FirstSQL/J Native Object/Relational Wrappers

Object/Relational wrappers are a tool for bridging the gap between object-oriented applications and database systems. O/R wrappers encapsulate database access in application-oriented wrapper objects. Applications use these special objects for all access to database entities.

There are a number of tools for generating O/R wrappers, including the Java Data Objects (JDO) initiative. In practice however, wrapper objects are often coded by the application developer. With the advent of Relational DBMSs supporting user-defined objects in the database, this task has become much easier. Creation of wrapper objects can take place in the DBMS before passing them to the client.

Why Use O/R Wrappers

Much has been made of the so-called *impedance mismatch* between object-oriented applications and traditional Database Management Systems (DBMSs). However, this conflict is quite natural. It is a reflection of the differing goals of the two sub-systems.

A DBMS is a resource shared among diverse client applications:

- Interactive and batch applications
- Reporting and analysis tools
- Ad hoc query interfaces.

It is the goal of a DBMS to provide effective services to all of its clients. A DBMS is also concerned with the overall integrity and security of the database.

A client application is oriented to performing or participating in a specific business task/process.

Object/Relational wrappers provide an excellent solution to the apparent conflict between the needs of the application and the needs of the DBMS. They allow the application to utilize the database in an application-specific manner. On the other side, they allow the DBMS to maintain a shared data structure that can service a rich set of application clients. This is accomplished through building wrappers that are specific to each application.

Application-specific wrappers support a concept important to both object-oriented and relational systems – a sub-system should have access to data only if relevant to its needs. In object-oriented terminology, this is known as *Encapsulation*.

The alternative of creating an O/R wrapper that services multiple applications is a daunting task. Such *universal* wrapper classes, that attempt to provide services to multiple applications, have two seemingly conflicting problems:

- Too generic – in order to satisfy the diverse application needs, the wrapper implementations often emphasize facilities common to all client applications.
- Too brittle – when the needs of individual client applications changes, the wrapper must accommodate this without affecting clients that haven t changed.

O/R wrappers oriented to individual applications are *lightweight* objects. This makes them easy to change as the needs of their client application changes.

Native O/R Wrappers Using Objects in the Database

An Object/Relational DBMS (ORDBMS) is a relational DBMS that supports cataloging of user-defined object classes in the database. ORDBMSs can directly create and manipulate objects of the user-defined classes, and they can return them to client software. Class level methods can function as Stored Procedures.

With these capabilities, Object/Relational DBMSs allow creation of wrapper object within the DBMS. They return these objects as result values to the client.

Building O/R Wrappers Using Stored Procedures

It is feasible to build simple object wrappers directly in a query using an ORDBMS. This is described in the next section. For more complicated objects, a stored procedure is the best choice for building wrapper objects.

Object-oriented stored procedures have the right capabilities to create complex wrapper objects. They have direct access to the DBMS for data retrieval (they run in the DBMS itself.) Their native object-oriented features allow them to easily create a network of related objects.

For a simplified example, we will describe creation of a wrapper object for an individual sales order. There are two associated tables in the database:

- sales_orders(ord_id,cust_id,ord_date)
- line_details(ord_id,line_no,product,qty,price,taxable)

Primary keys are underlined.

The primary wrapper class is SalesOrder. A SalesOrder object contains information from a row in the sales_orders table plus a collection of LineDetail objects. The secondary class – LineDetail, contains information from a row in the line_details table.

The stored procedure to create a complete SalesOrder object is named retrieveOrder(). For convenience, we will place it as a class-level method in the SalesOrder class. Such class-level methods are also called *Factory* methods because they manufacture objects.

The retrieveOrder() stored procedure receives a single argument – the id of the order to be retrieved. It returns a SalesOrder object as its result.

A pseudo-code version of retrieveOrder():

- Get an internal connection to the database.
- Execute a query on the sales_orders table to retrieve the row matching the order id argument.
- Use the information from the retrieved row to create a SalesOrder object with an empty collection of detail lines.
- Execute a query on the line_details table to retrieve all lines whose ord_id column matches the order id argument.

- For each row from line_details, create a LineDetail object and add it to the collection of lines in the SalesOrder object.
- Return the completed SalesOrder object as the result.

For expository purposes, retrieveOrder() uses a separate query for each different type of object created. The stored procedure could have used a single query to accomplish its purpose. The general case would use separate queries.

The application code to retrieve a SalesOrder wrapper for an order and its detail lines is just 5 lines in Java. Listing 1 shows sample Java code for client retrieval of a wrapper.

Listing 2 contains the SalesOrder class declaration in Java code. It includes the retrieveOrder() factory method used as a stored procedure. Listing 3 contains the class declaration in Java for the LineDetail class.

Building O/R Wrappers Using SQL Queries

It is also possible to create an O/R wrapper using a SQL query, eliminating the need to develop a stored procedure. It requires an ORDBMS supporting user-defined classes in its catalog. In most cases, the result of a query should be a single wrapper object, but a query can generate a set of interconnected objects. Wrapper objects that include lists of secondary objects, like orders and detail lines, customers and invoices, etc., would need to use a stored procedure, as described above.

As an example of a query producing wrapper objects, we will use a wrapper class – InvoiceWrapper, for wrapping customer invoices. InvoiceWrapper is a user-defined database class, cataloged in the ORDBMS. It has a constructor that receives the invoice #, customer #, invoice balance and invoice date.

The ORDBMS provides a NEW operator for creating objects for database classes. Using NEW, a query can dynamically create a wrapper object and return it as a result to the client. For example,

```
SELECT NEW InvoiceWrapper(inv_id, cust_id, inv_bal, inv_date)
FROM invoices
WHERE inv_id = 1384;
```

The result of this query is an InvoiceWrapper object containing the details about invoice #1384.

Changing the query to:

```
SELECT NEW InvoiceWrapper(inv_id, cust_id, inv_bal, inv_date)
FROM invoices
WHERE inv_date = CURRENT_DATE;
```

will return a set of InvoiceWrapper objects representing today's invoices.

Wrapper *Agents* Using SQL Queries

Most ORDBMSs also allow a database object instantiated on the client to be passed to the database using **?** parameters in SQL statements. This capability is useful for

creating database *agents*. A database agent is an object designed for a round-trip (from the client to the ORDBMS and back to the client).

A database agent would be an object created on the client with application-specific intelligence (data and methods). The client passes the agent object to a SQL query and receives the same object back as the result of the query. The result object has updated itself based on application-specific information and on database contents.

A round-trip object uses a specific type of method to accomplish this feat. The method must use the object itself as its return value. There are no other restrictions on the method.

Using the InvoiceWrapper class from the previous section, we could define a method with the following Java signature:

```
InvoiceWrapper retrieve(int invId, int custId, BigDecimal invBal, Date
        invDate)
```

The retrieve method would be used in a SQL query as follows:

```
SELECT (CAST(? AS InvoiceWrapper)).retrieve(inv_id, cust_id,
        inv_bal, inv_date)
FROM invoices
WHERE inv_id = 1384
```

The client would create an InvoiceWrapper object and pass it for the **?** parameter in the SQL statement. It would then retrieve the updated InvoiceWrapper object as the result of the query.

Conclusion

Application-specific O/R wrappers are an excellent solution for bridging the gap between applications and DBMSs. They allow the application to utilize the database in an object-oriented manner without changing the database structure to meet the needs of individual applications.

Modern Java ORDBMSs (Object/Relational Database Systems) make creation and use of O/R wrappers much easier. Java ORDBMSs allow the wrapper class to be cataloged in the database and wrapper objects to be created using SQL statements

Listings:

Listing 1: Sample O/R Wrapper Retrieve for Client

```
// client code to retrieve O/R wrapper from ORDBMS
CallableStatement call = conn.prepareCall("{?=CALL SalesOrder.retrieveOrder(?)}");
call.setInt(2, 2451);
call.execute();
call.registerOutParameter(1, Types.OTHER, "SalesOrder");
SalesOrder order = (SalesOrder) call.getObject(1);
```

Listing 2: O/R Wrapper SalesOrder Class

```
// SalesOrder.java -- O/R wrapper class for Sales Order

public class SalesOrder implements Serializable
```

```
{
  private int ordId, custId;
  private Date ordDate;
  private SortedMap lines = new TreeMap();
  public SalesOrder(int ordId, int custId, Date ordDate)
  {
    this.ordId = ordId;
    this.custId = custId;
    this.ordDate = ordDate;
  }
  public int getOrdId()
  {
    return ordId;
  }
  public int getCustId()
  {
    return custId;
  }
  public Date getOrdDate()
  {
    return ordDate;
  }
  // retrieve line detail by line number
  public LineDetail getLineDetail(int lineNo)
  {
    return (LineDetail) lines.get(new Integer(lineNo));
  }
  // get an ordered list of detail lines (Line Detail objects)
  public Iterator lineDetails()
  {
    return lines.values().iterator();
  }
  // add a detail line to this order
  public void addLine(LineDetail line)
  {
    lines.put(new Integer(line.getLineNo()), line);
  }

  // Stored Procedure to create a SalesOrder O/R Wrapper
  public static SalesOrder retrieveOrder(int ordId)
  {
    SalesOrder order = null;
    try
    {
      // Database is builtin class for retrieving an internal DBMS connection
      Connection conn = Database.getConnection( );
      PreparedStatement prep;
      ResultSet results;
      // Retrieve information for sales order
      prep = conn.prepareStatement("SELECT cust_id, ord_date " +
                                   "FROM sales_orders " +
                                   "WHERE ord_id = ?");
      prep.setInt(1, ordId);
      results = prep.executeQuery();
      if (results.next())
      {
        int custId = results.getInt(1);
        Date ordDate = results.getDate(2);
        results.close();
        prep.close();
        order = new SalesOrder(ordId, custId, ordDate);
        // Retrieve information for detail lines in sales order
        prep = conn.prepareStatement("SELECT line_no, product, qty, " +
                                     "       price, taxable " +
                                     "FROM line_details " +
                                     "WHERE ord_id = ?");
        prep.setInt(1, ordId);
        results = prep.executeQuery();
        while (results.next())
        {
          int lineNo = results.getInt(1);
```

```
          String product = results.getString(2);
          int qty = results.getInt(3);
          BigDecimal price = results.getBigDecimal(4);
          boolean taxable = results.getBoolean(5);
          order.addLine(new LineDetail(ordId,  lineNo, product, qty, price,
                                       taxable));
        }
      }
      results.close();
      prep.close();
    }
    catch (SQLException ex)
    {
      order = null;
    }
    return order;
  }
}
```

Listing 3: O/R Wrapper LineDetail Class

```
// LineDetail.java -- O/R wrapper class for Sales Order Detail Line

public class LineDetail implements Serializable
{
  private int ordId, lineNo;
  private String product;
  private int qty;
  private BigDecimal price;
  private boolean taxable;
  public LineDetail(int ordId, int lineNo, String product, int qty,
                    BigDecimal price, boolean taxable)
  {
    this.ordId = ordId;
    this.lineNo = lineNo;
    this.product = product;
    this.qty = qty;
    this.price = price;
    this.taxable = taxable;
  }
  public int getOrdId()
  {
    return ordId;
  }
  public int getLineNo()
  {
    return lineNo;
  }
  public String getProduct()
  {
    return product;
  }
  public int getQty()
  {
    return qty;
  }
  public BigDecimal getPrice()
  {
    return price;
  }
  public boolean getTaxable()
  {
    return taxable;
  }
}
```