

Alternative Datenhaltung

Datenstrukturen im .NET Framework – Teil 2

von Marcel Gnoth

Im ersten Teil des Artikels standen die Collection-Klassen des .NET Frameworks im Vordergrund. Im zweiten möchte ich Ihnen zeigen, wie Sie eigene Collection-Klassen bauen können, ohne auf die abstrakten Klassen des Frameworks zurückgreifen zu müssen. Anschließend sollen das Thema Sortieren und zwei Alternativen zur Datenhaltung – das ADO.NET Dataset und XmlDocument – zur Sprache kommen. Zum Abschluss stellen wir einige Betrachtungen zur Performance der einzelnen Collection-Klassen an.

Wer die vorgegebenen Pfade des Frameworks verlassen will und sich etwas ganz Spezielles „basteln“ möchte, dem soll hier geholfen werden. Nur in wenigen Situationen sollte es notwendig sein, eigene Collection-Klassen von Grund auf zu definieren. Schauen Sie sich vorher noch einmal die vielfältigen Möglichkeiten von *ArrayList*, *Hashtable* und *SortedList* an. Dort gibt es Methoden wie *ContainsKey*, *ContainsValue*, *IndexOf* und vieles mehr.

Wenn Sie dennoch eine eigene Collection-Klasse implementieren wollen, dann müssen Sie sich als erstes überlegen, wie Sie intern Ihre Daten halten wollen. Ob nun als Baum, verkettete Liste oder einfach nur als Array. Im folgenden Beispiel verwende ich ein String-Array. Dann müssen Sie sich entscheiden, welche Interfaces Sie implementieren wollen. Listing 1 zeigt Ihnen, wie Sie die Basis-Interfaces *IEnumerable* und *IEnumerator* implementieren, sodass Sie Ihre Collection mit *foreach* durchlaufen können.

Sie finden in dem Listing zwei Klassen. Eine Klasse, die äußere, ist die Collection-Klasse. Die innere Klasse implementiert den Enumerator, der für *foreach* benötigt wird. In dem String-Array *elements* speichert die Klasse *Tokens* ihre Daten. Der Konstruktor erwartet einen String und einen Delimiter, mit denen das interne Array gefüllt wird. *IEnumerable* verlangt, dass die Methode *GetEnumerator* implementiert wird. Die Methode gibt eine Objektinstanz der inneren Klasse *TokenEnumerator* zurück. Methoden wie *Add*, *Remove*, *Clear*,... können Sie analog wie oben bei den abstrakten Basisklassen be-

schrieben implementieren. Aber wie funktioniert nun so ein Enumerator?

Ein Enumerator muss folgende Methoden und Eigenschaften implementieren: *Current*, *MoveNext*, *Reset*. Nach jeder Iteration ruft *foreach* *MoveNext* auf

und holt sich anschließend mit *Current* das aktuelle Element. Intern müssen Sie nun aufpassen, dass Sie immer das nächste Element übergeben. Dafür ist die Variable *position* der Klasse *TokenEnumerator* verantwortlich. Bei *MoveNext* wird

Listing 1

```
using System;
using System.Collections;
namespace DataStruct {
    // Tokens Collection Klasse, ist aufzählbar
    public class Tokens : IEnumerable {
        private string[] elements; //enthält die einzelnen Items
        //Konstruktor
        public Tokens(string source, char[] delimiters) {
            //Zerteilt String mittels des Delimiters in einzelne
            Tokens (Items)
            elements = source.Split(delimiters);
        }
        //IEnumerable Interface Implementation:
        // Deklaration von GetEnumerator()
        // für IEnumerable
        public IEnumerator GetEnumerator() {
            return new TokenEnumerator(this);
        }
        #region TokenEnumerator
        /// <summary>
        /// Innere Klasse, die IEnumerator implementiert
        /// eine Instanz dieser Klasse wird rausgegeben
        /// wenn ein Enumerator von Tokens verlangt wird
        /// </summary>
        private class TokenEnumerator : IEnumerator {
            //aktuelle Pos bei For Each
            private int position = -1;
            //private Kopie der Referenzen die die org Tokens Klasse
            //hält
            private Tokens t;

            //Konstruktor bekommt das zu iterierende Objekt
            // (Tokens) übergeben
            //und speichert eine Referenz lokal zwischen

            public TokenEnumerator(Tokens t) {
                this.t = t;
            }

            // MoveNext für IEnumerator,
            // wird bei jedem ForEach Durchlauf aufgerufen
            // inkrementiert den Positionszeiger,
            // dadurch gibt die Property Current jedesmal
            // ein neues Objekt zurück
            public bool MoveNext() {
                if (position < t.elements.Length - 1) {
                    position++;
                    return true;
                }
                else {
                    return false;
                }
            }

            // Reset für IEnumerator, position wird zurückgesetzt
            public void Reset() {
                position = -1;
            }

            // Current Property für IEnumerator, wird bei jedem
            // ForEach Durchlauf nach dem MoveNext aufgerufen
            public object Current {
                get {
                    return t.elements[position];
                }
            }
        }
        #endregion
    } // Class Tokens
} // Namespace
```

sie so lange inkrementiert, bis das Ende des Arrays erreicht ist (dann wird *false* zurückgegeben), bei *Reset* wird *position* auf *-1* zurückgesetzt. *Current* benutzt diese Variable als Index für den Zugriff auf das interne String-Array der Klasse *Tokens*. Die Implementierung von *ICollection* sollte Ihnen nicht schwer fallen. Für die Eigenschaft *Count* brauchen Sie einfach nur die *Length*-Property des Arrays durchzureichen und für *CopyTo* verwenden Sie die gleichnamige Methode des Arrays. Die Synchronisation bei Multithreaded Anwendungen ist dann aber

Listing 2

```
public class CustSortByCountry : IComparer {
    public int Compare(object obj1, object obj2) {
        CCustomer cu1 = (CCustomer) obj1;
        CCustomer cu2 = (CCustomer) obj2;
        return String.Compare(cu1.Land, cu2.Land);
    }
}

public class CustSortByAge : IComparer {
    public int Compare(object obj1, object obj2) {
        CCustomer cu1 = (CCustomer) obj1;
        CCustomer cu2 = (CCustomer) obj2;
        if (cu1.Alter > cu2.Alter)
            return (1);
        if (cu1.Alter < cu2.Alter)
            return (-1);
        else
            return (0);
    }
}
```

Listing 3

```
DataSet dsCustomers; DataTable dt; DataColumn dc;
dsCustomers = new DataSet("Customers");
dt = dsCustomers.Tables.Add("Customer");

dc = dt.Columns.Add("ID", System.Type.GetType(
    "System.Int32"));
dc.AllowDBNull=false; //dc.Unique = true;
dc.AutoIncrement = false;
dt.Constraints.Add("PrimaryKey", dc, true);

dt.Columns.Add("Key",
    System.Type.GetType("System.String"));
dt.Columns.Add("Nachname",
    System.Type.GetType("System.String"));
dt.Columns.Add("Vorname", System.Type.GetType(
    "System.String"));

dt.BeginLoadData();
foreach (CCustomer xc in arrCustomer) {
    dt.LoadDataRow( new object[] {xc.ID, xc.Key,
        xc.Nachname, xc.Vorname}, true);
}
dt.EndLoadData();
```

noch einmal ein Thema für sich. Und dann sind Sie ja schon bei *IList* oder bei *IDictionary* angelangt.

Ordnung muss sein...

...sagte meine Oma schon immer! Also lassen Sie uns jetzt einmal einen Blick darauf werfen, wie man in diesem Framework sortiert. Egal, ob Sie nun die Objekte eines Arrays oder eines ListViews sortieren wollen, Sie werden immer wieder auf folgende Interfaces stoßen: *IComparable* und *IComparer*. Der Unterschied besteht darin, dass Klassen *IComparable* für sich selbst implementieren, um sich selbst mit einer anderen Objektinstanz zu vergleichen. So kann man aber normalerweise nur eine Art von Sortierung implementieren. Möchte man mehrere Sortierarten, wie z.B. nach Alter, Name und Geburtsort, dann verwendet man mehrere kleine Hilfsklassen, die jeweils *IComparer* implementieren. So eine Klasse vergleicht dann zwei Kunden-Objekte miteinander. Den meisten Sortiermethoden wird entweder gar nichts übergeben – dann wird vorausgesetzt, dass die Objekte *IComparable* implementieren – oder diese Methoden bekommen als Argument eine *IComparer*-Klasse übergeben, die dann die Reihenfolge der Objekte festlegt.

Das Interface *IComparable* wird direkt in der Klasse implementiert. Die einzige Methode *CompareTo* vergleicht ein übergebenes Objekt mit der eigenen Instanz und liefert dann *-1*, *0* oder *1* zurück:

```
public class CCustomer : IComparable {
    int IComparable.CompareTo(object oCu) {
        CCustomer Cu = (CCustomer) oCu;
        return String.Compare(this.Caption, Cu.Caption);
    }
}
```

IComparable wird meist in Hilfsklassen innerhalb der zu sortierenden Klasse implementiert. Beim Sortieren wird eine solche *IComparer*-Klasse übergeben, die dann zwei Objekte vergleicht und auch wieder *-1*, *0* oder *1* zurückgibt. Die einzige Methode, die implementiert werden muss, ist *Compare*. Listing 2 zeigt ein Beispiel. Durch den Aufruf *colArList.Sort(new CCustomer.CustSortByCountry());* wird zum Beispiel eine *ArrayList* mit *CCustomer*-Objekten sortiert, wobei *CustSortByCountry* eine Unterklasse von *CCustomer* ist.

Alternativen zu Collection-Klassen

Die Daten Ihres Programms müssen nicht immer in Collection-Klassen gehalten werden. Es gibt zwei interessante Alternativen, die Ihnen auch ganz andere Möglichkeiten bieten: das ADO.NET *Dataset* und *XMLDocument*. In beiden Fällen speichern Sie nicht Referenzen auf Objekte, sondern direkt Ihre Daten. Das *Dataset* eignet sich sehr schön dazu, größere Datenmengen in einer Art „in memory“-Datenbank zu speichern. Sie können mehrere Tabellen anlegen, Relationen zwischen den Tabellen und Beschränkungen wie *unique* und *foreign key* definieren, die Daten sortieren und filtern und Views bilden – wie in einer richtigen kleinen Datenbank. *Datasets* können auch auf der Festplatte persistiert werden.

XMLDocument-Objekte eignen sich für Daten, die in hierarchischer Form (Eltern-Kind-Beziehung) vorliegen. Auch auf XML-Dokumente kann mit einer Art SQL (XPath) zugegriffen werden und mit XSLT kann aus XML-Dokumenten deklarativ eine schicke Webseite erzeugt werden. XML-Dokumente sind wegen der vielen öffnenden und schließenden Tags aufgebläht, aber für Menschen gut les- und editierbar. Beide Klassen bieten Ihnen vielfältige Möglichkeiten, auf die Daten zuzugreifen.

ADO.NET Dataset

Über das Dataset wird und wurde schon viel geschrieben. Sie können es benutzen, ohne je mit einer Datenbank in Verbindung gestanden zu haben, und füllen es

Listing 4

```
s = String.Format („Customer[Key = \"{0}\"]“, xKey);
XmlNode xn = xmlCustomers.DocumentElement.
    SelectSingleNode(s);
s = „/Customers/Customer[Land = ‚PERU‘]“;
XmlNodeList xl = xmlCustomers.SelectNodes(s);

<Customers>
<Customer>
<ID>0</ID>
<Key>K0</Key>
<Nachname>Fox</Nachname>
<Vorname>Lisa</Vorname>
<Land>BAHRAIN</Land>
<Alter>39</Alter>
</Customer>
<Customer>
.....
</Customer>
</Customers>
```

mit Daten Ihrer Wahl. Bei dem alten ADO Recordset ging das auch schon sehr gut.

In Listing 3 sehen Sie wie es geht. Zuerst instanziiieren Sie ein neues *DataSet*-Objekt, fügen dann eine oder mehrere *DataTable*-Objekte hinzu und definieren *DataColumns*. Die Spalte *ID* wird als Primärschlüssel definiert, der keine *Null*-Werte erlaubt.

BeginLoadData beschleunigt das Laden von Daten, da intern einige Mechanismen abgeschaltet werden. *DataTable.LoadDataRow* oder *DataTable.Rows.Add* fügen Datensätze an.

Nun können Sie *DataViews* definieren, die Daten filtern und sortieren. Für den Zugriff gibt es verschiedene Methoden:

- *DataTable.Rows.Contains*
- *DataTable.Rows[Index]*
- *DataRow = DataTable.Rows.Find*
- *DataRow[] = DataTable.Select*

XMLDocument

XML-Dokumente können leicht über das Microsoft Document Object Model (DOM) oder als Textdatei erzeugt werden. Das *DataSet* kann auch als XML-Dokument gespeichert werden (intern wird im *DataSetXML* verwendet).

Für das Lesen haben Sie drei Varianten zur Verfügung. Sie können eine XML-Datei wie eine normale Textdatei öffnen und

selber parsen oder zwei fertige Parser verwenden. Der MS-XML Parser liest ein komplettes XML-File (oder String) ein und baut daraus im Speicher ein Objektmodell mit Eltern-Kind-Beziehungen auf. Bei großen Dokumenten dauert das natürlich, da sehr viele Objekte instanziiert werden müssen. Dafür können Sie dann mit XPath einzelne Knoten selektieren und gezielt auf bestimmte Elemente zugreifen.

Die Alternative besteht im SAX-XML-Parser. Dieser Parser ist Event-gesteuert. Sie teilen ihm mit, für welche Tags Sie sich interessieren, er liest sich dann das XML-Dokument durch und „feuert“ ein Event, wenn er ein solches Tag findet. Dabei wird im Speicher kein Objektmodell aufgebaut und viel Zeit und Speicher gespart.

In Listing 4 sehen Sie, wie Sie einzelne Knoten mit XPath selektieren. Dabei ist ein Select über mehrere Knoten wesentlich effizienter als die Ausführung mehrerer Aufrufe von *SelectSingleNode*. Unten sehen Sie einen Ausschnitt des XML-Files, auf das sich der Code bezieht.

Performance-Betrachtungen

Nicht ganz uninteressant ist das Laufzeitverhalten der einzelnen Collection-Klassen, wenn sie viele Objekte enthalten und häufig auf sie zugegriffen wird. Aus diesem Grund habe ich ein kleines Testprogramm geschrieben, welches die wichtigs-

ten Operationen (*Add*, *Random Access*, *Delete* und *Sort*) mit vielen Objekten getestet. Ich habe aus jeweils drei Versuchen den Mittelwert gebildet, um Schwankungen bei den Messwerten auszugleichen. Da sich die Werte größtenteils im Millisekundenbereich befinden, spielen auch kleine Abweichungen bereits eine Rolle. Wieder einmal bewahrheitet sich: „Wer misst, misst Mist!“. Das liegt zu einem guten Teil an der Garbage Collection, die im Hintergrund arbeitet und gelegentlich Speicher freigibt. Führt sie diese Arbeiten während der Messung durch, dauert die Operation entsprechend länger.

Lade-Operationen

Es zeigt sich, dass die indexbasierten Collection-Klassen schneller als alle anderen sind. Das Array ist ein bisschen langsamer als die *ArrayList*, weil ich *SetValue* für das Setzen der Objektreferenzen benutzt habe. Muss ein Zugriff über einen Schlüssel erfolgen, dann ist die *Hashtable* immer noch die beste Wahl, wie Tabelle 3 zeigt. Die alte VB-Collection ist drei mal langsamer. Collection-Klassen, wie *SortedList* und *NameValueCollection*, die beim Einfügen spezielle Sortierungen vornehmen, haben mehr zu tun und brauchen deshalb wesentlich länger. Das *ListDictionary* lief im 30.000er Test bereits außer Konkurrenz. Meine Teepause war zu Ende, als das Einfügen noch lief. Das *DataSet* habe ich Zeile für Zeile aufgebaut, was auch seine Zeit braucht. Aber oft kommen die Daten sowieso in Form eines *Dataset*s aus der Datenbank und dann lohnt es sich nicht, erst noch eine Collection daraus aufzubauen, sondern man behält die Daten gleich im *DataSet*. Sie können das *DataSet* auch in einer extra Klasse verstecken, die nach außen wie eine Collection aussieht. Property-Prozeduren greifen auf die einzelnen Spalten des *Dataset*s zu. Das hat außerdem den Vorteil, dass Sie Änderungen wieder direkt mit dem *DataSet* in der Datenbank speichern können.

Der Aufbau eines XML-Dokuments geht auch nicht gerade schnell, da bei sehr großen Datenmengen der Overhead von XML schnell zu groß wird. Aber vergessen Sie nicht, wir reden hier über das Laden von 30.000 Objekten auf einmal. Ihre Anwendung sollte normalerweise nicht dazu tendieren, so viele Datensätze zu laden, so viel kann sich der Anwender

| Datenstruktur | 1.000 Objekte | 30.000 Objekte |
|-----------------------------|---------------|----------------|
| ArrayFixedLength (SetValue) | 0,00 | 20,00 |
| ArrayList | 0,00 | 10,00 |
| Queue (enqueue) | 0,00 | 13,33 |
| Stack (push) | 0,00 | 10,00 |
| HashTable | 0,00 | 53,67 |
| SortedList | 10,00 | 2092,67 |
| ListDictionary | 40,00 | – |
| HybridDictionary | 0,00 | 60,33 |
| NameValueCollection | 10,00 | 190,33 |
| VBCollection | 3,33 | 173,33 |
| CCustDicBase | 0,00 | 57,00 |
| CustColBase | 0,00 | 13,33 |
| CustNamedObjColBase | 3,33 | 597,67 |
| StringCollection (Strings) | 0,00 | 20,00 |
| StringDictionary (Strings) | 0,00 | 86,67 |
| DataSet (LoadDataRow) | 20,00 | 1595,67 |
| DataSet (Rows.Add) | 13,33 | 1001,33 |
| XMLDocument (aus DataSet) | 190,67 | 6703,00 |

Tabelle 1: Gemessene Zeiten beim Hinzufügen von Objekten zu einer Collection-Klasse

sowieso nicht auf einmal sinnvoll ansehen (siehe Tabelle 1).

Auch hier zeigt sich wieder, dass ein Zugriff über Index immer performanter ist als einer über einen Schlüssel. Besonders gut kann man das bei Collection-Klassen, die beide Zugriffsarten erlauben, beobachten. *SortedList.GetByIndex* ist 100 mal schneller als der Zugriff über einen Schlüssel. Bei der „alten“ VB6-Collection ist es immerhin noch der Faktor 3. Die Hashtable hält wieder den ersten Platz bei den schlüsselbasierten Klassen.

Bei *Stack* und *Queue* kann immer nur das erste oder das letzte Element entnommen werden, was die Geschwindigkeit erklärt. Bei den anderen Collection-Klassen wurde wahllos auf Elemente zugegriffen. Das *ListDictionary* taucht wieder nicht auf, da es viel zu langsam ist.

Interessant sind die beiden alternativen Vertreter. Der Zugriff auf das Dataset kann durchaus mit dem Zugriff auf die Collection-Klassen mithalten, seine Stärken entfaltet es aber erst bei einem gleichzeitigen Filtern (Select) und Sortieren (Order By), wobei die Sortieroperation einen großen Teil der Zeit eingenommen hat. Bei *XMLDocument* zeigt sich, dass *SelectSingleNode* eine sehr teure Operation ist: ich habe nach 150 Zugriffen aufgehört, aber *SelectNodes* hat innerhalb kürzester Zeit 196 Knoten gefunden, die einem bestimmten Kriterium entsprechen (siehe Tabelle 2).

Das Sortieren mit den *IComparer*-Implementierungen hat bei Strings im Schnitt 500 ms und bei Zahlen 100 ms gedauert. Das Dataset war also gar nicht so schlecht, es musste ja auch weniger sortie-

ren, da es erst die Filteroperation durchgeführt hat.

Benötigen Sie Ihre Collection für weniger als 1.000 Objekte, dann spielt die Auswahl aus Performance-Sicht keine so große Rolle. Wollen Sie tausende von Objektreferenzen speichern, dann wird das Design Ihrer Anwendung immer wichtiger. Es gilt: „Die richtige Auswahl des Collection-Typs ersetzt kein gutes Anwendungsdesign!“.

Zusammenfassung

Wenn Sie sich zum ersten Mal mit dem Thema Datenstrukturen unter .NET beschäftigen, dann sind Sie jetzt wahrscheinlich ein bisschen erschlagen von der Vielzahl der Optionen. Deshalb möchte ich Ihnen hier abschließend noch einmal kurz die wichtigsten Typen auflisten.

Reicht Ihnen ein Zugriff über Index, dann ist die *ArrayList* die beste Wahl, brauchen Sie Schlüsselwerte, dann sollten Sie sich für die *Hashtable* entscheiden. Wenn Sie beides wollen, dann werfen Sie einen Blick auf die *SortedList* – aber Achtung, Einfügeoperationen sind entsprechend langsam und Sortierfunktionalität bietet Ihnen auch die *ArrayList*. Die *SortedList* verfügt aber über einige komfortable Zugriffsmöglichkeiten.

Müssen Sie nur Strings speichern, dann wählen Sie die dafür vorgesehenen Collection-Klassen *StringDictionary* und *StringCollection*. Wollen Sie eigene Collection-Klassen definieren, dann sind *CollectionBase* und *DictionaryBase* die Mittel Ihrer Wahl. Bevor Sie anfangen, eine Collection-Klasse von Grund auf selbst zu implementieren, schauen Sie sich erst einmal die Eigenschaften und Methoden der existierenden Collection-Klassen an. Die bieten Ihnen normalerweise alles, was Sie brauchen. Was im aktuellen Framework fehlt, sind Baum-Strukturen. Brauchen Sie so etwas, dann müssen Sie selbst Hand anlegen.

Wenn Ihr Interesse nicht primär darin besteht, Objektreferenzen zu halten, sondern Daten zu speichern, dann behalten Sie auch das *Dataset* und *XMLDocument* im Auge.

Marcel Gnoth ist Senior Consultant bei der Berliner NTeam GmbH. Seine Arbeitsschwerpunkte liegen im Bereich COM, .NET und verteilte Informationssysteme. Sie erreichen ihn unter mgnoth@nteam.de oder www.gnoth.net.

| Datenstruktur | Operation | 30.000 Zugriffe |
|--------------------------------------|--|-----------------|
| Array (Index) | Indexer | 0,00 |
| ArrayList (Index) | Random Access Indexer | 10,00 |
| Queue (Dequeue) | Dequeue | 3,33 |
| Stack (Pop) | Pop | 3,33 |
| HashTable (Key) | Random Access ByKey | 43,33 |
| HashTable (ContainsKey) | Random Access ContainsKey | 26,67 |
| SortedList (Key) | Random Access by Key | 387,67 |
| SortedList (GetByIndex) | Random Access GetByIndex | 3,33 |
| SortedList (ContainsKey) | Random Access ContainsKey | 377,33 |
| HybridDictionary (Key) | Random Access ByKey | 37,00 |
| HybridDictionary (ContainsKey) | Random Access Contains a Key | 30,00 |
| NameValueCollection (Index) | Random Access By Index | 20,00 |
| NameValueCollection (Key) | Random Access By Key | 70,00 |
| NameValueCollection (GetKey) | Random Access GetKey | 10,00 |
| VB-Collection (Index) | Random Access By Index | 50,00 |
| VB-Collection (Key) | Random Access By Key | 130,00 |
| CCustomersDicBase (Key) | Random Access by Key | 36,67 |
| CCustomersColBase (Index) | Random Access by Index | 10,00 |
| CCustomersNamedObjectColBase (Index) | Random Access by Index | 13,33 |
| CCustomersNamedObjectColBase (Key) | Random Access by Key | 66,67 |
| StringCollection (Index) | Random Access by Index | 10,00 |
| StringDictionary (Key) | Random Access by Key | 73,33 |
| Dataset (Rows[Index]) | Datatables.Rows[Index] | 104,00 |
| Dataset (Rows.Find) | Random Access Rows.Find | 397,00 |
| Dataset (Select+Sort=196 Rows) | Datatable.Select mit Sortieren 196 Zeilen gefunden | 220,33 |
| XMLDocument (SelSingleNode,150) | SelectSingleNode mit Key (max.150) | 16066,67 |
| XMLDocument (SelectNodes=196) | SelectNodes 196 gefunden | 0,00 |

Tabelle 2: Gemessene Zeiten beim zufälligen Zugriff auf Objekte einer Collection-Klasse