De eenvoudige kracht van extension methods

Zo leuk, dat je moet vrezen voor overmatig gebruik

Sander Hoogendoorn

Zoals waarschijnlijk niemand is ontgaan, heeft Microsoft in .NET 3.5 LINQ geïntroduceerd. Alhoewel deze language extension inmiddels op zo'n beetje alle grote en kleinere conferenties is gedemonstreerd, is de manier waarop LINQ queries uitvoert op reeds bestaande types minder voor de hand liggend. Hiervoor introduceert LINQ een aantal nieuwe query operators, daarbij gebruik makend van nog een andere nieuwe language feature: de extension method.

Er zijn twee sets LINQ query operators. Een set werkt op objecten van het type IEnumerable en de andere set op objecten van het type IQueryable. Om de methodes in deze sets te implementeren hebben de .NET architecten ervoor gekozen aan de taal iets toe te voegen dat extension methods wordt genoemd. Hiermee worden de door de beide interfaces vereiste methoden aan de bestaande Enumerable en Queryable klassen 'toegevoegd', zonder deze klassen direct uit te breiden, wat min of meer tot vervuiling van .NET framework zou leiden.

My first extension method

Naast de haast vanzelfsprekendheid van het introduceren van extension methods voor de query operators van LINQ, zit er echter meer aan vast. Op het eerste gezicht lijken extension methods het mogelijk te maken om iedere klasse in iedere namespace van extra functionaliteit te voorzien. Laat ik dit illustreren aan de hand van een voorbeeld.

Een van aspecten die mij al heel lang storen aan het werken met de DataSet klasse is dat je op tientallen plekken onderstaande of vergelijkbare code moet schrijven om de waarde van een kolom in een rij in een tabel te krijgen.

```
public void DoTraditionalTest()
{
    DataSet ds = new DataSet();
    object o = ds.Tables[2].Rows[4]["Name"];
}
```

Deze code is niet alleen omslachtig, maar ook twijfelachtig, omdat ze niet controleert of de tabel en de rij überhaupt beschikbaar zijn. Omdat vrijwel iedereen ooit wel eens dit soort code heeft geschreven, zou het dan ook veel makkelijker zijn om een methode zoals Get op de DataSet klasse te hebben zoals in het volgende voorbeeld. Get voert dan natuurlijk wel alle benodigde controles uit.

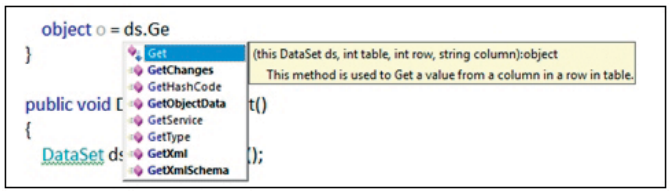
```
public void DoTest()
{
    DataSet ds = new DataSet();
      object o = ds.Get(0, 0, "Name");
}
```

Het probleem is echter dat je deze functionaliteit niet kunt toevoegen op de .NET framework klasse DataSet. Hier komen extension methods goed van pas. Met .NET 3.5 zou je de volgende methode kunnen schrijven.

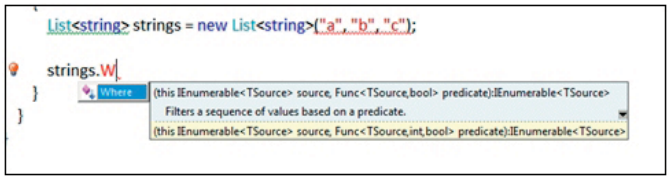
```
namespace Extensions
{
     public static class DataSetExtensions
    {

public static object Get(this DataSet ds, int table, int row,
string column)
        {
            if (table >=  0 && table <= ds.Tables.Count)
            {
                if (row >= 0 && row <= ds.Tables[table].Rows.Count)
                {
                    return ds.Tables[table].Rows[row][column];
                }
            }
              return null;
        }
    }
}
```

Als je nog nooit eerder een extension method van dichtbij hebt gezien zal de eerste parameter van de signature van deze methode wellicht wat vreemd voorkomen. Met name het this keyword. Dit wordt (altijd als eerste) toegevoegd om aan te geven dat de Get methode werkt op de DataSet klasse. Hoewel deze methode wordt gedefinieerd als een statische methode op een klasse met de naam DataSetExtensions, wordt ie door .NET herkend als een instance method op DataSet.



Maar het wordt nog mooier. Intellisense herkent mijn methode en zelfs het commentaar dat ik heb toegevoegd. Het enige verschil tussen een instance method en een extension method vanuit Intellisense gezien, is dat de this parameter terugziet in de tooltip. Ditzelfde geldt overigens voor de in het .NET framework zelf gedefinieerde extension methods. Kijk maar eens naar de LINQ Where methode, die ook de this laat zien.



Het Open Closed Principe

Cool toch? Toen ik voor het eerst over extension methods las en hoorde dacht ik onmiddellijk aan een heleboel functionaliteit die ik niet alleen aan onze eigen frameworks zou kunnen toevoegen, maar ook aan allerlei open source frameworks en zelfs aan het .NET framework zelf. Maar toch, alvorens een hele hoop code los te laten op de community, toch nog even iets dieper graven. Want mijn tweede gedachte over extension methods was heel anders. Mijn grootste zorg was of extension methods me in staat zouden stellen om het gedrag van bestaande framework code te veranderen, en dus te beschadigen.
Daarvoor is het belangrijk om een te kijken naar het Open Closed Principe (OCP). In het kort komt dit erop neer dat klassen open moeten staan voor uitbreidingen, maar gesloten moeten blijven voor wijzigingen. In zijn algemeenheid is dit een goed principe, fraai beschreven en bezongen door Bob Martin. De belangrijke vraag is dus: kan ik met extension methods het gedrag van bestaande klassen veranderen? Wat gebeurt er eigenlijk als ik de volgende extension method schrijf?

```
namespace Extensions
{
    public static class ObjectExtensions
    {
        /// <summary>
        /// Testing whether I can override the ToString() method.
        /// </summary>
        /// <param name="o">The object</param>
        /// <returns>My own string.</returns>
        public static string ToString(this object o)
        {
            return "Sander is always right.";
        }
    }
}
```

De vraag is: heb ik door deze eenvoudige extension method te schrijven de ToString methode voor alle klassen in mijn applicatie overschreven? Immers, deze ToString is geldig voor alle klassen van het type object, en ook voor alle subklassen hiervan. Deze en vergelijkbare extension methods zouden veel projecten in recordtempo om zeep helpen. Gelukkig (en natuurlijk) staan de architecten van .NET je niet toe dit te doen. Er zijn wel wat zaken waar je op moet letten over hoe extension methods werken en hoe ze worden gecompileerd.

Kenmerken van extension methods

Als eerste. Extension methodes zijn gedefinieerd binnen de context van hun namespace; in het voorbeeld hierboven de namespace Extensions. Om een bepaalde extension method te gebruiken in een klasse in je eigen code moet je dus een verwijzing naar de namespace opnemen die de extension method bevat. Zonder deze verwijzing wordt de methode al niet eens in Intellisense getoond.
Ten tweede, en misschien nog wel belangrijker, zijn extension methods in feite gewoon statische methodes op een statische klasse, zoals goed te zien is in de intermediate language (IL). Eigenlijk zijn extension methods alleen te zien in de editor als instance methodes van de doelklasse. Het feit dat ze als statische methodes buiten je doelklasse worden gedefinieerd heeft wel wat gevolgen. Je kunt namelijk in een extension method alleen de public properties van de doelklasse benaderen, net als in iedere normale methode met een parameter van dat type. Je bent dus niet in staat om op deze manier de protected en private fields van deze klasse aan te tasten. Sterker nog, je kunt in een extension method ook geen nieuwe properties of velden definiëren.
Nog belangrijker is het op te merken wat de compiler met je zelfgeschreven extension methods doet, vooral in het geval wanneer je een methode definieert die dezelfde signature heeft als een me-

Met extension methods kun je bestaande functionaliteit niet wijzigen. Je kunt geen bestaande methodes overrulen en geen private of beschermde velden wijzigen

thode in de doelklasse, zoals ik hierboven met de ToString heb geprobeerd.
De compiler volgt in dit soort gevallen een hele duidelijke regel: extension methods hebben altijd een lagere prioriteit dan echte instance methodes. Als de compiler een methode moet verwerken zoekt hij eerst in de doelklasse naar methodes met een overeenkomende signature, daarna in de overerfde klassen en gebruikt deze methode. Pas wanneer geen echte instance methode wordt gevonden gaat de compiler op zoek naar extension methods met dezelfde signature. Volgens deze regel zal de ToString methode die ik hierboven heb gedefinieerd nooit worden aangeroepen als de object klasse al een methode met dezelfde signature definieert.
Zelfs wanneer ik probeer om een (heel domme) extension method zoals hieronder te creëren, waarbij Equals(string s) met een string moet matchen, zal de compiler de Equals(object o) methode van object gebruiken, zelfs als deze een veel bredere scope heeft, en zelfs als ik Equals aanroep met daadwerkelijk een string als parameter. Hij zoekt naar een object, niet naar een string.

```
public static bool Equals(this object o, string match)
{
    return match.Contains("Sander");
}
```

Deze compilereigenschap stelt mij gerust. Met extension methods kun je dus bestaande functionaliteit niet wijzigen. Je kunt geen bestaande methodes overrulen en ook geen private of beschermde velden wijzigen. In feite leeft de extension method buiten de target klasse. Zoals viel te verwachten is het Open Closed Principe gegarandeerd door de regels die aan extension methods zijn gesteld door de architecten van .NET. Je bent dus behoorlijk veilig.

Geen zorgen dus?

Er is nog wel iets anders dat me verontrust over extension methods. Ik kan me redelijk druk maken over de meeste demo's over extension methods die ik heb gezien. Er lijkt maar geen eind te komen aan de stroom developers die extension methods gebruiken om de string klasse uit te breiden. Zo heb ik demo's gezien die controleren of een string een geldig emailadres is, een geldige url, en zelfs extension methodes die met behulp van een synthesizer een string kunnen uitspreken. Hoewel ik vooral van deze laatste een fraai staaltje vakmanschap verwacht, is dit niet iets wat ik in veel van mijn applicaties zal gebruiken.
Ik vind alleen dat dit soort zaken geen extension methods van de

string klasse mogen zijn. Tenzij tenminste vijftig procent van alle strings die ik in mijn code gebruik bestaat uit emailadressen, zou ik een toch ander patroon overwegen. Je kunt bijvoorbeeld ook value objects van het type email, zip of wat dan ook definiëren.
Eigenlijk maak ik me bij extension methods vooral zorgen over het feit dat het zo'n leuk technologietje is, dat veel mensen uitnodigt om het voor allerlei verkeerde redenen toe te gaan passen – alleen maar omdat het zo leuk is om een eigen extension methods te schrijven.
Voor je het weet hebben de ontwikkelaars in je project een hele library aan extension methods gecreëerd voor allerlei mogelijke .NET en andere framework klassen. Dit kan namelijk eventueel de applicatiearchitectuur ondermijnen en ongewenste afhankelijkheden creëren. Kijk maar eens naar het volgende voorbeeld.

```
using System.Workflow.Activities;
 namespace Extensions
{
    public static class CodeAcitivityExtensions
    {
        public static bool Kill(this CodeActivity activity)
        {
            activity.Dispose();
        }
    }
 }
```

Hier heb ik een extension method toegevoegd aan de klasse CodeActivity, die zich bevindt in de namespace System.Workflow.Activities. Hartstikke cool natuurlijk. Maar mijn Extensions namespace verwijst nu wel naar System.Workflow.Activities, maar ook indirect naar de onderliggende System.Workflow.ComponentModel namespace. Een van mijn vorige codevoorbeeld, waarin ik de klasse DataSet uitbreidde, bevindt zich ook in mijn mooie Extensions namespace. Vanaf dit moment verwijst nu alle code waarin ik de methode Get op DataSet aanroep indirect naar deze workflow namespaces. Ach, het is maar een voorbeeld .

Waarschijnlijk zul je veel meer extension methods van andere ontwikkelaars gebruiken dan dat je ze zelf schrijft. Toepassing ervan vergt nauwelijks extra kennis

Bovendien loop je ook nog een risico dat wanneer je een extension method definieert voor een type uit een extern framework, je extension method de code laat vastlopen wanneer de implementatie van dit type verandert of wanneer je project migreert naar een nieuwere versie van het framework waar deze klasse in is gedefinieerd.

Frameworks uitbreiden

Natuurlijk zijn er ook positieve punten aan extension methods. Waarschijnlijk zul je veel meer extension methods van andere ontwikkelaars gebruiken dan dat je ze zelf schrijft. Zo bezien is extension methods een krachtige feature; toepassing ervan vergt nauwelijks extra kennis. In mijn optiek zullen de meeste extension methods worden geschreven door framework developers, die ze gebruiken om hun eigen frameworks te patchen of het gebruik ervan te vergemakkelijken. Of – waarschijnlijker nog – door developers die behoefte hebben om de basisklassen van bestaande frameworks uit te breiden, zowel van Microsoft als uit de open source. Zie ook het volgende voorbeeld, dat het open source framework CSLA uitbreidt.
Een van de meest gebruikte basisklassen in het CSLA framework heet BusinessBase. Dit type dient als layer supertype voor alle domeinobjecten in applicaties die met het framework zijn gebouwd. De BusinessBase klasse heeft de mogelijkheid om zijn business rules te valideren. Om het resultaat van zo'n validatie te kunnen inzien, kent BusinessBase een collectie broken rules, waarin verwijzingen naar business rules zijn opgeslagen die in strijd zijn met de validatie van het domeinobject. In sommige gevallen, bijvoorbeeld wanneer de broken rules op het scherm moeten worden getoond, kan het handig zijn om per property na te gaan welke business rules worden geschonden. De BusinessBase klasse kent deze mogelijkheid niet.

```
public static IEnumerable<BrokenRule> GetBrokenRules(this BusinessBase bb, string property)
{
    var queryableList = new List<BrokenRule>(bb.BrokenRulesCollection);
    IEnumerable<BrokenRule> result;
    if (!string.IsNullOrEmpty(property))
    {
        result = from brokenRule in queryableList
                 where brokenRule.Property.Equals(property)
                 select brokenRule;
    }
    else
    {
        result = queryableList;
    }
    return result;
}
```

Dit codevoorbeeld toont een extension method voor BusinessBase, die de collectie broken rules doorloopt middels een LINQ query die de broken rules voor de geselecteerde property filtert. Deze methode is zeer bruikbaar en is een goed voorbeeld van hoe ontwikkelaars de functionaliteit van een framework klasse kunnen tweaken. Het zou me niet verbazen als op korte termijn veel meer van zulke extension methods beschikbaar komen voor populaire frameworks zoals Microsoft's Enterprise Library, NHibernate, ADF, CSLA en natuurlijk voor het .NET framework zelf.

Spaarzaam

Extension methods zijn een language feature die nog niet heel erg bekend is bij developers, maar erg nuttig kan zijn om het gebruik van frameworks te vergemakkelijken. Wanneer je extension methods ontwikkelt moet je je echter goed bewust zijn van de hierboven omschreven kenmerken. Ik raad daarom aan om extension methods spaarzaam te implementeren. Onder normale omstandigheden zul je bestaande typen nog steeds het meest uitbreiden door ervan te erven, zeker wanneer je daarbij gebruikt wilt maken van de protected fields van het type. En zelfs wanneer dit vereist dat de developers je nieuwe klasse gaan gebruiken in plaats van een van de basisklassen uit het .NET framework. Om maar met een quote te eindigen: "Creativity is a natural extension of our enthusiasm." Zo is dat.

Sander Hoogendoorn, is principal technology officer bij Capgemini. Voor meer informatie zie www.sanderhoogendoorn.com of www.smartusecase.com.

Met dank aan Joost van Schaik van Vicrea Solutions.