Yves Goeleven softwarearchitect bij Compuware Belgium

Het Unit of Work-patroon in LINQ To SQL

EEN KIJKJE ONDER DE MOTORKAP VAN LINQ

Al verschillende maanden wordt er veel gepraat over de nieuwste aanstormende technologie van Microsoft: Language Integrated Query of kortweg LINQ. Met het LINQ-framework worden queries een eersterangs onderdeel van jouw applicatie. De nieuwe features zorgen er onder andere voor dat het queryen van een databank automatisch lijkt te gaan. In dit artikel belicht de auteur wat er achter de schermen gebeurt.

Voordat we een kijkje gaan nemen onder de motorkap van dit nieuwe framework, is het wel even belangrijk te vermelden dat dit artikel gebaseerd is op een Community Technology Preview (CTP) van het LINQ-project, namelijk die van maart 2007. Dit houdt in dat de structuur van alle onderdelen van dit framework nog aan zware wijzigingen onderhevig kunnen zijn. Hoewel het gehele LINQ-project meer inhoudt dan enkel het queryen van objecten die uit een databank komen, vermoed ik toch dat voor de meesten van ons LINQ het meest toegepast zal worden in database-driven applicaties. Vandaar dat het me logisch leek dit artikel te richten op LINQ To SQL, het Object Relational Mapping (ORM) framework dat LINQ toelaat expressies uit te voeren op objecten die uit een database komen. LINQ To SQL zorgt ervoor dat de juiste objecten uit de database worden ingeladen aan de hand van een LINQ-expressie. Een LINQ-expressie is een C#- of VB.NET-statement dat door de compiler vertaald wordt in de correcte expressieobjecten en de nodige method calls om de gevraagde objecten te verkrijgen. Zo zie je in codevoorbeeld 1 een eenvoudig voorbeeld, waar uit een lijst van BlogEntry-objecten die entries worden geselecteerd waar de beschrijving van de geassocieerde Blog gelijk is aan “Dotnet Blog”. Het resultaat van deze syntax is een query-expressie die later uitgevoerd kan worden om de gewenste objecten te verkrijgen. Merk op dat het var-keyword door de compiler vervangen wordt door het juiste type dat deze expressie vertegenwoordigt, dit noemt men type inference.
Nadat bewerkingen op deze objecten zijn uitgevoerd, dienen ze uiteraard ook opnieuw in de database gepersisteerd te worden. Dit klinkt allemaal zeer eenvoudig, maar zoals je waarschijnlijk uit ervaring weet, is dat niet het geval. LINQ To SQL doet het nochtans allemaal, en wel automatisch.

Het Unit of Work-patroon

De kern van LINQ To SQL bestaat uit het Unit of Work-patroon. Dit patroon wordt door Martin Fowler omschreven als een object dat alle wijzigingen op een set van objecten monitort tijdens de uitvoering van je applicatie, zoals toevoegingen, wijzigingen en verwijderingen. Daarna bepaalt de Unit of Work zelf wat er in de database moet gebeuren aan de hand van de acties die je op de objecten hebt ondernomen. Het Unit of Work-patroon neemt een aantal taken op zich waardoor het synchroniseren van een in memory-objectmodel met een relationeel databasemodel wordt vereenvoudigd. Zo zorgt het onder andere voor:

- **Identity mapping**. In een databank wordt een object geïdentificeerd aan de hand van een primary key in een objectmodel aan de hand van het geheugenadres. De Unit of Work zorgt dat beide concepten met elkaar in overeenstemming blijven. Als bijvoorbeeld een query de identificatie van een object meermaals bevat, zal het Unit of Work-patroon ervoor zorgen dat je steeds dezelfde referentie krijgt.
- **Relationship management.** Objecten bevatten vaak referenties naar geassocieerde objecten. Het is uiteraard niet de bedoeling dat voor elke businessfunctionaliteit het gehele objectmodel in het geheugen wordt geladen. Anderzijds kan het ook niet de bedoeling zijn om elk object apart op te halen. Door middel van ‘lazy loading’, het ophalen van een object wanneer het gebruikt wordt, en ‘eager loading’, het op voorhand ophalen van een hele reeks objecten, kun je goed controleren wat de scope is van de objecten die je in het geheugen laadt.
- **Change tracking.** Het Unit of Work-patroon probeert ook te bepalen wat er met een object dient te gebeuren wanneer het gepersisteerd moet worden. Dit doet het door het object te vergelijken met zijn originele staat, een kopie die gemaakt werd toen het object uit de databank geladen werd. Als het object geen originele staat heeft, wordt het toegevoegd dan wel gewijzigd, of eventueel verwijderd als het deel uitmaakte van een collectie die werd gewijzigd.
- **Optimistic concurrency.** Wanneer een object uit een database geladen wordt voor een bepaalde operatie, bestaat de kans dat een ander proces ondertussen het zelfde record in de database aanpast alvorens jij jouw eigen aanpassingen doorvoert. Om dit te detecteren kan de Unit of Work de originele status in het geheugen vergelijken met de huidige status in de databank. Als deze niet overeenstemmen wordt een wijziging niet toegestaan.

Bijna al deze features kunnen verkregen worden door het bijhouden van een interne object-cache met daarin de originele en de huidige status van alle objecten. Het spreekt dus voor zich dat LINQ To SQL een zekere hoeveelheid aan geheugen vraagt waar je eventueel rekening mee moet houden.

```
var query = from b in blogEntries
   where b.Blog.Description == “Dotnet Blog”
   select b;
```

Codevoorbeeld 1. Een eenvoudige LINQ-expressie

DataContext

In LINQ To SQL wordt de Unit of Work geïmplementeerd met behulp van de DataContext-klasse. Alle features die ik zonet heb opgesomd, worden door deze klasse aangeboden. Het enige dat wij

```
string connectionString = ConfigurationManager.ConnectionStrings[“Blogs
”].ConnectionString;
IDbConnection connection = new SqlConnection(connectionString);

String xml = File.ReadAllText(“Mapping.xml”);
MappingSource mapping = XmlMappingSource.FromXml(xml);

DataContext datacontext = new DataContext(connection, mapping);
```

Codevoorbeeld 2. Een voorbeeldconfiguratie

moeten doen om van al deze features te genieten, is het configureren van een DataContext-instantie. Dit kan door zelf een instantie te creëren of met behulp van de LINQ To SQL-designer. Een objectrelationeel mapping framework maakt zich zorgen om twee dingen: enerzijds het objectmodel in het geheugen en anderzijds het relationeel model in de database. Wij moeten het framework vertellen waar de database zich bevindt en hoe de twee modellen met elkaar overeenkomen. Vandaar dat je bij de instantiëring van een DataContext-object in een IdbConnection-object moet voorzien dat de database vertegenwoordigt, alsook een MappingSource-object waarin de relatie tussen het objectmodel en het relationeel model beschreven staat. Codevoorbeeld 2 geeft een voorbeeldconfiguratie.

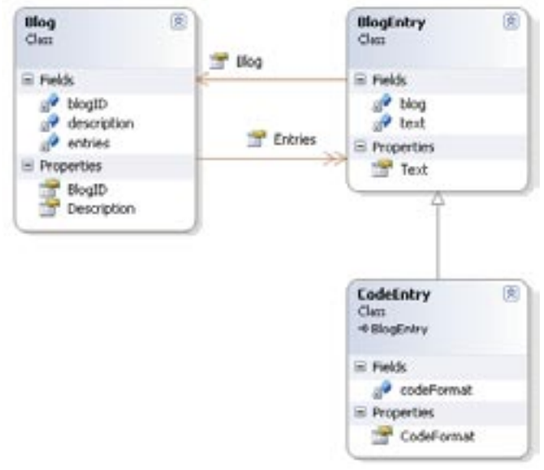
De databaseconnectie moet je momenteel nog voldoen met een SqlConnection. In de toekomst worden ook nog andere databanken ondersteund, hetzij door Microsoft, hetzij door de databasefabrikanten zelf. Het XmlMappingSource-object dien je te creëren door het, op één van de mogelijke manieren, te voorzien van objectrelationele mapping-informatie in XML-formaat.

Mapping

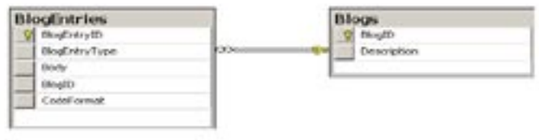
Het mapping-formaat dat je dient te voorzien, kan het best geïllustreerd worden aan de hand van een klein voorbeeld. Laten we een mapping voorzien van een eenvoudig objectmodel dat een blog moet voorstellen. Een blog is opgebouwd uit entries en iedere entry behoort tot slechts één blog. Indien de entry een CodeEntry is, kunnen we speciale formattering toepassen om de code mooi te tonen. Aangezien een tekening meer zegt dan duizend woorden geven afbeeldingen 1 en 2 respectievelijk het objectmodel en het relationeel model weer.

Hoewel het model ogenschijnlijk vrij eenvoudig is, zit er toch een aantal belangrijke concepten in. Enerzijds hebben we een lijst van entries in een blog, anderzijds moeten deze entries terug geassocieerd worden met diezelfde blog. Verder hebben we ook nog een inheritance-relatie tussen de BlogEntry- en de CodeEntry-klasses. De XML-specificatie in codevoorbeeld 3 beschrijft de mapping tussen het objectmodel en het relationeel model.

Het XML-mapping-formaat vertrekt steeds vanuit het standpunt van



Afbeelding 1. Het objectmodel



Afbeelding 2. Het relationeel model

de database: eerst definieer je de database en hoe de tabellen in die database overeenstemmen met .NET-types. In ieder type heb je uiteraard members, zoals properties en fields, die overeenstemmen met kolommen uit de database en aldus ook zo geconfigureerd dienen te worden. Belangrijk op te merken is dat je zowel de effectieve property samen met zijn type kunt specificeren, als ook de private member waarin die property zijn waarde opslaat, wat uiteraard zeer handig is wanneer je een readonly-property wilt opvullen. Dit doe je met behulp van het Storage-attribuut. Als je property ook een setter heeft, is dit attribuut uiteraard overbodig. Ieder object kan in de database geïdentificeerd worden aan de hand van een primary key. Deze primary key zal door de Unit of Work gebruikt worden als sleutel tot de identity-map. De kolom die als identificatie dient, wordt aangeduid met de IsPrimaryKey-Boolean, samen met eventueel een aanduiding of de waarde voor deze kolom door de database gecreëerd wordt. Sommige properties verwijzen naar een set van geassocieerde objecten of een enkelvoudig geassocieerd object. In beide gevallen specificeer je dit met een associatie en de multipliciteit van de associatie specificeer je met de

```
<?xml version=”1.0” encoding=”Windows-1252”?>
<Database Name=”Blogs”>
  <Table Name=”Blogs”>
    <Type Name=”LINQToSql.Domain.Blog”>
      <Column   Name=”BlogID” DbType=”Int NOT NULL IDENTITY”
                Member=”BlogID” Storage=”blogID”
                IsPrimaryKey=”True” IsDBGenerated=”True”/>
      <Column   Name=”Description” DbType=”NVarChar(50)”
                Member=”Description” Storage=”description” />
      <Association   Name=”FK_BlogEntries_Blogs” ThisKey=”BlogID”
                     Member=”Entries” Storage=”entries”
                     OtherTable=”BlogEntries” OtherKey=”BlogID”
                     IsParent=”False” />
    </Type>
  </Table>
  <Table Name=”BlogEntries”>
    <Type Name=”LINQToSql.Domain.BlogEntry”
          InheritanceCode=”Entry” IsInheritanceDefault=”True”>
      <Column Name=”BlogEntryID” DbType=”Int NOT NULL IDENTITY”
              Member=”BlogEntryID” Storage=”blogEntryID”
              IsPrimaryKey=”True” IsDBGenerated=”True” />
      <Column Name=”BlogID” DbType=”Int NOT NULL”
              Member=”BlogID” Storage=”blogID” />
      <Column Name=”Body” DbType=”NText”
              Member=”Text” Storage=”text” />
      <Column Name=”BlogEntryType” DbType=”NVarChar(10)”
              Member=”BlogEntryType” Storage=”blogEntryType”
              IsDiscriminator=”true” />
      <Association Name=”FK_Blogs_BlogEntries” ThisKey=”BlogID”
                   Member=”Blog” Storage=”blog”
                   OtherTable=”Blogs” OtherKey=”BlogID”
                   IsParent=”True” />
      <Type Name=”LINQToSql.Domain.CodeEntry” InheritanceCode=”Code”>
        <Column Name=”CodeFormat” DbType=”NVarChar(10)”
                Member=”Format” Storage=”format” />
      </Type>
    </Type>
  </Table>
</Database>
```

Codevoorbeeld 3. Het XML-mapping-formaat

IsParent-property. Zet je deze op true, dan creëer je een veel-op-één relatie of met andere woorden een enkelvoudige associatie, zoals de relatie tussen een entry en zijn blog. Zet je de IsParent-property op false, dan creëer je een één-op-veel relatie of een set van gerelateerde objecten, zoals de relatie tussen een blog en zijn entries. Ook moet je de betrokken tabellen en de sleutels specificeren, zowel de foreign key als de primary key. Wat overerving betreft, ondersteunt LINQ To SQL slechts één type, namelijk Single Table Inheritance, waarbij alle instanties van de basisklasse, BlogEntry, in één tabel staan gespecificeerd, de superklassen daarentegen worden geïdentificeerd door de waarde van een bepaalde kolom. De kolom die de verschillende waardes bevat, wordt aangeduid door middel van het attribuut IsDiscriminator.

Queries

In het begin van het artikel hebben we een overzicht gegeven van alle features die het Unit Of Work-patroon ons aanbiedt zoals identity mapping, relationship management, change tracking en optimistic concurrency. Al deze features kunnen we op een eenvoudige manier aantonen met behulp van een aantal unit-testen. Iedere test omvat een query en valideert daarna of de datacontext zich ook werkelijk gedraagt zoals verwacht.

De eerste unit-test (zie codevoorbeeld 4) bewijst dat we zonder na te denken gebruik kunnen maken van associaties tussen objecten, de datacontext zorgt ervoor dat de nodige objecten uit de database worden geladen wanneer we ze nodig hebben. Dit wil niet noodzakelijk zeggen dat de objecten effectief in het geheugen geladen worden wanneer de query wordt uitgevoerd. Het standaard gedrag is zelfs zo dat de geassocieerde objecten pas worden geladen wanneer de property aangeroepen wordt, dit concept wordt ook wel ‘lazy loading’ genoemd. Wens je toch op voorhand alle objecten te laden, dan kan dit eenvoudig geconfigureerd worden. Ook bewijst deze test de werking van de identity-map. Objecten die in de database vertegenwoordigd worden door dezelfde primary key hebben in het geheugen ook hetzelfde geheugenadres, het is namelijk hetzelfde object.

De tweede unit-test, codevoorbeeld 5, laat zien hoe de datacontext zorg draagt voor de levenscyclus van een object. De datacontext bepaalt zelf of een object in de database moet worden toegevoegd of gewijzigd wanneer wij verzoeken alles te bewaren door SubmitChanges aan te roepen. Houd er rekening mee dat iemand anders altijd de data kan wijzigen in de database terwijl jij je operaties aan het uitvoeren bent. Dit is moeilijk automatisch te testen, maar wijzig tijdens het uitvoeren van deze unit-test manueel maar eens een blog in de database. De DataContext zal je wijzigingen detecteren, door de originele status van het object te vergelijken met de waardes in de databank. Als deze niet overeenstemmen, wordt een ChangeConflictException gegooid. Standaard vergelijkt de DataContext een object in zijn geheel. Dit wil zeggen dat alle properties van het object worden vergeleken

```
[TestMethod()]
public void TestIdentityMap()
{
  using(DataContext datacontext = new DataContext(connection, mapping))
  {
    Table<BlogEntry> blogEntries = datacontext.GetTable<BlogEntry>();

    var query = from b in blogEntries
         where b.Blog.Description == “Dotnet Blog”
         select b;

    List<BlogEntry> results = query.ToList<BlogEntry>();

    Assert.IsTrue(results.Count > 0);
    Assert.IsTrue(Object.ReferenceEquals(results[0].Blog,
                  results[1]. Blog));   }
}
```

Codevoorbeeld 4. Test op associaties en de identity-map

```
[TestMethod()]
public void TestChangeTracking()
{
  using (DataContext datacontext = new DataContext(connection,
mapping))
  {
    Table<Blog> blogs = datacontext.GetTable<Blog>();

    var query = from b in blogs
         where b.Description == “Dotnet Blog”
         select b;

    Blog blog = query.Single();

    using (TransactionScope scope = new TransactionScope())
    {
      blog.Description = “Updated description”;
      datacontext.SubmitChanges();
    }// rollback

    datacontext.Refresh(blog, RefreshMode.OverwriteCurrentValues);
    Assert.IsTrue(blog.Description.Equals(“Dotnet Blog”));
  }
}
```

Codevoorbeeld 5. Test op change tracking

met alle velden in de database. Indien je dit gedrag wenst te optimaliseren, kun je gebruik maken van een Version-kolom die bijvoorbeeld een timestamp bevat en bij iedere wijziging verandert.

Transacties

In de tweede unit-test zie je ook dat ik de update heb geëncapsuleerd in een TransactionScope. Dat is dan ook alles dat je moet doen om de DataContext te laten registreren in een transactie. Deze transactie zal terugrollen, omdat de Complete-method op de TransactionScope niet werd aangeroepen. Alle wijzigingen die de datacontext heeft uitgevoerd toen SubmitChanges werd aangeroepen, worden ongedaan gemaakt in de database. Merk wel op dat enkel de database teruggerolt wordt en niet de objecten die in het geheugen aanwezig zijn. Indien je zelf geen controle neemt over de transactie, bijvoorbeeld via de Transaction-Scope-klasse, dan zorgt de DataContext zelf voor een lokale transactie.

Minder complex

LINQ To SQL biedt ons de benodigde features om op een eenvoudige manier met de verschillen tussen de objectgeoriënteerde en relationele wereld om te gaan. Zo biedt het Unit of Work-patroon ons een uniforme manier om met de database te communiceren en vermindert het de complexiteit die daar onherroepelijk mee verbonden is zoals identity mapping, relationship management, change tracking, concurrency en transacties. Wens je zelf wat te experimenteren met de codevoorbeelden in dit artikel, dan kun je deze vinden op de website van .NET Magazine, of op mijn persoonlijke blog. De solution bevat onder andere een voorbeelddatabase, een project met het domain-model, de mapping file en een unit-testproject met de testcode. Mag ik er wel nog even op wijzen dat dit artikel geschreven is aan de hand van de maart CTP van LINQ en dat alle code, inclusief de beschreven klassen, waarschijnlijk nog onderhevig zijn aan wijzigingen.

Yves Goeleven is werkzaam als softwarearchitect bij Compuware Belgium (www.compuware.be) en houdt zich graag bezig met de laatste nieuwe Microsoft-technologieën. Zijn email is: yves@goeleven.com en blog: http://www.goeleven.com/

Referenties

Unit of Work-patroon: http://www.martinfowler.com/eaaCatalog/unitOfWork.html
The LINQ-project: http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx
Orcas March 2007 CTP: http://msdn2.microsoft.com/en-us/vstudio/aa700831.aspx