Import aus proprietären Datenformaten

Import-Dompteur

Ein großer Datenbestand wie der von freedb.org eignet sich gut für die Durchführung von Performance-Tests, wenn die Produktionsdaten selbst für diesen Zweck nicht bereitstehen. dotnetpro zeigt, wie Sie sich eine Testdatenbank aufbauen können. Dabei lernen Sie viel über die Kapselung von proprietären Datenstrukturen in eigenen Cursor-Klassen.

Wenn Sie von ADO nach ADO.NET wechseln oder beispielsweise statt einer MS-Access-MDB-Datenbank lieber Oracle zur Datenspeicherung einsetzen wollen, dann lohnen sich Performance-Tests im Vorfeld. Ist das neue Produkt schneller als das bisherige? Ist die Zugriffsstrategie mit einem neuen API noch genauso performant? Aber auch, wenn Sie keinen Technologiewechsel durchführen wollen, sondern sich bei gegebener Technologie fragen, ob diese mit größer werdenden Daten- und Nutzerlasten umgehen kann, ob sie also skaliert, sind Tests wichtig.

Doch mit welchen Datenbeständen führen Sie Tests durch? Die üblichen Kunden mit Aufträgen und so weiter aus einer Nordwind-Datenbank sind zwar schnell zur Hand, aber was sagt es aus, wenn Sie die Aufträge von knapp 100 Kunden selektieren? Die Nordwind-Datenbank von Microsoft hat zwar eine nicht-triviale Struktur, aber mit weniger als 10 MByte ist sie ziemlich klein. Sie enthält bestenfalls einige tausend Datensätze.

Für echte Tests brauchen Sie mehr Daten. Sie brauchen 10.000 oder 100.000 oder gar ein paar Millionen Datensätze, um wirklich etwas über die Performance einer Technologie oder eine Zugriffsstrategie sagen zu können.

Aber woher können Sie solch große Datenbestände bekommen? Produktionsdatenbanken, die sicherlich oft genügend groß sind, dürfen Sie wahrscheinlich nicht „abziehen". So sind Sie denn meist wieder auf Auszüge oder gar selbst erfasste Testdaten angewiesen. Was also tun?

freedb.org als Rettung

Natürlich sind Tests mit Originaldatenbeständen am besten. Sie haben die für Sie relevante Struktur und ihre Daten sind typisch über die Wertebereiche verteilt. Wenn aber Originaldaten in ausreichender Menge fehlen, dann tun es für viele Tests auch spezielle Testdaten. Auch die Nordwind-Datenbank ist insofern eine Testdatenbank – aber eben eine sehr kleine.

Abhilfe kann da die Datenbasis von freedb.org [1] schaffen. freedb.org ist als freie Alternative zu CDDB [2] entstanden und bietet Informationen zu Musik-CDs. Über freedb.org und CDDB können Musikprogramme wie etwa der Windows Media Player Informationen zur abgespielten CD über das Internet ermitteln. Nähere Informationen zur Funktionsweise bietet beispielsweise [3].

Für das Testdatenproblem ist dabei interessant, dass freedb.org seine Datenbank zum Download anbietet. Sie enthält rund 1,2 Millionen Einträge zu Musikalben und mehr als 15 Millionen Musiktitel. Unter [4] können Sie die Datenbank als RAR-Archiv herunterladen. [5] liefert das passende Dekomprimierungs-Tool WinRAR dazu. Gepackt ist die Datenbank circa 220 MByte groß, entpackt mehr als 1,2 GByte! Abbildung 1 zeigt ein relationales Datenmodell für die Daten, die jedoch zunächst nur in Form von Textdateien vorliegen.

Im Mittelpunkt stehen bei freedb.org die Alben mit ihren Titeln (Tracks), die zueinander in einer 1:n-Beziehung stehen. Jedes Album und jeder Titel gehören über eine n:1-Beziehung zu einem Künstler (Artist). Außerdem gehört jedes Album über eine n:1-Relation zu einem Genre wie etwa Blues.

Das Datenmodell von freedb.org ist nicht trivial. Es bestehen hierarchische Be-

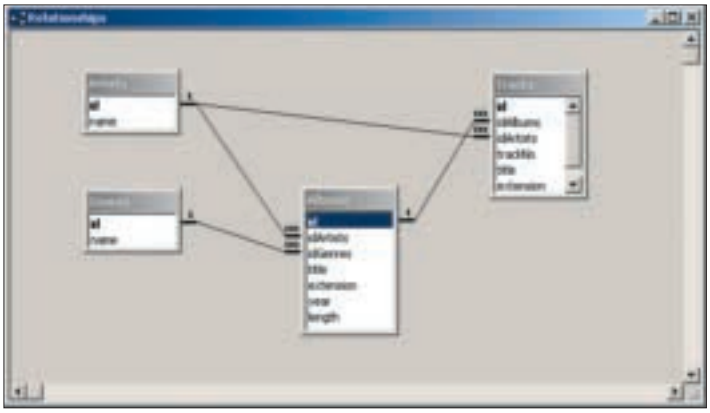


Abbildung 1 Relationales Datenmodell für die Daten von freedb.org.

Auf einen Blick

Autor

Ralf Westphal ist freier Software-Technologievermittler mit dem Schwerpunkt auf der Microsoft-Plattform. Er arbeitet als Fachautor, Berater, Software-Entwickler und Sprecher auf Entwickler-Events im In- und Ausland. Er ist Microsoft Regional Director und einer der .NET Twins (www.dotnettwins.de).

dotnetpro.code
A0406FreeDB

Sprachen VB.NET

Technik Datenimport über selbst entwickelte Reader-Klassen

Voraussetzungen .NET SDK

ziehungen über mehrere Ebenen und es können Fragen formuliert werden wie „Welches sind die Titel in einem Genre, die von einem bestimmten Künstler komponiert wurden?“ oder „Was ist die durchschnittliche Spieldauer von Titeln eines Genres?“, die sich bei einer relationalen Datenbank nur mit mehrstufigen Joins beantworten lassen.

Es ist also leicht, anspruchsvolle Zugriffe zu formulieren, die dank des Umfangs der Datenbasis große Datenmengen durchforsten müssen und auch noch große Datenmengen zurückliefern können. Schon ein kleiner Test wie in Listing 1 kann Performance-Unterschiede bei verschiedenen Datenbanksystemen aufdecken beziehungsweise Hinweise für das Query- und Datenbank-Tuning geben.

Listing 1

Alle Tracks von Alben suchen, deren Titel mit „A“ beginnt.

```
SELECT Tracks.*, Albums.titel, Artists.name
FROM Artists INNER JOIN
  (Albums INNER JOIN Tracks ON Albums.id =
Tracks.idAlbums)
  ON (Artists.id = Tracks.idArtists) AND
     (Artists.id = Albums.idArtists)
WHERE (Albums.title) Like 'a%'
```

1,2 GByte Rohdaten

Bevor Sie jedoch SQL-Abfragen oder andere Datenzugriffe formulieren können, müssen Sie die 1,2 GByte Rohdaten in ein geeignetes Datenformat überführen. Je nachdem, welche Technologie Sie testen wollen, brauchen Sie einmal eine MS-Access-MDB, ein anderes Mal eine SQL-Server-Datenbank (RDBMS) und dann vielleicht sogar einmal eine echte objektorientierte Datenbank (ODBMS). Der Beispielcode auf der Heft-CD liefert Importroutinen für alle Varianten.

Als ODBMS wurde das deutsche Produkt FastObjects (FO) [6] gewählt, das seit neuestem ein Binding für .NET-Sprachen bietet. An dieser Stelle kann nicht näher auf FO eingegangen werden. Aber ein Folgeartikel wird diese Alternative zu den „Platzhirschen“ näher betrachten. Das Interesse an O/R-Mapping-Technologien wie Microsofts ObjectSpaces zeigt, dass die Zeit reif für einen objektorientierten Umgang mit persistenten Daten ist. Und den bieten ODBMS wie FO. Eine

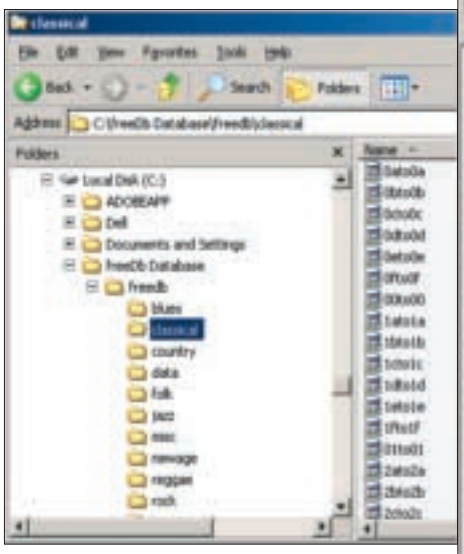


Abbildung 2 Organisation der CD-Rohdaten von freedb.org im Dateisystem.

Evaluationsversion von FO finden Sie unter www.versant.net.

Um Importroutinen schreiben zu können, müssen Sie zunächst das Quelldatenformat verstehen. Abbildung 2 zeigt die Organisation der Rohdaten nach Entpacken des RAR-Archivs.

Die Daten liegen in einem Baum von elf Verzeichnissen und mehr als 2.500 Dateien vor. Die Verzeichnisse sind nach Musikgenres benannt, haben letztlich aber keine Bedeutung (linke Seite in Abbildung 2). Gleiche Dateinamen einmal außen vor gelassen, könnten auch alle Dateien in einem Verzeichnis liegen.

Die Dateien in den Verzeichnissen sind Textdateien. Zeilentrennzeichen ist allerdings nur 0Ah (Line Feed). Sie müssen daher Windows WordPad statt Notepad benutzen, um ihren Inhalt zu betrachten (rechte Seite in Abbildung 2).

Innerhalb einer Datei können beliebig viele Musik-CDs mit ihren Titeln beschrieben sein. Eine genaue Erläuterung des Dateiformats finden Sie unter [7]. Hier nur einige Stichpunkte:

Das Textformat ist ausschließlich proprietär und nicht XML-basiert. Jede Textzeile enthält ein Name-Wert-Paar (Feld), wobei Name und Wert durch ein „=“ getrennt sind, zum Beispiel

```
DYEAR=1990
TTITLE2=Everybody Needs Somebody...
```

Eine Trennung zwischen Angaben zum Album (Disk) und zu einem Titel des Albums (Track) existiert nicht. Sie ergibt sich nur durch die logische Zuordnung der Feldnamen.

Zu einer Disk gehören die Felder DISCID, DTITLE, DYEAR, DGENRE, EXTD und PLAYORDER.

Zu einem Track gehören die Felder TTITLE und EXTT.

Mangels hierarchischer Struktur der Daten müssen die Felder der vielen Tracks eines Albums mit Indizes versehen sein. Ihren Feldnamen ist daher eine Zahl von 0..n-1 nachgestellt (n = Anzahl der Tracks eines Albums), zum Beispiel TTITLE12, EXTT9.

Darüber hinaus gibt es in den Dateien noch Kommentarzeilen, die mit „#“ beginnen. Ungewöhnlich ist, dass Kommentarzeilen jedoch nicht optional sind und nicht ignoriert werden dürfen. Jede Disk muss vielmehr mit einem speziell formatierten Header von Kommentarzeilen versehen sein, wie es auch in Abbildung 2 zu sehen ist. Der Header gibt die Gesamtlaufzeit der Disk an und spezifiziert den Anfangszeitpunkt jedes Tracks als so genannten Frame Offset. Ein Frame ist 1/72 Sekunde lang.

Warum so wichtige Informationen in Kommentarzeilen versteckt werden, ist

Listing 2

Disk- und Track-Informationen lesen.

```
Imports FreeDb.Import.Traverser

Sub Main()
    Dim r As New FreeDbFileReader _
        ("c:\freedb\blues\f8tof8")
    While r.Read
        Dim d As Disk
        d = r.current

        Console.WriteLine _
            (d.title & " by " & d.artist)

        For Each t As Track In d.tracks
            Console.WriteLine("  " & t.title)
        Next
    End While
    r.Close()
End Sub
```

natürlich unverständlich. Felder wie DLENGTH oder TOFFSET lägen nahe. Aber es hilft nichts, eine Importroutine muss damit leben.

Low-Level-Import

Da die freedb.org-Daten so universell verwendbar sind, können sie immer wieder für unterschiedliche Datenbanktechnologien eingesetzt werden. Mit einem einmaligen Import ist es also nicht getan. Eine Quick-and-Dirty-Lösung verursacht wahrscheinlich schon beim zweiten Import Nachteile. Sie sollten sich daher gut überlegen, wie Sie eine Importlösung aufbauen.

Der Beispielcode auf der Heft-CD definiert dafür mehrere Komponenten. Auf der untersten Ebene geht es zunächst einmal darum, eine freedb.org-Datei mit ihren Disks und Tracks zu lesen. Das Projekt *freedb.Import.Traverser* definiert dafür die Klasse *FreeDbFileReader*.

Im Sinne einer guten objektorientierten Programmierung ist es das erste Ziel, den Zugriff auf eine Textdatei in einer Klasse zu implementieren. Grundsätzlich bietet das .NET Framework dafür zum Beispiel *System.IO.StreamReader*. Aber *StreamReader* ist unspezifisch und zeilenorientiert. Die Dateien von freedb.org haben jedoch eine festgelegte Struktur, bei der zusammenhängende Informationen über mehrere Zeilen verteilt sind.

FreeDbFileReader hebt den Dateizugriff daher auf ein höheres Level und kapselt das Wissen über den genauen Dateiaufbau. Durch einen *FreeDbFileReader* sehen Sie nicht Zeilen in einer Datei, sondern *Disk*-Objekte. Zum *FreeDbFileReader* kommt also noch ein kleines Objektmodell, das jeweils eine Disk mit ihren Tracks repräsentiert.

Listing 2 zeigt, wie einfach die Klasse *FreeDbFileReader* es macht, die Daten in einer Datei zu lesen. Sie erzeugen den Reader für eine Datei und stellen sich auf die nächste Album-Information darin mit der *Read()*-Methode ein. Solange *Read()* *True* zurückliefert, sind noch Daten vorhanden. Auf die greifen Sie mit *current* zu und bekommen ein *Disk*-Objekt, an dem wiederum *Track*-Objekte für die Titel des Albums hängen.

Die Ähnlichkeit des *FreeDbFileReader* mit einem ADO.NET-*DataReader* ist offensichtlich – und natürlich gewollt. Wie ein *DataReader* stellt der *FreeDbFileReader* einen Cursor über Daten dar. Statt Datensätzen als Ergebnis einer SQL-Abfrage sind es jedoch Albuminformationen in einer Textdatei. Dieser Unterschied hat zwar Einfluss darauf, in welcher Weise sich Ihnen die jeweils anliegenden Daten darstellen. Bei einem *DataReader* erhalten Sie Spalten, bei einem *FreeDbFileReader* sind es *Disk*-Objekte. Keinen Einfluss hat der Unterschied allerdings darauf, *wie* Sie die Daten durchlaufen: nur lesend und nur vom Anfang zum Ende (read-only, forward-only). Für den Zweck des Imports ist das völlig ausreichend.

Also bietet es sich an, das Programmiermodell der *DataReader*- oder der Reader-Klassen allgemein, denn es gibt noch andere Klassen im .NET Framework, die dieses Muster implementieren, auch in diesem Fall zu übernehmen. Allgemein sieht das Muster so aus:

```
Dim r as New MyReader(Ressource beschreiben)
While r.Read()
    'Tue etwas mit den anliegenden Daten in r
End While
r.Close()
```

Beim *FreeDbFileReader* stellt die *Read()*-Methode den Cursor ein Album weiter und macht die gelesenen Informationen in einer Objekthierarchie verfügbar.

Listing 3 zeigt, dass die Struktur von *Read()* der Grobstruktur eines Albumeintrags in der Datei folgt.

Bei jedem Aufruf sucht die Methode zuerst mit *SkipToNextDiskEntry()* den nächsten Eintrag und parst ihn anschließend in zwei Schritten. Zuerst werden mit *CompileDiskHeader()* die führenden Kommentarzeilen und die D...-Felder verarbeitet, anschließend mit *CompileTracks()* die TTITLE- und EXTT-Felder für die Tracks.

Die Implementierung der Routinen fürs Parsen birgt keine Geheimnisse. Für Sie sollte vor allem das Muster interessant sein, wie eine Ressource mit proprietä-

Listing 3

Die Read()-Methode ist wie ein Albumeintrag aufgebaut.

```
Public Function Read() As Boolean
    _error = Nothing
    _currentDiskname = ""
    Try
        If SkipToNextDiskEntry() Then
            CompileDiskHeader()
            CompileTracks()
            Return True
        Else
            Return False
        End If
    Catch ex As Exception
        _error = ex
        Return True
    End Try
End Function
```

www.vbarchiv.net
täglich neu und immer aktuell

Developer CD für Visual Basic Entwickler-Komponenten für Visual Basic Entwickler
- 19 Entwickler-Komponenten
- freies Weiterverwenden in eigenen Projekten,
- sowohl privat wie kommerziell
- keine zusätzliche Lizenzgebühren!
- für Visual Basic 5 und 6
- Einzelwert der Komponenten: 1147,50 EUR
- jetzt nur 449,00 EUR*

Developer CD für MS-Access Entwickler-Komponenten für Access-Entwickler
- 12 Entwickler-Komponenten
- freies Weiterverwenden in eigenen Projekten,
- sowohl privat wie kommerziell
- keine zusätzliche Lizenzgebühren!
- für Microsoft Access 97/2000/XP
- Einzelwert der Komponenten: 919,00 EUR
- jetzt nur 369,00 EUR*

Alea jakta est!

vb@rchiv - Die deutschsprachige Internetadresse
SSL-Onlineshop: http://www.vbarchiv.net/shop/
* inkl. MwSt
Jetzt bestellen

rem Inhalt erschlossen werden kann: nämlich als Reader.

Gesamtstruktur traversieren

Mit einem *FreeDbFileReader* durchlaufen Sie die Albumdaten innerhalb einer Datei. Weil die Struktur der Dateien so komplex ist, ist es wichtig, diese Struktur in einer Klasse zu kapseln.

Aber wie durchlaufen Sie alle Dateien? Listing 4 zeigt einen einfachen, grundsätzlich ausreichenden Algorithmus. Da die Dateien in einer Verzeichnishierarchie liegen, genügt es, den Verzeichnisbaum rekursiv zu durchlaufen und auf jeder Ebene die Dateien nacheinander mit einem *FreeDbFileReader* zu öffnen.

Nachteilig an dieser Lösung ist jedoch, dass der importierende Code von der Gesamtstruktur der Daten abhängig ist. Der Code geht von der Voraussetzung aus, dass Album-Informationen in Dateien und diese Dateien in einer Verzeichnisstruktur stehen.

Wenn öfter oder durch verschiedene Personen importiert werden soll, dann wäre es aber einfacher, beim Import keine solchen Detailkenntnisse haben zu müssen. Für denjenigen, der Importcode schreiben muss, wäre es am bequemsten, er müsste ausschließlich wissen, dass alle Daten in einem Verzeichnis liegen und dass er mit *Disk*- und *Track*-Objekten umgehen muss.

Listing 5 demonstriert, wie einfach der Import dann aussehen kann. Auch hier kommt wieder das Reader-Muster zum Einsatz. *FreeDbDirReader* aus dem Beispielcode auf der Heft-CD verbirgt Verzeichnisbaum und Dateien vor dem Importcode und lässt die Alben als Strom von *Disk*-Objekten erscheinen. Woher sie dann genau kommen, ist unerheblich.

Das ist der Gewinn eines Dienstleistungsdenkens im Rahmen der komponentenorientierten Entwicklung. Immer wieder ist die Frage zu stellen: Wie kann ich den Umgang mit Daten oder Algorithmen noch einfacher machen? Wie kann ich die Schnittstelle für den Zugriff simpler, intuitiver gestalten? Das Wissen um die Komplexität wird damit vom Nutzer in eine Komponente verlagert. Die Komponente kapselt den Aufbau hinter einer Schnittstelle (hier: *FreeDbDirReader*) und vielleicht einem Datenmodell (hier: *Disk* und *Track*).

Der Preis der Bequemlichkeit

Die Bequemlichkeit des gekapselten Datenzugriffs mit einem Verzeichnis-Reader hat ihren Preis. Denn das grundsätzliche Problem bei Reader-Klassen ist, dass sie sich von *Read()*- zu *Read()*-Aufruf merken müssen, wo in der zu durchlaufenden Datenstruktur sie stehen.

Beim Algorithmus in Listing 4 ist das kein Problem. Dort wird dieser Zustand implizit durch die *IEnumerator*-Objekte hinter den *For-Each*-Schleifen, den jeweiligen *FreeDbFileReader* und auf dem durch die Rekursion aufgebauten Call Stack gehalten. Und innerhalb des *FreeDbFileReader* ist dafür der *IO.TextReader* verantwortlich, mit dem der Reader zeilenweise über eine Datei wandert.

Um jedoch einen kontinuierlichen Strom von *Disk*-Objekten über Verzeichnisse und Dateien hinweg zu erhalten, muss einiger Aufwand getrieben werden. Denn bei einem *Read()*-Aufruf wird vielleicht das letzte Album aus der Datei *c:\freedb\blues\fetofe* zurückgeliefert. Und beim nächsten muss das erste Album der Datei *c:\freedb\classical\0ato0a* anliegen. Im Sinne einer guten Kapselung müssen solche Details vor importierendem Code verborgen werden.

Die Leistung von *FreeDbFileReader* muss also darin bestehen, eine hierarchische Struktur flach erscheinen zu lassen. Das kann der Reader, indem er sich von Aufruf zu Aufruf der *Read()*-Methode merkt, wo im Strukturbaum er steht. Dessen allgemeiner Aufbau ist:

```
Verzeichnis V1
  Verzeichnis V11
    Datei D111
      Album mit Tracks A1111
      Album mit Tracks A1112
      ...
    Datei D112
      Album mit Tracks A1121
      ...
  Verzeichnis V12
    Datei D121
      Album mit Tracks A1211
      ...
    ...
  ...
```

Er muss die Information verwalten, auf welchem Album in welcher Datei in welchem Verzeichnis er sich befindet. Und wenn das letzte Album in einer Datei gelesen ist, muss er die nächste Datei im Verzeichnis finden. Und wenn die letzte Datei in einem Verzeichnis verarbeitet ist, dann muss er das nächste Verzeichnis auf derselben Ebene finden und so weiter. Der *FreeDbDirReader* löst diese Aufgabe mithilfe einer eingeschachtelten Reader-Klasse und eines Stacks.

Ein so genannter *DirReader* repräsentiert dabei die Position innerhalb einer Datei innerhalb eines Verzeichnisses beziehungsweise eines Unterverzeichnisses. Der Ast, in dem sich der *FreeDbDir-*

Listing 4

Ad-hoc-Code zum Durchlaufen aller freedb.org-Dateien.

```
Imports FreeDb.Import.Traverser

Sub Main()
    ImportAllFreeDbFiles _
        (New IO.DirectoryInfo("C:\freeDb"))
End Sub

Private Sub ImportAllFreeDbFiles _
    (ByVal d As IO.DirectoryInfo)
    For Each f As IO.FileInfo In d.GetFiles
        Dim r As New _
            FreeDbFileReader(f.OpenText())
        While r.Read
            Dim a As Disk
            Try
                a = r.current
                ' Ein Album importieren...
            Catch ex As Exception
                ...
            End Try
        End While
        r.Close()
    Next

    For Each d In d.GetDirectories
        ImportAllFreeDbFiles(d)
    Next
End Sub
```

Listing 5

Alle freedb.org-Daten traversieren.

```
Imports FreeDb.Import.Traverser
...
Dim dr As New FreeDbDirReader("C:\freeDb")
While dr.Read
    Try
        Dim d As Disk = dr.current
        ' Album importieren...
    Catch ex As Exception
        ...
    End Try
End While
dr.Close()
```

Reader gerade befindet, ist durch eine Reihe von aktiven *DirReader*-Objekten auf dem Stack festgehalten.

Da *Read()* nicht rekursiv absteigen kann wie *ImportAllFreeDbFiles()* in Listing 4, muss es den Call Stack explizit durch einen eigenen Stack nachbilden.

Wenn beim *FreeDbDirReader* das Album A1211 anliegt, sieht der Stack dann so aus:

```
DirReader für V12
DirReader für V1
```

Außerdem ist im *DirReader* für V12 notiert, dass die Datei im Zugriff D121 ist. Ein *FreeDbFileReader* steht dann innerhalb der Datei auf Album A1211.

Listing 6 zeigt die Arbeitsweise von *Read()*. Bei jedem Aufruf wird über *_dirStack.Peek* der *DirReader* zuoberst auf dem Stack herangezogen, um über *dr.currentFileReader* an den *FreeDbFileReader* mit dem nächsten Album zu kommen. Die in den meisten Fällen korrekte Annahme ist also, dass *Read()* das nächste Album in derselben Datei findet, in der auch das vorherige Album war.

Listing 6

FreeDbDirReader verwendet beim Aufruf von Read() einen Stack und DirReader-Objekte.

```
Public Function Read() As Boolean
    Do
        Dim dr As DirReader
        If _dirStack.Count > 0 Then
            dr = _dirStack.Peek

            Dim fr As FreeDbFileReader = dr.currentFileReader
            If Not fr Is Nothing AndAlso fr.Read Then
                _disk = fr.current
                Return True
            Else
                If Not dr.ReadFile Then
                    If dr.ReadSubdir Then
                        _dirStack.Push(New DirReader(dr.currentSubdir))
                    Else
                        dr.Dispose()
                        _dirStack.Pop()
                    End If
                End If
            End If
        Else
            Return False
        End If
    Loop
End Function
```

Alle Dateien durchlaufen

Falls die aktuelle Datei jedoch erschöpft ist, versucht *Read()* es mit *dr.ReadFile* bei der nächsten Datei des aktuellen *DirReaders* und fängt nochmal von vorn an. Das erklärt die *Do-Loop*-Schleife in dieser Routine.

Sind alle Dateien im Verzeichnis des aktuellen *DirReaders* abgearbeitet, dann versucht *Read()* es mit *dr.ReadSubDir()* beim nächsten Unterverzeichnis. Damit steigt die Methode jedoch im Verzeichnisbaum tiefer, das heißt, es muss ein neuer *DirReader* für das Unterverzeichnis erzeugt und auf den Stack gelegt werden:

```
If dr.ReadSubdir Then
    _dirStack.Push(New DirReader( _
dr.currentSubdir))
Else
    dr.Dispose()
    _dirStack.Pop()
End If
```

Falls auch alle Unterverzeichnisse des aktuellen *DirReaders* behandelt sind, ist die Verzeichnisbaumebene abgearbeitet und *Read()* kann wieder eine Ebene höher steigen, indem es mit *dr.Dispose()* den *DirReader* schließt und mit *_dirStack.Pop()* vom Stack entfernt.

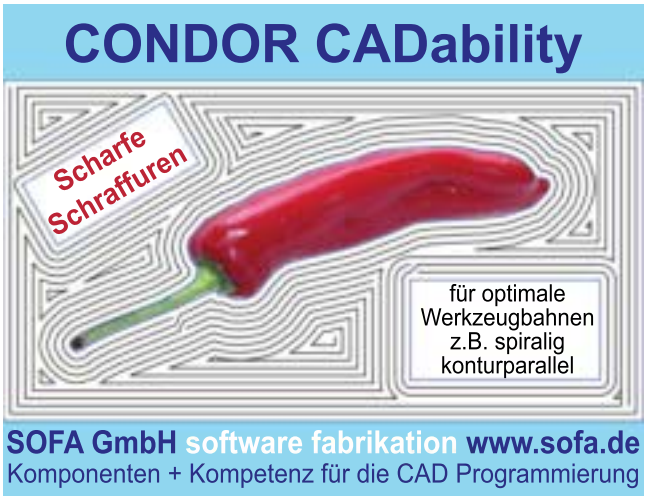
In jedem Fall startet *Read()* aber wieder von vorn, um mit dem jetzt auf dem Stack zuoberst liegenden *DirReader* an die nächste Datei und das nächste Album zu gelangen.

Die Arbeitsweise des *FreeDbDirReader* ist also nicht einfach, da eine Baumstruktur und auch noch verschiedene Speicherstrukturen (Verzeichnis, Datei, Liste von Alben) zu durchlaufen sind. Dafür ist das sich ergebende Programmiermodell einfach – und das ist das Ziel jeder guten Komponente: Komplexität hinter einer einfachen Fassade zu verstecken.

Für Reader-Klassen lässt sich allgemein ableiten: *Read()* muss über mehrere Aufrufe hinweg den Zustand darüber erhalten, wo in der zu durchlaufenden Ressource der Cursor jeweils steht. Bietet die Zugriffstechnologie einer Ressource selbst einen solchen Zustand, wie etwa ein Stream-Objekt, dann ist die Implementierung eines Readers einfach. Ansonsten muss eine Art Zeiger in die Ressource künstlich konstruiert werden. Wie komplex der Zeiger wird, hängt von der Komplexität der Ressourcenstruktur ab.

Einfach und doch bequem?

Ein Strom von Objekten, der in einer Schleife ausgelesen wird, ist wahrscheinlich die einfachste Abbildung für eine Ressource, die Sie importieren wollen.



Listing 7

Ein ereignisgetriebener Import während des rekursiven Abstiegs im Verzeichnisbaum.

```
Public Class FreeDbDirTraverser
    Public Event OnDisk _
        (ByVal disk As Disk, ByRef stopTraversal As Boolean)

    ...

    Private _rootDir As IO.DirectoryInfo
    Private _traversalCancelled As Boolean

    ...

    Public Sub Traverse(ByVal path As String)
        _traversalCancelled = False
        Traverse(New IO.DirectoryInfo(path))
    End Sub

    Private Sub Traverse(ByVal dir As IO.DirectoryInfo)
        If Not dir Is Nothing Then
            For Each f As IO.FileInfo In dir.GetFiles()
                Dim r As New FreeDbFileReader(f.OpenText())
                While r.Read
                   RaiseEvent OnDisk(r.current, _traversalCancelled)
                End While
                r.Close8)
                If _traversalCancelled Then Exit Sub
            Next

            For Each d As IO.DirectoryInfo In dir.GetDirectories
                Traverse(d)
                If _traversalCancelled Then Exit Sub
            Next
        End If
    End Sub
End Class
```

Unter Umständen müssen Sie dafür jedoch einigen Aufwand in einem eigenen Reader treiben. Es stellt sich daher die Frage, ob es nicht auch einfacher geht. Gibt es einen anderen Weg, um Importcode und Kenntnis über Datenstrukturen zu trennen?

Die Antwort lautet: Ja – wenn Sie bereit sind, Datentraversierung und Import zu trennen. Die Intuitivität einer Schleife über Daten, innerhalb derer sie verarbeitet werden, ist dann aufzugeben. Statt sequenziell arbeiten Sie dann ereignisgetrieben.

Listing 7 zeigt die Klasse *FreeDbDirTraverser* des Beispielcodes auf der Heft-CD, die dieses Vorgehen implementiert.

Listing 8

Ereignisgetriebener Import von Alben.

```
Imports FreeDb.Import.Traverser

Sub Main()
    Dim t As New FreeDbDirTraverser
    AddHandler t.OnDisk, AddressOf OnDisk
    ...
    t.Traverse _
        ("C:\freeDb Database\freedbsmall")
End Sub

Private Sub OnDisk (ByVal disk As Disk, _
    ByRef stopTraversal As Boolean)
    ' Importieren eines Albums...
End Sub
```

Die Grundfunktion ist dieselbe wie in Listing 4. *Traverse()* steigt den Verzeichnisbaum der freedb.org-Daten rekursiv hinab. Aber *Traverse()* enthält keinen Importcode. Stattdessen feuert die Methode für jedes gefundene Album das Ereignis *OnDisk()*.

Die Klasse verbirgt also wie der *FreeDbDirReader* die genaue Struktur der Daten vor aufrufendem Code. Dort jedoch ist die Datentraversierung durch den Aufruf von *Traverse()* vom Import eines Albums getrennt. Der Import findet in einer Ereignisbehandlungsroutine statt, wie es Listing 8 zeigt.

Der *FreeDbDirTraverser* verbirgt die genaue Datenstruktur genauso gut vor dem importierenden Code wie der *FreeDbDirReader*. Im Eventhandler kommen nacheinander einfach alle Alben an, egal wo sie in welcher Datei in welchem Verzeichnis stehen. Und *Traverse()* ist einfach zu realisieren, weil die Zustandshaltung wieder implizit erfolgt.

Wenn Sie das ereignisgetriebene Programmiermodell auch in solchen Szenarien, die nichts mit Benutzerschnittstellen zu tun haben, mögen, dann ist Ihnen diese Strategie für die Kapselung des Zugriffs auf Ressourcen wahrscheinlich sympathischer als ein Reader – zumindest wenn die Zustandshaltung im Reader nicht trivial ist.

Ressourcen-Objekte

Reader sind Objekte, die auf Ressourcen zugreifen – zum Beispiel auf Dateien oder Hardware-Schnittstellen. Es ist also wichtig, dass diese Ressourcen nach Gebrauch wieder freigegeben werden. Da Managed-Code-Objekte jedoch vom Garbage Collector zu unbestimmter Zeit nach Aufgabe der letzten Referenz gelöscht werden, verbietet sich eine Ressourcenfreigabe im Destruktor.

Microsoft empfiehlt daher das Implementieren einer Finalisierungsmethode und des *IDisposable-Musters [8]*. Diesen Vorschlag setzen auch die Reader des Beispielcodes auf der Heft-CD um. Sie leiten

Listing 9

Auf Ressourcen zugreifen und für die Freigabe sorgen.

```
Public Class FreeDbFileReader
    Inherits dnp.ResourceObject.ResourceObject

    ...

    Protected Overrides Sub _
        DisposeManagedResources()
        If Not _freeDbFile Is Nothing _
            Then _freeDbFile.Close()
    End Sub

    Protected Overrides Sub _
        DisposeUnmanagedResources()
    End Sub

    Public Sub Close()
        Me.Dispose()
    End Sub
End Class
```

Listing 10

Vom Persistenzmedium unabhängiger Importalgorithmus.

```
Public MustInherit Class Store
    …
    Private Shared _nImported, _nErr _
        As Integer

    Public Shared Sub Import(ByVal path _
      As String, ByVal importStore As Store)
        _nErr = 0
        _nImported = 0
        Try
            Dim dr As New _
                FreeDbDirReader(path)
            While dr.Read
                Try
                   Dim d As Disk = dr.current
                   importStore.ImportDisk(d)
                   _nImported += 1
                Catch ex As Exception
                    _nErr += 1
                End Try
            End While
            dr.Close()
        Finally
            importStore.Close()
        End Try
    End Sub
End Class
```

Listing 11

Schnittstelle für Store-Objekte.

```
Public MustInherit Class Store
    Inherits dnp.ResourceObject.ResourceObject

    Public MustOverride Sub Setup(ByVal template As String, ByVal destination As String)
    Public MustOverride Sub ImportDisk(ByVal disk As FreeDb.Import.Traverser.Disk)
End Class
```

Listing 12

Import von Daten mit dem Store für MDB-Datenbanken.

```
Imports FreeDb.Import.Store

Sub Main()
    Dim s As Store.Store = New MDBStore
    s.Setup("freedb template.mdb", "freedb.mdb")
    s.Import("C:\freeDb Database\freedbsmall", s)
End Sub
```

sich daher von der Basisklasse *dnp.ResourceObject.ResourceObject* ab, die das Muster versteckt und von Ressourcenklassen stattdessen verlangt, zwei virtuelle Methoden zu implementieren (*DisposeManagedResources()* und *DisposeUnmanagedResources()*).

Listing 9 demonstriert das am Beispiel des *FreeDbFileReader*. Nach Gebrauch rufen Sie *Dispose()* beziehungsweise *Close()* auf, um die *TextReader*-Ressource freizugeben. Falls Sie ein Reader-Objekt jedoch in einer *C#-using*-Anweisung benutzen, wird *Dispose()* automatisch an deren Ende aufgerufen.

Import von Albumdaten

Im Beispielcode auf der Heft-CD ist auch für den Import von Albumdaten ein Rahmenwerk zu finden. Da der Algorithmus unabhängig vom Ziel-Persistenzmedium ist, kann er wie in Listing 10 immer gleich aussehen.

Die Methode *Import()* durchläuft mit einem *FreeDbDirReader* alle Alben und ruft für jedes Album die Methode *ImportDisk()* auf einem *Store*-Objekt auf.

Ein *Store*-Objekt steht für ein Persistenzmedium, ist von der Klasse *Store* abgeleitet und liegt im Beispielcode für MDB-Datenbanken, SQL Server und FastObjects vor.

Store ist als abstrakte Klasse definiert und verlangt die Implementierung von nur zwei Methoden pro Persistenzmedium, wie es Listing 11 zeigt. *Setup()* initialisiert das Persistenzmedium und *ImportDisk()* speichert darin ein Album.

Wenn Sie freedb.org-Daten importieren möchten, wählen Sie also entweder einen der vorhandenen Stores aus oder implementieren selbst einen. Anschließend ist der Import wie in Listing 12 ein Dreizeiler.

Setup() hat zwei Parameter, weil es annimmt, dass das Zielmedium irgendwo vorbereitet als leeres Template liegt und vor dem Füllen an einen anderen Ort kopiert werden muss. Der *MDBStore* macht das exemplarisch vor.

Fazit

freedb.org liefert eine sehr große Menge an Daten in einer nicht trivialen Struktur, die sich gut für Testzwecke nutzen lässt. Angesichts des universell einsetzbaren Charakters der Daten lohnt es sich, ein allgemeines, robustes Importwerkzeug zu entwickeln.

Um es leicht handhabbar zu machen, sollte das Werkzeug die genaue Struktur der Daten vor einem Nutzer verbergen. Dafür bietet sich vor allem das Reader-Muster an, bei dem die Daten als Strom verwaltet werden.

Sind sie genügend komplex wie in diesem Fall, lohnt sich darüber hinaus eine Verpackung in ein Objektmodell. Je nachdem, wie komplex die Zustandshaltung in einem Reader wäre, lohnt sich alternativ die Implementierung einer Traversierungskomponente entsprechend dem ereignisgetriebenen Programmiermodell.

[1] www.freedb.org
[2] www.cddb.com
[3] Zugriff auf die freedb.org Datenbank, www.freedb.org/modules.php?name=Sections&sop=viewarticle&artid=30
[4] Herunterladen der freedb.org-Datenbank, www.freedb.org/modules.php?name=Sections&sop=viewarticle&artid=12
[5] RAR-Archiv Dekomprimierer WinRAR, www.win-rar.com
[6] FastObjects, www.versant.net/de_de/solutions/dotNet_de
[7] Beschreibung des freedb.org Dateiformats, www.freedb.org/modules.php?name=Sections&sop=viewarticle&artid=29
[8] .NET Framework Resource Management, gotdotnet.com/team/libraries/whitepapers%5Cresourcemanagement%5Cresourcemanagement.aspx