SIMPLIFYING DATA INTEGRATION – LINKING JAVA™ OBJECTS TO RELATIONAL DATABASES

TECHNOLOGY WHITE PAPER

For more information, contact:
Software Tree, Inc.
650 Saratoga Ave. San Jose, CA 95129
P: 408-557-6769
F: 408-557-6799
Email: info@softwaretree.com
http://www.softwaretree.com/

J-DATABASE EXCHANGE™ (JDX ™)

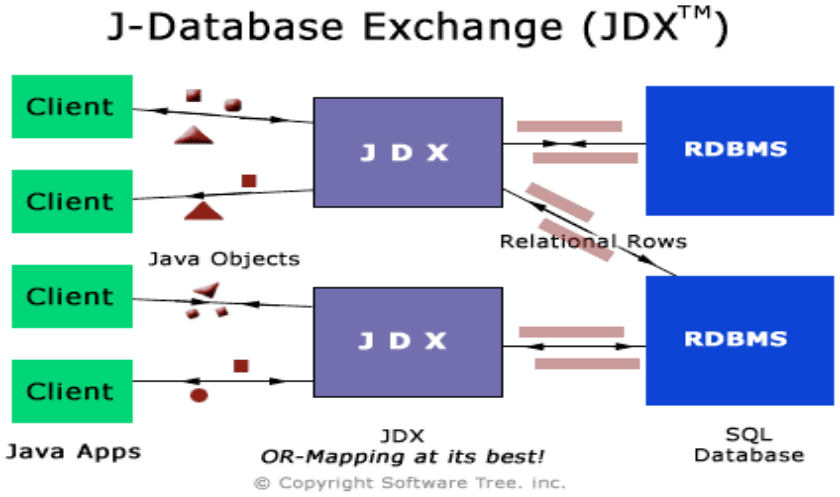
Object-oriented programming (OOP) has become the dominant programming paradigm these days. In OOP languages (e.g. Java, C++), a class encapsulates the structure and behavior of objects of a certain type. Business objects are easier to represent as instances of these classes.

One common need for virtually all business applications is the persistence (stable storage) of business objects. The options for persistent storage are file systems, object-oriented databases and relational SQL databases (RDBMS). For simple, non-mission critical applications requiring low data volume, a file system solution may be sufficient. For high performance, scalable, transactional and robust applications, a database management system is required for storing business objects.

Relational database management systems provide a popular and pragmatic repository for these business objects because of the maturity of the RDBMS technology and the availability of numerous third-party tools for analyzing and managing the relational data. However, there is an inherent conceptual paradigm-mismatch *(also known as impedance mismatch)* between an object-oriented model and a relational model. Furthermore, SQL as the standard language of interface for relational databases does not mix well with an OOP language like Java. However, there is a genuine need to use relational databases to store business objects. How to bridge this gap?

First need is to easily define the mapping between the object model and the relational model. The second need is to store this mapping information such that it can be used most naturally and conveniently. Thirdly, there is need for a product to understand this mapping and do appropriate translation between the object and relational data. That also includes the functionality of generating the relational schema definition given the class definitions and the mapping information. And finally, there is need for defining an intuitive application-programming interface (API), which will make the task of a programmer easier by relieving him of the burden of generating low-level SQL statements to store/get the object data into/from the relational database.

Software Tree provides an innovative solution to define the mapping between an object model and a relational model. We also describe an API that can effectively be used by application programmers to meet their object persistence needs using an RDBMS. The name of our product is JDX, which stands for J-Database Exchange.



WHAT IS JDX?

JDX is a high-performance, cross-platform, developer friendly and cost effective solution for transactional persistence of Java objects using relational databases. JDX's object-relational mapping (OR-Mapping) technology provides a natural object-oriented interface to relational data. It supports complex-object modeling including class-hierarchies, provides integration with legacy data and employs a scalable architecture.

JDX's adaptable design and 100% Java implementation makes it usable in any tier of a Java application – be they standalone, JSP/Servlet based or EJB based.

Using JDX requires three simple steps:

1. Define business objects (Java Classes),
2. Define object-relational mapping and,
3. Develop application using intuitive and powerful JDX API's.

WHY JDX? WHY CAN'T WE USE JDBC TO STORE JAVA OBJECTS IN RDBMS?

JDBC is a Java application programming interface to SQL databases defined by Sun. Although this is a useful technology, the use of JDBC forces application developers to generate SQL statements explicitly. This requires a lot of hand coding of SQL statements and then processing the results. It is a very mundane and time-consuming job.

Just to give an idea of how JDX can simplify the program development task of retrieving Java objects from the database, the following is an example of appropriate code segments - one using JDBC and the other using JDX.

For this example, we assume 2 classes - Title and RoySched (*Shown in right panel*) Each Title object has an array of RoySched objects. Primitive attributes of Title objects come from titles table and that of RoySched objects come from roysched table. We are trying to retrieve Title object(s) corresponding to a title_id stored in a String variable 'tid'.

Java Classes Definitions

Here are the Java class definitions for the examples used in this white paper. They are based on the sample database 'pubs' which comes with Microsoft SQL Server (version 6.5) RDBMS.

RoySched (table roysched)

```
public class RoySched implements
java.io.Serializable {
   public String title_id;
   public int lorange;
   public int hirange;
   public int royalty;
}
```

Title (table titles)

```
public class Title implements
 java.io.Serializable {
   public String title_id;
   public String title;
   public String type;
   public String pub_id;
   public java.math.BigDecimal price;
   public java.math.BigDecimal advance;
   public int royalty;
   public int ytd_sales;
   public java.sql.Timestamp pubdate;
   public String notes;
   public RoySched[] royscheds; //
   Array of RoySched objects for this
   title
}
```


RAW JDBC VERSUS JDX

JDBC CODE:

```
// Assuming a Connection object 'con' has
// been obtained to the database.

// Retrieve the Title object

Statement stmt = con.createStatement();

// First fetch the titles table row

String query = "SELECT title_id, type,
price, title, ytd_sales, pub_id," +
" pubdate, royalty, advance, notes FROM
titles" + " WHERE title_id = '" + tid +
"'";

ResultSet rs = stmt.executeQuery(query);

Title title = new Title();

while (rs.next()) {
   title.title_id =
   rs.getString("title_id");
   title.type = rs.getString("type");
   title.price = rs.getBigDecimal("price",
   2);
   title.title = rs.getString("title");
   title.ytd_sales =
   rs.getInt("ytd_sales");
   title.pub_id = rs.getString("pub_id");
   title.pubdate =
   rs.getTimestamp("pubdate");
   title.royalty = rs.getInt("royalty");
   title.advance =
   rs.getBigDecimal("advance",2);
   title.notes = rs.getString("notes");
   break; // Simplifying assumption –
          // only one row is returned.
}

// Now fetch all the corresponding
// roysched table rows

Vector royScheds = new Vector();
RoySched roySched;

query = "SELECT title_id, lorange,
hirange, royalty FROM roysched" +
   " WHERE title_id='" + title.title_id
   + "'" + " ORDER BY royalty";
rs = stmt.executeQuery(query);

while (rs.next()) {
   roySched = new RoySched();
   roySched.title_id =
   rs.getString("title_id");
   roySched.lorange =
   rs.getInt("lorange");
   roySched.hirange =
   rs.getInt("hirange");
   roySched.royalty =
   rs.getInt("royalty");
   royScheds.addElement(roySched);
}
stmt.close();

// Initialize title's royscheds
// attribute with the collection of
// roysched objects

title.royscheds = new
   RoySched[royScheds.size()];
royScheds.copyInto(title.royscheds);
```

JDX CODE:

```
// Assuming a handle 'jdx' to the JDX
// service object has been obtained.
// Retrieve the Title object(s). In
// general, many qualifying objects may
// be returned.


Vector queryResults = jdx.query("Title",
"title_id
      = '" + tid + "'", -1, 0, null);

Title title = (Title)
queryResults.firstElement();
```

That's it! Similar efficiencies are also gained for inserting, updating or deleting objects. JDX improves program development process by eliminating the need for hand coding of SQL statements. The resulting code is very intuitive, simple and, easy to maintain. All the SQL code is dynamically generated at runtime that avoids messy alternative approaches, which generate intrusive SQL code statically.

Following section highlights the issues associated with using raw JDBC in application code.

ISSUES WITH USING RAW JDBC

- **Generation of SQL statements (SELECT, INSERT, UPDATE and DELETE) for each class**
  You have to write these statements manually.
  What if you have hundreds of classes for your application?
- **Hard coding of database column names**
  A a a r g h !
- **What if a new attribute is added to a class?**
  All corresponding statements have to be updated.
- **What if an attribute name changes?**
  All corresponding statements have to be updated.
- **What if an attribute type changes?** *The getXXX call has to be changed appropriately. May involve database changes also.*
- **What if a class hierarchy is involved (e.g., a class PoliticalTitle may inherit from Title and its objects may come from a different table)?**
  We have to potentially collect objects from multiple tables. Lots of changes at many levels.
- **What if the class structure is more complex (more references, more levels)?**
  The code becomes exponentially complex!
- **What if we want to do directed queries for a complex object (i.e., follow some references and ignore a few of them etc.)?**
  How to specify such a query?
  Do we repeat the hard-coded SQL statements in different parts of the code?
- **How about directed insert, update, and delete operations?**
  Same issues as above.
- **How to define and use complex relationships between objects?**
  Can you easily define a notion of an object contained by value or by reference? How to implement persistence-by -reachability?
- **How scaleable is such an implementation?**
  Each application has to get its own connection to the database.
  The application process is directly talking to the database server, which may be many networking hops away.
- **How easy is the code to share between multiple applications?**
  Does each programmer in the team have to know where different components of an object are stored and how they are connected (primary keys, foreign keys)?
- **What if you want to use a new JDBC driver?**
  Do all JDBC drivers behave the same? Do they map uniformly?
- **Do you have tools to generate schema definition given your class definitions? Can you define Java classes based on existing relational data? What if you want to move your application to a different backend database?**
  Schema generation tool. Use of different JDBC drivers.
- **Do you like mixing SQL with Java?**
  Paradigm mismatch between object and relational views.
- **How easily can this code be maintained/enhanced?**
  Is a lot of time spent developing, debugging and enhancing this type of code?
  Would you rather be devoting more time to business logic?
- **How thick you want your application/applet to be?**
  All object-relational mapping code and the JDBC driver code may be attached to your application/applet.

WHY JDX AS A SOLUTION?

Persistence of business objects using RDBMS is an important requirement for modern applications that are mostly developed on Java platform. However, there is an inherent paradigm-mismatch between the Java object model and the SQL relational model. JDX OR-Mapping technology seamlessly bridges the gap between the two models by providing a very natural and powerful object oriented interface to store and retrieve Java business objects. JDX can generate database schema from Java object models (class definitions) and vice-versa. JDX leverages JDBC standard but hides all its complexities and thereby helps achieve significant reductions in overall time, risk and cost associated with Java database programming.

JDX uses a clean, non-intrusive approach, which does not require any pre-processing or post-processing of Java code. JDX, itself, is implemented in 100% Java making it portable to every Java enabled platform. JDX's adaptable and extensible design makes it usable in any tier of a Java/J2EE application.

Accelerate development lifecycle by avoiding tedious, time-consuming and error-prone tasks of low-level JDBC/SQL programming for persistence of business objects.

Accelerate data movement by using JDX's fine-tuned object-relational mapping engine.

Accelerate your Java/Servlets/JSP/EJB projects by concentrating on business logic instead of wasting time on infrastructure details.

http://www.softwaretree.com/products/jdx/Jdx1.htm

From Software Tree - enjoy the fruits!