André Obelink (MCSD) is eindredacteur bij de Nederlandse Visual Basic Groep

Debugging en Tracing met Visual Studio .NET

BETERE EN STABIELERE APPLICATIES DANKZIJ VERNIEUWDE DEBUGGER

De debugging-mogelijkheden van Visual Studio .NET zijn sterk verbeterd vergeleken met die van Visual Studio 6. Aan de ene kant bestaan die verbeteringen uit uitbreidingen in de IDE en aan de andere kant zijn er nu ook objecten beschikbaar waarmee de ontwikkelaar de sourcecode of de applicatie kan instrumenteren.

Met de komst van een nieuwe ontwikkeltaal en -omgeving wordt vaak het eerst gekeken naar de taaluitbreidingen. Daarnaast besteedt iedereen veel aandacht aan zaken als verbeterde Intellisense, automatische codegeneratoren of het kunnen toepassen van OOP-technieken. Vaak vergeten ontwikkelaars echter de eventuele verbeteringen te bestuderen die de 'vernieuwde debugger' biedt. Aangezien de debugger dagelijks wordt gebruikt, is dit eigenlijk heel vreemd. Door ook de nieuwe mogelijkheden van de debugger te bestuderen, kunnen ontwikkelaars snel betere en stabielere applicaties schrijven. En dat elke dag weer...

Soorten fouten

Wanneer we praten over fouten opsporen en het oplossen ervan, is het verstandig eerst eens te kijken naar welke typen fouten kunnen optreden. Er zijn in feite drie typen fouten: *Compile-Time errors*, *Run-Time errors* en *Logical errors*.We praten over 'Compile-Time errors' wanneer de compiler de code niet kan compileren. Compile-Time errors noemen we dikwijls syntaxfouten maar eigenlijk gaat het ook over lexicale- of semantische fouten. Vaak is de oorzaak een typefout of een functieaanroep zonder de vereiste parameters. Deze fouten zijn eenvoudig op te sporen. Telkens wanneer de code gecompileerd wordt, zijn de fouten te herkennen aan onderstreepte code. Ook komen ze op de 'Tasklist' te staan. Wanneer je dubbelklikt op zo'n fout, wordt je direct naar de desbetreffende code geleid. In veel gevallen is de foutomschrijving zo duidelijk, dat het oplossen van de fout eenvoudig is. 'Run-Time errors' treden op wanneer de applicatie iets probeert te doen wat op dat moment niet mogelijk of toegestaan is. Hierbij kun je denken aan delen door nul, of door een bestand te openen op een niet beschikbare netwerkbron. Dit type fouten is op te vangen met foutafhandeling en deels te voorkomen door defensief te programmeren. Wat ingewikkeldere fouten kun je opsporen met de beschreven hieronder tools. De zogenaamde 'Logical errors' zijn de lastigste fouten. Je krijgt geen foutmelding en alles lijkt goed te gaan. Het resultaat is echter niet wat je verwacht: berekeningen kloppen niet of onjuiste data zijn weggeschreven naar de database. Met name het tracen van informatie kan in dit soort gevallen uitkomst bieden, maar daarover later meer.

Visual Studio .NET 2003 IDE

De IDE van Visual Studio zit boordevol windows en tools om code te debuggen. Misschien zijn ze niet allemaal als nieuw voor deze versie te betitelen, maar ze zijn zeker het vermelden waard.

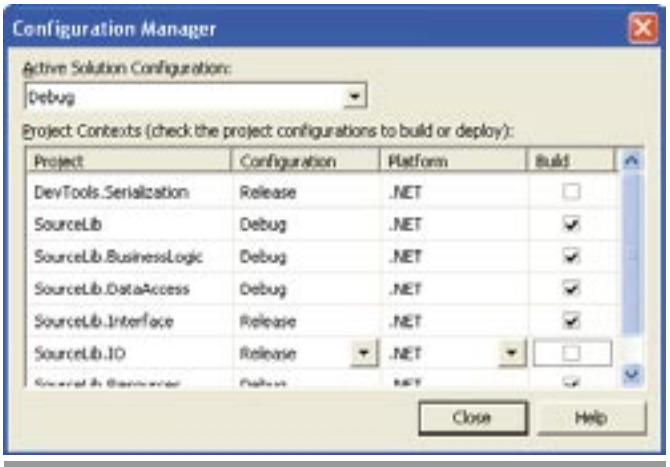
Configuration Manager

De Configuration Manager is een onderdeel dat zeer in het oog springt. Deze optie bevindt zich namelijk, als combobox, zeer prominent in de standaard toolbar. Met deze tool kun je zelf aangeven of een assembly gecompileerd moet worden, en zo ja op welke wijze dat moet gebeuren. Belangrijk om te weten is, dat je code alleen kunt debuggen wanneer het als zodanig - dus 'Debug' - wordt gecompileerd. In dit geval wordt er een 'debugging symbol'-bestand (*.pdb) aangemaakt. Dit bestand is essentieel voor het kunnen debuggen van de code. Het bevat informatie over functienamen of namen van controls. Het pdb-bestand maakt het mogelijk om tijdens het debuggen de corresponderende broncode te kunnen weergeven.

Het compileren in het algemeen, maar ook het aanmaken van zo'n pdb-bestand, is een tijdrovende bezigheid. Met name in trajecten waarbij de te bouwen solution bestaat uit meer projecten kun je hier kostbare tijd winnen. Je zet het opnieuw compileren uit bij projecten die al uitgebreid getest zijn. Build-configuraties bestaan op twee niveaus. Een daarvan is het solution-niveau, dat invloed heeft op een build van alle projecten in die solution; zie afbeelding 1. Het andere niveau is het projectniveau, dat invloed heeft op het afzonderlijke project. Deze instellingen hebben bijvoorbeeld betrekking op welke bestanden worden ingesloten, op welke locatie deze worden gemaakt of op welke wijze deze worden geoptimaliseerd. Je kunt deze instellingen bekijken en wijzigen via het 'Properties'-scherm van je project.

Breakpoints

Het instellen van breakpoints is waarschijnlijk wel de meest gebruikte manier van debuggen. De mogelijkheden van deze breakpoints zijn fors uitgebreid. Zo ben je flexibeler in het gebruik van condities, worden de 'breakpoint-instellingen' opgeslagen en kun je (bepaalde)



Afbeelding 1. Configuration Manager

breakpoints ook eenvoudig tijdelijk buiten gebruik stellen. Met name de laatste twee uitbreidingen werken erg prettig. Wanneer je de 'Breakpoints-window' selecteert heb je een duidelijk overzicht van alle breakpoints die gedefinieerd zijn en kun je deze vanuit hier ook makkelijk verwijderen of tijdelijk buiten gebruikstellen.

Ouput-window

De Output-window toont alle commandline-informatie tijdens de compilatie en de uitvoering van het programma. Bijvoorbeeld welke assemblies geladen zijn en de informatie die, via *Debug.Write*, beschikbaar komt. Je kunt dit scherm zichtbaar maken via View ⇒Other Windows ⇒Output.

Drie windows: Locals, Autos en Watch

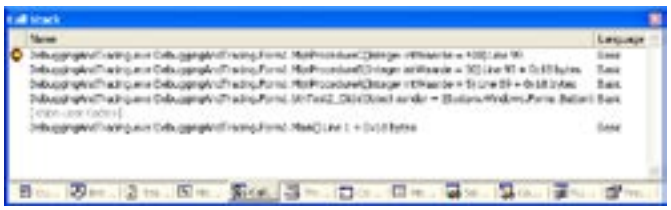
Drie windows die erg op elkaar lijken zijn Locals-, Autos- en Watch-windows. In alle drie windows kun je tijdens het debuggen de status en waarden zien van de variabelen. Elk window bestaat uit drie kolommen: de naam, de waarde en het type van de variabele. Complexe structuren of objectcollecties worden weergeven als een treeview, zodat de bijbehorende datastructuren eenvoudig opengeklapt kunnen worden.
De Locals-window toont de waarden van alle variabelen in de huidige scope; dit in tegenstelling tot de Autos-window. Dit window toont in C# de variabelen van de huidige en vorige regel en voor Visual Basic .NET de huidige coderegel en drie regels ervoor en erachter.
De Watch-window heeft tot doel de waarde van bepaalde variabelen te volgen gedurende het hele programmaverloop, ook al raken ze 'out-of-scope. Je hebt vier Watch-windows tot je beschikking, zodat je de 'watches' ook nog kunt structureren en groeperen. De inhoud van al deze windows wordt tijdens het afsluiten van de IDE opgeslagen bij het project, zodat je deze een volgende keer weer direct tot je beschikking hebt.
Je kunt de waarden van de numerieke en stringwaarden ook vanuit al deze windows aanpassen. Klik op de waarde (Value) en type een andere waarde. De tekst wordt rood en het programma gaat vervolgens met deze nieuwe waarden verder. Helaas is dit, hoewel wellicht wenselijk, niet mogelijk voor complexe class- of structure-variabelen.

Command- en Immediate-window

Het Command-window is een window dat door een groot aantal ontwikkelaars niet volledig benut wordt of zelf totaal onbekend is. In het Command-window kun je direct opdrachten uitvoeren, waarbij je kunt denken aan opdrachten die anders via het menu kunnen worden geselecteerd. Het is mogelijk het Command-window op te roepen met Ctrl+Alt+A en dan bijvoorbeeld het volgende te typen:

```
> nav msdn.microsoft.nl
```

Met de laatste actie open je een website in de Visual Studio .NET IDE. Het is verbazingwekkend hoeveel opdrachten beschikbaar zijn. Een complete lijst met Pre-defined Command Aliases is te vinden op MSDN-site, zie de lijst met nuttige internetadressen. Gelukkig is in dit scherm Intellisense aanwezig, zodat je ook zonder veel zoekwerk alle mogelijkheden kunt uitproberen. Het Command-window kun je ook draaien in de 'Immediate-mode'. Vanuit het Immediate-window is het mogelijk eigen methodes aan te roepen tijdens het debuggen, niet-complexe variabelen aan te passen of waarden direct naar het Command-window te schrijven met het vraagteken. Een voorbeeld:



Afbeelding 2. Call Stack-window

```
? 10*7
? btnTest.Text
```

Andere windows

Naast bovenstaande windows zijn er nog andere tools die behulpzaam zijn bij het debuggen. Zo is er in Visual Basic .NET het Me-window (in C# het This- window), waarin je een overzicht krijgt van alle members van het object dat geassocieerd is met de huidige method. De Call Stack-window toont de procedure-stack, dus welke weg de code bewandelt om uiteindelijk in de huidige functie te belanden. Afbeelding 2 laat zien dat er een break staat in de functie *MijnProcedureC* en dat deze is aangeroepen door *MijnProcedureB*. Deze functie is weer aangeroepen door *MijnProcedureA*, die zelf blijkbaar is geïnitieerd door *btnTest2_Click()*. Ook zie je waarden die de parameters hebben.

In de Threads-window staat een overzicht van de threads die door de applicatie in gebruik zijn. Vanuit dit scherm zijn ze ook tijdelijk te stoppen en kun je ze later weer laten vervolgen. De Module-window is eigenlijk niets meer dan een lijstje met alle assemblies die door de applicatie gebruikt worden. Wanneer je grote buffers, strings of andere data wilt bekijken die niet geschikt zijn om door één van de Watch-windows te laten tonen, kun je eens kijken naar de Memory-window. Voor de echte hardcore ontwikkelaar zijn er achtereenvolgens Disassembly-window en Registers-window, waarin je zelfs de assemblycode kunt debuggen of kunt controleren welke registerwaarden gewijzigd kunnen worden. De meeste van deze windows zijn te openen via het menu Debug ⇒Windows. Het is echter aan te raden om de bijbehorende toetscombinatie te gebruiken. Hierdoor kun je nog sneller tussen de diverse windows switchen. Met de sneltoetsen, bijvoorbeeld die voor 'Step Into' of 'Step Over', kun je in de debugcode navigeren. Dankzij de sneltoetsen kan de ontwikkelaar zijn code op een heel efficiënte wijze debuggen en stabieler maken.

Edit & Continue

Met name de Visual Basic-ontwikkelaars waren teleurgesteld dat 'Edit & Continue' (E&C) niet meer beschikbaar was in Visual Studio .NET. E&C heeft de mogelijkheid om tijdens de breakmode code aan te passen en, zonder opnieuw te hoeven compileren, deze verder te laten uitvoeren. Zoals je in .NET Magazine #6 (zie het artikel van Edwin Jongsma op pagina 13) hebt kunnen lezen, is deze feature vanaf Visual Basic .NET 2005 weer terug. En dit is fantastisch nieuws!

De Debug- en Trace-classes

Visual Studio .NET biedt een groot aantal tools waarmee ontwikkelaars efficiënt kunnen debuggen. Op codegebied zijn er met de komst van het .NET Framework ook uitbreidingen gekomen. De Debug- en Trace-classes beschikken over de mogelijkheid om op relatief eenvoudige wijze informatie over de huidige toestand van de applicatie te loggen. Dit alles zonder te hoeven overschakelen naar 'breakmode'. Vooral bij grote applicaties, waar we spreken over honderden, zoniet duizenden coderegels, is het stap-voor-stap door de applicatie lopen om een logische fout op te sporen haast ondoenlijk. In dit soort gevallen kunnen deze classes wellicht uitkomst bieden.

Afbeelding 3. Het resultaat van de Assert-opdracht

De grote lijnen

Met behulp van de Debug- en de Trace-classes kun je op basis van bepaalde condities informatie loggen. De informatie staat tijdens het debuggen in het eerder besproken Output-window. Daarnaast wordt de informatie ook gestuurd naar de Listeners-collectie. Deze Listeners-collectie kan een of meer classes bevatten die de ontvangen informatie vervolgens op een bepaalde manier verwerkt. Zo zijn er verschillende type *Listeners*, bijvoorbeeld een die de informatie wegschrijft naar een tekstbestand of naar het Eventlog. Ook zou je zelf een TraceListener kunnen maken die de informatie wegschrijft naar SQL Server. Achteraf kun je deze informatie bestuderen en krijg je inzicht in mogelijke oorzaken van de problemen. De Debug- en Trace-classes zijn in principe gelijk. Ze hebben beide dezelfde properties en methods en schrijven beide ook de informatie naar TraceListeners. Het enige verschil is dat de Trace-class ook haar werk kan doen wanneer de applicatie gecompileerd is en uitgerold bij de klant. Om de informatie ook werkelijk te kunnen loggen, staan zes methods tot ieders beschikking.

1. *Write*. Schrijft altijd de tekst weg naar de Listeners–collectie.
2. *WriteLine*. Hetzelfde als Write, maar voegt ook een regeleinde toe.

```
Dim intA As Int32 = 1
Dim intB As Int32 = 2

' Write --> Altijd loggen
Trace.Write("Deze melding wordt gewoon gelogd")

' WriteIf --> Alleen loggen wanneer conditie = True
Trace.WriteIf(intA > intB, "Deze melding wordt niet gelogd")

' WriteLineIf --> Conditie = True + Carriage Return
Trace.WriteLineIf(intA < intB, "Deze melding wordt wel gelogd")

' Assert --> Conditie = False, toont ook MessageBox
Trace.Assert(intA > intB, "Assertboodschap", "Meer details")
```

Codevoorbeeld 1.
Voorbeelden Trace-class methods

```
Dim objTr As Trace
Dim objTrL1 As New TextWriterTraceListener("c:\MijnLog.log")
Dim objTrL2 As EventLogTraceListener

' Bepaal of EventLog al bestaat...
If objTrL2.EventLog.Exists("MijnLog") = False Then
    '... zoniet maak de log aan
   objTrL2.EventLog.CreateEventSource("Test", "MijnLog")
End If
' Maak de TraceListenerEventLog aan
objTrL2 = New EventLogTraceListener("MijnLog")

' Voeg de TextWriter en EventLogger toe
objTr.Listeners.Add(objTrL1)
objTr.Listeners.Add(objTrL2)

' Schrijf data weg en flush gelijk
objTr.WriteLine("Geklikt om: " & Now.ToLongTimeString)
objTr.Flush()

' Sluit de Listeners
objTrL1.Close()
objTrL2.Close()

' Verwijder Listeners uit de collectie
objTr.Listeners.Remove(objTrL1)
objTr.Listeners.Remove(objTrL2)
```

Codevoorbeeld 2.
Toevoegen van TraceListeners

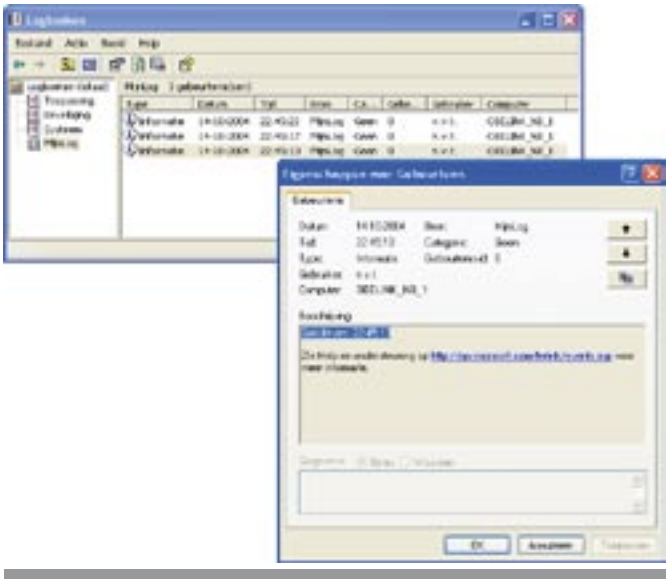
3. *WriteIf*. Schrijft de tekst weg naar de Listeners-collectie, maar alleen wanneer de conditie een True geeft.
4. *WriteLineIf*. Analoog aan WriteLine, schrijft deze methode de tekst alleen weg wanneer de expressie True is en voegt het gelijk een nieuwe regel toe.
5. *Fail*. Schrijft altijd de tekst weg naar de Listeners-collectie, maar toont tevens een Messagebox met daarin de informatie.
6. *Assert*. Schrijft de tekst weg naar de Listeners-collectie, toont een Messagebox, maar doet dit alleen wanneer de conditie False is.

Het gebruik van deze methods is erg intuïtief en zal waarschijnlijk geen problemen opleveren. Bekijk de code in codevoorbeeld 1. Doordat Fail in alle gevallen en Assert in sommige gevallen een MessageBox tonen, zijn deze methods eigenlijk alleen geschikt tijdens het debuggen. Ik denk niet dat het wenselijk is dat de gebruiker wordt geconfronteerd met een dialoogkader, boordevol met informatie, terwijl hij of zij daar feitelijk niets mee kan doen; zie afbeelding 3.
Om de informatie die je gaat wegschrijven wat overzichtelijker te maken, kun je met een aantal andere methods en properties teksten laten inspringen of weer laten terugspringen. Met de IndentSize-property leg je vast hoe groot de inspringing telkens is. Vervolgens realiseer je de inspringing door de Indent()-method aan te roepen. Met UnIndent() gaat de inspringing weer terug en met de IndentLevel-property kun je opvragen in welke hiërarchische laag de tracing zich momenteel bevindt. Deze methods roep je aan tussen de diverse WriteXXX-opdrachten.

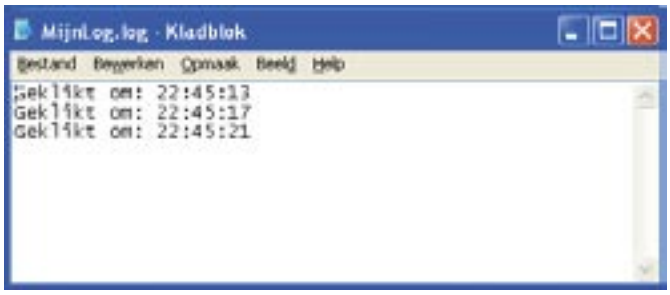
TraceListeners

Alle informatie die gelogd wordt, gaat dus door naar de TraceListeners-collectie. Elk lid van de collectie ontvangt alle informatie. Wat de desbetreffende TraceListener vervolgens met die informatie doet bepaalt dat object zelf. Wanneer de TraceListener geïnitialiseerd wordt, bestaat het direct al uit één lid, namelijk de 'DefaultTraceListener'. Doordat de DefaultTraceListener al aan de collectie is toegevoegd, zie je in de voorgaande voorbeelden de debug-informatie standaard in het Output-window verschijnen.

In productieomgevingen is het onwenselijk naar de Output-window loggen. Beter is het naar een tekstbestand of de Windows Eventlog te loggen. De keuze hangt mede af van het platform waarop de applicatie gaat draaien. Vergeet niet dat de Windows Eventlog niet beschikbaar is op Windows 98. Als dit platform ondersteund moet worden, zal de keuze meestal uitgaan naar een tekstbestand. Ik zeg 'meestal', omdat ook de mogelijkheid bestaat zelf een TraceListener te maken die de data naar een Access-bestand logt. Je zou er zelfs nog voor kunnen kiezen om alle drie de Listeners aan de collectie



Afbeelding 4. Loggen naar de Eventlog



Afbeelding 5. Loggen naar een tekstbestand

toe te voegen. Codevoorbeeld 2 laat zien hoe aan één Trace-class twee TraceListeners worden toegevoegd.

Nadat de Trace-class (objTr) is gedeclareerd, worden de twee TraceListeners gedeclareerd en geïnstantieerd. Indien nodig wordt eerst de Eventlog aangemaakt. De TraceListeners worden aan de collectie toegevoegd en data worden weggeschreven. Om de gegevens nu ook daadwerkelijk naar het bestand te schrijven moeten we 'flushen'. Als laatste sluiten we de TraceListeners en worden ze uit de collectie verwijderd. Het Trace- of Debug-object is standaard aanwezig, als een soort 'Public' object. Je hoeft het immers niet eerst te instantiëren door middel van het keyword New. Dit houdt ondermeer in dat eventuele TraceListeners op verschillende plaatsen in je code kunnen worden toegevoegd aan de collectie. Dit is ook de reden waarom we de TraceListeners weer netjes uit de collectie verwijderen. Wanneer we deze procedure nogmaals zouden aanroepen, zou er een Runtime-error optreden omdat we een TraceListener aan de collectie proberen toe te voegen die zich al in de collectie bevindt. Het resultaat kun je zien in afbeelding 4 en 5.

Trace Switches

Bij het debuggen wil je normaal gesproken alle data in het Output-window zien. Bij het tracen ligt dat anders. Tracen gebeurt alleen in bijzondere gevallen. Gelukkig kun je het functioneren van de Trace-class van buitenaf beïnvloeden met het Application Config-bestand. Dit gebeurt met 'Trace Switches'. Het .NET Framework kent twee typen: een 'BooleanSwitch' en de 'TraceSwitch'. De eerste geeft een Booleaanse waarde terug, dus 'aan' of 'uit'. Met het tweede type, de 'TraceSwitch', kun je een minimumniveau aangeven wanneer getraced moet gaan worden. Er worden vijf niveaus of TraceLevels onderscheiden:

1. *Off*. Er wordt niet getraced (niveau 0).
2. *Error*. Summiere foutmeldingen (niveau 1).
3. *Warning*. Summiere foutmeldingen en waarschuwingen (niveau 2).
4. *Info*. Summiere foutmeldingen, waarschuwingen en andere korte boodschappen (niveau 3).
5. *Verbose*. Alle bovenstaande punten uitgebreid met gedetailleerde boodschappen(niveau 4).

De waarde die is ingesteld op de TraceLevel-property kunnen indirect worden uitgelezen door vier read-only properties: TraceError, TraceWarning, TraceInfo en TraceVerbose. Wanneer de TraceLevel is ingesteld op TraceLevel.Info, dan is niet alleen Trace.TraceInfo True, maar ook de ondergeschikte levels: TraceWarning en TraceError zijn dan beide True. Je kunt zo'n App.Config-bestand eenvoudig aan je project toevoegen door te kiezen voor 'Add New Item' en vervolgens voor 'Application Configuration File'. Voeg de sectie <system.diagnostics> toe.

We hebben twee switches aangemaakt, zie codevoorbeeld 3. De BooleanSwitch heeft een waarde True. Het minimale level voor de TraceSwitch is ingesteld op TraceLevel.Info, oftewel 3. Let op: het is jouw verantwoordelijkheid om in de code wel of niet te loggen. Maak gebruik van bijvoorbeeld *WriteIf* of *WriteLineIf* en test op de waarden van de BooleanSwitch of TraceSwitch; zie codevoorbeeld 4.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <switches>
      <add name="MijnBooleanSwitch" value="1" />
      <add name="MijnTraceSwitch" value="3" />
    </switches>
  </system.diagnostics>
</configuration>
```

Codevoorbeeld 3.
BooleanSwitch in App.Config-bestand

```
Dim objBlnSwitch As New BooleanSwitch("MijnBooleanSwitch", "")
Dim objTrcSwitch As New TraceSwitch("MijnTraceSwitch","")

' Value = 1 --> Wel loggen
Trace.WriteLineIf(objBlnSwitch.Enabled = True, "Boolean")

' Value = 3, Error = 1, 1 < 3 -- > Wel loggen
Trace.WriteLineIf(objTrcSwitch.TraceError = True, "Error")

' Value = 3, Verbose = 1, 4 niet < 3 -- > Niet loggen
Trace.WriteLineIf(objTrcSwitch.TraceVerbose = True, "Verbose")
```

Codevoorbeeld 4.
Toepassen van Trace Switches

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <switches>
      <add name="MijnBooleanSwitch" value="1" />
      <add name="MijnTraceSwitch" value="3" />
    </switches>
    <trace autoflush="true">
      <listeners>
      <add name="MijnLog"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="c:\MijnLog.log" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

Codevoorbeeld 5.
TraceListeners in App.Config-bestand

Ik wil je één gave feature niet onthouden, namelijk het dynamisch aanmaken van een TraceListener. Dit kan ook met het App.Config-bestand. We breiden ons bestand uit met de Trace-sectie; codevoorbeeld 5.

Zonder de code uit codevoorbeeld 4 aan te passen, worden de Trace-data nu ook weggeschreven naar het tekstbestand op schijf. Het .NET Framework in combinatie met Visual Studio .NET beschikt over de tools en omgeving om zeer comfortabel te kunnen debuggen. Het is zeker de moeite waard om eens paar uur in te ruimen om alle mogelijkheden te verkennen. Het is een tijdsinvestering waarvan niemand spijt zal krijgen.

André Obelink (MCSD) is sinds 1995 als eindredacteur betrokken bij de Nederlandse Visual Basic Groep (http://www.vbgroup.nl). Voor vragen of opmerkingen kun je mailen naar andre@obelink.com

Nuttige internetadressen

http://www.developer.com/net/vb/article.php/1551871
http://www.informit.com/guides/content.asp?g=dotnet&seqNum=198
http://support.microsoft.com/default.aspx?scid=kb;en-us;313417
http://msdn.microsoft.com/library/en-us/cptutorials/html/trace___debug_classes.asp
http://msdn.microsoft.com/library/en-us/vsintro7/html/vxgrfpredefinedcommand-linealiases.asp