc:postgresql://sun2.kimble.co.uk:5432/orderdb', 'zardoz',  
'wizard', -1, 'postgres-3.2');
```

Entries for each persistent object must now be added, for example a Customer table:

```
INSERT INTO data_dictionary VALUES  
('uk.co.kimble.examples.Customer', 'primary', 'customer');  
INSERT INTO class_dictionary VALUES ('customer', 'customer_id', '  
customer_id', 'y');  
INSERT INTO class_dictionary VALUES ('customer', 'name', ' name',  
'n');
```

etc...

This states that the customer table is mapped to a Java class called uk.co.kimble.example.Customer, the customer_id attribute is the primary key. For brevity lines can be omitted where the attribute is not a primary key its name is the same as the database column name.

In the future it is intended to add two utilities to the COBRA utility suite. A graphic application for visually realizing data schemas and generating data dictionaries and an application to generate a data dictionary for legacy databases.

2-Tier Application

COBRA makes little distinction between 2 and N tier applications. The main difference is that classes are included from `uk.co.kimble.cobra`. The initialisation process is slightly different and the COBRA layer runs in the context of the same Java Virtual Machine.

Initialisation

As was mentioned above, normally all the bootstrap information is held in a properties file. The initialisation process is a case of initialising a `PersistentConnection` and initialising the connection broker as shown in the code fragment below:

```
Properties p = new Properties();

try {
    FileInputStream f = new FileInputStream(property_file);
    p.load(f);
} catch (Exception e) {
    return;
}

PersistentConnection cobra_connection;
try {
    cobra_connection = new PersistentConnection(
        p.getProperty("driver"),
        p.getProperty("url"),
        p.getProperty("username"),
        p.getProperty("password")
    );
} catch (PersistentException e) {
    return;
}

try {
    ConnectionBroker.init(cobra_connection,
        p.getProperty("database_table"));
} catch (Exception e) {
    return;
}
```

Creating Persistent Objects

Any Java class that should be persistent must inherit from the class `uk.co.cobra.PersistentObject`. Each attribute should have a getter and setter method, these follow the pattern established by the Beans framework, that is, first letter in capitals the rest in lower case.

For example for the table:

Customer		primary
customer_id	integer	y
name	character(40)	n

The class would look like:

```
public class Customer extends PersistentObject {
    int customer_id;
    String name;

    public Customer()
    {
        super();
    }

    public void setCustomer_id(int i) {
        customer_id = i;
    }

    public int getCustomer_id() {
        return customer_id;
    }

    public void setName(String v) {
        name = v;
    }

    public String getName() {
        return name;
    }
}
```

Persistent Object Meta Data

SQL supports a number of aggregate functions:

`SUM(column)`,
`AVG(column)`,
`MAX(column)`,
`MIN(column)`,
`COUNT(column)`,

The keyword DISTINCT can be used with COUNT, SUM and AVG to eliminate duplicates. It should be possible to access this information through the PersistentObject class.

Adding Retrieval Restrictions

Normally Persistent Objects will be retrieved, saved or deleted based on their primary key information. Primary keys are one or more database columns that uniquely identify a row. The attributes that comprise a primary key are given by the Data Dictionary entry for the class.

However for reasons of efficiency we often wish to retrieve or delete a number of rows simultaneously. SQL allows a range of restrictions to be applied to the SELECT, DELETE and UPDATE statements, these include:

- Logical AND/OR operators for concatenating restrictions.

- Mathematical operators: = != > < >= <

- Fuzzy matching: LIKE

The subclause: ORDER BY can be used to sort the retrieved data by specific columns.

Persistent Objects support these restrictions through the PersistentCriteria class to quickly return or delete a sets of objects.

Saving Objects

To save an object to the database create and instance of the object and set, as a minimum, its primary keys:

```
Customer c = new Customer();
c.setCustomer_id(10);
pos.save(c);
```

COBRA maintains state about the object so can decide whether to persist the object to the RDBMS using an INSERT (a new object) or UPDATE (an exsiting object).

Deleting Objects

To delete an object call the delete method, the primary key attributes must be set.

```
Customer c = new Customer();
c.setCustomer_id(10);
pos.delete(c);
```

Deleting with Criteria

For efficiency COBRA permits sets of objects to be deleted using criteria, for example to delete all orders belonging to a customer with customer_id of 10:

```
Order o = new Order();
PersistentCriteria criteria = new PersistentCriteria();
```

Concise **Object Relational Architecture**

```
criteria.addEqualTo("customer_id", new Integer(10));
t.delete(o.getName(), criteria);
```

Retrieving Objects

To retrieve an object set its primary key attributes

```
Customer c = new Customer();
c.setCustomer_id(10);
c = pos.retrieve(c);
```

Retrieving with Criteria

Sets of objects can be retrieved based on certain criteria, for example to retrieve all customers whose names begin with the letter A:

```
Customer c = new Customer();

PersistentCriteria criteria = new PersistentCriteria();
criteria.addEqualTo("customer_name", "A*");

PersistentSet ps = pos.retrieve(c.getName(), criteria);
while (ps.hasMoreObjects()) {
    c = (Customer) ps.nextObject();
    System.out.println(c.getCustomer_name());
} // while
ps.close();
```

Remember to close a PersistentSet after use. This releases database resources immediately rather than waiting for garbage collection to occur.

All objects in a set can be retrieved by omitting the criteria parameter in the above example.

Transactions

It is useful to group operations which alter the contents of the RDBMS into discrete units called Transactions. This can be achieved using the Trans object. For example we may wish to delete a customer and all its orders. If either the delete of the customer or order(s) fails the operation should fail as a whole, otherwise referential integrity may be affected in the database. Note another way of achieving this is using a cascading delete implemented as a stored procedure where this is supported by the RDBMS:

```
Customer c = new Customer();
Trans t = new uk.co.kimble.cobra.Trans();
t.begin();

c.setCustomer_id(10);
t.delete(c);

Order o = new Order();
PersistentCriteria criteria = new PersistentCriteria();
```

Concise **Object Relational Architecture**

```
criteria.addEqualTo("customer_id", new Integer(s));  
t.delete(o.getName(), criteria);  
t.commit();
```

Remote Method Invocation

Remote Method Invocation or **RMI** is Javasoft's native distributed object protocol. **IIOP** (Internet InterOrb Protocol) offers interoperation with **CORBA** (Common Object Request Broker Architecture).

RMI aims to make the location of objects transparent to the user. RMI objects can have the same method calls and can pass any base type as a parameter. User defined objects must inherit from `java.io.Serializable` in order to be passed as parameters or returned from remote objects. In addition all rmi methods throw the `RemoteException` in addition to any other exceptions.

However RMI objects differ in one key respect from their local counterparts, they are really only interfaces and therefore do not possess any data of their own. All data is held in the implementation on the server and can only be accessed through remote methods.

This has an effect on the design of the persistence layer. In a 2 – tier architecture the persistent object can provide the methods `save`, `delete` and `retrieve` and these methods can directly access the data within the persistent object.

```
User po = new User();  
po.name = "fred"  
po.save();
```

This is tidy. However in order to scale applications to 3 – tier and beyond the above approach is simply not possible with RMI.

One solution is to provide an interface and implementation for each and every persistent object in our system and then implement remote accessor methods for reading and writing attributes. Moving from 2 to multi tier architectures is then a question of replacing the local with the remote objects.

Local code	Remote code
<pre>import myobjects.*;</pre>	<pre>import rmi.myobjects.*;</pre>
<pre>User po = new User(); po.setName("fred"); po.save();</pre>	<pre>User po = remoteServer.getUser(); po.setName("fred"); po.save();</pre>

The above code fragment shows how similar the local and remote versions of the application are. However in order to achieve this similarity the developer has to go through a lot of hard work.

Concise **Object Relational Architecture**

In the local version User simply extends PersistentObject in order to provide the required functionality. In the remote version the following steps are necessary.

1. Write the User interface
2. Write the User implementation, this will forward requests to a local persistent object of the correct type
3. Extends the remote server to offer the new object type

In addition setting and getting attributes is now a remote operation with corresponding performance overhead.

A better approach is to encapsulate the save, retrieve and delete functionality into a separate class. The system can provide local and remote versions of this class. The remote version is an adapter (see Adapter pattern, Gamma et al) for the local version of the class. Persistent objects are then be real objects passed and returned by the adapter.

Initialising the RMI Interface

The principal difference between the COBRA local and RMI interface is in initialisation. A client wishing to communicate with COBRA over RMI must first locate a remote server, it may also have to set a security manager if the RMI server is not the server from where the RMI classes were loaded (the origin server).

```
/*
 * Set a security manager so we can connect anywhere
 */
System.setSecurityManager(new mySecurityMangler());
p = new Properties();
```

As in the local version we load a properties file that defines certain runtime constants.

```
try {
    FileInputStream f = new FileInputStream(property_file);
    p.load(f);
} catch (Exception e) {
    System.exit(-1);
}
```

The properties file gives the name of the remote server, we look this up and get an instance of a remote PersistentSource.

```
String server = "rmi://" + p.getProperty("url");

// find remote object server
try {
    PersistentObjectFactory pof = (PersistentObjectFactory)
        Naming.lookup(server);
    PersistentSource pos = pof.getPersistentSource();

    Customer c = new Customer();
```

Concise **Object Relational Architecture**

```
c.setCustomerId(1001);
c.setCustomerName("Interactive Industries Inc.");
pos.save( c );
c = (Account) pos.retrieve(this);
} catch (Exception e) {
    System.err.println("error saving customer");
}
```

The local examples shown will all work unchanged, however remember to import remote versions of PersistentObject and PersistentSet. Trans objects must be obtained from the object factory, apart from that their interface is unchanged.

Proxy Objects

TBD

Note: this should not be confused with **CORBA** (Common Object Request Broker Architecture) client proxy objects.

Java Server Pages

Integrating COBRA objects with Java Server Pages is a breeze. COBRA objects already support the Java Beans pattern of getter and setter methods. The following example (JSP 1.0) shows an HTML form and a login bean (that is also a COBRA persistent object).

First we have to specify which bean we want to use:

```
<%@ page import="uk.co.kimble.jams.Account" %>

<jsp:useBean id="loginBean" class="uk.co.kimble.examples.Account"
scope="session"/>
<jsp:setProperty name="loginBean" property="*" />
```

The COBRA object is called Account, the JSP engine creates an instance of account called loginBean that exists for the session, alternative scopes are page and application.

```
<%
// take user straight to admin page if they are logged in
if (loginBean.logged_in) {
    pageContext.forward("next.jsp10");
}
%>
```

If the user is already logged in, then the form is skipped and the user is forwarded to the next page. Otherwise the user is prompted to enter a username and password. To save the user re-typing unnecessary data the username value is extracted from the bean. We may even have saved this value on the user's browser using a cookie so that it is there at the start of a new session.

```
<FORM name="login">
  <TABLE>
    <TD><INPUT name=username TYPE=text value=
      "<%= loginBean.getUsername() %>"
    </TD>
  </TR>
  <TR>
    <TD><INPUT TYPE=password name=password value="" >
    </TD>
  </TR>
  <TR>
    <TD><INPUT TYPE=submit name=connect value="Enter">
    </TD>
  </TR>
</TABLE>
</FORM>
<%= loginBean.getReply() %>
```

Concise **Object Relational Architecture**

The account bean has two variables corresponding to the column names in the account table: username and password. There are setters and getters for these values as well as a getReply() method and a setConnect() method. These methods are ignored by COBRA because they do not correspond to columns in the class_dictionary.

On the first run through the form username is set to blank and the reply text "click enter to log in to server" is displayed at the bottom of the form.

When the form is submitted the data is sent to the server via the URL query string (as specified by the default GET method). Where the form data names correspond to the included bean method names each setter is called in turn. That is the username is set to the entered username value, the password is set to the entered password value and the setConnect() method is called with the String parameter "Enter".

```
package uk.co.kimble.examples;

import uk.co.kimble.cobra.rmi.*;
import java.rmi.Naming;

public class Account extends PersistentObject {
    String username = "";
    String password;
    public boolean logged_in = false;
    public String reply;

    public Account() {
        super();
        init();
    }

    void init () { reply = "Click enter to login to server"; }

    public String getPassword() { return password; }
    public void setPassword(String s) { password = s; }
    public String getUsername() { return username; }
    public void setUsername(String s) { username = s; }

    public String getReply() { return reply; }

    public void setConnect(String s) {
        try {
            PersistentObjectFactory rof = (PersistentObjectFactory)
                Naming.lookup("//monte-carlo/JamsServer");
            PersistentSource pos = rof.getPersistentSource();

            Account a = (Account) pos.retrieve(this);
            if (a.getPassword().equals(password) {
                logged_in = true;
            } else {
                reply = "Incorrect username or password<br>";
            }
        }
    }
}
```

Concise **Object Relational Architecture**

```
        } catch (Exception e) {  
            reply = "Could not log in, please check your username and  
password<br><i>" + e.toString() + "</i><br>";  
            logged_in = false;  
        }  
    }  
}
```

The setConnect() method connects to the COBRA server using RMI (yes this is a 3 or 4 tier application depending on how you look at things). The user account corresponding to the submitted username is retrieved and the passwords compared. If there is a match logged_in is set to true (so when we drop through the form again we are taken to the new page) otherwise the reply message is changed to warn the user of the error.

In practise we wouldn't want to do the RMI lookup every time we action the bean so this would be shifted off somewhere else, probably to another bean with session scope.

CORBA

The original remote interace to COBRA was through CORBA, however more time has been spent on the RMI interface and COBRA is no currently supported in this release. See the package `uk.co.kimble.cobra.corba` for work in progress.

Statement Caching

JDBC offers the possibility of pre-compiling frequently used statements for efficiency. This has direct application to the object persistence mechanism where the statements to retrieve, update, create and delete a class will be constant for all instantiations of the object, only the data will change.

Callback

TBD

Object Locking

An example lock manager using Object Ids is included in *uk.co.kimble.cobra.extras*.

Known Bugs

Performance

Local Test Machine: AMD K5/133 running Redhat Linux 5.1, Postgres 6.3.4, JDK 1.1.5

Remote Test Machine: AMD K6/180 running Windows NT 4.0 sp 3, Symantech 2.5 JIT

Network: 10 Mbps Ethernet

Loopback Test

In this test RMI is used over the TCP/IP loopback interface. Connection to the database over the same interface.

PersistentObject.Retrieve :	50 ms
PersistentObject.retrieveSet:	100 ms
PersistentObject setter/getter:	5 ms
PersistentObject.nextObject:	45 ms
Overall Retrieve Time	70ms

Remote Test

RMI is used over a 10 Mbps LAN. The remote machine runs the COBRA middle-tier but connects back to the local machine where the database is stored. Getter/Setter/nextObject speed is better because the operation is divided amongst two processors and network latency is not significant. The retrieves take longer as a connection is made back to the database on the local machine.

PersistentObject.Retrieve :	341 ms
PersistentObject.retrieveSet:	365 ms
PersistentObject setter/getter:	3 ms
PersistentObject.nextObject:	20 ms
Overall Retrieve Time	398 ms

Local (no RMI)

PersistentObject.Retrieve :	80 ms
PersistentObject.retrieveSet:	91 ms
PersistentObject setter/getter:	0 ms
PersistentObject.nextObject:	5 ms

Serialized Objects

The remote interface was now modified so that rather than getting a reference to a remote object on the server a local object was created, its primary keys set and then it was sent over the network, an initialised object was then returned to the user. The overall time is about the same as with the remote object case.

PersistentObject.Retrieve :	-
PersistentObject.retrieveSet:	-
PersistentObject setter/getter:	0 ms
PersistentObject.nextObject:	-
Overall Retrieve Time	73 ms

PersistentObject.Retrieve :	-
PersistentObject.retrieveSet:	-
PersistentObject setter/getter:	0 ms
PersistentObject.nextObject:	-
Overall Retrieve Time	412 ms