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

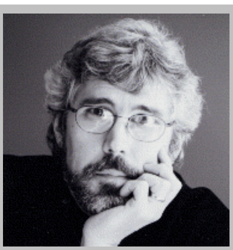
Complex Object Structures, Persistence, and db4o

By Rick Grehan

One of the strengths of object-oriented programming -- its core strength, in fact -- is the ability it gives developers to model data relationships in a natural way. That doesn't mean that the modeling process is easy. The architecture of many real-world systems can be complicated trees of sub-systems and more sub-systems -- all interacting with other systems equally complex. The subsequent class architecture for such systems can be dizzyingly intricate, and tricky to program.

It gets even trickier when you add persistence to the mix. It's one thing to code the data and methods for complex, interconnected, multi-class systems; it's quite another to store the resulting object structures in a database in such a way that those objects can be swiftly and easily fetched, searched, updated, and deleted ... and all the while their relationships preserved. We're going to explore that trickiness; we'll look at some of difficulties encountered when large or complicated object structures are persisted (a fancy way of saying: 'written to a database.') To do this, we'll use several examples that, though hypothetical, are nonetheless representative of real-world programming situations:

- flat object structures
- tall object trees
- evolving object structures



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.

We'll show how those difficulties are mitigated by a powerful object-oriented database, db4o. You'll see how db4o allows you to focus on creating the optimal class and object structures for the application at hand. You don't have to worry whether those structures might thwart - or be thwarted by - the operations of the underlying database. In other words, db4o provides persistence while placing a minimal burden on the developer. No special coding; no re-arrangements of object relationships to suit the database. Just store the objects you need to store; fetch the objects you need to fetch, and delete the objects you need to delete.

db4o does the rest.

Table 1. While an RDBMS does well with a flat object structure, and non-native OODBMSs add to that the ability to easily handle complex object structures, only db4o tackles both AND adds the ability to deal nimbly with object structures that change over time.

ability to easily store...	RDBMS	non-native OODBMS	db4o
flat object structures	X	X	X
tall object trees		X	X
evolving object structures			X

What is db4o?

The OO database library db4o (database for objects), from db4objects, is a pure object oriented database that is native in Java and native in .NET languages. Its noteworthy advantages are its speed, its simplicity, and its small memory footprint. db4o can persist “ordinary objects.” Non-native OO databases must instrument executable code to provide persistence. Others require you to build separate “object schema” files. But, with db4o, you need adopt no unusual (or additional) programming practices to place objects in the database, or to manipulate them once they’re there.

The examples we will show are written in Java. Keep in mind that, as we said above, a version of db4o is also available for .NET. Consequently, the concepts presented by the examples are as applicable to .NET as to Java.[1]

Flat Object Structures

A simple object structure that nevertheless contains many objects is a “flat” structure. It might include a large array, or perhaps a linked list of some sort. As part of working out how to persist such a structure, you need to determine first whether the array (or list) must be in memory at all times. (Here, we are assuming very LARGE lists, that might involve thousands or millions of entries.) That determination will, in turn, require determining the potential size of the list, and considering whether the list will fit in available memory.

If the array can fit, then perhaps the entire thing can be persisted as a single unit. Otherwise, its contents will have to be read from or written to the database in some piecemeal fashion.

Example 1: Monitor Web Application Transaction

An organization provides services to its customers via a web-based application. Client sites connect to this application, and post transactions to it (reading, updating, modifying, etc.). The organization would like to monitor – at arbitrary times and for varying intervals – the quantity and turnaround times of requests made to this web application.

So, the organization's developers create a monitoring system that consists of a “stub” application attached to the server, and a separate monitoring console. The stub is hooked into the web application, and sends transaction tracking data to the console (via some sort of remote IPC mechanism). A user running the console program can initiate a “session.” During a session, the stub monitors the type, arrival, and departure of client requests and responses. That information is transmitted back to the console, which receives, organizes, and stores the information in a database.

[1] In db4o, the database is referred to as an `ObjectContainer`. An `ObjectContainer` is associated with a specific disk file. Opening an `ObjectContainer` is a simple API call:

```
    ObjectContainer db = Db4o.openFile(“<path>”);
```

where `<path>` is the path to the database file. When all operations on the database are completed, you call:

```
    db.close();
```

For the remainder of this paper, we will assume that an `ObjectContainer` object named db has been opened.

Later, administrators can query the database, examine trends, and generate reports and graphs that explore such information as the number of transactions received at different times of the day, the average turnaround time for a transaction plotted against the quantity of transactions per second, and so on. The organization can use this information to determine whether the server is scaling well under client transaction loads.

Implementation

Developers of this hypothetical application modeled a session using two classes. One, a `Session` class, includes global session information: a unique name, the date and time the session was started, and the date and time the session was stopped. Also included in the `Session` object is a `List` of the captured transaction information.

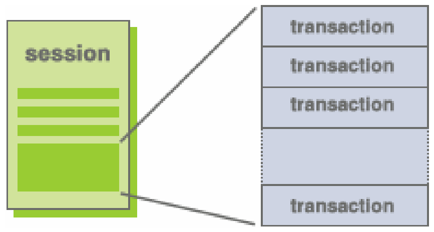


Figure 1. A simple object structure for modeling a session. The `Session` object includes an `ArrayList` of Transaction objects.

The `Session` class looks like this:

```
// Session
public class Session
{ private String sessionName; // Unique name for the session
  private timestamp start;    // Start of reading
  private timestamp finish;   // End of reading
  private List transactions;  // Transactions
  // Constructor
  Session (String sessionName,
  timestamp start)
  {
    this.sessionName = sessionName;
    this.start = start;
    this.transactions = new ArrayList();
  }
  // Add a transaction
  public void addTransaction(Transaction trans)
  {
    transactions.add(trans);
  }
      ... other methods for Session ...
}
```

And the `Transaction` class models transaction data “packets” delivered by the stub:

```
// Transaction objects
public class Transaction
{ private int type;          // Transaction type
  private int quantity;      // # of bytes of data exchanged
  private timestamp start;   // Request arrival
  private timestamp finish; // Response transmission
      ... remainder of Transaction class ...
}
```
[2]

[2] Because we are interested primarily in how to persist an object's data members, and less so with the class' methods behavior, the above classes are abbreviated. Also, these classes presume the availability of a `timestamp`

When a session is begun, a `Session` object is instantiated, and given a unique name (`sessionName`). The console then “tells” the stub to begin collecting transaction data. As each packet of data arrives from the stub, the console application instantiates a new Transaction object, copies the data into object, and stores that object into the session's transaction `ArrayList`:

```
theSession.addTransaction(theTransaction);
```
[3]

When the session has concluded, saving it into the db4o database could not be simpler, and here we make our first acquaintance with db4o's built-in intelligence that makes a programmer's task so much simpler than it would be otherwise. It requires but a single line of code to store the entire object – `List` and all:

```
db.set(theSession);
```

db4o silently traverses the object tree, seeking out and storing not only the `theSession` object, but all members of the transaction `ArrayList` as well.

Retrieving the session object is equally simple. db4o provides a straightforward query capability based on the “query-by-example” paradigm. To query a db4o database, you build a 'prototype' object of the class you wish to search. Load the appropriate fields with those values that you want to match, and call the `ObjectContainer`'s `get()` method.

Suppose, for example, you want to retrieve a session whose name was "`FridayCapture`:”

```
Session proto = new Session("FridayCapture",
  (timestamp)0);
ObjectSet result = db.get(proto);
```
[4]

The result of a query is an `ObjectSet`, through which you can iterate to fetch the matched objects. Because the above query returns a single object, you can retrieve it directly from the set with:

```
Session theSession = (Session)result.next();
db.activate(theSession, 2);
```

That second line instructs db4o to 'activate' `theSession` to a depth of 2. When db4o activates an object, it loads the objects fields from the database. So, in the above example, an activation depth of 2 causes db4o to retrieve not only is `theSession` object, but it's "children" (the content of the `ArrayList`) as well. Specifying an activation depth of 3 would retrieve children and grand-children; 4 retrieves children, grand-children, and great-grandchildren; and so on.[5]

---

datatype – possibly `java.sql.timestamp` – as well as a set of `int` constants defined to enumerate the transaction `type`.

[3] Assuming Session object `theSession` and Transaction object `theTransaction`

[4] Empty or zero fields do not participate in the query.

[5] db4o lets you specify the activation depth for all the objects in an entire database, if you wish, rather than on a per-object basis – as shown above. We chose the explicit, per-object option in order to explain the concept of activation.

It is just that simple. Whole object structures are persisted with a single call. And whole object structures can be retrieved with a simple, easy-to-grasp query API.

This brings us to an important principle regarding db4o:

db4o lets you work with object structures almost as though they were “in memory” structures. Little additional coding is required to manage object persistence.

Variation on a Theme

The preceding example was straightforward and functional, but it may not be ideal. Because a session's entire data resides in memory, it places an upper limit on the size of a session – that limit being, of course, the amount of available memory. (It may be an entirely realistic upper limit, but we will pretend that it is not for the purposes of our discussion.)

In →Appendix A – How db4o Deals with Memory Leaks we’ll show that when we solve a memory leak problem – one that would have existed regardless of which back-end persistence engine used – db4o had not been an impediment; it has fitted itself smoothly into the problem’s solution. Its simple query mechanism, in fact, made the solution much easier.

Deleting

Before we leave this section, we should cover a database operation that we have not yet covered. This will permit us to illustrate another of db4o's powerful capabilities.

Suppose we need to delete an entire session's worth of data. Deleting a single object is simple enough. Provided that we've fetched a persistent Session object into `theSession`, we can delete the object from the database with:

```
db.delete(theSession);
```

But, deleting an arbitrarily long linked list would be a tedious loop. Because deleting a persistent object requires that we first fetch the object, the loop would look like:

1) Fetch the head Transaction object
2) Save its next link
3) Delete the Transaction
4) Fetch the Transaction referenced by next
5) Return to (2) and repeat until next is null.

It would work, but lacks elegance to a degree that makes it almost embarrassing to implement.

Luckily, db4o saves our dignity with its "cascaded delete" feature. All we need do is tell db4o that we want cascaded delete activated for a particular class before we open the database. For this example, the code would look like:

```
Configuration config = Db4o.configure();
ObjectClass sessionObjectClass =
  config.objectClass("<package>.Session");
sessionObjectClass.cascadeOnDelete(true);
```
[6]

Once our application has executed the above code, opened the database, and fetched from the database a Session we wish to delete, we can delete the entire session – `theSession` and the complete linked list of `Transaction` objects – with a single call:

```
db.delete(theSession);
```

db4o will traverse the entire object tree rooted in `theSession`, deleting all child objects, grandchild objects, great-grandchild objects, and so on. In this case, db4o will march down the linked list, deleting the entire chain of Transaction objects, at the push of a single button – as it were.[7]

Compare to a Relational Database

Pause, for a moment, look over the code examples in this section, and consider what we would have to have written, were we using a relational database. Somewhere, there would have been a schema definition for the table structures. We would have created two tables: one for session data, the other for transaction data. Columns in each table would be chosen as key fields, and a one-to-many relationship defined (in the schema), linking a session record with multiple transaction records. (It's likely that we would have had to put a column in the transaction table that would have existed for no other reason than to provide the relationship link between session and transaction.) There would have been SQL INSERT, UPDATE, and DELETE commands – and assignment operations to move data to and from object members and bound variables in the SQL statements. Fetching an entire session would have involved a join. And so on.

In short, a sizeable quantity of code, needed to move data back-and-forth across the invisible boundary that divides an object-oriented program and a relational database backend …

… is, thanks to db4o, not there.

[6] Where `<package>` is the Session class' package.

[7] Cascaded delete can significantly simplify your programming. Cascaded delete can also wreak havoc if you're not careful. As with any powerful capability, it must be handled gingerly.

Was That Necessary?

At this point, some readers may be suspecting that we have created a mountain out of a molehill. The Transaction data is arriving in sequential fashion, and it is being stored in sequential fashion. Why not simply create a sequential file, and write transaction data to that file as it comes in? Why the fuss over a database?

In some circumstances, that would certainly be the best route. But, if you are creating a repository that will be mined for information in the future, using a database provides you with out-of-the-box searching capabilities that you would otherwise have to create yourself, were the data stored in a sequential file.

Using our example from this section, suppose you wanted to discover the average number of bytes transferred by a particular kind of transaction ... for all sessions. Had you used a set of flat sequential files, you would have to write code to search through all those files for the matching transaction type to gather the information.

Meanwhile, db4o would let you find those transactions with a simple query. Assume that the transaction type is identified by the constant `READ_TRANSACTION`, and the code would look something like this:

```
Transaction theTrans;
double totalbytes = (double)0.0;
int totaltrans = 0;
double average;
      ...
Transaction proto = new Transaction();
proto.setType(READ_TRANSACTION);
ObjectSet result = db.get(proto);
while(result.hasNext())
{
  theTrans = (Transaction)result.next();
  totalbytes + (double)theTrans.getQuantity();
  totaltrans++;
}
if(totaltrans!=0)
  average = totalbytes/(double)totaltrans;
      ...
```

We can do this because `Transaction` objects are being stored as, well, objects. They are therefore accessible to us – independent of their `Session` objects – through db4o's querying mechanism.

This is certainly simple to code, simpler than would be necessary were we to have to write code to search through all the lists for all the sessions stored. (And note that the above code will work, even as new sessions are added to the database.)

Tall Object Trees

One of the beauties of object-oriented programming is that it allows you to model complex and elaborate real-world relationships in a way that is more natural than would be necessary were you still forced to use structured programming techniques (filled to the brim with error-prone memory pointers). And some of the more elaborate structures that programmers model are expressed as one form of "tree" structure or another.

When we speak of "tall object trees," we are considering object structures that differ from the preceding in regards to dimension. The previous example was "shallow, but wide." We are now concerned with configurations that are "deep" and "multi-branched." That is, a root object references multiple child objects, each child object references its own multiplicity of children, and so on to an unknown depth.

Example 2: Material Database for Electrical Generators

A company products a line of electrical generators, and would like to create a bill-of-materials database. In such a database, each end product "points to" a collection of assemblies and sub-assemblies. They, in turn, point to further sub-assemblies. In addition, assemblies and sub-assemblies point to lists of component parts. In effect, the entire structure of a product is mirrored in the structure of assemblies, sub-assemblies, and component parts, so that completely traversing an object tree reveals all the pieces that go together to make a product.

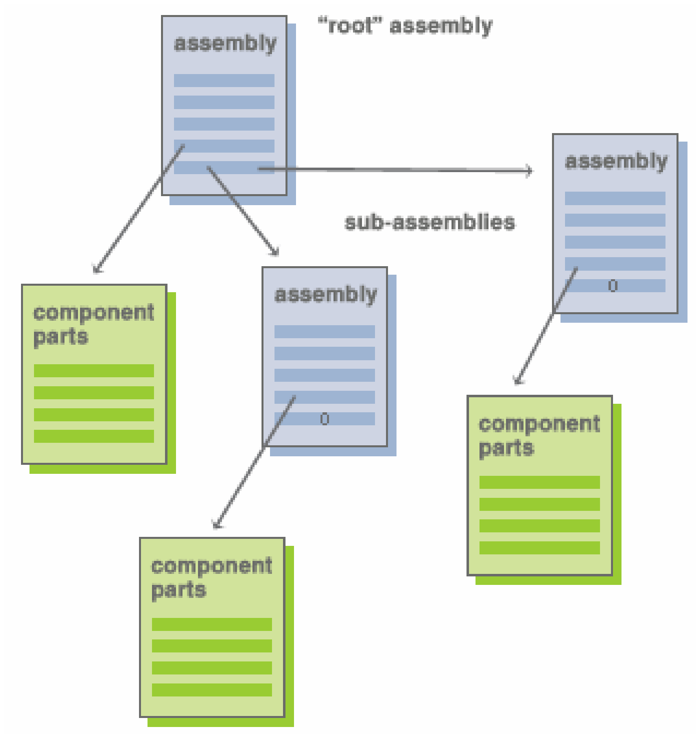


Figure 2. An object structure for capturing the components of a finished product. The "root" Assembly consists of a set of component parts, as well as one or more sub-assemblies. Sub-assemblies are themselves Assembly objects, referencing component part collections and (possibly) further subassemblies.

This is a structure especially suited to an object-oriented database. A "product" in the database is naturally represented as a tree. So, the developers for the organization define a triad of classes: `Assembly`, `ComponentParts`, and `RawPart`.

An Assembly consists of an arbitrary number of subAssemblies and an arbitrary number of componentParts (hence, subAssemblies and componentParts are modeled as arrays). A component part represents an actual, physical inventory item such as a screw, a bolt, a fastener, and so on. (As you'll see, each entry of componentParts also carries a quantity. So, if an assembly includes as a component part, 15 #2 screws, that appears as a single entry in the componentParts array – not 15 entries.) In addition, the structure is recursive: a subassembly is itself Assembly (nested down), also composed of componentParts and (possibly) more subAssemblies.

Each entry in the componentParts array references a RawPart item, which carries the information required to identify a specific, material item: the company (or companies) from which the part can be obtained, the part's SKU, and its cost.

The description of a product, then, is a “tree”. The “root” (at the top) is the product itself. As you follow the tree down into its branches, you ultimately arrive at the "leaves" – sub assemblies composed entirely of componentParts.

```
public class Assembly
{
  public String name;     // Name or description
  public Assembly subAssemblies[];
  public ComponentParts componentParts[];
      ... remainder Assembly class ...
}
public class ComponentParts
{
  public int productID;   // RawPart product ID
  public int quantity;
      ...remainder of ComponentParts class...
}
public class RawPart
{
  public int productID;   // RawPart product ID
  public string SKU;      // Raw part's SKU
  public List suppliers;  // List of suppliers
  public BigDecimal cost; // Cost
      ...remainder of RawPart class...
}
```

At this point, if you’re looking for a complex piece of code to add a new item with all its children to the database, you won’t find it. db4o can handle this structure as easily as it handled the objects in the preceding section. So, given that you’ve created an Assembly object called theAssembly, and have created and linked together all its subassemblies and componentParts, you can put theAssembly into the database with:

```
db.set(theAssembly);
```

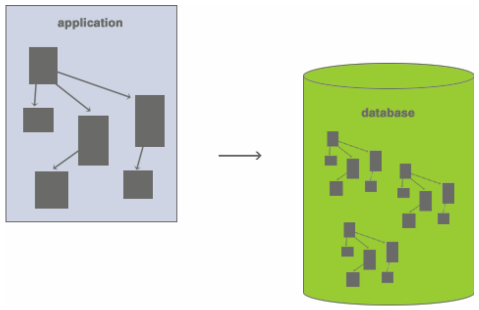


Figure 3. With db4o, complex object structures are stored in the database as easily as simple structures. A single call does the trick.

And, as before, if you've turned on cascaded delete and need to delete an Assembly object along with all its subassemblies and `componentParts`, you need only call:

```
db.delete(theAssembly);
```

Notice, however, that we have borrowed the notion of foreign key from the database world in order to create a relationship between a `ComponentParts` object and a `RawPart` object. The `productID` field serves to link the two together. So, if we want to fetch, say, the cost of a particular raw inventory part, we use something like this:

```
int prodID = theAssembly.componentParts[<index>];
RawPart rawPartProto = new RawPart; // Build prototype for query
rawPartProto.productID = prodID;    // Field to match
ObjectSet result = db.get(rawPartProto);
theRawPart = (RawPart) result.next();
```

Because the relationship between a `ComponentParts` and a `RawPart` object is not an object reference, this allows us to keep the raw parts inventory component of the database untouched whenever we cascade-delete an Assembly item.

We return to the principle we stated earlier:

db4o lets you work with object structures
almost as though they were "in memory" structures.
Little additional coding is required to manage object persistence.

As you've seen, manipulating a "deep" tree structure is as simple as manipulating a "flat" array or list structure. So, we add another principle:

As the persistence requirements become more complex,
db4o's unique design takes care of the added complexity
so you can continue to work
as though that complexity never happened.

Evolving Object Structures

“I don't make things complicated. That's just how things get all on their own.”

Martin Riggs, Lethal Weapon

The underlying theme of that famous quote is: like it or not, things evolve, and they tend to do so in the direction of increased complexity. Programmers who have struggled with projects that time and evolving requirements have modified, morphed, and mutated will surely nod their heads in understanding. When data structure is changed, the necessary code changes that follow can be tedious and error-prone, particularly if a database is filled with “old” data that must be made “new.”

But, with db4o, change is easy. Let’s go back to the first example.

Example 1 Revisited: Monitor Web Application Transaction

Over time, the organization has grown, and it now runs application servers from various locations. The company's developers have been asked to expand the monitoring system to collect data from multiple servers simultaneously.

So, they alter the class definitions accordingly:

```
// Session objects
public class Session
{ private String sessionName; // Unique name for the session
  private List hosts;       // List of hosts involved in the reading
  private timestamp start;  // Start of reading
  private timestamp finish; // End of reading
  private Transaction head; // Reference to head of list
  private int length;       // Number of items in the list
      ... remainder of Session class ...
}
// Transaction objects
public class Transaction
{ private String host;      // Origin of transaction
  private int type;         // Transaction type
  private int quantity;     // # of bytes of data exchanged
  private timestamp start;
  private timestamp finish;
  private Transaction next; // Reference to next transaction
      ... remainder of Transaction class ...
}
```

In these new versions, Session objects carry a list of the server hosts from which data is captured. In addition, each Transaction also includes a host identifier. This permits a single session to receive data from an arbitrary number of server hosts. [8]

[8] We chose to use a string to identify host systems within each transaction. This will make our example easier to illustrate. In the real world, we would be more frugal with storage space; possibly using a single byte identifier instead, and associating each host in our server network with an ID between 1 and 255.

Now, when a session is started, the names of the hosts to be monitored are stored in the hosts List. And, as transactions arrive, each is tagged with its origin before being appended to the linked list and written to the database. What we showed of the application in the first example – the mechanics of persisting the data, how the link-list can be written to the database – can remain largely unchanged.

Changed Schema

But, wait. Something has changed: the class structure. What happens to the data that's already been written to the database? Have we lost all the information gathered when the organization ran a single application server? Suppose the application starts writing “new” Session and Transaction objects into the original database: will the application crash when "old" data is read into objects instantiated from the new classes? Will we have to create a new database, and write an application to copy-and-convert old data to new data?

Happily, no. db4o assimilates evolving classes without missing a beat. We need only add a new method to each of the class definitions. To the Session class, we add this:

```
public void objectOnActivate(ObjectContainer container)
{
  if(hosts == null)
  { hosts = new ArrayList();
    hosts.add("MainServer");
  }
}
```

And to the Transaction class, we add this:

```
public void objectOnActivate(ObjectContainer container)
{
  if(host == null)
    host = "MainServer";
}
```

That’s it. Dust your hands off. The conversion is done.

When db4o activates an object, that activation requires db4o to instantiate an object from a class definition. And when db4o performs that instantiation, it checks the class for the presence of an `objectOnActivate()` method with the signature as shown in the two methods above. If db4o finds such a method, it calls that method at the end of the activation process. Consequently, we can use that method to examine the content of the object instantiated, and populate empty or null object members as required to “fix” the object before it gets handed to the application.

So, we can invisibly make "old" objects look like "new" objects. The application code is never aware – need never be aware – that the change is taking place. To the application, all objects fetched from the database will be new ones. In fact, once the application code sees them, old objects are new objects. And if we were to write them back out to the database, the new version would overwrite the old.

In the above examples, we presume that the original application server is called "MainServer." So, all the old `Session` objects and `Transaction` objects will look to the application as though they were captured from a single source: "MainServer."

Finally, notice that db4o has permitted us to absorb the changed object structure – object schema, really – without polluting the application code. The alteration required just two small methods tucked away in the class definitions, called only when needed, and called by db4o without our having to intervene.

Compare, Again, to a Relational Database

Now, pause again and consider what we would have had to do, had the data been in a relational database system. We would have had to either:

1) Create a separate set of tables; the first set holding the old data, and the second set holding the new data. We would have had to write painfully unsatisfying routines to manage the simultaneous use of new and “legacy” data, or …
2) Create a new database, define the new schemas, write an application to convert old data to new, and hope that future events don’t require us to change the object structure again.

In fact, not even a non-native OODBMS could handle a changed object structure with db4o’s agility. Many would become confused by the mixture of “old” and “new” objects. Some sort of conversion application would be almost mandatory.

We return to the second principle presented above:

As the persistence requirements become more complex,
db4o’s unique design takes care of the added complexity
so you can continue to work
as though that complexity never happened.

Indeed, we are tempted to say that db4o has gone beyond even that. Even though object structures might change “right under our feet,” db4o manages the shifting situation so that we never really lose our balance. db4o lets us mange the change tidily, using an API that produces easy-to-maintain code, thereby allowing us to proceed to worthwhile tasks, rather than dribble away our time writing “cleanup” code.

A Customer Case - 10% Faster to Market

We have focused primarily on db4o's simplicity, which allows developers to manage even the most complicated persistent object arrangements easily. But, how does this translate into "time-to-market" benefits for new products?

The BOSCH Packaging Technology Group is No. 1 in the worldwide market for packaging technology. The engineers at BOSCH Sigpack Systems AG saw "time to market" as their primary objective. And that's why they chose db4o as the underlying persistence engine for the application that controls their Delta XR31 high-speed, reliable "pick-and-place" robotic packaging systems.

That application models the robotic system using 3,000 robot objects, each connected to an array of feeder objects. Each feeder object is, in turn, linked to an array of sensor objects. All together, there are 39,000 objects being managed between memory and the database simultaneously. Whenever the robot "assembly line" is to be set up to assemble a particular product, the robots must be configured – which is another way of saying that all those objects have to be configured, quickly and accurately. Any time that the assembly line sits idle is time wasted.

Sebastian Hubrich, lead engineer with BOSCH Sigpack Systems AG, quantified, how db4o's ability to handle complex object structures with ease translated into faster "time to market":

"The use of db4o on the data-backend has helped us to achieve a time-saving effect of at least 10% on each project. This project is highly demanding and had a tough time schedule. db4o was the ideal choice on both counts."

Read the whole story: http://www.db4o.com/about/customers/boschsigpack.aspx

In Sum

We hope that, in this whitepaper, we have shown that managing object persistence is – thanks to db4o – pretty simple. And that it is pretty simple regardless of the complexity of the structure of the data being persisted.

We hope as well that we have adequately illustrated that db4o is undismayed by object complexity in space as well as in time; that is, not only can db4o manage "flat" and "deep" object structures, but that it can handle object structures that change, too. In addition, we hope that we have shown that, if you must use a database in your next Java (or .NET) application, you do not need to expend time educating yourself on the nuances of technologies such as JDBC, ODBC, SQL, OQL, etc. You have already invested time in learning the details of object-oriented programming; db4o optimizes that investment.

Not that we mean to suggest that these database technologies are in any way inadequate or mediocre. But, as we've shown, with db4o it takes only one line of code to do what an RDBMS will do in several lines. And non-native OODBMSs demand additional code instrumentation steps or "extra-curricular" materials (such as schema files) that are blessedly absent from db4o.

Simply put, db4o uncomplicates even the most complex persistent object tasks.

Appendix A – How db4o Deals with Memory Leaks

The example “Monitor web application transaction” in our paragraph on “Flat Object Structures” was straightforward and functional, but it may not be ideal. Because a session's entire data resides in memory, it places an upper limit on the size of a session – that limit being, of course, the amount of available memory. (It may be an entirely realistic upper limit, but we will pretend that it is not for the purposes of our discussion.)

Suppose the developers have determined that the application must allow for arbitrarily large sessions. Consequently, they alter the classes so that transaction data is maintained on a singly-linked list:

```
public class Session
{ private String sessionName; // Unique name for the reading
  private timestamp start;    // Start of reading
  private timestamp finish;   // End of reading
  private Transaction head;   // Reference to head of list
  private int length;         // Number of items in the list
      ...
  public void incrementLength()
  { length++; }

  public int getLength()
  { return(length); }

  public void setFinish(timestamp theTime)
  { finish = theTime; }

  public void setHead(Transaction theHead)
  { head = theHead; }
      ... remainder of Session class ...
}


public class Transaction
{ private int type;         // Transaction type
  private int quantity;     // # of bytes of data exchanged
  private timestamp start;  // Request arrival
  private timestamp finish; // Response transmission
  private Transaction next; // Reference to next transaction
      ... remainder of Transaction class ...
}
```

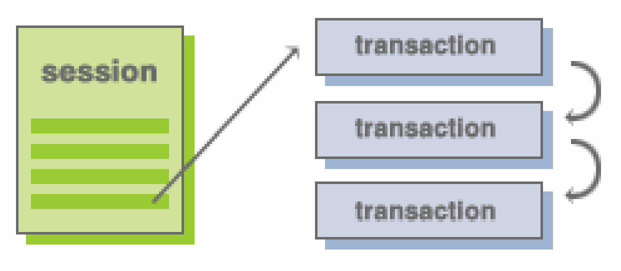


Figure 4. A more elaborate structure. Now, Session objects carry a reference to a singly-linked list of Transaction objects.

The goal, of course, is to put transactions into the database in such a way that less memory is consumed. So, as transaction data arrives, it is attached to the list – each Transaction's next pointer references subsequent Transaction objects, the last being set to null to indicate the list's terminal object – and written into the database. When the session has finished, the final timestamp is written into the Session object, and the Session object is stored in the database. Significant code snippets for this follow:

```
// Globals
Transaction tail;
Transaction newTransaction;
Session theSession;
      ...
// This code initializes the session
theSession = new Session("<Session Name>",
new timestamp(System.currentTimeMillis()));
tail = null;
      ...
// This code is executed when new transaction data arrives
newTransaction = new Transaction(<fill in from incoming data>);
if (theSession.getLength() == 0)
  Session.setHead(newTransaction);
else
{
  tail.next = newTransaction;
  db.set(tail); // Persist the Transaction object
}
tail=newTransaction;
theSession.incrementLength();
      ...
// This code is executed when the session is finished
theSession.setFinish(new timestamp(System.currentTimeMillis());
if(tail != null) db.set(tail); // Persist last Transaction
db.set(theSession);
      ...
```

Happy with the new code, the developers install it and execute it and ...

... discover that it does not work. All the data is still entirely in memory.

Observant programmers will have noticed that this is a classic Java memory leak. Because `theSession` object holds a reference to the head of the list, appending new objects simply lengthens the chain in memory, even though members of the list are persisted. As long as the connection to the head is in memory, the entire list stays in memory.

Fortunately, there's an easy fix that not only solves the leak, but gives you the ability to fine-tune the 'pacing' at which items are written to disk. The fix entails:

1) Writing the Session object to disk and removing any references to it, thus "cutting off the head."
2) Caching incoming transaction objects as they arrive, so that write to the database occurs in "chunks" (when the cache fills up).
3) Reading the Session object back in at the end to fix up the length and finish fields.

Here is the fixed code:

```
// Globals
// Cache
Transaction transactionCache[];
int transactionCacheNext;
int numTransactions;
boolean sessionWritten; // True if the session has been written
Session theSession;
String theSessionName;
      ...
// Initialization
// This code executes at the very start of a Session,
// before any data has arrived.
// (It assumes that theSession has already been
// created and its description and start fields initialized.)
sessionWritten=false;
numTransactions=0;
theSessionName = theSession.getSessionName();
dbTransCacheInit(CACHE_SIZE);
...
// This method initializes the cache to size entries
public void dbTransCacheInit(int size)
{ transactionCache = new Transaction[size];
  transactionCacheNext = 0;
}
      ...
// This method is a "cached set()". Whenever a transaction
// comes in, the console application calls this routine to
// write the transaction to the database (in place of a .set()).
// Transactions are first placed in the cache. When the
// cache fills, its contents are written to disk (see the
// following method dbCacheFlush())
// Note that the first time the cache is written, the Session
// object is also written. This cuts off the head of the list,
// so the entire list is not kept in memory.
public void dbCachedSet(Transaction obj)
{
  // If the cache is full flush it
  if((transactionCacheNext+1)==transactionCache.length)
    dbCacheFlush();
  // Put the new object in the cache
  transactionCache[transactionCachNext++]=obj;
}
public void dbCacheFlush()
{
  // If the session object has not yet been written,
  // write it
  if(sessionWritten==false)
  { db.set(theSession);
    theSession=null;  // Cut off the head
    sessionWritten=true;
  }
  // Flush whatever is still in the cache
  if(transactionCacheNext>0)
  for(int i=0; i<transactionCacheNext; i++)
  { db.set(transactionCache[i];
    numTransactions++;
    transactionCache[i]=null;
  }
  transactionCacheNext=0;
}
      ...
// This code is executed when the Session has
// completed.
// Set finish time
finishTime = new timestamp(currentTimeMillis());
// Flush the cache one last time
dbCacheFlush();
// Fetch the session object from the database to
// fix it up.
Session proto = new Session(theSessionName,0);
ObjectSet result = db.get(proto);
theSession = (Session) result.next();
theSession.setFinish(finishTime);
theSession.setLength(numTransactions);
// Write the updated session
db.set(theSession);
      ...
```

The new code provides a linked list structure that is persistent, and that can grow to an arbitrarily large size. In addition, the size of the cache is tunable, so developers can adjust memory consumption as needed.

Notice what we've demonstrated here. We have solved a memory leak problem – one that would have existed regardless of which back-end persistence engine is used – with a minimum of effort. Once again, db4o has not been an impediment; it has fitted itself smoothly into the problem's solution. Its simple query mechanisms, in fact, make the solution that much easier.