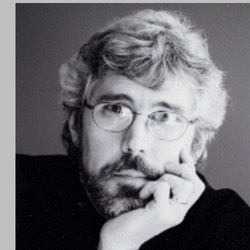
db4o | Die Open Source Objektdatenbank | Java und .NET

The Database Behind the Brains

von Rick Grehan



Rick Grehan ist QA Engineer bei Compuware/Numega labs, wo er an Java und .NET Projekten gearbeitet hat.

Er ist auch mitwirkender Autor beim InfoWorld Magazin. Seine Arbeit wurde in Embedded Systems Programming, EDN, The Microprocessor Report und Computer Design veröffentlicht. Bevor er zu Compuware kam, war Rick am Discover DSP Projekt bei Metrowerks, Inc. beteiligt.

Früher war Rick Senior Editor beim BYTE Magazin, wo er Lab Director war und BYTEs JavaTalk Kolumne geschrieben hat.

Mobile Devices, Informationsgeräte, intelligente Steuerungssysteme; Es ist kaum zu übersehen, dass die Einsatzgebiete von Computeranwendungen immer breiter werden. Diese Systeme werden üblicherweise als „eingebettete Systeme" bezeichnet, aber dieser Ausdruck berücksichtigt nicht die wachsende Finesse derartiger Anwendungen.

Taktraten von Prozessoren steigen, Speicherdichte wächst, Festplatten schrumpfen, Preise fallen, und die Aufgaben, für die jene Systeme entwickelt wurden, werden immer komplexer. Eingebetteten Systemen wird mehr als je zuvor abverlangt – „schlauer" als je zuvor zu sein. Das ist in Wirklichkeit nur eine andere Art zu sagen, dass sie immer mehr Daten speichern, abrufen und manipulieren müssen; und das mit einer Reaktionszeit, die von Usern in der heutigen Welt wachsender Prozessor-Leistungsfähigkeit und steigender Kommunikationsintensität erwartet wird.

Anders betrachtet, hängt die Intelligenz eines intelligenten Geräts nicht nur von den Algorithmen ab die es ausführt, sondern auch von den Daten mit denen es diese Algorithmen füttert. Mit steigender Ausgereiftheit dieser Geräte steigt folglich auch der Bedarf nach einer entsprechend ausgereiften Datenorganisations-, Speicher-, und Abfragesoftware, um die Erwartungen an die eingebettete Anwendung zu erfüllen.

Was brauchen wir

Wir könnten die Frage "Was brauchen wir?" ziemlich leicht beantworten: „Eine Datenbank-Bibliothek, die klein, schnell, leistungsfähig und leicht zu bedienen ist."

Das ist eine ziemlich wage Beschreibung, die nur wenig aussagt. Deshalb sollten wir die einzelnen Kriterien näher untersuchen.

- **Minimaler Ressourcenverbrauch.** Obwohl ständiger technologischer Fortschritt es Systemdesignern ermöglicht, immer mehr Speicher für weniger Geld zu bekommen, wird die Optimierung des Speicherverbrauchs einer Anwendung, nie aus der Mode kommen. Der Trade-Off ist simpel: Wenn unsere Datenbank weniger Speicher verbraucht, steht mehr Speicher für andere Anwendungskomponenten zur Verfügung, und der Designer kann mehr Features einbauen (oder die Performance von bereits bestehenden Features erhöhen).

Beachten Sie, dass wir das Wort „Optimierung" verwendet haben. Es geht nicht darum, nur die Grösse der Datenbank-Bibliothek zu reduzieren – das kann man einfach tun, indem man Features weglässt. Aber das Eintauschen von Funktionalitäten gegen Bytes ist gefährlich: Man wird sicherlich ein kleineres Produkt als Ergebnis bekommen, aber es könnte auch weniger nützlich sein.

- **Hoher Durchsatz.** Diese Vorraussetzung ist axiomatisch. Nur in sehr beschränkten Anwendungen werden niedrige Datenbank-Zugriffszeiten toleriert. Um es anders auszudrücken: Wir haben noch nie von jemandem gehört, der sich beschwert hat, weil eine Datenbank ihre Ergebnisse zu schnell liefert.

Testen Sie db4o kostenlos | www.db4o.com

• **Einfache Implementierung.** Trotz aller Verbesserungen bei den Entwicklungsumgebungen, den Fortschritten bei der objekt-orientierten Programmierung und den Framework-Bibliotheken, die randvoll mit vorgefertigten Algorithmen und Datenstrukturen gefüllt sind, müssen Entwickler immer noch die Businesslogik der Anwendung entwerfen und programmieren – ein Unterfangen, das weiterhin schwierig bleibt. Deshalb sollte die API einer Datenbank-Bibliothek keine Lernherausforderung darstellen. Entsprechend dem bekannten Zitat soll gelten: „as simple as possible, but no simpler.“; daraus resultiert eine minimale Lernkurve; Entwickler können sich auf die Anwendung konzentrieren, anstatt die Funktionsweise der Datenbank-Bibliothek zu studieren.

Ausserdem sollte das Eingliedern einer Datenbank-Bibliothek in das Entwicklungsprojekt in einem Schritt möglich sein. Idealerweise besteht die Bibliothek nur aus einer einzigen Datei (für Java eine .JAR Datei; für .NET eine .DLL Datei). Bei kommandozeilen-basierten Projekten sollte das Hinzufügen der Bibliothek lediglich die Modifikation einer einzelnen Zeile in einem Build Script erfordern; für IDE-basierte Projekte sollte das Hinzufügen per Drag-und-Drop erfolgen.

• **Portabel.** Eine Datenbank-Bibliothek sollte “plattform-agnostisch” sein, um ihre möglichen Einsatzziele maximieren – und somit auch die potenzielle Kundenbasis der Anwendung. Breite Portabilität bietet Programmierern auch den Luxus, auf einem grösseren Betriebssystem zu entwickeln – und somit auch die Vorteile von State-of-the-art Entwicklungs- und Debugging Tools – mit der Gewissheit, dass das Ergebnis nicht an das Betriebssystem gebunden ist.

• **Zuverlässig.** Zuverlässigkeit ist eine weitere axiomatische Voraussetzung – und wahrscheinlich auch die wichtigste. Eine unzuverlässige Datenbank ist schlichtweg unbrauchbar. Für den Grossteil der eingebetteten Anwendungen, und insbesondere für diejenigen in Echtzeitsystemen, ist Zuverlässigkeit eine nicht verhandelbare Eigenschaft ALLER Komponenten.

Darüber hinaus muss die Performance der Datenbank den industrie-anerkannten Kriterien genüge leisten. Die Datenbank muss speziell die ACID Eigenschaften aufweisen. (siehe „ACID“ Textblock)

Eine relationale Möglichkeit

Eine relationale Datenbank stellt sicherlich das bekannteste und wahrscheinlich weit verbreitetste Datenbank-Speicherungs-Paradigma dar. Auf Grund ihrer Bekanntheit, ist eine relationale Datenbank die wahrscheinlichste Wahl für die meisten Datenbankanwendungen. Jedoch, trotz den Vorteilen, funktioniert die Eingliederung einer relationalen Datenbank in eine objekt-orientierte Anwendung alles andere als reibungslos. In Zusammenhang mit der Verwendung von relationalen Datenbanken in einer objekt-orientierten Umgebung spricht man oft vom „Impedance Mismatch“. Dieser Ausdruck stammt aus der Elektronikwelt und drückt den Widerspruch zwischen dem relationalen und dem objekt-orientierten Modell aus.

Der „Impedance Mismatch“ kommt daher, dass relationale

ACID

ACID ist ein Akronym für vier Eigenschaften, die eine Datenbank aufweisen muss, bevor sie guten Gewissens als brauchbares Datenbank System bezeichnet werden kann. Sie sind:

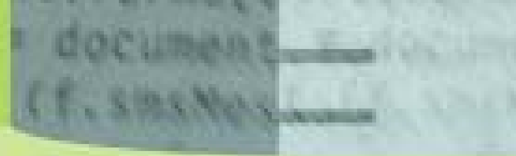
Atomarität – Eine Transaktion muss ganz oder gar nicht ausgeführt werden. Beispiel: Wenn eine Trans-aktion das Löschen von 4 Objekten beinhaltet, müssen alle diese vier Objekte so gelöscht werden, als ob es sich um ein einziges Objekt handelt. Es reicht nicht, wenn drei Objekte ge-löscht werden, aber eines davon ungelöscht übrig bleibt.

Konsistenz – Operationen schieben die Datenbank von einem festdefi-nierten Zustand in den nächsten, ohne dass Zwischenzustände sichtbar sind. Wenn ein User bspw. Objekt A zur Datenbank hinzufügt, sollte es zu keinem Zeitpunkt für ihn (oder einen anderen User) möglich sein, ein unvollständiges Objekt A abzurufen. Die Datenbank sollte niemals in einem Zustand erscheinen, bei der Operationen nur teilweise ausgeführt wurden.

Isolation – Parallele, in Ausführung befindliche Transaktionen sind unter-einander unsichtbar und beeinflussen sich nicht gegenseitig. Wenn zwei User gleichzeitig versuchen, dasselbe Objekt zu modifizieren, muss die Datenbank einen Mechanismus besitzen, um Userzugriffe auf das Objekt zu serialisieren, so dass sich die Handlungen der beiden User nicht gegenseitig beeinträchtigen oder sogar füreinander sichtbar für werden.

Dauerhaftigkeit – Sobald eine Trans-aktion an die Datenbank “übergeben” wurde, sind deren Arbeitsabläufe und Ergebnis dauerhaft. Sie gehen nicht einmal dann verloren, wenn es zu einem Hardware oder Software-Absturz kommt. Wenn ein User also eine Transaktion zum Löschen von drei Objekten ausführt und das System während dem Löschen des zweiten Objekts abstürzt, wird die Datenbank nach dem Neustarten des Systems, nicht nur sich selbst wiederherstellen, sondern auch die nicht beendete Transaktion. Somit wird gewährleistet, dass die Trans-aktion abgeschlossen wird.

Testen Sie db4o kostenlos | www.db4o.com



Datenbank Management Systeme (RDBMS) Informationen in Zeilen innerhalb von Tabellen speichern. Folglich muss beim Speichern eines Objekts in einer solchen Datenbank, das Objekt zerlegt sein – auseinander genommen und in Einzelteilen in den Feldern der Datenbanktabellen verteilt. Um das Objekt abzurufen, müssen seine verstreuten Teile gesammelt und wieder zusammengesetzt werden.

Ein Beispiel (in Java): Ein intelligenter Verkaufsautomat

Diese Arbeit kann durch ein Beispiel leicht illustriert werden.

Nehmen wir an, wir haben einen intelligenten Verkaufsautomat entworfen: Er protokolliert die Art und Anzahl der verkauften Snacks, behält die Übersicht über das Wechselgeld im Speicher, führt Buch über Verkäufe und – vielleicht hat er ja sogar eine drahtlose Verbindung ins Internet – sendet eine Email an die zugehörige Geschäftsstelle wenn er nachgefüllt werden muss. (Glauben Sie uns; das ist gar nicht mal so weit hergeholt, wie es sich anhört.)

Das Abrufen eines `Snack` Objekts aus einem RDBMS würde in etwa den folgenden Code erfordern:

```
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 ...
}
}
```

Auflistung 1. Das Abfragen der Daten eines Snacks aus einem RDBMS. Dieser Code ruft Informationen über einen Snack ab, dessen ID über die Inputvariable `searchID` bestimmt ist.

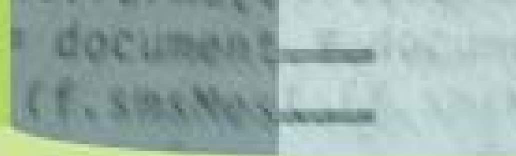
Das Ergebnis dieser Abfrage ist erst einmal in der Collection `results` gespeichert. Dann werden durch Zuordnungsoperationen, die Inhalte der einzelnen Felder des `results` Objekts in das `snack` Objekt kopiert. Zusätzlich könnte noch jede Zuordnungsoperation eines Feldes, die Übersetzung des Wertefomats der Datenbank, in die von Java benutzte interne Darstellung, beinhalten (diese Übersetzung findet jedoch innerhalb der spezifischen Methode statt und ist nicht explizit sichtbar). Der umgekehrte Version dieses Aufwands wird benötigt, wenn man ein Objekt in die Datenbank einfügen will. Sie können sich vorstellen, wie der Code bei grossen, komplexen Objektstrukturen aussehen würde.

Beachten Sie auch, dass der Prozess mit der Ausführung eines SQL Statements, das in einem String ausgedrückt wird, beginnt. Wenn das Statement noch nicht pre-compiled ist, werden noch mehr CPU Zyklen verbraucht um die SQL, welche die eigentliche Query in der Backend-Datenbank verrichtet, zu parsen und auszuführen.

Der "Impedance Mismatch" wird in objekt-relationalen Datenbanksystemen etwas gemindert. Ein objekt-relationales Datenbanksystem (ORDBMS) übernimmt die Übersetzung zwischen Objekten und der relationalen Datenbank unsichtbar. (Die relationale Datenbank ist immer noch da, sie versteckt sich lediglich im Backend). Als Ergebnis wird der – oben dargestellte – explizite Code nicht zum Auseinanderbauen und Wiederzusammensetzen von Objekten benötigt.

Aber obwohl der Code nicht explizit ist, ist er immer noch da. Er ist üblicherweise in einem „Mapping Layer" verschleiert, einer Sammlung von Klassen und Methoden innerhalb der objekt-relationalen Datenbank-Bibliothek. Dieser Mapping Layer kümmert sich um die

Testen Sie db4o kostenlos | www.db4o.com



Übersetzung zwischen Objekten in der Anwendung, und den Tabellen, Zeilen und Feldern in der relationalen Datenbank.

Während der Programmierer also Objekte als Objekte manipulieren kann und somit weniger Code schreiben muss, werden die CPU Zyklen immer noch für das Übersetzen zwischen relationaler Datenbank und Objektdaten verbraucht. Und irgendwo in der Datenbank Bibliothek muss noch SQL arbeiten, um Zeilen aus den Datenbanktabellen zu extrahieren, oder um sie darin zu speichern.

Im Ergebnis entsteht durch ein RDBMS ein Speicherplatz und Zeit Overhead in der Anwendung. Speicherplatz wird durch den Objekt-in-RDBMS Übersetzungscode verbraucht; Zeit wird durch die Ausführung dieses Code verbraucht. Ein ORDBMS erleichtert die Angelegenheit etwas; Programmierer sind zumindest vom Schreiben des Übersetzungscodes befreit. Aber der Übersetzungscode ist – obwohl versteckt – immer noch präsent und frisst Speicherbytes und Taktzyklen der CPU Zeit.

Die Lösung

Die Lösung für die oben genannten Probleme ist ein rein objekt-orientiertes Datenbanksystem. Eine besonders praxisnahe, objekt-orientierte Datenbank ist db4o (von db4objects, Inc.). db4o erfüllt die im ersten Abschnitt erwähnten Anforderungen und umgeht gleichzeitig die im zweiten Abschnitt beschriebenen Probleme. db4os Stärken werden am besten beleuchtet, wenn wir uns ansehen, wie es mit den bereits diskutierten Problemen umgeht.

- **Minimaler Ressourcenverbrauch.** Die db4o Bibliothek braucht nur etwa 400K. Wie Sie sehen werden, bedeutet sein speichergenügsamer Footprint dennoch keine Einschränkung bzgl. seinen Fahigkeiten.

- **Hoher Durchsatz.** db4os Performance ist auf der gleichen Stufe mit den besten Datenbanksystemen. Die Benchmark Ergebnisse unten zeigen db4os Performance verglichen mit typischen SQL Datenbanken.

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

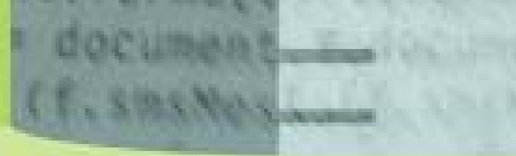
fastest … slowest

db4o Benchmark. Die obere Tabelle zeigt db4os Performance verglichen mit typischen SQL Datenbanken bei read, write, query und delete Operationen. Die komplexen Operationen beinhalten Datenbankaktionen für Objekte mit einer fünfstufigen Vererbungsstruktur.[1]

- **Einfache Implementierung.** Die Java Version von db4o ist eine einzige JAR Datei; die .NET Version ist eine einzige DLL Datei. Sie können die Bibliothek zu Ihrer Anwendung hinzufügen, indem Sie sie in Ihren CLASSPATH einbringen (bei der Benutzung von Kommando-Zeilen Tools für Java Entwicklung), oder indem Sie die Datei in Ihrem Projekt

[1] Mehr Informationen sind auf der db4objects Website verfügbar http://www.db4o.com/about/productinformation/benchmarks/

Testen Sie db4o kostenlos | www.db4o.com



ablegen (bei der Benutzung einer IDE für Java oder .NET).

Die db4o API ist nicht mit komplizierten und komplexen Klassen und Methoden überladen. Beispiel: Ein db4o Programmierer arbeitet hauptsächlich mit db4os `ObjectContainer` Klasse (welche die Datenbank selbst darstellt). Das `ObjectContainer` Interface definiert lediglich 10 Methoden; Dennoch stellen diese 10 den Grossteil der Datenbankmanipulation bereit – Hinzufügen, Suchen und Löschen von Daten.

Wenn wir also annehmen, dass ein `ObjectContainer` namens `vendingmachineDB` schon geöffnet ist, und wir ein `Snack` Objekt in der Datenbank speichern wollen, ist der Java Code einfach:

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

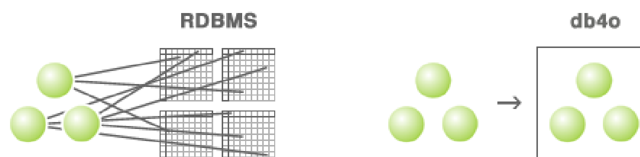
Das selbe Codefragment kann benutzt werden um ein `Snack` Objekt, das schon in der Datenbank gespeichert ist, zu aktualisieren. (Beachten Sie folgendes: Falls es sich bei der oben angegebenen Datenbankoperation, um die letzte in der Anwendung handelt, sollten Calls an die `ObjektContainer` `commit()` und `close()` Methoden folgen, um die Aktivitäten in der Datenbank ordnungsgemäss zu beenden. Ähnliche Anweisungen würden auch bei einem RDBMS oder ORDBMS benötigt werden.)

- **Portabel.** Wie bereits erwähnt, gibt es Java und .NET Versionen von db4o. Die Java Version läuft auf allen Java Plattformen ab Java 1.1.x aufwärts (einschliesslich PersonalJava und der J2ME CDC Konfiguration). Die .NET Version ist kompatibel mit .NET 1.0, 1.1 und dem CompactFramework. Darüber hinaus kann die .NET Version mit allen .NET Sprachen benutzt werden und läuft auch auf dem open-source Mono Framework.

- **Betriebssicher.** db4o unterstützt alle ACID Charakteristiken. Mehrere, gleichzeitige User eine db4o Datenbank werden entsprechend isoliert und ihre Operationen unsichtbar von der db4o Bibliothek serialisiert. Transaktionen werden von den `commit()` und `rollback()` Methoden der `ObjectContainer` Klasse beendet. Und im Falle eines Systemabsturzes während eines Datenbankupdates, wird jede unterbrochene Transaktion beim Wiederöffnen des db4o `ObjectContainer` ordnungsgemäss zu Ende gebracht.

Kein Impedance Mismatch

db4o ist eine echte Objektdatenbank. Objekte werden “wie sie sind” gespeichert; Es gibt keine Objekt-zu-relational-Übergangsschicht, weder explizit noch unsichtbar. Ausserdem kann db4o mit beliebig komplexen Objektstrukturen umgehen. Sie müssen auch keine Schema Definitionen erstellen um Objekte in relationale Datenbanktabellen zu mappen. Die Klassenhierarchie und die Objektbeziehungen Ihrer Anwendung selbst definieren das Datenbankschema:



Kein Impedance Mismatch. Die Klassenhierarchie und die Objektbeziehungen Ihrer Anwendung selbst definieren das Datenbankschema.

Die Leichtigkeit mit der man db4o einsetzen kann, wird am besten anhand einer Reihe von Beispielen dargestellt. Um auf das Beispiel mit der Verkaufsautomat-Datenbank

zurückzukommen (das wir oben in SQL dargestellt haben): Nehmen Sie an, wir erstellen die gleiche Datenbank in db4o. Der Code zum Öffnen der Datenbank und das Hinzufügen eines neuen Snack „Eintrags“ würden folgendermassen aussehen:

```
// 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();
```

Auflistung 2. Speichern eines Snack Objekts in einer db4o Datenbank.

Zum Speichern eines Objekts muss also lediglich eine `set()` Methode auf den `ObjectContainer` aufgerufen werden, wodurch der Methode eine Referenz zu dem zu speichernden Objekt übermittelt wird. Die `commit()` Methode stellt sicher, dass jegliche Modifikationen seit dem Öffnen der Datenbank (oder seit dem letzten `commit()`) auch in die Datenbank geschrieben werden. (Sie stellt auch sicher, dass die Operation nicht verloren geht, falls Sie durch einen Systemabsturz unterbrochen wurde.)

Beachten Sie, wie leicht es ist. Der Programmierer muss nicht mehr die Bestandteile des Objekts manipulieren, um Daten in die Datenbank zu bekommen. Das Objekt wird wie – naja, ein Objekt behandelt. Alles was der Programmier tun muss, ist db4o zu sagen: „Bitte lege dieses Objekt in der Datenbank ab.“ db4o kümmert sich, für den User unsichtbar, um die Details.

Das Abrufen des Snacks ist genauso leicht. db4o nutzt eine neuartige Query By Example (QBE) Technik um Datenbankobjekte zu lokalisieren. Wir statten db4o mit einem „Template“ Objekt aus, das benutzt wird, um unser Suchziel zu lokalisieren. Das Template Objekt entstammt der selben Klasse wie unser Suchziel, und seine Elemente enthalten Daten, welche die Suchkriterien spezifizieren.

Von der Annahme ausgehend, dass unser `vendingmachineDB ObjectContainer` bereits geöffnet wurde, liefert uns der folgende Code den Snack, den wir gerade eingegeben haben:

Testen Sie db4o kostenlos | www.db4o.com

```
// 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.
   . . .
}
```

Auflistung 3. Abrufen eines `Snack` Objekts aus einer db4o Datenbank.

Vergleichen Sie das mit dem Beispiel für RDBMS. Mit db4o wird das `Snack` Objekt als Ganzes abgerufen; der Programmierer muss nicht mehr eine Reihe von Zuweisungs-statements schreiben, um die Objektfelder auszufüllen. (Dies trifft immer zu, ganz unabhängig von der Anzahl der Felder im abgerufenen Objekt.) Es gibt auch keinen SQL Command String der an eine `executeQuery()` Methode zum Parsen und Verarbeiten weitergegeben werden muss.

Das Löschen eins Objekts ist genauso einfach. Sobald ein Objekt aus der Datenbank abgerufen wurde, geben Sie einfach seine Referenz an die `delete()` Methode des `ObjectContainer` weiter. Der Code ist trivial:

```
// 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);
}
```

Auflistung 4. Das Löschen eines `Snack` Objekts aus der Datenbank.

Wie immer werden Operationen am Objekt ganzheitlich ausgeführt, Objekte werden als die untrennbaren Entitäten behandelt, die sie darstellen.

Es mag der Aufmerksamkeit des Lesers entgangen sein, dass alle dargestellten Beispiele für db4o mehr als nur ein einziges `Snack` Objekt beinhalten. Wir haben die `Snack` Klasse folgendermassen definiert:

Testen Sie db4o kostenlos | www.db4o.com

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

Auflistung 5. Die `Snack` Klasse.

Die zweiten und letzten Elemente der `Snack` Klasse sind String Objekte. Java Strings werden aber als Objekte und nicht als Primitive implementiert. Somit werden beim Speichern eines `Snack` Objekts auch zwei String Objekte gespeichert, und beim Löschen des `Snack` werden auch die String Objekte, die mit Name und Anbieter verknüpft sind, gelöscht. db4o erledigt sämtliches Speichern und Löschen im Hintergrund, ohne dass wir es dazu anweisen müssen. (Wie sich herausstellt, behandelt db4o String Objekte und andere einfache Typen wie Objekte der Unterklasse; sie haben keine eigene Identität und werden zusammen mit ihrem Elternobjekt gelöscht und aktualisiert.)

Nichts desto trotz handhabt db4o mit Leichtigkeit beliebig komplexe Objekthierarchien. Nehmen wir an, dass wir tiefer in die Struktur unserer Verkaufsautomaten-Datenbank eintauchen. Wir definieren eine Klasse namens `SnackSlot`, die einen der zahlreichen Slots im Verkaufsautomaten modelliert, welche die Snacks enthalten. (Jeder Slot kann mehrere Instanzen eines gegebenen Snacks enthalten.) Die Klasse könnte so aussehen:

```
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
...
}
```

Auflistung 6. Die `SnackSlot` Klasse.

Ein `Snackslot` hat also eine Referenz zu dem Snack Objekt, welches von diesem Slot ausgegeben wird. Wir beobachten auch die ursprüngliche Anzahl der Snacks, die in einen bestimmten Slot gesteckt wurden, sowie die Anzahl der verbleibenden Snacks. Dadurch kann der Automat feststellen, wieviele Snacks in einem bestimmten Slot verkauft wurden.

Ausgehend von der Annahme, dass wir die Möglichkeiten unseres Verkaufsautomaten erweitern, indem wir einen neuen Snack und einen neuen Ausgabeslot hinzufügen, könnte man meinen, dass dafür zwei einzelne `set()` Operationen notwendig sind. Dies ist jedoch nicht der Fall.

Wenn ein Objekt zum ersten Mal zur Datenbank hinzugefügt wird, erforscht db4o alle Referenzen dieses Objekts und speichert alle referenzierten Objekte ebenfalls in der Datenbank ab – alles automatisch. Folglich wäre der Code um gleichzeitig einen neuen `Snackslot` und einen zugehörigen neuen `Snack` hinzuzufügen:

```
// 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);
```

Auflistung 7. Hinzufügen von komplexeren Objekten zur Datenbank.

Das Abrufen und Löschen von `Snack` und `SnackSlot` Objekten erfordert nur ein kleines bischen mehr Aufwand, aber das auch nur, weil db4o das Feintuning des Managements von Objekten erlaubt, sobald sie in der Datenbank sind.

Um ein zusammengesetztes Objekt abzurufen, müssen wir db4o sagen, wie weit es in den Refrenztree des Objekts beim Abruf „hineingreifen“ soll (beim Abruf eines `get()`). Dies wird als „Activation Depth“ bezeichnet. Eine Activation Depth von 0 ruft nur das Root-Objekt ab, wenn wir also ein `SnackSlot` Objekt mit einer Activation Depth von 1 abrufen, erhalten wir von db4o sowohl das `SnackSlot` als auch das `Snack` Objekt.

Da die standardmässige Activation Depth bei db4o auf 5 gesetzt ist, können wir ein `SnackSlot` Objekt und sein zugehöriges `Snack` Objekt mit dem gleichen Code wie in **Auflistung 3** abrufen. (Wir zeigen im nächsten Abschnitt wie man mit db4os Activation Depth arbeitet.)

db4o kann auch alle Objekte löschen, die von einem Root-Objekt referenziert sind. Jedoch müssen wir db4o explizit anweisen, dass es beim Löschen eines Objekts auch alle Objekte löschen soll, die es referenziert – ansonsten wird nur das Root-Objekt gelöscht. Wenn wir also das `Snack` Objekt beim Löschen eines `SnackSlot` Objekts mitlöschen wollen, müssen wir bei dem `SnackSlot` Objekt ein „Kaskadiertes Löschen“ Flag setzen. Wir tun dies in db4os „globalem Konfigurationsobjekt“. Der Code wird in etwa so aussehen:

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

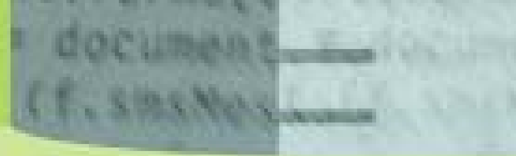
Auflistung 8. Einrichten des „Kaskadiertes Löschen“ Flag

Dieser Code sagt db4o, dass jedesmal, wenn ein `SnackSlot` Objekt gelöscht wird, alle untergeordneten Objekte (in diesem Fall, das zugehörige `Snack` Objekt) ebenfalls gelöscht werden sollen. Mit dem Einfügen des Fragments aus **Auflistung 8**, ist der Code in **Auflistung 4** ausreichend, um einen `SnackSlot` und den zugehörigen `Snack` zu löschen.

Natürlich ist kaskadiertes Löschen nicht in allen Fällen angebracht (deshalb wird es von db4o wahlweise angeboten). In unserem hypothetischen Verkaufsautomaten Beispiel könnte es gut möglich sein, dass – aus welchem Grund auch immer – ein `SnackSlot` entfernt wird, aber der zugehörige `Snack` auch noch in einem anderen Slot angeboten wird. In diesem Fall würden wir auf kaskadiertes Löschen verzichten, weil wir das `Snack` Objekt in der Datenbank behalten wollen.

Native Queries

Der Kern von db4os QBE entspricht mehr oder weniger einem “WO … IST GLEICH ZU …” Filter innerhalb einer Query. Das ist sinnvoll, um das Root-Objekt in einer Hierarchie von



Target-Objekten zu lokalisieren. Sobald die Root in den Speicher geladen wurde, erlaubt Ihnen db4o die Benutzung von gewöhnlichen Objektreferenzen um zu einem beliebigen Kind- oder Geschwister-Objekt zu navigieren. Natürlich wird die Navigation durch einen Objekt-Tree nicht dadurch beeinflusst, dass der Objekt-Tree ursprünglich Teil der db4o Datenbank war. Das Ergebnis ist ein leistungsfähiger, unkomplizierter Abfrage-Mechanismus, der für eine breite Auswahl von Query-Bedürfnissen, vollkommen ausreichend ist.

Natürlich gibt es auch Fälle, in denen ausgefeiltere Queries benötigt werden; z.B. wenn Gegenstände von einem persistenten Speicher, basierend auf anderen Kriterien als Äquivalenz, ausgewählt werden müssen. Man könnte glauben, dass kompliziertere Queries auch eine komplizierte Query Syntax erfordern. Für viele Datenbank-Systeme mag das zutreffen – nicht aber für db4o.

db4os Native Query (NQ) API bewältigt Queries, welche die Fähigkeiten von QBE überschreiten, mit Leichtigkeit. Und – entsprechend db4os einfachem Nutzungsprinzip – muss der Entwickler bei Native Queries weder eine separate Sprache beherrschen, noch muss er sich mit einer unübersichtlichen API abmühen. Vielmehr benutzt der Entwickler bei Native Queries nicht mehr (und auch nicht weniger) als die Sprache, in der auch der Rest der Anwendung geschrieben ist.

Denken Sie an die Query, die wir in **Auflistung 3** gezeigt haben, zurück. In diesem Code Ausschnitt haben wir das `Snack` Objekt abgerufen, dessen ID gleich 100 war. Der selbe Code würde als Native Query in etwa folgendermassen aussehen:

```
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
     ...
}
```

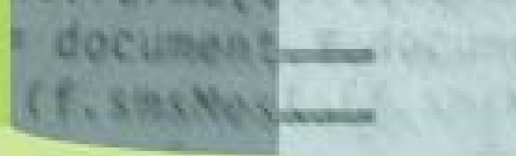
Auflistung 9. Eine Native Query entsprechend dem Beispiel aus Auflistung 3.

Hier haben wir nicht nur db4os NQ API eingesetzt, sondern auch Javas kürzlich eingeführte generische Fähigkeiten in Anspruch genommen, um unsere Abfrage vollständig typsicher zu machen. (Generics wurden mit JDK 5 eingeführt; db4os Native Queries können allerdings auch mit „weit zurückliegenden" Java Versionen bis hin zu JDK 1.1 genutzt werden.)

Bei näherer Betrachtung des obigen Query-Codes, können wir die hervorstechenden Merkmale von db4os Native Query API entdecken. Als erstes definiert NQ die `Predicate` Klasse Instantiierungen, die die eigentliche Query-Engine darstellen. Innerhalb dieser Klasse gibt es eine einzige Abstraktionsmethode – `match ()` – welche die Erweiterungen der `Predicate` Klasse implementieren müssen. Die `match()` Methode benötigt ein einzelnes Objekt Argument. Dieses Argument identifiziert die Klasse der Ziele, die in der Datenbank abgefragt werden. Im Falle der **Auflistung 9**, würden Zielobjekte des Typs `Snack` in der Query miteinbezogen werden.

Die `match()` Methode liefert ein `boolean` zurück, woraus man gut erkennen kann, dass die eigentliche Aufgabe von `match()` das Filtern der Zielobjekte ist. Einfacher ausgedrückt: `match()` liefert true zurück wenn ein Objekt den Suchkriterien entspricht.; ansonsten false. Beim Ausführen einer Abfrage wird also jedes `Snack` Objekt in der Datenbank an die `match()` Methode weitergereicht und dann mit dem empfangenen boolean festgestellt, ob

Testen Sie db4o kostenlos | www.db4o.com



ein gegebenes Objekt in der zurückgelieferten `List` Collection enthalten sein soll.

Native Queries sind aus mehreren Gründen ansprechend, nicht zuletzt wegen der Tatsache, dass die Sprache der Anwendung selbst die Query-Sprache ist. Nehmen wir beispielsweise an, dass wir wissen möchten, welche Snacks sich besonders gut verkaufen (z.B. 5 oder mehr Einheiten).

Mit der NQ API ist das 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...
```

Auflistung 10. Eine komplexere Query mit db4os NQ.

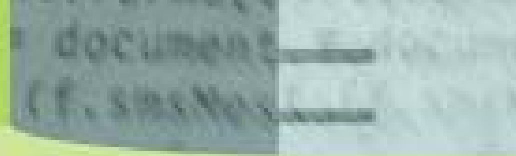
Mit einer Native Query können wir Berechnungen auf den Objektfeldern der Kandidaten durchführen und somit feststellen, welche `SnackSlots` mehr als 5 Einheiten verkauft haben. In der While Schleife untersuchen wir die Treffer und nutzen db4os `activate()` Methode um die `Snack` Objekte in den Speicher einzulesen. (Wie wir vorhin bereits erwähnt haben, sagt die `activate()` Methode db4o, dass es die zugehörigen Objekte aus der Datenbank bis zu einer festgelegten „Tiefe“ in den Speicher laden soll. Eine Tiefe von 2 stellt sicher, dass nicht nur das `SnackSlot` Objekt, sondern auch das `Snack` Objekt, im Speicher ist.)

Nachdem diese Query ausgeführt wurde, kann die Anwendung die `popularSnacks ArrayList` bearbeiten, welche möglicherweise den Besitzer des Verkaufsautomaten benachrichtigen kann, welche Snacks in der nahen Zukunft nachgefüllt werden müssen.

Natürlich hätten wir den Abgleichsanteil der Query beliebig komplex gestalten können. Wir hätten `activate()` benutzen können, um andere zugehörige Objekte in den Speicher zu laden. Das Tolle an NQ ist, dass wir jeglichen Java Code verwenden können, der nötig ist um die Query zu implementieren. (Es gibt ein paar kleinere Einschränkungen, die alle mit dem Vermeiden von unerwünschten Nebeneffekten zu tun haben. Für mehr Informationen zu diesem Thema sollten Sie die Dokumentation auf der db4o Website lesen.)

Das Beste ist jedoch, das unsere Queries vollständig typsicher und komplett refakturierbar sind. Wenn wir eine string-basierte Query-Sprache wie SQL verwendet hätten, würden

Testen Sie db4o kostenlos | www.db4o.com



sämtliche Fehler im Referenzfeld eines Objekts (Tippfehler, falsche Feldnamen, etc.), erst beim Ausführen zum Vorschein kommen. Bei Native Queries sorgt der Compiler dafür, dass wir nicht aus der Reihe tanzen. Falls wir – mit fortschreitender Entwicklung unserer Anwendung – den Namen einer Klasse, Klassenbestandteile, Felder, etc. geändert hätten, könnte die Refactoring-Funktion unserer Integrated Development Environment, die Namensänderungen zuverlässig in einem Zug durchführen. Mit einem relationalen, SQL-basierten System, wären wir gezwungen, Query-Strings manuell zu lokalisieren und Änderungen daran vorzunehmen.

An der Oberfläche gekratzt

Wir haben lediglich an der Oberfläche gekratzt. Obwohl wir nur einen kleinen Ausschnitt von db4os Fähigkeiten gezeigt haben, konnten wir zumindest zeigen, dass es den Ansprüchen an eine zuverlässige, responsive Datenbank mit kleinem Footprint, gerecht wird. Darüber hinaus haben wir gezeigt, dass es bei der Verwendung von db4o zu keinerlei Impedance Mismatch Problemen wie bei RDBMS und ORDBMS in einer objekt-orientierten Umgebung kommt. Ausserdem besitzen wir dank db4os Native Query Fähigkeit beliebig komplexe Queries, die einfach zu erstellen und zu warten sind – viele der Fallgruben, in die Programmierer bei der Arbeit mit RDBMS Queries stolpern, können somit umgangen werden.

Dennoch gibt es viel, das wir nicht vorgestellt haben:

• **Änderungen der Objektversionen.** Wir haben beispielsweise nicht gezeigt, wie leicht db4o mit Änderungen der Objektversionen umgeht. Angenommen, wir wollen die Struktur unserer `Snack` Klasse modifizieren, indem wir einen `snackType` Teil hinzufügen, der es uns ermöglicht, zwischen Schokoriegeln und Chips zu unterscheiden. Wir können diese Änderung mit einer db4o Datenbank durchführen, die bereits erstellt ist und „alte" `Snack` Objekte enthält. Und wir können das ohne eine Unterbrechung in der Datenbanknutzung erledigen. db4o erlaubt es uns – mit einem einzigen Methodenaufruf – eine Klasse umzubennen, die bereits in einer Datenbank gespeichert ist. (Folglich könnten die alten `Snack` Objekte in `OldSnack` Objekte umbenannt werden.) Demgegenüber würde eine Lösung mit einem relationalen Datenbank Backend, sowohl Änderungen an den gesamten Datenbank-Tabellen, als auch am Abfrage-Code, erfordern.

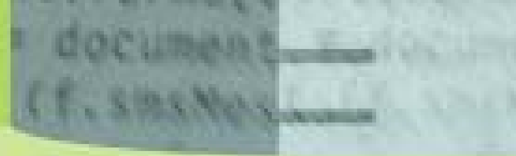
• **Einfache Objekt-Replikation und Synchronisierung.** Software, die auf intelligenten, aber periodisch-verbundenen Geräten (z.B. Handheld Organizer oder Scanner) läuft, muss die Möglichkeit besitzen, ein „Subset" von persistenten Objekten aus einer Master-Datenbank zu akzeptieren. Sie muss auch der User-Anwendung erlauben, mit dieser derivativen Datenbank (getrennt von der Master-Datenbank) zu arbeiten, und sich wieder mit der Master-Datenbank zu verbinden, um vorgenommene Änderungen zu synchronisieren. Bei db4o ist diese Möglichkeit unter dem Namen db4o Replication System (dRS) direkt in die Datenbank eingebaut[2].

• **Wartungsfreiheit.** Wir haben noch nicht viel über db4os Wartungseinrichtungen erzählt – und das aus gutem Grund: db4o benötigt praktisch keine Wartung, und damit ist so ziemlich alles gesagt. Daher ist es ideal für Handheld Mobile Device Anwendungen, Informationsgeräte, intelligente medizinische Systeme und alle anderen Anwendungen bei denen die Datenbank für den User unsichtbar ist.

• **Mit offenem Quellcode verfügbar und unter der GPL kostenlos.** Für den Schluss haben wir uns die vielleicht attraktivste Eigenschaft von db4o aufgehoben: Sowohl die Java, als auch die .NET Version sind open-source. Und falls Sie db4o in Ihrer nächsten kommerziellen Anwendung miteinbeziehen wollen, sprechen die geringen Kosten von db4o für sich selbst.

[2] Mehr Informationen über das db4o Replication System: http://www.db4o.com/about/productinformation/features/drs.aspx

Testen Sie db4o kostenlos | www.db4o.com



Aber alle diese Eigenschaften wären wertlos, ohne die Essenz von db4o: Es ist eine kompakte, leistungsfähige und praktische objekt-orientierte Datenbank für Java und .NET.

Testen Sie db4o kostenlos | www.db4o.com