Op het gebied van applicatieontwikkeling speelt het modelleren een steeds belangrijkere rol. Het succes van UML en de opkomst van MDA zijn hier sprekende voorbeelden van. Microsoft heeft in eerste instantie de kat uit de boom gekeken, maar heeft sinds enige tijd een belangrijk initiatief in de richting van het modelleren genomen. Binnen VisualStudio 2005 zijn de Domain Specific Language Tools (DSL Tools) gelanceerd.

thema

Domain Specific Languages

Modelleren met Microsoft

DSL Tools bieden niet, zoals UML, een aantal vaste modellen die gebruikt kunnen worden. Het idee achter DSL's is dat een ieder zijn eigen domeinspecifieke modelleertaal kan definiëren. Met DSL Tools kan een complete editor, code-generatie en naadloze integratie met Visual Studio gemaakt worden.

Wat is een DSL? De meeste mensen zijn bekend met zogenaamde 'general purpose languages'. Deze talen zijn geschikt voor een veelheid aan taken. Voorbeelden hiervan zijn de programmeertalen C# en Java en natuurlijk de modelleertaal UML. Een DSL daarentegen is een 'special purpose language', geschikt voor één bepaald domein. Het is dus toegespitst op één bepaalde taak. We kennen allemaal voorbeelden zoals SQL, spreadsheet formules, of allerlei visuele user interface designers. Een DSL bevat gespecialiseerde concepten uit het domein en is binnen zijn domein bijzonder krachtig. Verder is een DSL in de meeste gevallen executeerbaar. Dit gebeurt meestal via code-generatie, maar het kan eventueel ook met behulp van interpretatie. De gegenereerde code maakt in zo'n geval vaak gebruik van een domeinspecifiek framework dat speciaal voor de betreffende DSL ontworpen is. Omdat een DSL op één specifiek domein gericht is, wordt een DSL vaak gebruikt om slechts een deel van een applicatie te ontwikkelen. Andere delen worden vaak met andere DSL's gedaan, of gewoon op de traditionele wijze met de hand gecodeerd. Alleen als de te ontwikkelen applicatie gespecialiseerd is voor het betreffende domein kan met één DSL volstaan worden. Een domein kan in deze context van alles zijn. Je kunt technische domeinen definiëren, zoals webservices of relationele databases, maar ook meer aan business gerelateerde domeinen zijn mogelijk, zoals bijvoorbeeld hypotheken of verzekeringen. De DSL-ontwikkelaar bepaalt dit zelf.

Waarom DSL's? DSL's kunnen om verschillende redenen worden ingezet. Ze hebben een enorm potentieel. Een hoge productiviteit kan behaald worden als standaardcode uit het model gegenereerd wordt. Dit zorgt tevens voor een snelle time-to-market. Code-generatie heeft bovendien ook voordelen op het gebied van het leveren van kwaliteit. Programmeurs willen om allerlei redenen wel eens slordig zijn, al is het maar omdat het maandagochtend is. Code-generatoren kennen geen speciale maandagochtend-modus en zullen dus code opleveren die consistent van dezelfde kwaliteit is. Wanneer er onder architectuur gewerkt dient te worden is dit van nog groter belang. Controle hierop is lastig, zo niet onmogelijk. Code-generatie uit een DSL kan opgezet worden om aan de gewenste architectuur te voldoen en alle applicaties die hiermee gebouwd worden zullen dan ook netjes aan de architectuur voldoen.

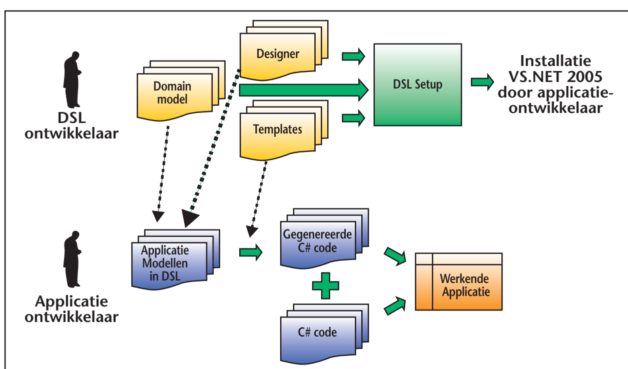
Dit is een aspect dat misschien wel meer appelleert aan klanten / gebruikers van de software die we bouwen. We kunnen meer tijd besteden aan onze klanten, en door een snelle iteratieve werkwijze. Zeker als we in plaats van een technisch domein een business domein met onze DSL kunnen modelleren wordt de aansluiting van IT bij de business beter.

DSL versus UML en MDA De duidelijke overeenkomst tussen DSL's en UML is dat beide gebruik maken van een visuele taal om modellen te maken. Deze modellen bevinden zich op een hoger abstractie-

niveau dan de source-code. Zowel vanuit UML als vanuit DSL's kan code worden gegenereerd. De verschillen tussen UML en DSL's zijn opvallend. In UML probeert men zoveel mogelijk één groot samenhangend model te maken van een compleet systeem. Dit leidt vaak tot complexe modellen. Verder is de UML-taal zelf, mede vanwege dit uitgangspunt erg complex geworden. Binnen de DSL Tools is het uitgangspunt dat een DSL klein is, op zichzelf staat en slechts één bepaald onderdeel van een applicatie ondersteunt. Voor een complete applicatie worden dan ook vaak meerdere DSL's gebruikt. Een ander verschil is dat UML gebaseerd is op internationale standaarden, de DSL Tools van Microsoft niet. Zij werken alleen met Visual Studio.

In de UML-wereld is het startpunt een hoog niveaumodel zoveel, van waaruit men zoveel mogelijk naar de code probeert te gaan. Bij de DSL Tools is het precies tegenovergesteld. Men probeert vanuit de codemodellen te abstraheren, die direct op de code afgebeeld kunnen worden. Het voordeel van de DSL-aanpak is dat code-generatie eenvoudiger wordt, het nadeel is dat het abstractieniveau soms lager is. Hoewel MDA en UML vaak in één adem genoemd worden is het zeker mogelijk om MDA zonder hulp van UML te doen. Binnen MDA kunnen we met behulp van de MOF (Meta Object Facility) standaard onze eigen modelleertalen definiëren. Daarbij zijn we vrij om deze al dan niet domeinspecifiek te maken. UML zelf is op dezelfde wijze via de MOF gedefinieerd. De DSL-aanpak kan derhalve ook in de MDA-wereld gebruikt worden. Aangezien je zelf het domein van jouw DSL kunt bepalen kunnen DSL's ook generiek zijn. Hoewel de gekozen weg verschilt probeert Microsoft met de DSL Tools hetzelfde doel te bereiken als MDA. De DSL Tools kunnen dan ook gezien worden als de Microsoft-manier om MDA te verwezenlijken.

Ontwikkeling en gebruik van een DSL Een DSL wordt in vier stappen ontwikkeld. Vervolgens wordt hij bij een willekeurig aantal ontwikkelaars geïnstalleerd en gebruikt. Dit proces is weergegeven in Afbeelding 1.



Afbeelding 1. Een DSL wordt in vier stappen ontwikkeld.

De DSL-ontwikkelaar voert de volgende stappen uit:

- In stap één definieert hij het domein model. In deze stap beschrijft hij de elementen van de DSL. Hij definieert wat hij met zijn taal kan beschrijven.
- In stap twee definieert hij de visuele designer. In deze stap beschrijft hij de grafische vorm van zijn taal. Dit geeft aan hoe hij de taal gebruikt in de model-editor.
- Vervolgens ontwikkelt hij de templates. Een template vertaalt de DSL naar een concreet artefact (bijvoorbeeld C# of XML). Hiermee geeft hij zijn taal een concrete betekenis.
- Tenslotte zet hij de set-up op. Dit stelt anderen in staat zijn DSL te gebruiken.

De DSL-gebruiker installeert de DSL met behulp van de set-up en hoeft slechts het volgende te doen:

- De ontwikkelaar modelleert het applicatiemodel met behulp van de DSL-editor.
- In de tweede stap genereert hij de code met behulp van de templates.
- Vervolgens schrijft de ontwikkelaar additionele code.
- In stap vier wordt de gegenereerde en handgeschreven code gecombineerd met het beschikbare framework en kan de applicatie uitgevoerd worden.

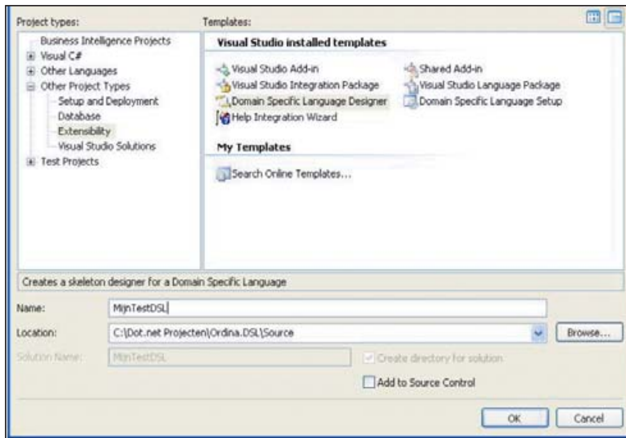
Het concept van DSL is inmiddels duidelijk. In het restant van het artikel nemen we u mee in de stappen van het ontwikkelen van een DSL.

Het ontwikkelen van de DSL Voordat we een DSL kunnen ontwikkelen moeten we eerst de Visual Studio SDK geïnstalleerd hebben. De meest recente versie is te downloaden van de Microsoft-site. Deze installeert twee project-templates voor het ontwikkelen van DSL's:

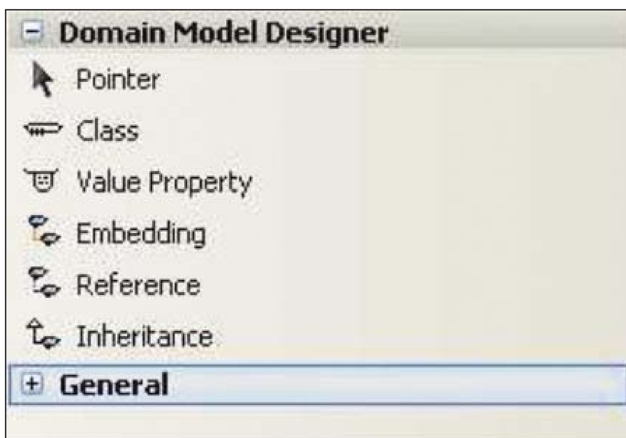
- Domain Specific Language Designer voor het ontwikkelen van onze DSL.
- Domain Specific Language Setup voor het ontwikkelen van de set-up voor onze DSL.

Door het uitvoeren van de 'Domain Specific Language Designer'-wizard creëren we een solution met twee projecten; één met het domeinmodel, één met de designer.

Stap 1 – Domeinmodel In het domeinmodel-project bevindt zich een bestand met de naam 'DomainModel.dsldm'. Dit is een XML-geformatteerd bestand waarin we de taalelementen beschrijven van onze taal. Gelukkig is er een grafische editor beschikbaar. Deze editor voelt heel natuurlijk aan voor Visual Studio-gebruikers. Allereerst is er de toolbox met de meta-taalelementen. Deze kunnen worden vergeleken met de UML-elementen uit Tabel 1. Het aantal meta-



AFBEELDING 2. De Visual Studio SDK installeert twee project-templates voor het ontwikkelen van DSL's.



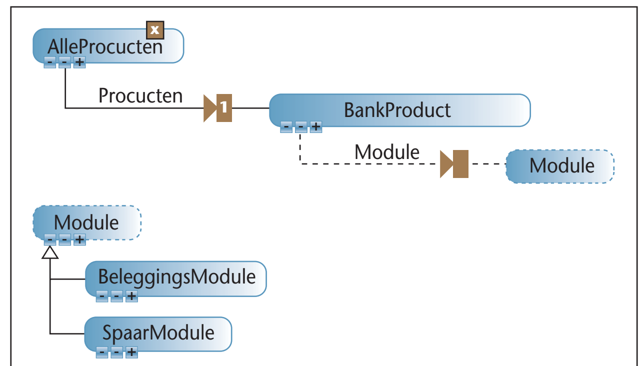
AFBEELDING 3. De grafische editor toont de toolbox met de metataalelementen.

taalelementen is beperkt tot vijf. Met deze elementen kunnen we onze taal beschrijven. Afbeelding 4 toont een taal die ontwikkeld is voor bancaire producten.

Klasses worden aangegeven met afgeronde rechthoeken. Een 'Reference'-relatie, herkenbaar aan de stippellijn, is gebruikt tussen BankProduct en Module. De levenscyclus van een Module staat los van die van een product. De 'Embedding'-relatie, de doorgetrokken lijn, is gebruikt tussen AlleProducten en BankProduct. Het sterretje (*) geeft aan dat AlleProducten meerdere BankProducten kan bevatten, dev 1 geeft aan dat een

DSL Metataalelement	UML equivalent
Class	Class
Value Property	Class or aggregation property
Embedding	Compose association
Reference	Association
Inheritance	Generalization

TABEL 1. Overzicht van UML-elementen.



AFBEELDING 4. Met vijf meta-taalelementen wordt een taal beschreven voor bancaire producten

BankProduct bij precies een Alleproducten hoort. 'Inheritance' is gebruikt tussen BeleggingsModule en SpaarModule en Module.

Evenals in de meest populaire ontwikkeltalen (C# en Java) kan een klasse van slechts één andere klasse worden afgeleid. Omdat het aantal meta-taalelementen beperkt is kan een complexe DSL's soms wat lastig gespecificeerd worden. Aan zowel klasses als relaties kunnen we 'Value Property's' toevoegen. Deze zijn voor de DSL-gebruiker zichtbaar als property's in de 'Properties View' als het object is geselecteerd. Het opzetten van het domain model is vooral drag-and-drop werk. Voor eenvoudige talen volstaan de vijf meta-taalelementen. Voor complexere talen is wat meer kunst en vliegwerk vereist om het gewenste doel te bereiken.

STAP 2 – DESIGNER In het designer-project definiëren we de grafische aspecten van onze nieuwe taal. Ook dit is een XML-bestand, namelijk 'Designer.dsldd'. Helaas wordt in dit geval (nog) geen grafische editor meegeleverd om het schrijven te vereenvoudigen. Deze is wel door Microsoft beloofd in de volgende versie van de DSL Tools. Voorlopig moeten we het doen met de standaard XML-editor. De designer bevat drie delen; de toolbox, de shapes en de mapping. In het toolbox-deel beschrijf je wat er in de toolbox wordt getoond. In onderstaand voorbeeld wordt een toolbox gedefinieerd met één tool:

```
<toolbox>
  <items>
     <shapeTool iconId="Product" captionId=
     "ProductCaption" order="0">
     <shape>Mijn1eDSL.Designer/Shapes/
ProductShape</shape>
     </shapetool>
  </items>
</toolbox>
```

Hier wordt een tool gedefinieerd waarmee een 'ProductShape' gemaakt wordt. Een DSL zal normaliter

uit meerdere tools bestaan, maar ten behoeve van de ruimte is het voorbeeld beperkt tot één. Na het definiëren van de toolbox kunnen we de vormen van de zichtbare taalelementen beschrijven.

```
<shapes>
  <imageShape imageId="Product"
name="ProductShape">
     <decorators>
        <shapeText name="ProductName"
        position="OuterMiddleRight"/>
     </decorators>
  </imageShape>
</shapes>
```

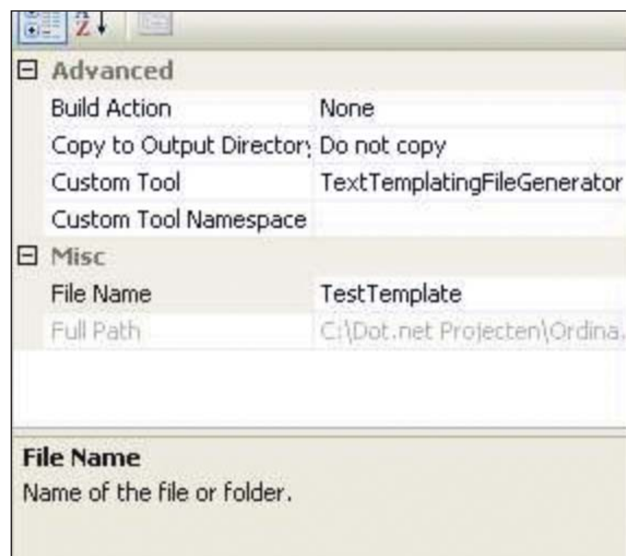
In dit stukje XML wordt de 'ProductShape' gedefinieerd. Dit is dezelfde shape waarnaar in de toolbox gerefereerd wordt. Ook hier beperken we ons tot één eenvoudige vorm. De SDK komt met een aantal standaardvormen. Naast de hier gebruikte ImageShape, welke verwijst naar een plaatje, zijn ook geometrische vormen en compartment-vormen beschikbaar (zoals een klasse in een UML-klassediagram). Tenslotte moet de vorm nog worden gerelateerd aan een taalelement uit het domeinmodel.

```
<shapeMap>
  <class>Mijn1eDSL.DomainModel/
  BankProduct</class>                         (1)
  <melCollectionExpression>
     <roleExpression>
        <role>Mijn1eDSL.DomainModel/
        AlleProducten/Producten</role>         (5)
     </roleExpression>
  </melCollectionExpression>
  <shape>Mijn1eDSL.Designer/Shapes/
ProductShape</shape>                          (2)
  <textMaps>
     <shapeTextMap>
        <textDecorator>
        Mijn1eDSL.Designer/Shapes/
ProductShape/Decorators/ProductName           (3)
        </textDecorator>
        <valueExpression>
           <valuePropertyExpression>
              <valueProperty>
              Mijn1eDSL.DomainModel/
              BankProduct/Name                (4)
              </valueProperty>
           </valuePropertyExpression>
        </valueExpression>
     </shapeTextMap>
  </textMaps>
</shapeMap>
```

In het eerste deel geven we aan dat de domeinklasse 'BankProduct' (1) gekoppeld wordt aan de 'ProductShape' (2). Verder wordt de property 'Name' van BankProduct (4) gekoppeld aan de tekst die bij de 'ProductShape' (3) staat. Tenslotte geven we aan dat het nieuwe 'BankProduct' toegevoegd moet worden aan de verzameling 'Producten' van het object 'AlleProducten' (5).

Met de designer XML kan relatief eenvoudig een grafische editor voor onze taal worden gedefinieerd. Voor meer finesse moet worden teruggegrepen naar C# en de Visual Studio SDK. Hoewel we na enige tijd wel gewend zijn geraakt aan nogal cryptische XML wachten we met smart, zoals je wel zult begrijpen, op de grafische designer voor deze XML-file.

Stap 3 – Templates Als derde stap ontwikkelen we templates om de DSL-taal te transformeren naar werkende code. Met deze templates kunnen allerlei soorten bestanden kunnen worden gecreëerd zolang het tekstbestanden betreft (C#, HTML of XML). C# bestanden kunnen partial classes bevatten. De code van een partial class kan verspreidt zijn over meerdere bestanden. Extra functionaliteit kan in een separaat bestand worden toegevoegd. Dit maakt partial classes ideaal voor code-generatie omdat handmatige uitbreidingen niet in het gegenereerde bestand staan, maar in een apart bestand. Hergeneratie met behoud van de handmatige toevoegingen is op deze manier eenvoudig. Om een template te kunnen ontwikkelen en testen is een werkende DSL nodig. Een template is een tekstbestand. Door de 'TextTemplatingFileGenerator' als Custom Tool binnen VisualStudio te registreren wordt de template uitgevoerd. De template-file begint met een drietal directives, welke voorafgaan aan de template zelf:



Afbeelding 5. Door de 'TextTemplatingFileGenerator' als Custom Tool binnen VisualStudio te registreren wordt de template uitgevoerd.

```
<#@ template inherits=
  "Microsoft.VisualStudio.TextTemplating.
  VSHost.ModelingTextTransformation"
#>
<#@ output extension=".htm" #>
<#@ alleProducten
  processor="Mijn1eDSL_DesignerDirective
  Processor"
  requires="fileName='Sample.my1stdsl'"
  provides="AlleProducten=AlleProducten"
#>
```

De template directive geeft aan welke template host te gebruiken. De template host geeft aan in welke omgeving de template engine zijn werk doet. De standaard-host zal in de meeste gevallen voldoen, maar voor speciale zaken kun je een eigen host definiëren. De output-directive geeft aan wat de extensie van het resultaatbestand is. Het moge duidelijk zijn dat één template tot één resultaatbestand leidt. In het voorbeeld wordt een HTML-bestand gegenereerd. De derde directive laadt een specifiek model, zodat we in de template gebruik kunnen maken van de informatie uit het model. We kunnen nu verder op een ASPX-manier de template definiëren. ASPX-constructies, zoals <%= block %> en <% block %> zijn ook in iets andere vorm beschikbaar; <# block #> en <#= block #>. De afwijking maakt het eenvoudig ook ASPX-pagina's te creëren. De blokken bevatten C# code (of VB) en kunnen we gebruiken om een dynamisch eindresultaat te creëren, bijvoorbeeld op basis van de modelinformatie. Onderstaand voorbeeld resulteert in een lijst van productnamen in het gerefereerde model. Let erop dat AlleProducten en Product-elementen uit het domeinmodel zijn.

```
<# foreach(Product p in AlleProducten) { #>
  <#= p.Name #>
<# } #>
```

Stap 4 – Ontwikkelen van de set-up Als de DSL geheel af is, dienen we een set-up te maken om de DSL te distribueren naar de ontwikkelaars die de DSL gaan gebruiken. Een set-up maken voor een DSL is heel eenvoudig. Het is simpelweg een kwestie van de project template 'DSL set-up' uitvoeren. Microsoft heeft de DSL set-up gebaseerd op haar WiX standaard (Windows Installer XML). Deze standaard maakt het mogelijk om via XML-bestanden de set-up applicatie te beschrijven. Een DSL set-up project kunnen we uitbreiden met onze eigen XML-bestanden, om bijvoorbeeld extra registry-settings te zetten tijdens het uitvoeren van de set-up. Nu kan de DSL rechtstreeks geïnstalleerd worden door alle ontwikkelaars die Visual Studio 2005 hebben. Deze ontwikkelaars zullen verder alleen de DSL gebruiken, dat wil zeggen ze maken modellen en genereren daar-

SYSTEEMONTWIKKELING
EN -ONDERHOUD
voor uw comfort

U onderneemt met een rotsvast vertrouwen. U kijkt vooruit, staat af en toe stil en kijkt goed om u heen. Anticiperen op veranderingen maakt slagvaardiger in de strijd met uw concurrenten. Een goede automatisering en een up-to-date informatiseringssysteem zorgen voor de basis. U ontwikkelt softwareapplicaties die aansluiten op de kennis, kunde en wensen van uw organisatie.
Kies voor Solipsis. Onze ervaren specialisten ontwerpen, bouwen, testen en implementeren informatiseringssystemen op gestructureerde wijze. Jarenlange ervaring maakt ICT onze tweede natuur en ons uw betrouwbare en stabiele partner. *Wel zo comfortabel.*

SOLIPSIS
COMFORT CLASS IN ICT

www.solipsis.nl
telefoon: (0418) 57 61 00 info@solipsis.nl

uit code (en andere bestanden) door het uitvoeren van de templates.

Ervaringen De DSL Toolkit is verrassend robuust en daardoor goed werkbaar. Wel doet vooral het ontbreken van een grafische editor voor de designer zich gelden. Foutjes in de XML zijn soms lastig op te sporen, wat zeker voor de beginnende DSL-ontwikkelaar tot frustraties leidt. Gelukkig is dit een tijdelijke situatie. Ondanks het beperkt aantal meta-taalelementen is de DSL-toolkit in de meeste gevallen expressief genoeg. Dit geldt voor zowel het domeinmodel als de grafische designer. DSL's en de daarmee gemaakte modellen zijn strikt stand-alone, dat wil zeggen er is geen enkele voorziening om relaties tussen DSL's of tussen modellen te maken. Zelfs wanneer er meerdere modellen volgens één DSL gemaakt worden, dan zijn de modellen strikt gescheiden. Referenties tussen modellen zijn dan ook niet mogelijk. Het is bijvoorbeeld niet mogelijk om delen uit een model te knippen en te plakken in een ander model, zelfs niet als de modellen van hetzelfde type zijn. Voor mensen die UML kennen is dat even wennen. Daar waar in UML een diagram niets meer is dan een view op de informatie uit één groot samenhangend model, is een model in DSL analoog aan zowel de view als de informatie.

Als het delen van informatie tussen modellen voor onze oplossing belangrijk is, moeten we daar zelf voorzieningen voor treffen, zowel in het model als in de designer. Omdat de DSL-toolkit gebaseerd is op de Visual Studio SDK is veel mogelijk, maar dit kent wel een grote leercurve. In onze ervaring is dit al snel nodig en we hebben hier dan ook een eigen oplossing voor ontwikkeld. De manier om templates te ontwikkelen is zeer krachtig. De template-taal is krachtig genoeg om een model te vertalen naar een artefact. Een template transformeert naar één resultaatbestand. Geen probleem voor een model dat naar een vast aantal artefacten wordt vertaald. Maar als het aantal artefacten afhankelijk is van de modelinhoud schiet de template-taal tekort.

Ondersteuning De gebruiker van een DSL moet voor het genereren van de code een template per model maken. De template-file verwijst via een harde padnaam naar de modelfile. Wanneer we dus vijf modellen volgens dezelfde DSL maken krijgen we ook vijf kopieën van dezelfde template. Deze wijze van omgaan met templates vervuilt het project met veel te veel bijna identieke template-bestanden en is lastig voor de gebruiker van de DSL. Verder wordt nog geen ondersteuning geboden om het template-resultaat te combineren met een eerdere getransformeerde versie (regeneratie met behoud van handmatige wijzigingen). Dat is voor C# bestanden met gebruik van partial classes niet zo'n probleem, maar voor XML-bestanden of daarvan afgeleide standaards wel. Voor de meeste beperkingen is het mogelijk om een eigen oplossing te ontwikkelen en deze netjes in te passen in de DSL Tools. Voor een aantal van de in dit artikel genoemde beperkingen hebben we met behulp van de Visual Studio SDK al een eigen oplossing gemaakt.

Conclusie Microsoft biedt met haar DSL-toolkit een hulpmiddel waarmee snel een eigen, krachtige visuele modelleertaal ontwikkeld kan worden. Het resultaat is een model editor en bijbehorende codegenerator die volledig met Visual Studio geïntegreerd zijn. Hiermee is de bruikbaarheid voor de ontwikkelaar goed gewaarborgd. Het ontwikkelen van zo'n eigen tool is relatief eenvoudig. Eigen tools kunnen bijdragen aan het verbeteren van de productiviteit van ontwikkelaars. Er is alle reden om alvast met deze nieuwe technologie aan de slag te gaan, ondanks het feit dat de toolkit nog zijn beperkingen heeft en zich nog in de bèta-fase bevindt. Wij zijn in ieder geval overtuigd van de mogelijkheden van de DSL Tools.

John Dekker en Jos Warmer zijn beiden werkzaam bij Ordina.