Jeffrey van Gogh
is Software Developer Engineer voor het .NET Framework SDK Team bij Microsoft in Redmond

.NET Design Guidelines

Meer tijd voor het schrijven van nieuwe uitdagende code

Tijdens de ontwikkeling van het .NET Framework is een eigen set ontwerprichtlijnen opgesteld en zijn deze beschikbaar gesteld op internet[1]. Deze set kan een goede basis vormen voor de richtlijnen van uw bedrijf. Bij Microsoft zijn deze richtlijnen zo belangrijk dat nieuwe code alleen maar mag worden toegevoegd als deze voor 100% voldoet aan de richtlijnen.

Er bestaat een wereldwijde competitie om zo onleesbaar mogelijke programmacode te schrijven[2]. Het doorlezen van een aantal codefragmenten van de verschillende winnaars toont een overduidelijk beeld hoe onleesbaar programmacode kan zijn. Iedere programmeur die zijn code een half jaar na programmeren terugleest zal even flink achter zijn oren moeten krabben om te herinneren hoe de code precies in elkaar zat, zelfs als er prima documentatie bestaat. Code van een andere programmeur doorgronden is de oorzaak voor vroegtijdige kaalheid bij een flink aantal programmeurs. Zeker nu het gebruik van componenten en web services aan het toenemen is, zal het schrijven van fatsoenlijke code en interfaces nog belangrijker worden. Ontwerprichtlijnen zijn daar het belangrijkste instrument voor. Microsoft's ontwerprichtlijnen zijn in de volgende groepen te verdelen:

- Benaming
- Class member-gebruik
- Typegebruik
- COM Interop
- Foutafhandeling
- Array-gebruik
- Operator overloading & gelijkheids overloading
- Type casting
- Een aantal vaak voorkomende patronen
- Security
- Threading gebruik
- Gebruik van asynchronisch programmeren

Voor veel van deze regels geldt dat het afspraken zijn zonder speciale logica. Het is standaard om een bepaald stuk code voor alle ontwikkelaars eenduidig te krijgen.

Een voorbeeld hier van is dat classnamen met een hoofdletter beginnen en parameter-namen met een kleine letter. Deze keuze is alleen gemaakt vanwege de wens om onderscheid te maken, niet vanwege de achterliggende technieken.

Regels veranderen ook mee met veranderende programmeeromgevingen: vroeger codeerde men vaak in de naam van een variable het type van die variable. Tegenwoordig is dit niet meer nodig omdat de typeinformatie beter in de metadata van .NET staat beschreven en met behulp van IntelliSense in Visual Studio .NET zichtbaar wordt. De programmacode wordt nu juist duidelijker door de typeinformatie juist niet mee te coderen in de naam. Een aantal regels heeft een dieper achterliggende gedachte. Kijk bijvoorbeeld naar het stuk programmacode in afbeelding 1. De code voldoet aan de Naming Guidelines, de class begint zoals het hoort met een hoofdletter en heeft netjes als postfix de naam Exception. En als de applicatie wordt 'gedraaid' werkt deze bij veel programma's prima. Toch kan deze class flink wat problemen opleveren, vooral als er later functionaliteit aan het programma wordt toegevoegd zoals het ontsluiten via web services. Wat is er namelijk aan de hand? Er zijn twee ontwerpproblemen hier:

1. Exceptions moeten uitbreiding zijn van ApplicationException of SystemException. Op die manier kan een ontwikkelaar ervoor kiezen om alle applicatie-afhankelijke exceptions af te vangen en de systeem exceptions op een hoger punt te laten afvangen

1 MSDN Design Guidelines - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp

2 The International Obfuscated C Code Contest - http://www.ioccc.org/

```
public class DemoException : Exception
{

  private string oops;

  public string Oops { get { return oops; } set { oops = value; }}

  public DemoException(string oops):
   base(oops)
  {
   Oops = oops;
  }
}
```

Afbeelding 1.

```
[Serializable]
public class DemoException : ApplicationException
{
  private string oops;

  public string Oops { get { return oops; } set { oops = value; }}

  public DemoException() :
   base()
  {
  }

  public DemoException(string message, Exception e) :
   base(message, e)
  {
  }

  public DemoException(string oops):
   base(oops)
  {
   Oops = oops;
  }

  protected DemoException(SerializationInfo info,
   StreamingContext context)
   : base(info, context)
  {
   Oops = info.GetString("oops");
  }

  public override void GetObjectData(SerializationInfo info, Stre-
amingContext context)
  {
   info.AddValue("oops", Oops);
   base.GetObjectData(info,context);
  }
}
```

Afbeelding 2.

(met bijvoorbeeld een OutOfMemory-Exception kan het programma toch niet veel, hoogstens de gebruiker van een probleem op de hoogte stellen).

2. In .NET kunnen objecten op andere machines worden aangeroepen (via web services, remoting, COM+ etc.), dus ook Exceptions moeten worden overgestuurd. Om dit te kunnen doen moet een Exception serializable zijn. Hiervoor moet aan de Exception de volgende elementen worden toegevoegd:
   - Serializable attribuut.
   - Public constructor zonder parameters (om het object te kunnen instantiëren na serializatie).
   - Public constructor met string & exception parameter (om exceptions vanuit COM te genereren).
   - Protected constructor met een serializatie info & context object (om exceptions via Remoting & XML Web Services door te geven) waardoor de lokale variabelen worden gedesynchroniseerd.
   - Protected methode GetObjectData die de lokale variabelen synchroniseert

De exception class komt er dan uit te zien zoals in afbeelding 2. Een ander nog minder snel te vinden probleem in code staat in afbeelding 3. Dit stukje code wil een /i of /I switch meekrijgen. Ook dit stukje code lijkt perfect te werken, echter zodra het product internationaal wordt gebruikt, blijkt het bijvoorbeeld in Turkije niet te werken. Dit komt doordat in Turkije twee vormen van de letter i bestaan:

Hoofdletter	Kleine letter
I	ı
İ	i

De hoofdletter I zonder puntje wordt vertaald naar kleine letter ı zonder puntje. En de kleine letter i met puntje naar hoofdletter İ met puntje. .NET ondersteunt door de ingeweven globalisatie dit alfabet (binnen het hele framework wordt gebruikt gemaakt van Unicode) en daardoor klopt onze code niet meer. We kunnen dit oplossen door .NET te vertellen dat hij de parameter cultuuronafhankelijk moet behandelen (zie de code in afbeelding 4). Het 'Turkse' voorbeeld

```
static void Main(string[] args)
{
  foreach(string arg in args)
  {
   if(arg.Length<2 || arg[0]!='/')
       Console.WriteLine(
       "Please give correct parameter switches");
   else
   {
     String UpperCase = arg.ToUpper();
     switch(UpperCase[1])
     {
       case 'I':
          Console.WriteLine(
             "Magic /I switch used, Cool!");
          break;
       default:
          Console.WriteLine(
             "Boring other switch used :-(");
          break;
     }
   }
  }
}
```

Afbeelding 3.

```
static void Main(string[] args)
{
  foreach(string arg in args)

  {
   if(arg.Length<2 || arg[0]!='/')
     Console.WriteLine(
       "Please give correct parameter switches");
   else
   {
     String UpperCase = arg.
       ToUpper(CultureInfo.InvariantCulture);
     switch(UpperCase[1])
     {
       case 'I':
         Console.WriteLine(
             "Magic /I switch used, Cool!");
         break;
       default:
         Console.WriteLine(
             "Boring other switch used :-(");
         break;
     }
   }
  }
}
```

Afbeelding 4.

FxCop

De meest ervaren .NET-programmeur zal een aantal van de vele ontwerpvoorschriften af en toe vergeten tijdens zijn programmeerwerk. Om toch te zorgen dat de code aan de ontwerprichtlijnen blijft voldoen hebben we bij Microsoft een tool ontwikkeld, genaamd FxCop. Deze tool toetst de code automatisch aan de ontwerprichtlijnen. Omdat .NET meerdere talen ondersteunt, moet de tool dat natuurlijk ook kunnen. Vandaar dat FxCop via Reflectie en IL-Parsing de gecompileerde code inspecteert. In FxCop kunnen de assemblies van uw code worden opgegeven, waarna ze vervolgens worden geanalyseerd. Daarna geeft FxCop u een lijst met afwijkingen van de ontwerprichtlijnen inclusief het advies hoe deze op te lossen.

Zoals eerder gezegd zijn ontwerprichtlijnen niet iets wat alleen door Microsoft wordt ontworpen, maar kunnen ze ook binnen uw eigen bedrijf worden ontworpen. Daarom biedt FxCop u ook de mogelijkheid om eigen regels te ontwerpen. Het programmeermodel hiervoor is zo simpel mogelijk opgezet. U kunt de volgende items analyseren: Assemblies, Resources, Namespaces, Types, Events, Methods, Properties, Fields, Parameters
Om een zo'n item te analyseren implementeert u de bijbehorende interface: ITypeRule voor Types. De interface bevat 1 functie: `string Check(Type type)`; De parameter is het type dat gechecked moet worden. Vindt u een fout dan retourneert u een string met een mogelijke oplossing. Wordt het type goed bevonden dan retourneert u null. Naast deze interface per verschillende regel, moet u ook nog de interface IRule implementeren. In deze interface definieert u met behulp van properties de gegevens van de regel, zoals beschrijving, contact email etc. Vervolgens compileert u deze regel en voegt de assembly in de FxCop User Interface toe. De volgende keer dat u uw code analyseert zal uw nieuwe regel automatisch mee worden genomen in de check.

laat zien dat fouten snel over het hoofd worden gezien, maar ongemerkt snel in de hele applicatie kunnen opspelen.
Programma's zijn vaak grote lappen code. Om het nalezen (en repareren) van deze code een stuk minder tijdrovend te maken zijn ontwerprichtlijnen een zeer goed gereedschap. Daarnaast voorkomen de richtlijnen ook een flink aantal bugs. Een zaak dus om goed aandacht aan te schenken. De ontwerprichtlijnen van Microsoft bieden een goede basis hiervoor en samen met de FxCop tool kan er veel ergernis en tijd worden bespaard die weer in het programmeren van uitdagende nieuwe programmacode kan worden gestopt.

Nuttige internetadressen

- http://www.ioccc.org/
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp
- FxCop is te downloaden vanaf http://www.gotdotnet.com/team/libraries/