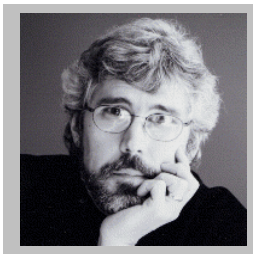
db4o | The Open Source Object Database | Java and .NET

The Database Behind the Brains

By Rick Grehan | 2nd, Updated Edition | March 2006



Rick Grehan is a QA Engineer at Compuware/Numega labs, where he has worked on Java and .NET projects.

He is also a contributing editor for InfoWorld Magazine. His work has appeared in Embedded Systems Programming, EDN, The Microprocessor Report, and Computer Design. Before coming to Compuware, Rick was on the Discover DSP Project at Metrowerks, Inc.

Earlier, Rick was a Senior Editor at BYTE Magazine, where he was the Lab Director, and authored BYTE's JavaTalk column.

Mobile devices, information appliances, intelligent control systems; no deep investigation is needed to uncover the spreading frontiers of computer applications. These systems are usually referred to as “embedded systems,” but that term does not express the growing sophistication of such applications.

Processors’ clock cycles rise, memory densities grow, hard disks shrink, prices fall, and the tasks these systems are called upon to perform become ever more complex. Embedded systems are being asked to do more than ever – to be ‘smarter’ than ever. This is really another way of saying that they are being asked to store, retrieve, and manipulate more and more data; and to do so with a responsiveness that their users have come to expect in a world of growing processor horsepower and increasing communication throughput.

Looked at another way, an intelligent device’s intelligence does not depend only on the algorithms it executes, but also on the data it feeds to those algorithms. So, as device sophistication increases, there follows an increasing need for correspondingly sophisticated data organization, storage, and retrieval software to meet the demands placed on the embedded application.

What Is Needed

We could answer the question "What is needed?" quite simply: "A database library that is small, fast, powerful, and easy to use."

That’s quite a condensation, and tells us little by itself. We should examine the criteria more closely.

• **Minimal consumption of resources.** Although ongoing technological advances permit system designers to purchase more memory for less, optimizing an application's memory consumption will never go out of fashion. The tradeoff is simple: as our database library consumes less memory, more is available for other application components, and the designer can add more features (or boost the performance of existing ones).

Notice that we used the word “optimizing” above. We are not talking of simply reducing the database library's size – that can easily be done by amputating features. But, exchanging capabilities for bytes is dangerous: you’ll certainly end up with a smaller product, but it may be less useful as well.

• **High throughput.** This requirement is axiomatic. Only in very limited applications are low database access times tolerated. Put another way, we are unaware of anyone complaining of a database that returned its results too quickly.

• **Easy implementation.** In spite of all the improvements in development environments, in spite of the advances in object-oriented programming, of framework libraries brim-full with pre-written algorithms and data structures, developers must still design and code the application’s business logic – a chore that remains difficult. Hence, a database library’s API

should not present a learning challenge. It should be “as simple as possible, but no simpler,” to borrow a well-wrought expression. This creates a minimal learning curve; developers can concentrate on the application, rather than studying the database library’s mechanics.

In addition, incorporating a database library into a development project should be a one-step process. Ideally, the library will be a single file (for Java, a .JAR file; for .NET, a .DLL file). For command-line based projects, adding the library should involve modifying a single line in a build script; for IDE-based projects, the addition should be a drag-and-drop operation.

- **Portable.** A database library should be 'platform-agnostic' to maximize its run-time targets – and thereby maximize the application's potential customers. Wide portability also gives programmers the luxury of developing on a major OS – and so take advantage of state-of-the-art development and debugging tools – with the assurance that the result is not bound to that OS.

- **Reliable.** Reliability is another axiomatic requirement – and likely the most important. An unreliable database library is simply unusable. For most embedded applications, particularly those employed in realtime systems, reliability is a non-negotiable property of ALL components.

Furthermore, the database library must perform according to industry-recognized criteria. Specifically, the database library must implement the ACID properties. (See the “ACID” sidebar.)

A Relational Possibility

A relational database is certainly the best known, and possibly the most widely used database storage paradigm. Its popularity makes a relational database a likely choice for any database application. However, in spite of the advantages, incorporating a relational database into an object-oriented application is not altogether smooth. Commentaries on the use of a relational database system in an object-oriented environment often speak of an ‘impedance mismatch’. The term is borrowed from the electronics world, and expresses the dissonance between the relational and object-oriented models.

This mismatch arises from the fact that a relational database management system (RDBMS) stores information in rows within tables. Consequently, storing the contents of an object into such a database requires that the object be dissected – pulled apart, and its pieces distributed into fields within the database’s tables. To retrieve the object, its disparate parts must be gathered and re-assembled.

An example (in Java): An Intelligent Vending Machine

This effort is easily illustrated by an example.

Suppose we have designed an intelligent vending machine: one that tracks the kinds and quantities of snacks it dispenses, monitors the change available in its till, tracks its own sales and – perhaps it is outfitted with a wireless connection to the internet? – emails the home office when in need of restocking. (Trust us; this is not as far-fetched as it sounds.)

ACID

ACID is an acronym for four characteristics that a database must provide before it can reasonably be considered a usable database system. They are:

Atomicity - The components of a transaction on the database must execute in an all-or-nothing fashion. For example, if a transaction on the database involves deleting 4 objects, then those 4 objects must be deleted as though they were a single object. It won't do if 3 of the objects are deleted, but one is somehow left unremoved.

Consistency - Operations on the database move it from one well-defined state to the next, with no intermediate states visible. If, for example, a user added Object A to the database, then it should at no point be possible for that user (or any other user) to retrieve a 'partial' Object A. The database should never appear to be in a state where operations are only partially complete.

Isolation - Multiple transactions at work on the database are unaware of one another. So, if two users attempt to modify the same object simultaneously, the database must implement some mechanism for serializing their access to the object, so that neither user's work interferes with -- or even 'sees' -- the other's.

Durability - Once a transaction has been 'committed' to the database, its work is not lost, even in the face of hardware or software failure. So, if a user executes a transaction on the database to delete 3 objects, and the system crashes in the process of deleting the second object, then when the system is rebooted, the database will recover not only itself, but the incomplete transaction. And it will finish the transaction.

Retrieving a `Snack` object from an RDBMS would require something like the following code:

```
  ResultSet results = statement.executeQuery("SELECT ID, Name, Cost," +
   " Retail, Supplier FROM Snack WHERE ID = " + searchID);
  if(results.next())
  {  snack = new Snack();
     snack.ID = results.getLong("ID");
     snack.name = results.getString("Name");
     snack.cost = results.getLong("Cost");
     snack.retail = results.getLong("Retail");
     snack.manufacturer = result.getString("Supplier");
     //  ... Do something with the Snack object ...
  }
}
```

Listing 1. Retrieving a snack's data from an RDBMS. This code retrieves information for the snack whose ID is given by the input variable `searchID`.

The result of the query is first stored in the collection `results`. Then, assignment operations copy the contents of individual fields from the `results` `ResultSet` into the `snack` object. In addition, each field's assignment operation could involve a translation from the format of the value in the database, to the internal representation used by Java (though this translation takes place within the specific method, and is not explicitly visible). The reverse of all this effort is required to put an object into the database. You can imagine what the code would look like for large, complex object structures.

Notice also that the process begins with the execution of an SQL statement that is expressed in a string. If the statement is not pre-compiled, yet more CPU cycles are consumed to parse and execute the SQL that performs the actual query on the back-end database.

The impedance mismatch is somewhat alleviated in object/relational database systems. An object/relational database system (ORDBMS) handles the translation between objects and the relational database invisibly. (The relational database is still there, it's just "hiding" on the back end.) As a result, the explicit code – shown above – for disassembling and reassembling objects is not needed.

But, even though the code is not explicit, it's still there. It is typically veiled in a "mapping layer", a collection of classes and methods within the object-relational database library. This mapping layer accomplishes the translation between the objects in the application and the tables, rows, and fields in the relational database.

So, while the programmer can manipulate objects as objects, and therefore has less code to write, the CPU cycles are still consumed in translating between relational database and object data. And, somewhere in the database library, SQL must execute to extract rows from or store rows into the database tables.

In summary, an RDBMS adds space and time overhead to the application. Space is consumed by the presence of the object-to-RDBMS translation code; time is consumed by the execution of that code. An ORDBMS improves matters some; programmers are at least relieved of writing translation code. But, though hidden, the translation code is still present, eating bytes of memory and clock cycles of CPU time.

The Solution

The solution to the problems cited above is a purely object-oriented database system. And a particularly practical object-oriented database is db4o (from db4objects, Inc.). db4o simultaneously meets the needs outlined in the first section, and sidesteps the difficulties described in the second section. db4o's strengths are best illuminated if we explore how it addresses the issues we have already discussed.

• **Minimal Consumption of Resources.** The db4o library uses only about 400K. As you will see, however, its memory-frugal footprint does not betoken a deficit of capabilities.

• **High Throughput.** db4o's execution is on par with the best database systems. The benchmark results below show db4o's performance compared to typical SQL databases:

barcelona benchmarks	read	write	query	delete
db4o/4.5.200	1.0	1.0	1.0	1.0
Hibernate / MySQL	20.8	32.2	6.7	17.3
Hibernate / HSQLDB	10.4	5.4	536.0	3.9
JDBC / MySQL	10.8	14.6	1.7	6.5
JDBC / HSQLDB	0.4	1.7	677.8	0.7
JDBC / Derby	3,696.0	12.9	1,299.7	7.1
JDO/VOA/MySQL	4.4	14.8	3.0	2.4

Benchmarking db4o. The above table shows the performance of db4o as compared to typical SQL databases and O/R mappers for read, write, query and delete operations. The Complex operations involve database actions on objects with a tree structure involving 5 levels of inheritance.[1]

• **Easy Implementation.** The Java version of db4o is a single JAR file; the .NET version is a single DLL file. You add the library to your application by placing it in your CLASSPATH (if using command-line tools for Java development) or dropping the file into your project (if using an IDE for either Java or .NET).

The db4o API is not overrun with elaborate, complex classes and methods. For example, a db4o programmer works primarily with db4o's `ObjectContainer` class (which is a representation of the database itself). The `ObjectContainer` interface defines only 10 methods; yet those 10 provide the bulk of database manipulation – adding, searching, and deleting data.

So, if we assume that an `ObjectContainer` called `vendingmachineDB` has already been opened, and we want to store a `Snack` object to the database, the Java code is simply:

```
vendingmachineDB.set(snack);
```

This same fragment of code can be used to update a `Snack` object already stored in the database. (Note that, if the above is the last database operation in the application, it should be followed by calls to the `ObjectContainer commit()` and `close()` methods so that activity on the database is properly teminated. But, similar commands would be needed with an RDBMS or ORDBMS.)

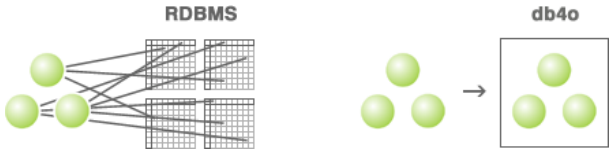
• **Portable.** As already stated, versions of db4o exist for Java and .NET. The Java version can run on all Java platforms from Java 1.1.x forward (including PersonalJava and the J2ME CDC configuration). The .NET version is compatible with .NET 1.0, 1.1, and the CompactFramework. Furthermore, the .NET version can be used with all the .NET languages, and runs on the open-source Mono framework.

[1] More information is available on db4objects' Website www.db4o.com/about/productinformation/benchmarks/

• **Reliable.** Finally, db4o supports all of the ACID characteristics. Multiple simultaneous users of a db4o database are appropriately isolated, their operations invisibly serialized by the db4o library. Transactions are terminated by the `ObjectContainer` class's `commit()` and `rollback()` methods. And, in case of a system crash during a database update, when the db4o `ObjectContainer` is re-opened, any interrupted transaction is properly completed.

No Impedance Mismatch

db4o is a true object database. Objects are stored 'as-is'; there is no object-to-relational translation layer, either explicit or invisible. In addition, db4o can handle arbitrarily complex object structures. And you don't have to create schema definitions for mapping objects to relational database tables. Your application's class hierarchy and object relationships themselves define the database schema:



No Impedance Mismatch. Your application's class hierarchy and object relationships themselves define the database schema.

The ease with which one employs db4o can best be shown via a series of examples. Returning to the vending machine database example (that we showed in SQL above), suppose we create an identical database in db4o. Code to open the database and add a new snack "record" would look like this:

```
// Open an ObjectContainer
// (openFile creates it if it does not exist)
ObjectContainer vendingmachingDB = Db4o.opernFile("vmachine.YAP");

// Create a new Snack object and populate
// its fields.
// Constructor fields are:
//  ID code; Product name; Cost in pennies; Retail in pennies
//    Supplier's Name
snack = new Snack(100,
  "Cheeze Zaps",
  1500,  // Cost is 0.15
  5000,  // Retail is 0.50
  "Sooper Cheeze Inc.");

// Put the snack into the database
vendingmachineDB.set(snack);

// A transaction is automatically started when
//  the ObjectContainer is opened. Before closing,
//  we should commit() the transaction that included
//  the set() operation
vendingmachineDB.commit();
vendingmachineDB.close();
```

Listing 2. Storing a `Snack` object to a db4o database.

So, storing an object is simply a matter of issuing a `set()` method on the `ObjectContainer`, passing to that method a reference to the object to be stored. The `commit()` method ensures that any modifications since the database was opened (or since the last `commit()`) are written to the database. (It also ensures that the operation will not be lost if interrupted by a system failure.)

Notice the simplicity. The programmer need not manipulate the object's members in order to get data into the database. The object is treated like – well, like an object. All the programmer must do is tell db4o: "Please put this object into the database." db4o works out the details invisibly.

Retrieving the snack is just as simple. db4o uses a novel Query By Example (QBE) technique for locating database objects. We supply db4o with a "template" object, which db4o uses to locate our search target. The template object is of the same class as the search target, with elements filled with data that specifies the search criteria..

Assuming that our `vendingmachineDB` `ObjectContainer` has already been opened, the following code retrieves the snack just entered:


```
// Create a template object.
// Retrieve the snack by ID number.
// Other fields are null or 0, and will be
//  ignored by the query
snackTemplate = new Snack(100,null,0,0,null);


// Issue the query
ObjectSet result =
  vendingmachineDB.get(snackTemplate);


// Fetch the retrieved snack
if(result.hasNext())
{  Snack snack = (Snack)result.next();
   // Do something with the snack.
   . . .
}
```

Listing 3. Retrieving a `Snack` object from a db4o database.

Compare this with the example given for the RDBMS. With db4o, the `Snack` object is retrieved wholesale; the programmer need not write a series of assignment statements to populate the object's fields. (And this is true regardless of the number of fields in the fetched object.) Nor is there an SQL command string that need be passed to an `executeQuery()` method for parsing and processing.

Deleting an object is equally straightforward. Once an object has been retrieved from the database, you simply pass its reference to the `ObjectContainer`'s `delete()` method. The code follows:

```
// Create a template object
// This time, query the snack by name
snackTemplate = new Snack(
  0, "Cheeze Zaps", 0, 0, null);

// Issue the query
ObjectSet result =
  vendingmachineDB.get(snackTemplate);

// Get the retrieved object
if(result.hasNext())
```

```
{ Snack snack = (Snack)result.next());
  // Delete the snack
  vendingmachineDB.delete(snack);
}
```

Listing 4. Deleting a `Snack` object from the database.

Again, operations on the object are wholesale, objects are treated as the indivisible entities that they are.

It may have escaped the reader's notice that all of the examples given for db4o have involved more than just a single `Snack` object. We have defined the `Snack` class as:

```
public class Snack {
  private int id;
  private String name;
  private long cost;
  private long retail;
  private String supplier;
  // Constructors and accessors follow
  ...
}
```

Listing 5. The `Snack` class.

The second and last elements of the `Snack` class are String objects. But, Java strings are implemented as objects, not as primitives. So, storing a `Snack` object also stores two String objects, and deleting the `Snack` also deletes the String objects associated with name and supplier. db4o does all this storing and deleting behind the scenes without our having to ask. (As it turns out, db4o treats Strings object and other simple types as second class objects; they have no identities of their own, and are deleted and updated in conjunction with their parent object.)

Nevertheless, db4o easily handles arbitrarily complex object hierarchies. Suppose we move further into the structure of our vending machine's database. We define a class called `SnackSlot` that models one of the numerous slots (or posts) in the vending machine that actually holds snacks. (Each slot can hold multiple instances of a given snack.) The class might look like this:

```
public class SnackSlot {
  int slotnum;      // Slot number
  Snack thisSnack;  // Snack in this slot
  int original;     // Original number stocked
  int current;      // Number of snacks left
  // Constructors and accessors follow
...
}
```

Listing 6. The `SnackSlot` class.

So, a `SnackSlot` holds a reference to the `Snack` object that is dispensed by that slot. We also track the original amount of snacks placed in a given slot, as well as the number of snacks remaining. That allows the machine to determine how many snacks have been sold on a given slot.

Assuming that we were expanding the capabilities of our vending machine, adding a new snack, and a new dispensing slot, one might think that two separate `set()` operations were required. That, however, is not so.

When an object is first added to a database, db4o explores all that object's references, and stores all referenced objects to the database as well -- all automatically. So, the code to simultaneously add a new `SnackSlot` and an associated new `Snack` would be:

```
// Create the Snack object
snack = new Snack(101,
  "Nacho Novas",
  1200,  // Cost is 0.12
  7500,  // Retail is 0.75
  "Sooper Cheeze Inc.");

// Create the SnackSlot object
snackslot = new SnackSlot(
  10,     // Slot number 10
  snack,  // Connect the snack to the slot
  10,     // 10 bags on this slot...
  10);    // ...none sold yet

// Put the new SnackSlot object in the database
//  The Snack is stored as well.
vendingmachineDB.set(snackslot);
```

Listing 7. Adding more complex objects to a database.

Retrieving and deleting `Snack` and `SnackSlot` objects are only slightly more complicated, but only because db4o lets you fine-tune how objects are managed once they are in the database.

To retrieve a compound object, we have to tell db4o how far into the object's reference tree we want db4o to "reach" when the object is fetched (when we call a `get()`). This is referred to as the "activation depth." An activation depth of 0 will retrieve only the root object, So, if we fetch a `SnackSlot` object with an activation depth of 1, db4o will retrieve both `SnackSlot` and `Snack` objects.

Because db4o's default activation depth is set to 5, we can retrieve a `SnackSlot` and its associated `Snack` object using the same code as shown above in **listing 3**. (We will demonstrate working with db4o's activation depth in the next section.)

db4o can also delete all objects referenced by a root object. However, we have to explicitly tell db4o that – when it deletes an object, it should also delete all objects it references – otherwise, it will only delete the root object. So, if we want the `Snack` object deleted when we delete a `SnackSlot` object, we must associate a 'cascaded delete' flag with the `SnackSlot` object. We do this in db4o's 'global configuration object.' The code will look something like this:

```
Configuration config = Db4o.configure();
ObjectClass oc = config.objectClass("<package>.SnackSlot");
oc.cascadeOnDelete(true);
…
```

Listing 8. Setting the cascaded delete flag.

This code tells db4o that whenever a `SnackSlot` object is deleted, then all member objects (in this case, those referenced by the `Snack` object member) are also deleted. With the little fragment of **listing 8** tacked on its front, the code in **listing 4** is sufficient to delete a `SnackSlot` and associated `Snack`.

Obviously, cascaded delete is not appropriate in all cases (which is why db4o makes it optional). In our hypothetical vending machine example, it may well be that a `SnackSlot` is –

for some reason – removed, but the associated `Snack` is also provided by a different slot. In that case, we would not want to enable cascaded delete, because we would want the `Snack` object to remain in the database.

Native Queries

The core of db4o's QBE is more or less equivalent to a “WHERE ... IS EQUALS TO ...” filter within a query. This is adequate for locating the root object in a hierarchy of target objects. Once the root has been fetched into memory, db4o allows you to use ordinary object references to navigate to any child or sibling objects required. That is, navigation throughout an object tree is unperturbed by that object tree's being in a db4o database. The result is a powerful, uncomplicated query mechanism that is entirely sufficient for a wide array of querying needs.

There are times, however, when more sophisticated queries are required; when items must be selected from persistent storage based on criteria other than equivalence. And, you would think that more complicated queries would demand a more complicated query syntax. For many databases systems, that's true – but not with db4o.

db4o's Native Query (NQ) API easily tackles those queries that exceed the capabilities of QBE. And, in keeping with db4o's principle of ease-of-use, Native Queries do not require the developer to master a separate language, nor struggle through a labyrinthine API. In fact, with Native Queries, the developer uses nothing more (nor less) than the language in which the rest of the application is written.

Recall the query we presented in **listing 3**. In that code snippet, we fetched the `Snack` object whose ID equaled 100. That same code, expressed as a Native Query, would look something like this:

```
List<Snack> snacks =
   vendingmachineDB.query<Snack>(new Predicate()
 {
   public boolean match(Snack snack)
   {  return( snack.id == 100); }
 })

if(snacks.hasNext())
{
     Snack snack = snacks.next();
     // Do something with snack
     ...
}
```

Listing 9. A native query equivalent to the example given in listing 3.

Here, we've employed not only db4o's NQ API, but we've also called upon Java's recently-added generic capabilities to make our query completely type-safe. (Generics were added as of JDK 5; but db4o's native queries can be used with Java versions even as “far back” as JDK 1.1.)

Peeling apart the above query code, we can discover the salient features of db4o's Native Query API. First, NQ defines the `Predicate` class, instantiations of which serve as the actual engine of the query. Within this class, there is a single abstract method – `match()` – which extensions of the `Predicate` class must implement. The `match()` method takes a single object argument. This argument identifies the class of targets that will be queried in the database. In the case of **listing 9**, target objects of type `Snack` will participate in the query.

The `match()` method returns a `boolean`, and it's easy to see from this that the fundamental purpose of `match()` is to filter the target objects. Put simply, `match()` returns true if an object satisfies the search criteria; false otherwise. So, when the query is executed, it passes to the `match()` method each `Snack` object in the database, and uses the returned `boolean` to determine whether a given object should be placed in the returned `List` collection.

Native Queries are appealing for a number of reasons, not the least of which is the fact that the language of the application is the query language. Suppose, for example, we wanted to determine which snacks are selling particularly well (say, 5 or more items).

Using the NQ API, this is trivial:

```
// Fetch list of snacks that have sold
//  5 items or more.
// The list is returned in
//  popularSnacks ArrayList

List<SnackSlot> slots =
   vendingmachineDB.query<SnackSlot>(new Predicate()
{
   public boolean match(SnackSlot snackslot)
   { if(snackslot.original == 0)
       return false;
     return((snackslot.original -
       snackslot.current) > 4);
   }
}
while(slots.hasNext())
{
     SnackSlot goodSlot = slots.next();

     // Read the snack object in, too
     vendingmachineDB.activate(goodSlot,2);
     popularSnacks.add(goodSlot.thisSnack);
}
... process popularSnacks...
```

Listing 10. A more complex query with db4o's NQ

Using a native query, we can perform mathematics on the candidate objects' fields and determine which `SnackSlots` have sold 5 or more items. In the `while` loop, we examine the matches, and use db4o's `activate()` method to read the `Snack` objects into memory. (As we hinted at earlier, the `activate()` method tells db4o to fetch an object's members into memory from the database, up to a specified "depth". So, a depth of 2 ensures that, not only is the `SnackSlot` object in memory, but the `Snack` object is as well.)

After this query has executed, the application can process the `popularSnacks` `ArrayList`, perhaps alerting the vending machine's owner to which snacks will need to be re-stocked in the near future.

Of course, we could have made the comparison portion of the query arbitrarily complex. We could have used `activate()` to fetch other member objects into memory. The beauty of NQ is that we can use whatever Java code is necessary to implement the query. (There are some minor restrictions, all of which have to do with the avoidance of unwanted side-effects. You should read the documentation on the db4o website for more information.)

Best of all, our querying is entirely type-safe, and completely re-factorable. If we had used a string-based query language like SQL, any mistakes we made in an object's field references

(typos, incorrect field names, etc.) would not be caught until runtime. With Native Queries, the compiler keeps us in line. In addition, if – as we continued to develop our application – we had changed the name of a class, class member, field, etc., our integrated development environment's re-factoring capabilities would be able to safely make the name changes wholesale. With a relational, SQL-based system, we would be forced to manually locate and make changes to the query strings.

A Surface Scratched

We have only scratched the surface of db4o's features. While we have only shown a small subset of db4o's capabilities, we have at least demonstrated that it satisfies the requirements for a small-footprint, reliable, responsive database. Furthermore, we've shown that the impedance mismatch problems of RDBMS and ORDBMS solutions in an object-oriented environment are completely absent when db4o is applied. And, db4o's Native Query capabilities give us arbitrarily complex queries that are easy to create, easy to maintain, and sidestep many of the development pitfalls that programmers tumble into when working with RDBMS queries.

Still, there is much we have not shown:

- **Changes in Object Versions.** For example, we have not shown how easily db4o deals with changes in object versions. Suppose we want to modify the structure of our `Snack` class, adding a `snackType` member that permits us to distinguish between, say, candy-bars and chips. We can make that change with a db4o database already created and populated with "old" `Snack` objects. And we can do that with no disruption in the use of the database. db4o allows us – with a single method call – to rename a class already stored in a database. (So, the old `Snack` objects could be renamed to `OldSnack` objects.) By contrast, a solution employing a relational database backend would require wholesale changes to the database tables, as well as alterations to the querying code.

- **Easy Object Replication and Synchronization.** Software running on intelligent, but intermittently-connected devices (e.g., handheld organizers or scanners) must possess the means to accept a "subset" of persistent objects from a master database, allow the user application to work with that derived database (disconnected from the master), and then re-connect to the master database and synchronize the changes made. db4o has such a capability built right into the database, called the db4o's Replication System (dRS)[2].

- **Zero Administration.** In addition, we have not said much about db4o's administration facilities – though for good reason: db4o requires virtually no administration, and that pretty much says it all. Hence, it is ideal for handheld mobile device applications, information appliances, intelligent medical systems, and all applications where the database is invisible to the user.

- **Available with open sources and free under the GPL.** We have also left for last perhaps the most attractive of db4o's qualities: both Java and .NET versions are open-source. And if you want to include db4o in your next commercial application, the low cost of ownership puts db4o in a class by itself.

But all these attributes would be nothing if not for db4o's essence: it is a lean, powerful, no-nonsense object-oriented database for Java and .NET.

[2] More information on the db4o Replication System: www.db4o.com/about/productinformation/features/drs.aspx