Denne guide er oprindeligt
udgivet på Eksperten.dk

COMPUTERWORLD

Refleksion med .NET

Refleksion bruges til at undersøge klasser på runtime tidspunktet. Se hvordan du f.eks. loader en klasse fra en DLL, gennemkigger dens metoder (funktioner) og kalder dem. Helt uden at kende klassen i forvejen.

Skrevet den **03. Feb 2009** af **nielle** I kategorien **Programmering / C#** | ★★★☆☆

Indledning

Refleksion giver dig muligheden for - på runtime tidspunktet - at undersøge hvordan en klasse, et interface, en struct, eller noget helt fjerde, er skruet sammen. Med disse oplysninger i hånden kan man bruge klassen som om den var kendt på kompileringstidspunktet.

Man behøver imidlertid ikke at have haft adgang til klassen på det tidspunkt. Man behøves faktisk ikke engang at have haft kendskab til dens eksistens.

.NET frameworket er gennemsyret af såkaldte *metadata* ("data om data"), og refleksion er teknologien som man bruger til at udtrække disse metadata og handle på baggrund af dem.

Hvad kan det så bruges til? En af mulighederne er muligheden for at lave en plugin arkitektur, hvor andre kan berige dit program med ny funktionalitet uden at din kode skal rekompileres: Dette har jeg skrevet om i en tidligere artikel

http://www.eksperten.dk/artikler/1164

Dvs. at give dit program muligheden for at loade og bruge plugins som andre har lavet.

En anden mulighed er et program som NUnit der bruges til unittest:

http://www.nunit.org/index.php

Du placere dine tests i en DLL som NUnit så loader. Derefter skanner NUnit DLL'en for tests og udføre dem. Hvis du ikke allerede kender NUnit bør du tage et seriøst kig på det. Du kan starte med at læse arne_v's udmærkede artikel om det her på Eksperten:

http://www.eksperten.dk/artikler/607

Denne her artikel afslutter i øvrigt med at skitsere skelettet for hvordan NUnit kunne være kodet.

Andre steder, hvor der gemmer sig refleksion under kølerhjelmen, er sådan noget som Visual Studio's Intellisence.

v. 1.0: 18/01/2008 - Første version.

Type-klassen

Type-klassen er den centrale klasse inden for refleksion; enhver klasse har en *type* som indeholder oplysninger om klassen - klassens metadata.

Man kan enten bruge typeof()-operatoren eller GetType()-metoden til at finde typen af en klasse eller et objekt:

```
using System;

namespace RefleksionNS1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Typen af en klasse.
            Type type1 = typeof(EnKlasse);
            Console.WriteLine(type1.Name);

            // Typen af et objekt.
            EnKlasse klasse2 = new EnKlasse();
            Type type2 = klasse2.GetType();
            Console.WriteLine(type2.Name);

            // GetType() må ikke forveksles med ToString():
            Console.WriteLine(klasse2);
        }
    }

    class EnKlasse
    {
        public override string ToString()
        {
            return "Hvis ikke ToString() overstyres vil den returnere:
RefleksionNS1.EnKlasse";
        }
    }
}
```

Når man har typen, kan man udfritte den om forskellige oplysninger om klassen:

```
using System;
using System.Reflection;

namespace RefleksionNS2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Prøv selv med en anden klasse som f.eks. int
            Type typen = typeof(EnKlasse);

            Console.WriteLine(typen.Name);
```

```
            Console.WriteLine(typen.FullName);

            Type baseTypen = typen.BaseType;
            if (baseTypen != null)
                Console.WriteLine("Nedarver fra: " + baseTypen);
            else
                Console.WriteLine("Nedarver ikke fra noget.");

            Console.WriteLine();

            MemberInfo[] miArr = typen.GetMembers();
            foreach (MemberInfo mi in miArr)
            {
                Console.WriteLine(mi.Name);
                Console.WriteLine("\ter en " + mi.MemberType);
                Console.WriteLine("\ter defineret i " + mi.DeclaringType);
            }
        }
    }

    class EnKlasse
    {
        public int SvaretPåLivetUniversetOgAltDetDer()
        {
            return 42;
        }

        public string klasseLærer;

        public string KlasseLærer
        {
            get { return klasseLærer; }
            set { klasseLærer = value; }
        }
    }
}
```

Resultat:

```
EnKlasse
RefleksionNS2.EnKlasse
Nedarver fra: System.Object

SvaretPåLivetUniversetOgAltDetDer
    er en Method
    er defineret i RefleksionNS2.EnKlasse
get_KlasseLærer
    er en Method
    er defineret i RefleksionNS2.EnKlasse
set_KlasseLærer
    er en Method
    er defineret i RefleksionNS2.EnKlasse
```

```
GetType
    er en Method
    er defineret i System.Object
ToString
    er en Method
    er defineret i System.Object
Equals
    er en Method
    er defineret i System.Object
GetHashCode
    er en Method
    er defineret i System.Object
.ctor
    er en Constructor
    er defineret i RefleksionNS2.EnKlasse
KlasseLærer
    er en Property
    er defineret i RefleksionNS2.EnKlasse
klasseLærer
    er en Field
    er defineret i RefleksionNS2.EnKlasse
```

Som man i øvrigt kan se af output, vil en property give anledning til to metoder (foruden sig selv), en for hver af get()- og set()-accessorerne.

Funktionen GetMembers() giver altså oplysninger om klassens indhold; dens metoder inkl. constructore, properties og fields.

BindingFlags

Hvis man derimod specifikt er ude efter en bestemt af typerne, kan man i stedet for GetMembers() bruge en af GetMethods()-, GetConstructors()-, GetProperties()- eller GetFields()-metoderne.

Som udgangspunkt vil en metode som f.eks. GetMethods() returnere oplysninger om alle de metoder som klassen har og som i øvrigt er erklæret som public. Man kan ændre på dette, og man kan endda få fat i metoder som er erklæret private. Dette gøres med den rette kombination af BindingFlags:

```
using System;
using System.Reflection;

namespace RefleksionNS3
{
    class Program
    {
        static void Main(string[] args)
        {
            // Klassens type.
            Type typen = typeof(EnKlasse);

            // Klassens private metoder.
            MethodInfo[] miArr = typen.GetMethods(
```

```
                BindingFlags.NonPublic |
                BindingFlags.Instance |
                BindingFlags.DeclaredOnly);

            // Udskriv dem.
            foreach (MethodInfo mi in miArr)
                Console.WriteLine(mi.Name);
        }
    }

    class EnKlasse
    {
        public int Metode1() { return 42; }
        private int Metode2() { return 7 - 9 - 13; }
        public static int Metode3() { return 100; }
    }
}
```

Resultat: Metode2

BindingFlags.NonPublic giver de metoder som ikke er erklæret public. Dette er selvfølgelig ikke det samme som at sige at de er erklæret private. BindingFlags.Instance giver de metoder som er kræver en instans af klassen - altså modsætningen til metoder som er erklæret som static. BindingFlags.DeclaredOnly giver de metoder der er erklæret i selve klassen - i modsætning til de metoder der er erklæret i klasser der ligger tidligere end EnKlasse i arve-hierarkiet. Uden denne ville man også have fået udskrevet:

MemberwiseClone
Finalize

foruden altså Metode2.

Kodestumperne:

```
            MethodInfo[] miArr = typen.GetMethods(
                BindingFlags.Public |
                BindingFlags.Instance |
                BindingFlags.DeclaredOnly);
```

og:

```
            MethodInfo[] miArr = typen.GetMethods(
                BindingFlags.Public |
                BindingFlags.Static |
                BindingFlags.DeclaredOnly);
```

resulterer i: Metode1 hhv. Metode3.

MethodInfo-klassen

GetMethods() returnere et array af MethodInfo-objekter. Disse beskriver klassens metoder i flere detaljer:

```
using System;
using System.Reflection;

namespace RefleksionNS4
{
    class Program
    {
        static void Main(string[] args)
        {
            Type typen = typeof(EnKlasse);

            MethodInfo[] miArr = typen.GetMethods(
                BindingFlags.Public |
                BindingFlags.Instance |
                BindingFlags.DeclaredOnly);

            foreach (MethodInfo mi in miArr)
            {
                // Metodens navn.
                Console.WriteLine(mi.Name);

                // Hvilken type returnere metoden.
                Console.WriteLine("\t" + mi.ReturnType);

                // Array som beskriver metodens argumenter.
                ParameterInfo[] piArr = mi.GetParameters();

                // Udskriv oplysninger om de enkelte argumenter.
                foreach (ParameterInfo pi in piArr)
                {
                    Console.WriteLine("\t\t{0} {1}",
pi.ParameterType.ToString(), pi.Name);
                    Console.WriteLine("\t\t\tout={0}", pi.IsOut);
                }
            }
        }
    }

    class EnKlasse
    {
        public int Metode1()
        {
            return 42;
        }

        public string Metode2(string argument)
        {
            return "For held: " + (7 - 9 - 13);
        }
```

```
        public void Metode3(ref string arg1, string arg2, out double arg3,
params int[] arg4)
        {
            arg3 = 42D;  // arg3 er defineret som "out" og skal derfor sættes
til en værdi.

            Console.Write("{0} {1} {2}", arg1, arg1, arg3);
            for (int idx = 0; idx < arg4.Length; idx++)
                Console.Write(arg4[idx] + " ");
            Console.WriteLine();
        }
    }
}
```

I samme ånd giver funktioner som GetConstructors(), GetEvents(), GetFields() og GetProperties() oplysninger om hhv. klassens constructores, dens events, dens fields (variable) og dens properties.

Funktionen GetMembers() giver information om samtlige members - dvs. metoder, properties og fields samlet under én hat.

Assembly-klassen

En af de grundlæggende ting, man kan bruge refleksion til, er at loade en kompileret assemply - DLL eller EXE - og undersøge hvad der er inde i den.

Man kan endda oprette instanser af, og kalde funktioner på de klasser man finder på den måde!

NUnit, og muligheden for at lave plugins, er gode eksempler på hvad man kan gøre, hvis man kan undersøge kompileret kode - ikke på kompileringstidspunktet, men på det langt senere runtime tidspunkt.

Lad os kigge nærmere på det. Opret en DLL med følgende kode:

```
using System;

namespace RefleksionNS5
{
    public class EnKlasse
    {
        public void Metode1()
        {
            Console.WriteLine("Metode1() kaldt");
        }

        public string Metode2(string argument)
        {
            return argument + (7 - 9 - 13);
        }
    }

    delegate int EnDelegate(string argument);
```

```
    enum EnEnum { }
    interface EtInterface { }
    struct EnStruct { }
}
```

Dette er det assembly som vi ønsker at undersøge nærmere i det følgende. Læg det i roden af C-drevet eller ret stien i den næste blok kode.

Opret derefter et Console projekt:

```
using System;
using System.Reflection;

namespace RefleksionNS6
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Indlæs assemblyet fra dens position på disken.
                // Andre muligheder er at indlæse den fra f.eks. GAC'en.
                Assembly assemply =
Assembly.LoadFile(@"C:\RefleksionNS5.dll");

                // Undersøg de typer der er i assemplyet.
                Scan4Types(assemply);
            }
            catch (ReflectionTypeLoadException) { }
        }

        ...
    }
}
```

Med Assembly.LoadFile() indlæses den kompilerede DLL, sådan at vi kan kigge efter hvad der ligger i den. Det er Scan4Types() som efterfølgende gør dette:

```
        private static void Scan4Types(Assembly assemply)
        {
            // Hent alle typerne fra assembly'et.
            Type[] typeArr = assemply.GetTypes();

            // Undersøg dem mens de gennemløbes.
            foreach (Type type in typeArr)
            {
                // Hvilken type er det?
```

```
                Console.WriteLine(type.Name);

                // Hvis det er en klasse ...
                if (type.IsClass)
                {
                    // Klassens navn.
                    Console.WriteLine("\tEn klasse");

                    // Undersøg de metoder der er i klassen.
                    Scan4Methods(type);
                }

                // Et par andre muligheder.
                if (type.IsEnum) Console.WriteLine("\tEn enum");
                if (type.IsInterface) Console.WriteLine("\tEt interface");
                if (type.IsValueType) Console.WriteLine("\tEn value type
(f.eks. en enum eller struct)");
            }
        }
```

Koden starter med at bede assembly'et som dens typer. Af udskriften kan man se at det ikke kun er EnKlasse som vises på denne måde - også EnDelegate, EnEnum, EtInterface og EnStruct har en type. Det er ikke kun klasser som har en type.

De viste er ikke de eneste C# konstruktioner som har en type - det er blot dem af de mulige som kan optræde helt ude på namespace niveau.

Derefter bruger koden properties, som IsClass, IsEnum, IsInterface og IsValueType, til at spørge ind til hvad der egentlig gemmer sig i typen.

Det kan måske komme som en overraskelse at en delegate faktisk er en klasse. Men det er den, som man kan se af udskriften (når vi er færdige).

Hvis programmet i øvrigt konstatere at det er en klasse, som gemmer sig bag typen, kaldes Scan4Methods() som løber igennem dennes metoder:

```
        private static void Scan4Methods(Type classType)
        {
            // Hent oplysninger om alle metoderne i klassen.
            MethodInfo[] methodInfoArr = classType.GetMethods();

            // Gennemløb metoderne.
            foreach (MethodInfo methodInfo in methodInfoArr)
            {
                // Udskriv metodens navn
                Console.WriteLine("\t\t" + methodInfo.Name);

                // Hvis det endda er EnKlasse klassen ...
                if (classType.Name == "EnKlasse")
                {
                    // ... så "leger" vi lidt med den og dens metoder.
```

```
                    // Afprøv Metode1().
                    if (methodInfo.Name == "Metode1")
                        CallMetode1(classType, methodInfo);

                    // Ditto for Metode2().
                    if (methodInfo.Name == "Metode2")
                        CallMetode2(classType, methodInfo);
                }
            }
        }
```

Denne udskriver i denne version bare metodernes navne - hvilket jo kan være interessant nok i sig selv.

Activator-klassen og Invoke funktionen

Hvis det endeligt konstateres at det endda er EnKlasse klassen, så går programmet et skridt videre og prøver at kalde metoderne Metode1() og Metode2(). Først koden for Metode1():

```
        private static void CallMetode1(Type enKlasseType, MethodInfo method1)
        {
            // Metoden var defineret som:
            // public void Metode1()

            // Opret en instans af klassen EnKlasse.
            object enKlasseObj = Activator.CreateInstance(enKlasseType);

            // Kald Metode1() på instansen.
            Console.Write("\t\t\tResultat: ");
            method1.Invoke(enKlasseObj, null);
        }
```

En klasse kan have både static-erklærede og instans metoder. I dette tilfælde er der kun tale om instans-variable. For at kalde sådanne metoder skal man først have en instans - dvs. et objekt af klassen. Normalt ville man f.eks. kalde Metode1() på denne måde:

```
EnKlasse ek = new EnKlasse();
ek.Metode1();
```

Det kan vi imidlertid ikke gøre i dette tilfælde. Det første problem er at vi faktisk slet ikke har inkluderet definitionen af EnKlasse i vores konsol applikation. Husk på at EnKlasse jo bare er repræsenteret ved en type som vi tilfældigvis fandt i en DLL som vi loadede direkte fra harddisken.

Alligevel kan vi lave en instans af denne klasse vha. refleksion. Det gør Activator-klassen med denne kommando:

```
object enKlasseObj = Activator.CreateInstance(enKlasseType);
```

Det kunne være fristende at skrive:

```
EnKlasse enKlasseObj = Activator.CreateInstance(enKlasseType);
```

Men den går heller ikke. EnKlasse-definitionen er jo som sagt ikke inkluderet i vores applikation.

Her er det så på sin plads at nævne en smule om constructoren: I EnKlasse-koden er der ikke angivet nogen constructor. Activator.CreateInstance() bruger derfor klassens default-constructor. En klasse har altid en costructor, og hvis man ikke selv har angivet en så vil .NET selv tilføje en. Det er den man får udfør nor man skriver:

```
new EnKlasse();
```

Hvis man selv har udvidet klassen med en eller flere constructore, har man mulighed for at kalde nogen af overload versionerne af CreateInstance() for at få Activator-klassen til at bruge dem i steder for default-constructoen.

Vi har også en MethodInfo som beskriver den metode vi ønsker at køre; et objekt af typen MethodInfo har en Invoke() metode. Syntaksen er lidt sjov - essentielt set beder man metoden om at køre sig selv fra instansen af klassen:

```
object enKlasseObj = Activator.CreateInstance(enKlasseType);
method1.Invoke(enKlasseObj, null);
```

hvilket altså svare nogenlunde til dette:

```
EnKlasse enKlasseObj = new enKlasseObj()
enKlasseObj.Metode1();
```

fordi method1-argumentet, i kaldet til CallMetode1(), i dette tilfælde repræsenterer en MethodInfo for metoden Metode1(). Tjek selv efter hvor den kaldes i Scan4Methods() ovenfor.

Koden for at kalde Metode1() var relativt simpel. Dette skyldes at den hverken tager argumenter eller returnere noget som resultat af kaldet.

Der skal lidt mere med i kaldet af Metode2():

```
        private static void CallMetode2(Type enKlasseType, MethodInfo method2)
        {
            // Metoden var defineret som:
            // public string Metode2(string argument)

            // Opret en instans af klassen EnKlasse.
            object enKlasseObj = Activator.CreateInstance(enKlasseType);

            // Et array af argumenter til metoden. Der skal bruges
            // et array også selv om der som her kun er et enkelt
            // argument, nemlig en string.
            object[] parameters = { "Lykken er: " };

            // Kald Metode2() på instansen med parametrene
            // i arrayet. Gem returværdien i resultObj.
            object resultObj = method2.Invoke(enKlasseObj, parameters);

            // Vi ved at det faktisk var enKlasseObj string.
            string resultStr = resultObj as string;

            // Udskriv resultatet.
            Console.WriteLine("\t\t\tResultat: " + resultStr);
        }
```

Argumenterne skal serveres som et array af objekter. Dette kan selvfølgelig godt virke lidt som overkill, hvis der nu kun er ét argument som det er tilfældet her. Ovenfor, for Metode1() brugte jeg i øvrigt bare en null på denne plads.

Resultatet returneres i et objekt. Dette castes til en string og udskrives.

Jeg har "snydt" lidt her og brugt at jeg kender metoderne på forhånd. Både metodernes argumenter, typerne og antallet af dem, samt deres returtype var kendt på forhånd. I et tidligere eksempel viste jeg hvordan man kunne detektere disse oplysninger på runtime tidspunktet.

Attribute-klassen

Refleksion handler primært om metadata - hvad kan klasser, interfaces, metoder, fields, osv. fortælle om sig selv. Alle disse ting er indbygget i .NET frameworket på forhånd.

Man har imidlertid selv tilføje oplysninger. Dette gøres vha. *attributter* - hvilket blot er klasser som arver fra Attribute-klassen.

Du har sikkert allerede stødt på attributter uden at vide det. F.eks. hvis du har set en af disse linjer i noget af den kode du har været igennem:

[assembly: AssemblyTitle("ReflectionNS6")]
[DllImport]
[Obsolete]
[Serializable]
[STAThread]

[Test]

Attributter er en måde at hægte ekstra metadata på en klasse uden ellers at ændre den måde selve klassen fungere på. Normalt er de simpelthen en slags besked til andre programmer om at behandle klassen på en speciel måde.

Det nævnte NUnit bruger attributter til at markere om noget er en klasse med test-metoder (TestFixtureAttribute) og om en given metode i en klasse er en test (TestAttribute).

Hvis vi selv skulle lave en lille NUnit klon, kunne starten på denne se nogenlunde sådan her ud:

```
using System;
using System.Reflection;

namespace NUnitKlon
{
    [AttributeUsage(AttributeTargets.Class)]
    class TestFixtureAttribute : Attribute { }

    [AttributeUsage(AttributeTargets.Method)]
    class TestAttribute : Attribute { }

    ...
}
```

Det er en konvention at attributten navngives med et "Attribute" sidst i navnet.

Læg i øvrigt mærke til at der endda er brugt attributter på definitionen af de to attributter. I dette tilfælde betyder det at TestFixture kun kan bruges på klasser og at Test kun kan bruges på metoder.

Under normale omstændigheder ville man have sine tests liggende i en separat DLL og ikke blande dem sammen med NUnit koden. Lad os se bort fra det, og med det sammen definere to klasser med test metoder i:

```
    [TestFixture]
    class UnitTestSuite1
    {
        [Test]
        public void Test1() { Console.WriteLine("Test1 udført."); }

        [Test]
        public void Test2() { Console.WriteLine("Test2 udført."); }
    }

    [TestFixture]
    class UnitTestSuite2
    {
        [Test]
        public void TestX() { Console.WriteLine("TestX udført."); }
```

```
        public void TestY() { Console.WriteLine("Bliver ikke testet."); }
    }
```

Læg mærke til at der skrives [TestFixture] og [Test] i stedet for [TestFixtureAttribute] og [TestAttribute]. Det er faktisk .NET som tillader at man kan udelade "Attribute" hvis man ønsker det.

Herefter selve programmet, som starter med at indlæse sig selv og skanne for typer:

```
    class Program
    {
        static void Main(string[] args)
        {
            // Load dette assembly ...
            Assembly thisAssembly = Assembly.GetExecutingAssembly();

            // ... og led alle typerne igennem.
            Scan4Types(thisAssembly);
        }

        ...
    }
```

Programmet går målrettet efter at finde de klasser som er dekoreret med en TestFixture attribut. Dette er de klasser hvor at der er test i. Ved at markere klasserne med attributten undgår man at skulle lede samtlige klasser igennem efter noget som kunne ligne en test metode - man kan dermed gå målrettet efter dem:

```
        private static void Scan4Types(Assembly thisAssembly)
        {
            Type[] typeArr = thisAssembly.GetTypes();

            foreach (Type type in typeArr)
            {
                // Er det en klasse som er dekoreret med en TestFixture-
attribut.
                if (type.IsClass && IsTestFixture(type))
                {
                    // Led alle metoderne igennem.
                    Scan4Tests(type);
                }
            }
        }

        private static bool IsTestFixture(Type type)
        {
            // Hent alle de attributter som er dekoreret med en
TestFixtureAttribute.
            object[] attrArr =
```

```
type.GetCustomAttributes(typeof(TestFixtureAttribute), false);

            // Returnere sand hvis der mindst er en. Ellers false.
            return (attrArr != null && attrArr.Length > 0);
        }
```

Når programmet har fundet en passende klasse skannes den for metoder som er markeret med [Test]. Ved at skelne disse fra dem som ikke har denne attribut, kan man skelne de rigtige tests far de andre metoder (hjælpe metoder):

```
        private static void Scan4Tests(Type type)
        {
            MethodInfo[] methArr = type.GetMethods(
                BindingFlags.Public |
                BindingFlags.Instance |
                BindingFlags.DeclaredOnly);

            foreach (MethodInfo meth in methArr)
            {
                // Er det en metode dekoreret med en Test-attribute?
                if (IsTest(meth))
                {
                    // Kør testen.
                    RunTest(type, meth);
                }
            }
        }

        private static bool IsTest(MethodInfo meth)
        {
            // Hent alle de attributter som er dekoreret med en TestAttribute.
            object[] attrArr = meth.GetCustomAttributes(typeof(TestAttribute),
false);

            // Returnere sand hvis der mindst er en. Ellers false.
            return (attrArr != null && attrArr.Length > 0);
        }
```

Til sidst køres de test-metoder der er fundet:

```
        ...

        private static void RunTest(Type type, MethodInfo meth)
        {
            object testFixture = Activator.CreateInstance(type);
            meth.Invoke(testFixture, null);
        }
    }
```

```
}
```

I sin nuværende form indeholder koden ikke noget tjek af at test-metoderne rent faktisk er på formen:

```
public void MethodeNavn() { ... }
```

Der er et par enkelte andre steder hvor programmet ligeledes kunne strammes om, men det viste burde forhåbentlig vis demonstrerer grundprincippet.

Efterord

Ovenstående har blot ridset overfladen af nogen af de muligheder man har: F.eks. kan man også få fat i selve MSIL koden. Eller man kan endda generere ny kode på runtime tidspunktet.

Jeg håber dog at det er nok til at du har fået blod på tanden til at gå videre.

Kommentar af miclarsen d. 12. Feb 2008 | 1

Kommentar af bertelsenbo d. 15. Feb 2008 | 2

Dækker det essentielle og er godt forklaret

Kommentar af sph1nx d. 06. Aug 2008 | 3