



Thinktecture.DataObjectModel

Vermittler zwischen den Welten

Bei der Entwicklung verteilter Anwendungen kommen verschiedenste Technologien zum Einsatz: WCF für die Kommunikation, ein O/R-Mapper für den Datenzugriff, eine Clienttechnologie zur Präsentation und vieles mehr. Das Zusammenspiel funktioniert jedoch nicht immer reibungslos. Besonders bei den Datenobjekten treffen teilweise gegensätzliche Anforderungen aufeinander. Mit dem Thinktecture.DataObjectModel steht ein Open-Source-Framework zur Verfügung, das zwischen den Welten vermittelt und vor allem auf Clientseite eine Menge fehlender Funktionalität nachrüstet.

Die Zeiten, in denen DataSets zwischen den Schichten einer verteilten Anwendung übertragen wurden, sind endgültig vorbei. Das hat vor allem einen Grund: Interoperabilität. Aber auch Typsicherheit und Flexibilität beim Datenzugriff haben dazu beigetragen, dass in das verbindungslose Modell von ADO.NET nicht mehr investiert wird. In Silverlight ist es sogar nicht einmal mehr enthalten.

Stattdessen geht der Trend zu typisierten Datenobjekten. Sowohl die Bezeichnungen als auch die Anforderungen gehen jedoch, je nach Blickwinkel, weit auseinander. WCF spricht von Data Contracts und meint damit einfache Datentransferobjekte, die keine Logik enthalten und nach dem POCO-Prinzip (Plain Old CLR Object) entwickelt werden. O/R-Mapper kennen hingegen nur Entitäten und verfolgen in erster Linie einen datengetriebenen Ansatz, bei dem es um die Verfolgung von Änderungen geht. Eine ganz andere Sicht auf Datenobjekte hat wiederum die Präsentationsschicht. Hier geht es vor allem um Datenbindung und Eingabevalidierung.

Wie Sie sehen, verfolgen die verschiedenen Technologien unterschiedliche Ziele, was zur Folge hat, dass Sie als Entwickler einigen Herausforderungen gegenüber stehen. Zudem haben die Anwender eigene Anforderungen wie die Undo-/Redo-Funktionalität oder das dynamische Sortieren und Filtern der Daten zur Laufzeit.

Hier kommt nun das Thinktecture.DataObjectModel (TT.DOM) [1] ins Spiel. Dabei handelt es sich um eine Open-Source-Bibliothek des Autors, die eine Brücke zwischen den Technologien schlägt. Zu den wichtigsten Fähigkeiten zählen:

- Erweiterte Unterstützung für Datenbindung in Windows Forms und WPF
- Flexibles Modell zur Eingabevalidierung
- View-Modell, das ein dynamisches Sortieren, Filtern und Aggregieren ermöglicht
- Änderungsverfolgung für die prozessübergreifende Aktualisierung von Objekten

- Unbegrenztes Zurückrollen und Wiederherstellen von Änderungen (Undo/Redo)
- Unterstützung von lokalen und verteilten Transaktionen
- Sicherheitsmodell für die listen-, spalten- und zeilenbasierte Berechtigungssteuerung
- Erzeugen von dynamischen Runtime-Proxies, um beliebige Datenobjekte ad-hoc mit der TT.DOM-Funktionalität auszustatten

Basis des Frameworks ist die Klasse DataObject. Sie implementiert alle nötigen Interfaces für die Datenbindung und verfügt zudem über einen Mechanismus zur Änderungsverfolgung. Sie müssen lediglich Ihre Datenobjekte von DataObject ableiten und haben damit das nötige Rüstzeug. Das folgende Beispiel zeigt die notwendigen Arbeiten:

```
[DataContract]
public class Person : DataObject
{
    private string _name;
    [DataMember]
    public string Name
    {
        base.OnPropertyChanging(„Name“);
        _name = value;
        base.OnPropertyChanged(„Name „);
    }
}
```

Wie Sie sehen, müssen Sie lediglich die Änderungen an den Eigenschaften der Basisklasse melden. Notwendig ist das sowohl für die Änderungsverfolgung als auch für die Datenbindung und die Transaktionsverarbeitung. Wollen Sie Ihre Klassen zusätzlich mit Eingabevalidierung ausstatten, überschreiben Sie den Indexer der Basisklasse. Über diesen kommuniziert die Datenbindungsinfrastruktur, um die Gültigkeit der Daten während der Eingabe zu prüfen. Das kann folgendermaßen aussehen:

```
public override string this[string columnName]
{
    Get
    {
        if ((string.IsNullOrEmpty(columnName) ||
            columnName == „Name“) &&
            (string.IsNullOrEmpty(this.Name)))
        {
            return „Das Feld \“Name“\ ist ein
            Pflichtfeld!";
        }
    }
}
```

Zudem steht mit `DataObjectList<T>` eine generische Listenklasse mit gleicher Funktionalität zur Verfügung. Sie verwaltet alle Objekte eines bestimmten Typs und kann sowohl zur Bindung an die Oberfläche als auch zur Kommunikation mit dem jeweiligen Middle-Tier-Service verwendet werden. Das folgende Beispiel wandelt eine Liste von Person-Objekten, die über einen WCF-Service abgerufen wird, in eine `DataObjectList<T>`-Instanz:

```
List<Person> persons = personService.GetPersons();
DataObjectList<Person> list = persons
    .ToDataObjectList(„Id„);
list.BeginEdit();
```

Die Umwandlung von `List<T>` in `DataObjectList<T>` übernimmt an dieser Stelle die Extension-Methode `ToDataObjectList`. Ihr werden der oder die Namen der Eigenschaft übergeben, die den Primärschlüssel der Daten repräsentieren. Der abschließende Aufruf der Methode `BeginEdit` startet die Änderungsverfolgung. Der Aufruf muss explizit erfolgen, um zu

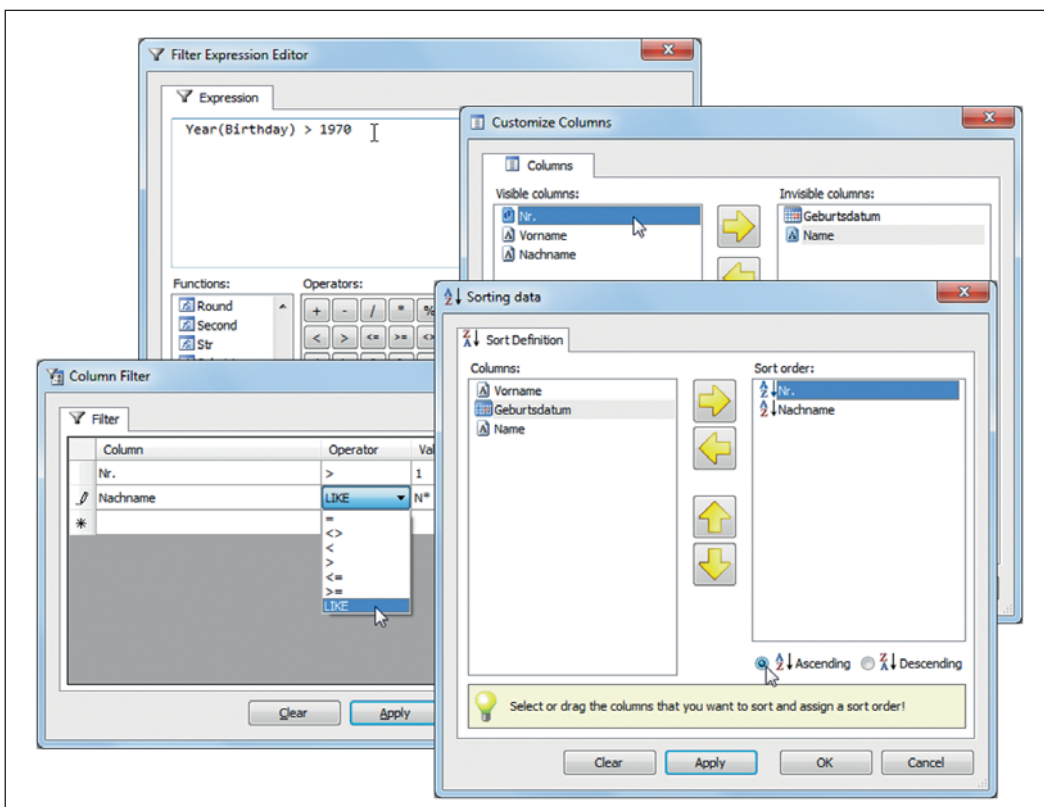
vermeiden, dass bereits beim Füllen oder Serialisieren der Objekte Änderungen protokolliert werden. Alternativ können Sie dies auch über die `Mode`-Eigenschaft von `DataObject` einstellen.

Nun können Sie die Liste oder einzelne Objekte an die Oberfläche binden. Um die Darstellung der Liste zu beeinflussen, bietet TT.DOM ein View-Konzept, das dem der ADO.NET-Klassen `DataTable` und `DataView` ähnelt. Über die Eigenschaft `DefaultView` haben Sie Zugriff auf die Standardansicht der Liste. Alternativ können Sie über die Methode `CreateView` weitere Sichten auf die Daten erzeugen. In beiden Fällen erhalten Sie eine Instanz von `DataObjectView` zurück, mit der Sie alle anzeige-relevanten Einstellungen vornehmen können. So bietet beispielsweise die Eigenschaft `ColumnAttributes` die Möglichkeit, die Darstellung jeder einzelnen Eigenschaft zu steuern:

```
list.DefaultView.ColumnAttributes[„Id“]
    .DisplayName = „Nr.“;
list.DefaultView.ColumnAttributes[„Id“]
    .ReadOnly = true;
list.DefaultView.ColumnAttributes[„Name“]
    .DisplayName = „Nachname“;
```

Darüber hinaus können Sie ausdrucksbasierte Sortier- und Filterregeln hinterlegen. Hierfür stellt `DataObjectView` die Eigenschaften `Sort` und `Filter` bereit.

Die zeichenkettenbasierte Definition der Regeln bietet im Vergleich zu LINQ den Vorteil, diese zur Laufzeit generieren zu können. Auf diese Weise können Sie die Funktionalität auch dem Anwender bereitstellen. Hierfür gibt es im TT.DOM entsprechende Dialoge.



Die Änderungsverfolgung

Nachdem der Benutzer Änderungen an den Daten vorgenommen hat, können Sie diese mit der Methode `GetChanges` auf Listenebene ermitteln. Hierbei haben Sie die Möglichkeit, entweder alle oder nur spezifische Änderungen (eingefügt, geändert, gelöscht) abzufragen. In folgendem Beispiel werden alle Änderungen ermittelt und an die `Save`-Methode des WCF-Service übertragen:

```
List<Person> changes = list.GetChanges();
List<Person> refreshed =
    personService.Save(changes);
list.MergeItems(refreshed);
```

Hierbei sollte die `Save`-Methode des Service nicht nur eine Liste von Objekten entgegennehmen sondern diese auch zurückgeben. Dies ist erforderlich, um Änderungen, die von der Datenbank vorgenommen wurden (zum Beispiel das Erzeugen von Identity- oder TimeStamp-Werten), an den Client zurückzugeben. Auf Clientseite sorgt daraufhin die `MergeItems`-Methode dafür, die serverseitigen Änderungen mit der lokalen Liste zu synchronisieren.

Die `DataObject`-Klasse bietet über die `State`-Eigenschaft die Möglichkeit, den Änderungsstatus eines Objekts zu erfragen. Hierdurch kann der WCF-Service alle geänderten Objekte in einer einzelnen Methode entgegennehmen und diese entsprechend filtern. Um zu gewährleisten, dass nur Datensätze in der Datenbank aktualisiert werden, die nicht in der Zwischenzeit von anderen Benutzern geändert wurden, kann der Service die Originalwerte der Objekte abfragen und diese beim Datenzugriff mit angeben. `DataObject` bietet hierfür die Methode `GetOriginalValues`.

Runtime-Proxies

In manchen Fällen wollen oder können Sie Ihre Datenklassen vielleicht nicht von `DataObject` ableiten. Sei es, weil Sie Ihre Data-Contracts nach dem POCO-Modell generieren möchten oder weil Ihr O/R-Mapper selbst eine Ableitung von einer Basisklasse erfordert. Hierfür bietet TT.DOM die Möglichkeit, Runtime-Proxies für die entsprechenden Typen zu erzeugen. Dazu wird zur Laufzeit ein neuer Typ angelegt, der von `DataObject` ableitet und alle Eigenschaften des Ursprungstyps enthält. Das Lesen und Schreiben der Eigenschaften wird dabei an das



Jörg Neumann ist Principal Consultant bei der Acando GmbH in Hamburg, Associate bei thinktecture und Microsoft MVP im Bereich Client App Dev. Er berät Firmen zu Client- und Datenbank-Technologien und vermittelt sein Wissen regelmäßig in Form von Artikeln, Büchern und als Referent auf Entwickler-konferenzen.

jeweilige Datenobjekt delegiert. Auf diese Weise können Sie Ihre Typen unberührt lassen und dennoch die volle Funktionalität von TT.DOM nutzen. Das folgende Beispiel demonstriert diesen Weg:

```
List<Person> results = personService.GetPersons();
IDataObjectProxyList list = results
    .ToDataObjectListProxy("Id");
list.BeginEdit();
"
```

Die Extension-Methode `ToDataObjectListProxy` generiert im Hintergrund den notwendigen Proxy-Typ und gibt eine Liste vom Typ `DataObjectList<T>` zurück. Da T an dieser Stelle mit dem jeweiligen Proxy-Typ gefüllt wird, können Sie nicht direkt auf die Liste zugreifen. Stattdessen verwenden Sie das Interface `IDataObjectProxyList` als Platzhalter.

Zum Speichern der Änderungen werden hingegen die Originalobjekte an den Service übertragen. Da diese jedoch über keinen Status verfügen, können Sie dies nicht – wie im oberen Beispiel – mit einer einzigen Methode abbilden. Stattdessen sollte Ihr Service `Insert`-, `Update`- und `Delete`-Methoden enthalten, die jeweils ein einzelnes Objekt aufnehmen.

```
public Person UpdatePerson(Person p);
public Person InsertPerson(Person p);
public void DeletePerson(Person p);
```

Folgt Ihr Service diesem Aufbau, können Sie die `SaveChanges`-Methode von `DataObjectList<T>` verwenden und dieser die entsprechenden Methoden in Form von Lambda-Expressions übergeben:

```
list.SaveChanges<Customer>(
    c => service.UpdateCustomer(c),
    c => service.InsertCustomer(c),
    c => service.DeleteCustomer(c));
```

Die Liste kümmert sich selbstständig um die Aktualisierung der Daten und die Synchronisation der serverseitigen Änderungen.

Fazit

Wie Sie gesehen haben, kann Sie das `Thinktecture.DataObjectModel` auf sehr einfache Art und Weise bei der Entwicklung von verteilten Anwendungen unterstützen. In diesem Artikel haben Sie nicht alle Funktionen der Bibliothek kennen gelernt. Weitere Informationen über das TT.DOM finden Sie in meinem Blog [2]. Für Fragen und Anregungen erreichen Sie mich unter joerg.neumann@thinktecture.com.

[1] [Thinktecture.DataObjectModel auf CodePlex:](http://dataobjectmodel.codeplex.com/)
<http://dataobjectmodel.codeplex.com/>

[2] [Blog von Jörg Neumann:](http://headwriteline.blogspot.com)
<http://headwriteline.blogspot.com>