Robert van der Kleij is technisch directeur bij ETX.

MSBuild: Vol verwachting klopt ons hart

DE MOGELIJKHEDEN VAN DE NIEUWE MICROSOFT BUILD ENGINE

Voor het bouwen en testen van grote applicaties is meer dan alleen Visual Studio nodig. Microsoft heeft voor het ontwikkelen van zijn software een eigen build-lab in gebruik, waar iedere nacht de software wordt gebouwd en getest. Tot nu toe was er nog geen goede software beschikbaar om dat soort activiteiten ook in je eigen organisatie uit te voeren. Bij de volgende versie van Visual Studio gaat dat veranderen. MSBuild wordt dan als nieuwe build-engine meegeleverd bij het .NET Framework.

MSBuild werkt met gestandaardiseerde en gedocumenteerde XML-projectbestanden die een serie zelfstandige taken uitvoeren. Denk hierbij aan het genereren van resourcebestanden of het ophalen van code uit Microsoft SourceSafe. Met deze nieuwe build-engine van Microsoft kun je het bouwen en compileren van software uitbreiden en aanpassen aan de wensen van je organisatie. Het uitvoeren van een unit-test of het controleren van de assemblies met FxCop kan zonder problemen als extra taak worden toegevoegd aan de build van elk Visual Studio project. Visual Studio 2005 - voorheen Visual Studio .NET 'Whidbey' genoemd - maakt gebruik van MSBuild-projectbestanden die de bestaande Visual Studio projectfiles volledig gaan vervangen. In dit artikel worden de basisconcepten van MSBuild uitgelegd, zodat je gericht kunt inschatten wanneer en op welke wijze je met dit product aan de slag wilt gaan.

Waarom MSBuild?

Voor ontwikkelaars die nu tevreden zijn over de build-mogelijkheden van Visual Studio kan MSBuild als een rare noviteit overkomen, zonder duidelijke toegevoegde waarde of direct aanwijsbare voordelen. In mijn opinie heeft MSBuild ons, als ontwikkelaars, echter veel te bieden. MSBuild kan namelijk als een SDK voor het bouwproces worden gezien. Op het moment dat je het Visual Studio bouwproces wilt aanpassen of dat je een eigen buildapplicatie wilt maken, kun je MSBuild gebruiken om het bouwproces naar je hand te zetten. Een scenario dat in iedere organisatie voorkomt is het uitrollen en installeren van software. Meestal worden scripts en msi-bestanden met de hand uitgevoerd. Dit kan in de toekomst ook met een buildproject. Door aanvullende installatietaken aan het projectbestand toe te voegen kan een ontwikkelaar of beheerder de applicatie direct op een testmachine laten installeren. Iedere .NET-ontwikkelaar krijgt in de toekomst overigens met MSBuild te maken. Het is een standaard onderdeel van de volgende versie van het .NET framework waarmee je projecten kunt bouwen zonder dat je Microsoft Visual Studio nodig hebt. Wil je wél Microsoft Visual Studio gebruiken, dan komt kennis van MSBuild goed van pas bij het oplossen van compilatieproblemen. Microsoft Visual Studio 2005 laat namelijk al zijn compilatiewerk over aan MSBuild.

```
F:\Build>msbuild build2b.csproj

Microsoft (R) .NET Build Engine version 1.2.30703.27
[Microsoft .Net Framework, Version 1.2.30703.27]
Copyright (C) Microsoft Corporation 2003. All rights reserved.

Target "Build" in project "build2b.csproj"
   Task "CSC"
      Csc.exe /out:"TestApp.exe" "TestApp.cs"
```

Listing 1

```
<Project>
   <Target Name="Build">
      <Task Name="CSC"
            Sources="TestApp.cs"
            OutputAssembly="TestApp.exe" />
   </Target>
</Project>
```

Listing 2

MSBuild.exe

De snelste manier om met MSBuild te beginnen, is het opstarten van de applicatie via de commandline. MSBuild is echter ook te benaderen via nieuwe framework classes. Met de commandlineversie kun je een build starten door een projectbestand als parameter mee te geven; zie listing 1. Elk projectbestand met als extensie '.*proj' kan zo worden meegegeven. Indien jouw huidige directory slechts één project bevat, selecteert MSBuild dat bestand automatisch zonder dat je daarvoor een parameter hoeft op te geven.

Het projectbestand kan er bijvoorbeeld uitzien als in listing 2. In dit voorbeeld bevat het een C# compilatietaak die een executable TestApp.exe oplevert. Alle taken die nodig zijn om projecten te bouwen worden standaard in MSBuild meegeleverd. Zo zijn er taken voor het genereren van resourcefiles, het compileren van VB- en C#-projecten, het uitvoeren van externe programma's en het bouwen van andere MSBuild-projecten.

Concepten

De drie belangrijkste concepten van MSBuild zijn *tasks*, *targets* en *items*. Het eindresultaat van een build wordt altijd geformuleerd als een target. Om tot dat resultaat te komen, moeten taken worden uitgevoerd op basis van de bronbestanden die in de items staan. Het zijn wel begrippen die kort een nadere omschrijving verdienen.

Task

Al het werk in een projectbestand wordt uitgevoerd door een task. Een task is dus eigenlijk het werkwoord van MSBuild. In feite is de engine alleen maar bezig te bepalen welke taken in welke volgorde moeten worden uitgevoerd, en om deze vervolgens op de juiste

manier aan te roepen. Tasks kunnen zowel invoer- als uitvoerparameters bevatten.

Item

De invoer en uitvoer van een task worden door een item gespecificeerd. Een item kan bijvoorbeeld uit een verzameling bronbestanden bestaan die gebruikt worden bij een C# Compilatie Taak (CSC Taak). In listing 3 staat een voorbeeld van een item dat gebruik maakt van wildcards. De dubbele * zorgt ervoor dat recursief in alle folders wordt gezocht naar C#-files. Een item kan via de @(ItemType) syntax worden doorgegeven als parameter aan een taak.

Tasks zijn vaak afhankelijk van elkaar. De uitvoer van de vorige task fungeert als de invoer voor de volgende. Na bijvoorbeeld de task 'compileer resource files' zijn de namen van de gecompileerde resourcefiles nodig als input voor de task 'compileer C# code'. Dit is dus een voorbeeld waarbij een item ook als uitvoer wordt gebruikt.

Target

De target bepaalt het eindresultaat van een build. Een target bestaat uit een verzameling van tasks en andere targets. Een typisch voorbeeld van targets zijn Build en Rebuild. Per project kan een standaard target worden gespecificeerd en iedere target kan op zijn beurt ook weer afhankelijke targets specificeren. Hiermee kun je een hele boom aan afhankelijkheden modelleren.

Besturing van de build

Het hard coderen van instellingen in een MSBuild-bestand maakt het onderhoud van een project lastiger. Dat probleem kun je oplossen door properties te gebruiken. Een property kan op drie manieren worden gespecificeerd: in het projectbestand, via de command prompt of via een environmentvariabele. In listing 3 zou bijvoorbeeld de uitvoerdirectory het beste vervangen kunnen worden door een OutputDirectory property. Het is ook gebruikelijk om een andere directory te nemen voor een Release of Debug-build. Door condities toe te voegen kan dit, afhankelijk van de situatie, worden ingesteld. Een conditie kan zowel op property, target als task worden gespecificeerd. Dit geeft je de mogelijkheid om sommige acties optioneel te maken. Listing 4 maakt bijvoorbeeld alleen een directory aan als het nodig is.
Een soortgelijke functionaliteit is ook mogelijk met een afhankelijkheidsanalyse van targets. Door op een target de input en output te specificeren kan MSBuild bepalen wat de afhankelijkheden zijn voor die target. Als de inputbestanden onveranderd zijn,weet MSBuild dat de tasks in de target niet uitgevoerd hoeven te worden, omdat de target up-to-date is. Dit wordt gedemonstreerd in listing 4 met behulp van Input- en Output-properties op de BouwApplicatie-target.

Tasks en uitbreidingen

Een van de krachtigste mogelijkheden van MSBuild is dat je nieuwe tasks kunt maken en die vervolgens kunt gebruiken in projecten. Veel .NET-projecten hebben nog te maken met de integratie van VB6-componenten. Een zelfgemaakte task kan dan bijvoorbeeld een VB6-compilatietask zijn. De legacy VB6 COM-componenten compileren mee tijdens een .NET-build, zodat problemen met verkeerde guids of interop-libraries tot het verleden behoren.
Het bouwen van een task bestaat uit drie delen. De eerste stap is het afleiden van de framework class Microsoft.Build.Utilities.Task. De Task-class is een base class die standaard logging-functionaliteit bevat. Deze class implementeert de interface ITask die de Build Engine gebruikt om een taak uit te voeren. De structuur van de ITask-interface is eenvoudig. Deze bevat slechts één methode 'bool Execute()' die door de engine wordt aangeroepen. De methode levert true of false op, afhankelijk van het succes van de Execute. In het codevoorbeeld van listing 5 maken we zo een RemoveDirectory-taak.

```
<Project DefaultTargets="BouwApplicatie">
  <Item Type="BronBestanden" Include="**\*.cs" />

  <Target Name="BouwApplicatie">
    <Task Name="CSC"
          Sources="@(Compile)"
          OutputAssembly="bin\TestApp.exe" />
  </Target>

  <Target Name="Clean">
    <Task Name="RemoveDirectory"
          Directories="bin" />
    <Task  Name="MakeDir"
          Directories="bin" />
  </Target>

  <Target Name="HerbouwApplicatie"
          DependsOnTargets="Clean;BouwApplicatie" />
</Project>
```

Listing 3

```
<Property BinDir="Bin\Release"  />
<Property BinDir="Bin\Release"
               Condition="'$(BuildConfig)'=='Release' />

<Target Name="BouwApplicatie" Inputs="@(Compile)"
        Outputs="$(BinDir)\TestApp.exe">

     <Task Name="MakeDir" Directories="$(BinDir)"
           Condition="!Exists('$(BinDir)')" />

     <Task Name="CSC"
           Sources="@(Compile)"
           OutputAssembly="$(BinDir)\TestApp.exe" />
</Target>
```

Listing 4

De tweede stap is het toevoegen een directory-property op de task. MSBuild kan met properties van verschillende types werken en met arrays omgaan. De execute-methode bevat geen parameters, maar door public properties toe te voegen aan de class worden het automatisch parameters van een task in een projectbestand. MSBuild maakt gebruik van reflectie om dynamisch de public properties van de class te vullen. Het gedrag van de engine is verder nog te beïnvloeden door het required attribuut toe te voegen aan een property. Tijdens het aanroepen van een task controleert MSBuild of alle required properties ingevuld zijn. De laatste stap is het implementeren van de execute-methode zelf. Met behulp van logging-functies van de base class kan tracing-informatie worden teruggegeven aan de engine. Deze logging-infrastructuur is volledig open. De listener is als commandline-parameter op te geven. Zo kun je bijvoorbeeld een eigen logger bouwen om de resultaten in je bug-registratiesysteem op te slaan.

Als de task slechts bedoeld is voor één project, kan hij in het projectbestand worden geregistreerd met een UsingTask-tag, zoals in listing 6. In het geval van een generieke task, kun je deze registreren in het Microsoft.BuildTasks-bestand in de framework-directory. Zorg dan dat de task-assembly ook in deze directory staat. De task is zo te gebruiken zonder using-statement in het projectbestand.

MSBuild en NAnt

MSBuild is niet de eerste op XML gebaseerde build-engine voor .NET. Ant, een open source build-engine voor de Java-gemeenschap, is al succesvol geport naar .NET onder de naam NAnt. Deze tool is te dowloaden via de website http://nant.sourceforge.net. NAnt was één van de eerste engines die het concept van XML projectfiles en zelfstandige tasks introduceerden voor het aansturen

```
using System;
using System.Diagnostics;
using System.IO;
using Microsoft.Build.Framework;
using Microsoft.Build.Utilities;

public class RemoveDirectory : Task
{
  private string[] _directories;

  [Required]
  public string[] Directories
  {
    get{ return _directories; }
    set{ _directories = value; }
  } // Directories

  public override bool Execute()
  {
    // Delete each directory
    bool success = true;

    foreach (string directory in Directories)
    {
      try
      {
        if (Directory.Exists(directory))
        {
          LogCommentFromText("Removed");
          Directory.Delete(directory, true);
        }
      }
      catch (Exception)
      {
        LogErrorFromText("Something has gone wrong");
        success = false;
      }
    }
    return success;
  } // Execute
} // RemoveDirectory
```

Listing 5

```
<UsingTask
    AssemblyName="SampleTasks"
    TaskName="FxCop"
/>

<Target Name="Build" >
    <Task
    Name="RemoveDirectory"
    Directories="c:\Build\TempDir"
    />
</Target>
```

Listing 6

van de build. MSBuild en NAnt lijken op het eerste gezicht heel veel op elkaar. Laten we beide tools met elkaar vergelijken, zodat je zelf kunt beoordelen welke voor jou het meest geschikt is.

Een belangrijk verschil is dat bij MSBuild functionaliteit zoals afhankelijkheidsanalyse en communicatie tussen taken in de engine is geplaatst. NAnt stopt deze functionaliteit meer in de tasks zelf. Het voordeel bij NAnt is dat de gebruiker meer controle kan uitoefenen op het verloop van de taken. In NAnt is het mogelijk nieuwe concepten voor afhankelijkheidsanalyse toe te voegen die méér kunnen dan het controleren van timestamps op de lokale harde schijf. Een ftp-task zou bijvoorbeeld bestanden kunnen controleren op een internetserver. Het nadeel van deze aanpak is wel dat de functionaliteit in de taken moet worden gebouwd en dat het daarmee bewerkelijker is om alle taken eenzelfde soort gedrag te geven. In MSBuild is het bouwen van een task gemakkelijker omdat een task minder hoeft te doen. Bovendien kan men ontwikkelomgevingen op een consistente manier integreren en gebruik maken van de algemene engine-functionaliteit, zonder kennis te hebben van de interne werking van de afzonderlijke taken.

Een heel praktisch verschil is ook dat NAnt op dit moment al beschikbaar is en dat je het kunt inzetten met de oudere .NET Frameworks 1.0 en 1.1. MSBuild wordt daarentegen voorlopig alleen nog maar ondersteund in Visual Studio 2005 met het 1.2 Framework. De ondersteuning en integratie van MSBuild projectfiles in Visual Studio 2005 is vanzelfsprekend erg goed, omdat het voor alle projecten wordt ingezet. Kiezen tussen MSBuild en NAnt heeft naar mijn idee voornamelijk met de ontwikkelomgeving en de geschiedenis van het project te maken. Begin je een nieuw project in Visual Studio 2005, dan is MSBuild de meest logische keuze. Heb je een bestaand project in een oudere versie van Visual Studio en werk je al met NAnt, dan zul je het voorlopig bij NAnt moeten houden.

Beperkingen

MSBuild is nog in ontwikkeling en er zijn nu al genoeg wensen voor een tweede versie. Twee belangrijke kanttekeningen: MSBuild biedt op dit moment nog geen ondersteuning voor C++ projecten. Dat betekent dat C++ gebruikers nog steeds met een eigen projectstructuur blijven zitten. Daarnaast heeft MSBuild dezelfde vervelende eigenschap als de twee vorige versies van Visual Studio: het bouwt alleen maar software voor één specifieke versie van het .NET framework. Vooralsnog ga ik ervan uit dat dit slechts een kwestie van tijd is en dat het bij volgende versies mogelijk wordt om zelf de .NET Framework-versie aan te geven waarmee de software gebouwd moet worden.

Goede infrastructuur en integratie

Voor ontwikkelaars die al gewend zijn om te werken met nachtelijke builds en gecontroleerde uitlevering van software kan MSBuild een geschenk uit de hemel zijn. Het biedt een goede infrastructuur voor complexe buildscenario's; het integreert goed in Visual Studio, en zelf ontwikkelde buildtasks kunnen als eersterangs burgers worden toegevoegd aan de projectbestanden. Dit betekent dat in grotere projecten meer focus op het test- en bouwproces van de software zelf mogelijk is en dat minder tijd nodig is voor een goede inrichting van de ontwikkelomgeving. Elk hulpmiddel om betere software te bouwen moet naar mijn idee met twee handen worden aangegrepen. Het is jammer dat we daarmee moeten wachten tot de uiteindelijke Visual Studio 2005 release, maar dan zal ik MSBuild ook met plezier omarmen.

Nuttige URL's

msdn.microsoft.com/vstudio/whidbey
msdn.microsoft.com/vstudio/productinfo/roadmap.aspx
msdn.microsoft.com/Longhorn/toolsamp/default.aspx
msdn.microsoft.com/library/default.asp?url=/library/en-us/dnlong/html/msbuildpart1.asp
nant.sourceforge.net

Robert van der Kleij is lead-designer van BuildMonkey.NET, een onderdeel van de door Microsoft gecertificeerde ETX SoftwareStraat.NET. Voor vragen of opmerkingen kun je mailen naar robertk@etx.nl