Reinhard Brongers en Loek Duys zijn werkzaam bij VX Company

Reflectie: een andere kijk op objecten

GOED GEBRUIK VAN REFLECTION LEVERT FLEXIBILITEIT OP IN EEN PROJECT

Reflection is moeilijk. Een opmerking die je als ontwikkelaar vast wel gehoord hebt, of zelf geroepen. Maar eigenlijk valt dat best mee. In dit artikel willen we dat illustreren door twee voorbeelden van reflection uit de praktijk te geven. Stapsgewijs wordt uitgelegd hoe reflection veel ontwikkelwerk uit handen kan nemen en onderdeel kan zijn van een herbruikbare component die in veel projecten gebruikt kan worden.

Systeemontwikkeling bestaat vaak voor een groot deel uit het schrijven van code die niet direct met het businessprobleem te maken heeft. Bijvoorbeeld het schrijven van testmethods of lijstjes waarin databaseparameters voorzien worden van een waarde. Wijzigingen in de business of database hebben vaak wijzigingen in de datalaag en testcode tot gevolg. Gebruik van reflection kan het omgaan met wijzigingen veel flexibeler en zelfs configureerbaar maken.

Case 1: Het testen van data access code.

Laten we beginnen met een testfunctie die controleert of een update en het ophalen van functies in de datalaag naar verwachting werken. Dat wil zeggen, het dataobject dat wordt aangeboden, moet na update en opnieuw ophalen identiek zijn aan het origineel. In dit voorbeeld gaat het om een object van het type Weer; zie codevoorbeeld 1. Overigens is alle code in uitgebreidere vorm te downloaden via de site van .NET Magazine.

Een nadeel van deze code is dat wanneer het *Weer*-type wordt uitgebreid, alle bijbehorende code moet worden aangepast om de wijziging te ondersteunen in opslag en test. Ook moet voor ieder zelf gedefinieerd type een soortgelijke *Compare*-method geschreven worden. Wanneer dat laatste wordt vergeten, worden wijzigingen zelfs ongemerkt niet meegenomen in tests.

Wat is reflection?

Een .NET-assembly bestaat uit een dll, of executable plus een manifest. Het manifest bevat meta-informatie over de code in de assembly. De meta-informatie bestaat uit een verzameling tabellen, waarin informatie over bijvoorbeeld alle types, methoden en referenties naar andere assemblies staat. Reflection is de techniek waarmee deze informatie over types kan worden uitgelezen. De informatie wordt geleverd in de vorm van classes. In het .NET Framework zijn deze classes verzameld in de *System.Reflection*-namespace. Maar naast de *System.Reflection*-namespace is *SystemType* eigenlijk het belangrijkste type bij reflection. Het kan op basis van een assembly alle aanwezige types representeren en aangeven of een type een reference- of valuetype is. Ook kan *System.Type* informatie leveren over properties in de vorm van onder andere *PropertyInfo*-objecten en methods in de vorm van *MethodInfo*-objecten. De geleverde informatie bestaat uit onder andere de naam, het property-type van de property en functies om de waarde van de property te kunnen lezen en schrijven. *MethodInfo* kan worden gebruikt om methods aan te roepen door middel van *Invoke*. Als alternatief voor het gebruik van *PropertyInfo* of *MethodInfo* kan *System.Type*. *InvokeMember* gebruikt worden. Wanneer reflection gebruikt wordt, dient *System.Type* dus steeds als uitgangspunt.

```
public class Weer
{
   public int? WeerID
   { [..] }
   public int Luchtdruk
   { [..] }
   public int Temperatuur
   { [..] }
}

public class DataAdapter
{
   public void Update(Weer weer) {
      [..]
   }
   public void Select(Weer weer) {
      [..]
   }
}

public class TestDataAccess
{
   public bool TestWeerUpdateSelect(){
      [..]
      return Compare(weer, updatedWeer);

   private bool Compare(Weer left, Weer right){
      bool equal;
      equal = (left.Luchtdruk == right.Luchtdruk
         && left.Temperatuur == right.Temperatuur
         && left.WeerID == right.WeerID);
      return gelijk;
   }
}
```

Codevoorbeeld 1.

Hoe kan reflection helpen in onze case?

Het vergelijken van twee objecten betekent vaak dat alle properties met elkaar vergeleken worden. *System.Reflection.PropertyInfo* komt hierbij goed van pas.
Codevoorbeeld 2 beschrijft hoe PropertyInfo gebruikt kan worden om properties met elkaar te vergelijken. De code werkt als volgt:

1. De method *GetType* levert informatie over het type waarop de method wordt aangeroepen.
2. De method *GetProperties* levert informatie over alle properties van het type, in de vorm van *PropertyInfo*-objecten.
3. De waarde van een property kan worden achterhaald door de *GetValue*-method erop aan te roepen. De waarde wordt geleverd als object.
4. Waarden kunnen worden vergeleken door de static method op het *Object*-type te gebruiken. Door deze method te gebruiken in plaats van de leftValue.*Equals*-method is ondersteuning voor vergelijkingen met *null* ingebouwd.

De *Compare*-functie kan nu gebruikt worden om twee instanties van ieder willekeurig type met elkaar te vergelijken. In de praktijk zullen types aanzienlijk complexer zijn, bijvoorbeeld door compositie van types. Als we complexere objecten met elkaar willen vergelijken, moeten we nog iets meer weten over reflection.

Complexe(re) objectstructuren

Omdat we tijdens het ontwikkelen (design-time) weten hoe de structuur van het object is, kunnen we een verzameling 'paden' maken naar de locatie waar alle properties te vinden zijn. Vergelijkbaar met de statements die je zou schrijven in C#, maar dan in de vorm van strings.
In geval van Weer zouden deze paden er als volgt uit zien:

- Temperatuur
- Luchtdruk

De 'paden' zijn dus de namen van de properties. Wanneer het type Weer uitgebreid wordt
om regenval te beschrijven, kan de code er uit gaan zien als in codevoorbeeld 3.
De nieuwe verzameling paden gaat er dan zo uit zien:

- Temperatuur
- Luchtdruk
- Regenval.Minimum
- Regenval.Maximum

Twee instanties van het nieuwe *Weer*-type kunnen vergeleken worden door de waarden van de properties, die zijn beschreven door de paden, te vergelijken. Er moet dus een functie geschreven worden die op basis van een pad en een object de waarde van een property teruggeeft; zie codevoorbeeld 4. De code werkt als volgt:

1. Het meegegeven pad wordt opgeknipt bij alle puntkarakters.
2. Het eerste pad wordt geëvalueerd op het meegegeven object.
3. De eropvolgende paden worden geëvalueerd op het resultaat van de vorige evaluatie.
4. De GetPropertyValue-method wordt gebruikt om een PropertyInfo-object te vinden op het meegegeven type, op basis van de naam van de property.

De code die we nu hebben is bruikbaar voor ieder object. Alleen voor verzamelingen is nog een uitbreiding nodig die in dit artikel niet wordt behandeld. Wanneer een object nieuwe properties krijgt, is het toevoegen van een zoekpad voldoende om de properties mee te testen. De zoekpaden moeten in deze method als parameters worden meegegeven. Maar als we de zoekpaden toch al weten, kunnen we net zo goed de properties rechtstreeks opvragen. We missen nog een stukje, namelijk het afleiden van de zoekpaden.

Het afleiden van zoekpaden

Het maken van de paden wordt foutgevoeliger wanneer het object complexer wordt. Ook hier kan reflection weer helpen. Door van een type en alle geneste types de property-namen recursief te verzamelen, kunnen alle paden automatisch worden aangemaakt. Een uitwerking hiervan staat in codevoorbeeld 5.

De code werkt als volgt:

1. Op basis van het type van een object worden zijn properties bepaald. Door de waarde van een property als parameter te gebruiken in plaats van het type, ontstaat ondersteuning voor polymorfisme.
2. Wanneer een property een genest object is, wordt deze meegegeven aan de *GetPaths*-method. Zo ontstaat een recursieve functie. Ook wordt het pad naar de property meegegeven als startpad.

```
private bool Compare(object left, object right){
   bool equal = true;
   Type type = left.GetType();
   if (right.GetType() != type)
      equal = false;
   else{
      PropertyInfo[] properties = type.GetProperties();
      foreach (PropertyInfo pi in properties){
         object leftValue = pi.GetValue(left, null);
         object rightValue = pi.GetValue(right, null);
         equal = equal && Object.Equals(leftValue, rightValue);
      }
   }
   return equal;
}
```

Codevoorbeeld 2.

```
public class Weer
{
   [..]
   public Regen Regenval
   { [..] }
}

public class Regen
{
   public int Maximum
   { [..] }
   public int Minimum
   { [..] }
}
```

Codevoorbeeld 3.

```
protected object Parse(object target, string path){
   string[] splitProperties = path.Split('.');
   object propertyValue = target;
   foreach (string propertyName in splitProperties){
      if (propertyValue == null)
         break;
      propertyValue = GetPropertyValue(propertyValue, propertyName);
   }
   return propertyValue;
}

private object GetPropertyValue(object target, string propertyName){
   object value = null;
   if (target != null){
      Type propertyType = target.GetType();
      PropertyInfo info = propertyType.GetProperty(propertyName);
      value = info.GetValue(target, null);
   }
   return value;
}
```

Codevoorbeeld 4.

```
private void GetPaths(object value, StringCollection list, string
    pathStartsWith){
  if (value == null)
    return;
  Type type = value.GetType();
  PropertyInfo[] properties = type.GetProperties();
  foreach (PropertyInfo pi in properties){
    if (pi.PropertyType.IsClass && pi.PropertyType !=
      typeof(string)){
      string newPath;
      if (pathStartsWith == null)
        newPath = pi.Name;
      else
        newPath = String.Concat(pathStartsWith, ".",
          pi.Name);
      object nestedClass = pi.GetValue(value, null);
      GetPaths(nestedClass, list, newPath);
    }
    else{
      if (pathStartsWith == null)
        list.Add(String.Concat(pi.Name));
      else
        list.Add(String.Concat(pathStartsWith, ".",
          pi.Name));
    }
  }
}
```

Codevoorbeeld 5.

```
public static void FillObject(SqlDataReader reader, object target){
  if (target != null && !reader.IsClosed
    && reader.Read() && reader.FieldCount > 0)
  foreach (PropertyInfo property in target.GetType().GetProperties())
  {
    object databaseValue = reader[property.Name];
    if (databaseValue != Convert.DBNull)
    {
      property.SetValue(target, databaseValue, null);
    }
  }
}
```

Codevoorbeeld 6.

```
public static void FillParameters(SqlParameterCollection parameters,
    object source, char parameterStart){

  foreach (SqlParameter parameter in parameters){
    string parameterName = parameter.ParameterName.TrimStart('@');
    Type type = source.GetType();
    PropertyInfo info = type.GetProperty(parameterName);
    object parameterValue = info.GetValue(source, null);
    if (parameterValue != null)
      parameter.Value = parameterValue;
    else
      parameter.Value = Convert.DBNull;
  }
}
```

Codevoorbeeld 7.

3. Wanneer een property een valuetype is, wordt zijn naam toegevoegd aan de lijst met paden (*list*), eventueel voorafgegaan aan het startpad.

Configurabel gedrag

Het grote voordeel van het gebruik van paden is de flexibiliteit. Het is mogelijk paden op te slaan in een database en daarop onderhoud te doen. Het gedrag van een applicatie is daarmee dus configureerbaar te maken. Als uitbreiding op deze case zouden bepaalde properties kunnen worden uitgesloten tijdens het genereren van de paden, zodat zij tijdens het vergelijken van objecten niet worden geëvalueerd.

Case 2: Reflection in combinatie met ADO.NET.

We hebben gezien hoe reflection bij het vergelijken van objecten gebruikt kan worden. Een andere toepassing is het gebruik van reflection om in de datalaag *SqlParameter*-objecten van een waarde te voorzien, of andersom, objecten van een waarde te voorzien met behulp van een *SqlDataReader*-object. Een veelgebruikte werkwijze bij het opzetten van een datalaag bestaat uit het maken van een type voor iedere tabel in de database van een applicatie en vier stored procedures. Daarbij wordt ook vaak voor ieder type code geschreven om de types te kunnen persisteren. Er wordt niet alleen een *Update*-method gemaakt, maar vaak ook een of meer *Select*-methods, een *Insert*-method en een *Delete*-method. Een grote hoeveelheid tabellen leidt dus tot zeer veel code in de datalaag.

Het toevoegen van een kolom aan een tabel in de database heeft tot gevolg dat:

1. Stored procedures moeten worden aangepast.
2. De code in de datalaag moet worden uitgebreid om de extra stored procedureparameter te ondersteunen.
3. Het type dat op de betreffende tabel is gebaseerd wordt uitgebreid met een nieuwe property, die de kolomwaarde gaat bevatten.

Hoe kan reflection hierbij helpen?

Om flexibeler te kunnen omgaan met wijzigingen in de database, kan het vullen van een object op basis van een *DataReader* met behulp van reflection gedaan worden. Wanneer alle kolomnamen van een tabel overeenkomen met property-namen in een type, kan een *DataReader* eenvoudig worden uitgelezen door alle property-namen af te gaan. Het *PropertyInfo*-type heeft behalve een *GetProperty*-method ook een *SetProperty*-method. Deze kan in dit scenario gebruikt worden om de waarde van een property binnen een object te specificeren. Op een vergelijkbare manier als in de eerste case kunnen waarden van een bronobject naar een doelobject worden gekopieerd; zie codevoorbeeld 6. Het doelobject wordt samen met een bronobject, een *SqlDataReader*, meegegeven. Wanneer het bronobject geen waarden bevat, of het doelobject null is, kan niets worden gekopieerd. Anders wordt iedere property-naam in de resultset gezocht. Wanneer de gevonden waarde geen database-null-waarde (*DBNull*) is, wordt deze gekopieerd naar het doelobject.

Parameters

Het ligt voor de hand om ook de stored procedureaanroep zo aan te pakken. Wanneer alle kolomnamen overeenkomen met parameternamen in een stored procedure, kan een parameter-object van een waarde worden voorzien door de bijbehorende property-naam in het type te zoeken. De code kan eruit zien als in codevoorbeeld 7. De verzameling parameters wordt doorlopen, van iedere parameter wordt de naam ontdaan van het prefixkarakter. Wanneer SqlServer als database wordt gebruikt, moet het karakter een '@' zijn. Omdat de namen overeenkomen kan de waarde voor de parameter gevonden worden door een property te zoeken in het bronobject, met dezelfde naam.

Wanneer nu een kolom wordt toegevoegd aan een tabel, heeft dat geen invloed meer op de code die verantwoordelijk is voor het aanroepen en gebruiken van stored procedures. Het aanpassen van het type en de stored procedures is voldoende. De bovenstaande code komt pas echt tot zijn recht als de verzameling stored procedures automatisch wordt aangemaakt. Alleen het vullen van de *Value*-properties van de *SqlParameter*-objecten zou

omslachtig zijn. Gelukkig biedt het .NET Framework daarvoor al ondersteuning. Het *System.Data.SqlClient.SqlCommandBuilder*-type heeft een static *DeriveParameters*-method, die een *SqlCommand*-object voorziet van alle stored procedureparameters.

Beperkingen en uitbreidingen

Het is niet altijd even handig als de namen van kolommen in een database gelijk moeten zijn aan de property-namen in een type, bijvoorbeeld wanneer tegen een bestaande database wordt geprogrammeerd. Ook daar kan reflection uitkomst bieden. Door bijvoorbeeld optioneel een *Attribute* toe te voegen op een property kan zijn naam worden 'overschreven'. De code uit voorbeelden 6 en 7 moet worden aangepast, waarbij eerst wordt gekeken of het *Attribute* voorkomt op de property. Wanneer dat het geval is, moet de aangegeven naam worden gebruikt in plaats van de naam van de property. Een andere manier om het mappingprobleem op te lossen is door gebruik te maken van een XML-bestand waarin de namen gekoppeld worden. Door deze aanpassing kun je in korte tijd een basale Object-Relational (O/R) mapper bouwen.

Ten slotte

Naast alle voordelen zijn er ook nadelen verbonden aan het gebruik van reflection. Vergeleken met het directe gebruik van properties is er een performancepenalty. Het meeste werk dat anders tijdens compileren gedaan werd, moet nu tijdens het draaien van de applicatie gebeuren. En dan niet één keer, maar iedere keer dat de code wordt uitgevoerd. Er zijn diverse optimalisaties aanwezig in het .NET Framework. Voor de method *GetType* is bijvoorbeeld in de JIT-compiler ondersteuning ingebouwd, gebruik van deze method wordt gedetecteerd en er wordt wat extra IL-code voor gegenereerd om de performance te verbeteren. Een andere optimalisatie is het feit dat alle gevraagde *PropertyInfo*-objecten van een type automatisch intern worden gecached. Ondanks de optimalisaties, kan bijvoorbeeld in een ASP.NET-webapplicatie intensief gebruik van reflection in veel geraakte code een bottleneck vormen.
Ook wat security betreft is er een aandachtspunt. Code zonder de juiste permissies (*ReflectionPermission*) heeft geen rechten om non-public waarden van een type uit te lezen, zelfs niet zijn eigen type. In de beschreven scenario's wordt alleen gebruikgemaakt van public properties en is deze permissie niet nodig. Wanneer het Weer-type bijvoorbeeld internal gemaakt zou worden en in een andere assembly dan de testcode staat, dan is *ReflectionPermission* wel nodig. De vuistregel is dat de normale zichtbaarheidregels ook bij reflection gelden. In een scenario waarbij code van onbekende partijen (internet) wordt gebruikt, is het zelfs verstandig om *ReflectionPermission* te ontzeggen in je code.
Nieuw in .NET Framework 2.0 is een hybride vorm tussen puur gebruik van reflection en standaard calls, in de vorm van lightweight code generation. Het wordt gebruikt om tijdens het draaien van de applicatie eenmalig code te genereren en deze uit te voeren. Erg interessant, maar te veel om nog in dit artikel op te nemen. De laatste url bij dit artikel levert meer informatie. Op de juiste manier toegepast, levert het gebruik van reflection als ondersteunende component in een project veel flexibiliteit en kan het de impact van wijzigingen verkleinen.

Loek Duys is als senior developer werkzaam bij VX Company (www.vxcompany.com). Zijn werkzaamheden richten zich op het ontwerpen en implementeren van .NET-oplossingen. Hij is te bereiken via lduys@vxcompany.com.

Reinhard Brongers is werkzaam bij VX Company (www.vxcompany.com) als technisch architect. Hij is te bereiken via rbrongers@vxcompany.com.

Referenties

Meer informatie over het gebruik van reflection:
http://msdn2.microsoft.com/en-us/library/cxz4wk15.aspx
Programming guide voor reflection:
http://msdn2.microsoft.com/en-us/library/ms173183.aspx
Artikel over performance en reflection:
http://msdn.microsoft.com/msdnmag/issues/05/07/Reflection/default.aspx
Gebruik van Lightweight Code Generation, dynamic methods:
http://msdn2.microsoft.com/system.reflection.emit.dynamicmethod.aspx