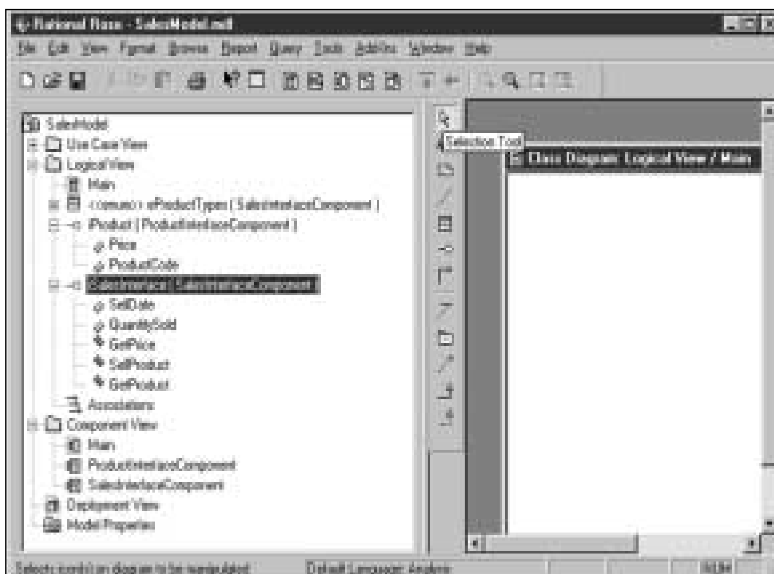
In recente toespraken heeft Ivar Jacobson (UML-grondlegger en vice president software development bij Rational) zijn toekomstvisie geëtaleerd. Hij verwacht onder andere een ontwikkeling richting het layman programming: als de ontwikkeltools maar voldoende intelligent en transparant worden, heb je nauwelijks nog bijzondere technische vaardigheden nodig om een nieuw systeem te ontwikkelen. Uiteraard voorziet Jacobson ook toenemend gebruik van UML als programmeertaal: ontwerpen in UML en vervolgens de code reverse engineeren. Welke tool zou Jacobson in gedachten hebben? Hoe kan het ook anders: Rational Rose.

achtergrond

De programmerende leek

IDL-generatie vanuit Rational Rose

We zijn nog lang niet zover dat je je business wensen kunt intypen en dat een tool hieruit je nieuwe systeem genereert, maar het moet gezegd: vanuit Rational Rose kun je vanuit je design model code genereren in talen als Java, Visual Basic, Ada en IDL. Het is zelfs mogelijk om vanuit die talen je model weer te updaten. Dit artikel beschrijft het genereren van IDL-bestanden vanuit Rational Rose (versie 2000). IDL is een technologie-onafhankelijke syntax waarmee interfaces beschreven kunnen worden (IDL = Interface Definition Language). Door een IDL-bestand vervolgens te compileren tot een type library, kan de interface beschikbaar worden gesteld aan bijvoorbeeld een Visual Basic-applicatie.



FIGUUR 1. **Een 'component' betekent in Rational Rose een fysieke component**

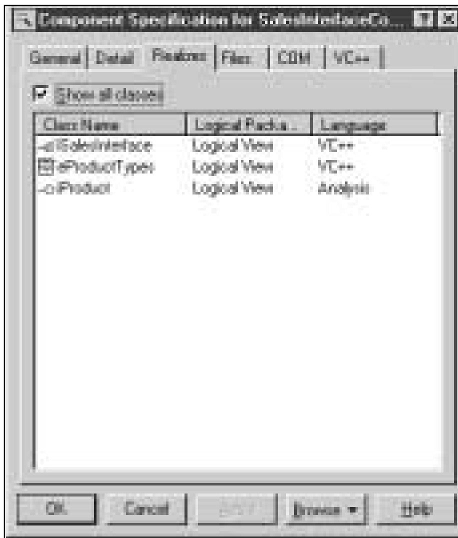
Hoe makkelijk is dat genereren van IDL nou? Welke haken en ogen zitten hier aan vast?

GENEREREN VAN IDL-BESTANDEN We hebben in Rose twee soorten elementen nodig als we code willen genereren. Aan de ene kant hebben we een design element nodig dat we om willen zetten in fysieke code; dit bevindt zich normaalgesproken in de Logical View van een Rose-model. Aan de andere kant hebben we een model-element dat de fysieke component vertegenwoordigt die we gaan genereren of bijwerken. Dat laatste element is dan ook gelinkt aan een fysieke component (bijvoorbeeld een class, een project, een IDL-file) en we vinden het terug in de Component View van het model. Een "component" betekent in Rose dan ook een fysieke component.

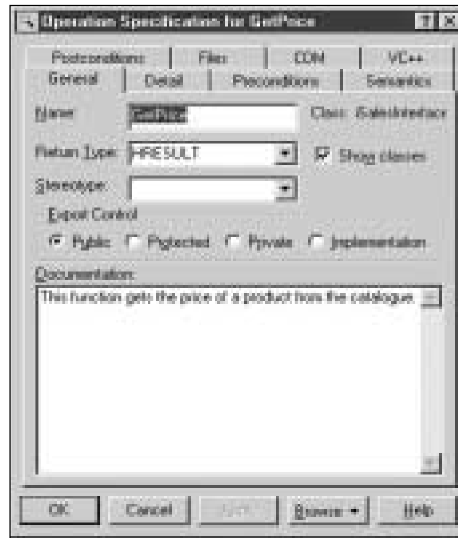
Het element in de Component View draagt drie belangrijke stukjes informatie in zich:

- De link met de fysieke component die het element weergeeft
- Een link met de design elementen uit het model die in de fysieke component gegenereerd gaan worden (zie figuur 2)
- De taal waarin de code gegenereerd gaat worden.

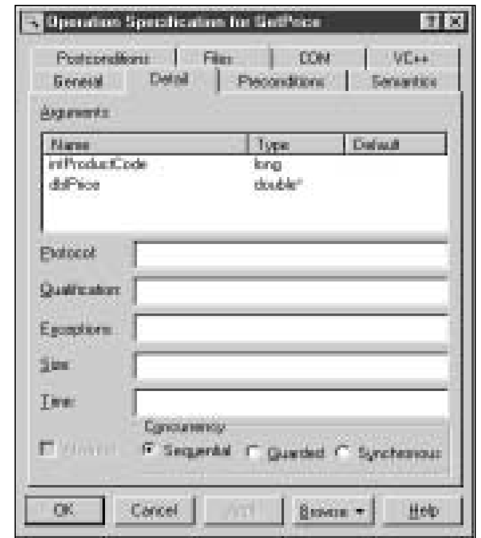
Rose bepaalt de lijst met mogelijke stereotypen overigens aan de hand van de ingevoerde taal! Als we dus een class willen genereren in IDL, moeten we eerst de taalwaarde op "VC++" zetten en op Apply klikken. Daarna krijgen we van Rose pas de mogelijkheid om "MIDL" als stereotype te kiezen. Om VC++ te kunnen kiezen moet de VC++ add-in zijn geselecteerd in het



FIGUUR 2. Een link met de design elementen uit het model



FIGUUR 3. Het opgegeven Return Type moet altijd "HRESULT" zijn.



FIGUUR 4. Op de tab Detail zijn de parameters te zien.

Add-Ins menu. Om die add-inn te kunnen selecteren moet echter wel Visual C++ op de computer geïnstalleerd zijn. Het zijn de design elementen in de Logical View die de eigenlijke functionaliteit beschrijven: methoden, attributen et cetera. Idealiter zouden de elementen in de Logical View gelijk moeten blijven wat de taal ook is waarin we ze genereren. Helaas zullen we nog een aantal instellingen van het design element moeten wijzigen voordat we de fysieke component gaan genereren. Doen we dat niet, dan heeft Rose onvoldoende informatie om te bepalen hoe de component precies gegenereerd moet worden. De tool zal natuurlijk wel een poging wagen, maar het resultaat zal syntactisch onjuist zijn, waardoor we naderhand de code nog handmatig zouden moeten aanpassen. Om dit te voorkomen moeten we extra aandacht besteden aan de Specification van het design element. Het belangrijkste bij genereren van een IDL-bestand is dat we een gedetailleerde beschrijving geven van de parameters die in de ontworpen methodes gebruikt worden.

PARAMETER TYPEN De datatypes waar we in Rational Rose standaard uit mogen kiezen (op het Specification scherm), zijn afhankelijk van de taalinstelling van de bijbehorende component in de Component View. Default staat die instelling op "Analysis", maar de bijbehorende datatypes zijn niet bruikbaar in IDL. En worst of all: er is geen taalinstelling die de gewenste lijst met datatypes oplevert. We zijn dus toch gedwongen om ze zelf in te tikken. Overigens zal de bruikbaarheid van de datatypes die we invoeren ook nog afhangen van de programmeertaal waarin we het interface-bestand gaan gebruiken. Als de type library (het gecompileerde IDL bestand) een datatype opgeeft dat de betreffende programmeertaal niet kent, zal de implementatie van dat datatype een ongeldig resultaat opleveren.

Voor het resultaat van een functie geldt nog een aparte instelling. Als we er van uitgaan dat we een COM-interface genereren in IDL, moet het opgegeven Return Type altijd "HRESULT" zijn, zowel voor functies als voor eigenschappen. Rose voert geen controle meer uit op de ingevoerde waarde voor "Return Type", dus wees voorzichtig: alles wat je daar intikt zal letterlijk zo gegenereerd worden. Zoals eerder aangegeven hangt het van de uiteindelijke programmeertaal van het systeem af welke datatypes we in de IDL kunnen gebruiken. Stel dat we een type library aan het maken zijn voor Visual Basic, dan komen de volgende datatypes in aanmerking:

IDL	Wordt in VB gegenereerd als
VARIANT_BOOL	Boolean
Unsigned char	Byte
CURRENCY	Currency
SampleEnum	CustEnum
AllMethods*	CustObj
DATE	Date
Double	Double
Short	Integer
Long	Long
Idispatch*	Object
Float	Single
BSTR	String
VARIANT	Variant

TABEL 1

INPUT EN OUTPUT Afgezien van het datatype, moeten we ook nog specificeren of de parameter een pointer is naar een daadwerkelijke waarde (ByRef door-

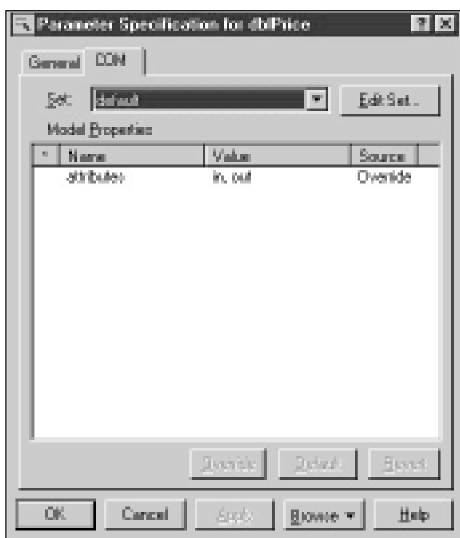
by reference/ by value	input/output	datatype (regulier[1] of object)	waarde van het input/output attribuut in Rose	gevolgen voor het datatype
by value	input	regulier	in[2]	-
by reference	input & output	regulier	in, out[2]	1 asterisk toevoegen
by reference	output	regulier	out, retval[3]	1 asterisk toevoegen
by reference	input & output	object	in, out[2]	2 asterisken toevoegen
by reference	output	object	out, retval[3]	2 asterisken toevoegen

[1] met 'reguliere' datatypes bedoelen we hier alles behalve een object (string, long, etc.)
[2] optioneel
[3] resultaat van een functie

TABEL 2

gegeven in VB) of de waarde zelf (ByVal doorgegeven in VB). Als je in Rose een parameter dubbelklikt in het Detail scherm, wordt het specificatiescherm voor die ene parameter getoond. Op dit scherm moet ook het datatype van de parameter worden ingevuld. Er is een belangrijke relatie tussen het datatype en de waarde voor by-reference/by-value: als de parameter by reference wordt doorgegeven, moeten we een asterisk (*) aan het datatype vastplakken. Als de parameter tegelijkertijd ook nog een object(-referentie) is, moeten we twee asterisken (**) toevoegen aan het datatype. Als regel kunnen we dus stellen dat je een asterisk moet toevoegen als de parameter ByRef is, en nog eentje als die parameter ook een object is. Als we nu op de COM-tab van de parameter-specificatie klikken, kunnen we de input- of output-specificatie opgeven (zie figuur 5).

> Belangrijk bij het genereren van een IDL-bestand is een gedetailleerde beschrijving van de parameters



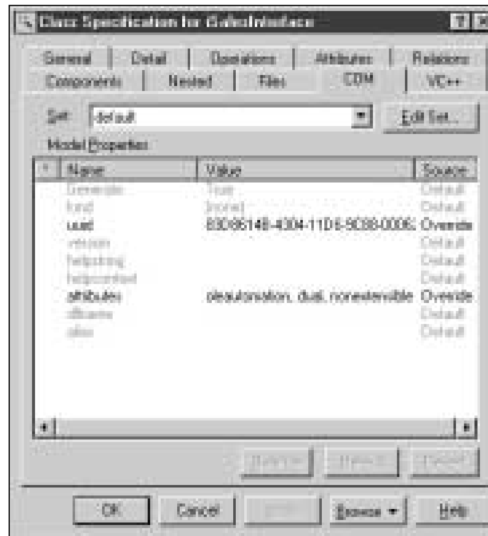
FIGUUR 5. Hier kunnen de input- of output specificaties worden opgegeven

Opvallend genoeg zal de waarde die we invoeren voor het input/output-attribuut in de meeste gevallen door Rose genegeerd worden. Een of twee asterisken achter het datatype impliceert bijvoorbeeld al dat het gaat om een parameter die zowel input als output is, dus hoeven we niet expliciet het attribuut in, out op te geven. Verder is het zo dat als we geen asterisk en geen expliciet attribuut opgeven, Rose de aanname zal doen dat het hier gaat om een ByVal input-parameter. Ook in dit geval hoeven we dus niet expliciet een attribuut in op te geven.

De enige keer dat we een attribuut moeten opgeven, is als we willen aangeven dat het gaat om het resultaat van een functie. In dit laatste geval moet er een input/output attribuut worden toegevoegd met de waarde out, retval. Doen we dit niet, dan zal de parameter worden gegenereerd als een ByRef input parameter. Als we nu alle mogelijke combinaties naast elkaar zetten krijgen we het overzicht zoals in tabel 2.

PROPERTIES Properties voeg je in Rational Rose toe aan een interface alsof het gewone operaties zijn. Als we van een property IDL willen brouwen, splitsen we hem in een aparte operatie voor de Put- en de Get-actie. Er zijn hierbij twee belangrijke dingen om rekening mee te houden:

- Op het tabblad General van de Specification voor deze operaties moeten we een stereotype invullen: propget voor de Get en propput voor de Put.
- Op het COM-tabblad van de specificatie van iedere operatie is het mogelijk om een identificerend nummer op te geven dat ook meegenomen zal worden bij de IDL-generatie. Dit is echter niet noodzakelijk. Als



Figuur 6. Tenslotte moeten nog drie attributen aan de specification worden toegevoegd

je besluit om wel een ID op te geven, onthoud dan dat als een Put en Get methode (in IDL) op een-en-dezelfde property slaan, deze twee operaties verplicht dezelfde ID moeten hebben. Zo niet, dan zal Rose deze twee niet herkennen als een paar.

Enumerated types Je kunt in een Rose-model enumerated types aan je interface toevoegen, die ook in de IDL gegenereerd zullen worden. Je voegt een enumerated type toe in Rose als een class met het stereotype enum. Een enumerated type bevat zoals de term al zegt een opsomming van mogelijke waarden. In Rose wordt ieder van die waarden gezien als een apart attribuut. Op de Specification-tab van ieder attribuut kunnen we die waarde identificeren door een specifiek nummer toe te voegen in het Initial value veld. Controleer goed of er niet meerdere waarden zijn met hetzelfde nummer, aangezien dit tot grote verwarring kan leiden.

COMpleteren van de instellingen Omdat de type libraries die we aan het maken zijn COM-compliant zijn, gelden er enkele restricties waar we ons aan moeten houden:

- De ontworpen interface moet de IDispatch interface implementeren. Als we in Rose een nieuw model creëren met de Visual Basic template, zal de IDispatch interface (samen met een hoop andere interfaces) automatisch aan het model worden toegevoegd. Een andere manier om IDispatch in het model te krijgen is door het reverse engineeren van een VB-class.
- Om IDispatch vervolgens te implementeren in de onze zelf ontworpen interface, kun je een Overview Class Diagram in Rose maken. Voeg hier de IDispatch interface en onze eigen interface aan toe, en teken hier een Realize-relatie tussen. Als het goed is kunnen we nu deze nieuwe relatie terugzien op de tab Relations van de Specification van onze interface.
- Tenslotte moeten we nog drie attributen toevoegen op de COM-tab van de Specification van onze interface: oleautomation, dual, nonextensible (zie figuur 6). Controleer wel eerst of het design element al gelinkt is aan een component in de Component View met als taalinstelling VC++. Zonder die instelling zal er namelijk bij het design-element helemaal geen COM-tab te zien zijn!

Eindelijk is onze interface volledig voorbereid: we kunnen er nu code mee gaan genereren.

Code buiten het design model Soms zal het gewenst zijn om code te genereren die we niet op een nette manier in het design model kunnen verwerken. Om dit te ondervangen, bevat Rational Rose een Model Assistant, waarmee we specifieke code aan een bepaald model element kunnen toevoegen. Die code zal dan in het gegenereerde IDL-bestand worden gekopieerd vanaf een positie die je zelf kunt specificeren. Om de Model Assistant te openen klik je met de rechtermuisknop op het element waaraan je code wil toevoegen (bijvoorbeeld een class of een interface) en kies je uit het menu de optie Model Assistant... Onthoud dat deze optie pas te selecteren is als het betreffende design element is toegewezen aan een VC++ component in het Implementation model.

Source Control tool Als je een nieuw systeem aan het ontwikkelen bent, zul je vaak gebruik maken van een source control applicatie zoals Rational's ClearCase of Microsoft's SourceSafe. Vanzelfsprekend zullen er door het genereren van nieuwe code bestanden veranderd worden, dus deze bestanden moeten eerst "gereserveerd" worden in je Source Control tool (het bijbehorende commando heet meestal check out). Als je de betreffende VC++ COM-component voor het eerst genereert, zullen er ook een aantal bestanden in je Rose model aangepast worden tijdens de generatie.

Ten eerste zullen er Globally Unique Identifiers (GUID's) gegenereerd worden voor de interfaces die we implementeren in de fysieke component, en ook voor die fysieke component zelf. Rose zal proberen die GUID's toe te voegen aan de juiste elementen in het

> Rational's jongste telg, XDE probeert de spraakverwarring al wat te beperken

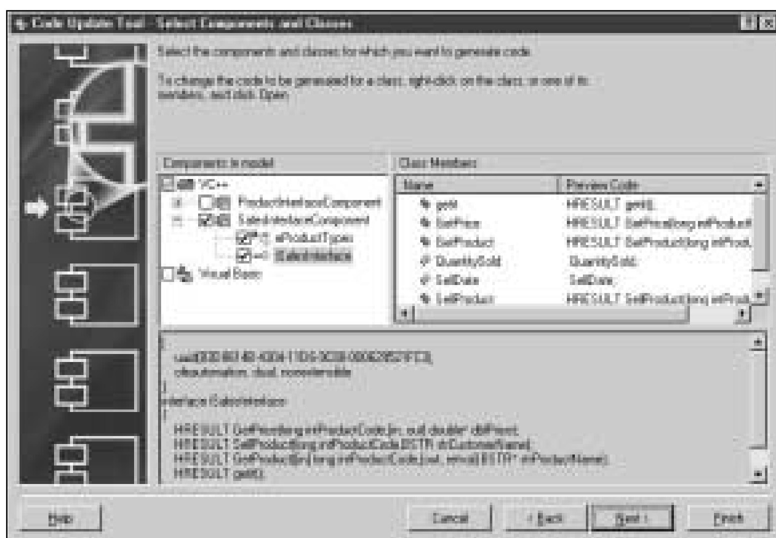
model, dus moeten de packages waar deze elementen in zitten ook write-enabled zijn, anders zal de generatie fouten opleveren. Behalve de GUID's genereert Rose ook Model ID's. Als je code van een component opnieuw genereert vanuit een model, of als je het model bijwerkt vanuit de code, zal Rose de Model ID's gebruiken om te bepalen welk stukje code bij welk model element hoort.

Als je alleen een implementation component aan het updaten bent die je eerder al gecreëerd hebt, hoef je alleen de code-bestanden die bijgewerkt gaan worden write-enabled te maken:

- het workspace bestand
- het project bestand
- het IDL-bestand
- het TLB (type library) bestand

GENEREREN OF UPDATEN Om te beginnen met de codegeneratie klikken we met de rechtermuisknop op het te genereren element uit het design model (Logical View). Op die manier zul je default alleen dat specifieke design element bijwerken in de code. Dat kan nuttig zijn als je fysieke component diverse design elementen tegelijk implementeert.

We spreken nog niet allemaal Esperanto, dus we gaan ook niet in dezelfde taal code genereren



FIGUUR 7. Onderin het beeldscherm is al een preview van de code te zien

FIGUUR 8. We zien in Visual C++ de code die is gecreëerd of bijgewerkt

Het is ook mogelijk om met de rechtermuisknop te klikken op de fysieke component (in de Component View) die we gaan genereren. Default zullen we dan alle design elementen bijwerken die de component implementeert. Ook als een component meerdere design elementen implementeert, en we hebben Update Code... geklikt op de component, krijgen we in het volgende scherm nog de mogelijkheid om alleen bepaalde design elementen bij te werken. Op datzelfde scherm is overigens onderin al een preview te zien van code die gegenereerd gaat worden (zie figuur 7).

Let op: logischerwijs zouden we alle design elementen selecteren als we voor de eerste keer een component gaan genereren. Er zit echter een bug in Rose die dit belet als er meerdere enumerated types in de interface zitten. De codegenerator lijkt dan de kluts kwijt te raken en de code wordt een rommeltje; gegenereerde code van het ene enumerated type begint ergens midden in de code van het vorige, wat natuurlijk grammaticaal onjuiste IDL oplevert. De beste manier om dit te voorkomen is het apart genereren van de enumerated types, een voor een dus.

Als we de te genereren elementen hebben gekozen klikken we op Next. We komen dan op een scherm dat een overzicht geeft van de elementen die nu gegenereerd zullen gaan worden. Nu kan eindelijk de daadwerkelijke codegeneratie gestart worden door op Finish te klikken. Terwijl we het IDL-bestand genereren, zal de MS Visual C++ omgeving opgestart worden. Rose laat

intussen op een apart scherm de voortgang van de generatie zien. Na de code-update wordt er in Rose een overzichtsscherm getoond dat aangeeft welke elementen zijn gegenereerd, en daarnaast welke fouten of waarschuwingen eventueel zijn opgetreden.

De code die we nu hebben is dus het IDL-bestand. Om hier de gewenste type library van te maken, moeten we deze code nog compileren. We klikken hiervoor op Build in het menu met dezelfde naam. Nu zal de type library aangemaakt worden, een binary bestand met de extensie .tlb. In een Visual Basic project kunnen we nu een referentie zetten naar deze type library. In een code unit in VB die deze type library gebruikt worden we nu vanzelf gedwongen om alle functies, property's, etc. te implementeren zoals beschreven in de type library.

Er treedt een bug in Rational Rose op als er meerdere enumerated types in de interface zitten

Programmerende techneut Toch handig: je interfaces zijn nu klaar voor gebruik en ze zijn tegelijkertijd al gedocumenteerd. Kunnen we dus de business consultant binnenkort al op de stoel van de ontwikkelaar laten plaatsnemen? De bovenstaande (beknopte!) handleiding lijkt het tegendeel te bewijzen. Het genereren van code vanuit Rose werkt, maar is verre van doorzichtig. Afgezien van een aanzienlijk aantal stappen die je moet doorlopen binnen Rose, vergt het ook nog een redelijke hoeveelheid hands-on kennis van de taal waarin je code genereert. Om te zorgen dat zelfs een leek nu code kan genereren, zal Rational Rose in ieder geval zo intelligent moeten worden, dat programmeertaalspecifieke kennis overbodig worden.

Het is maar zeer de vraag of dat mogelijk is. Rational's jongste telg, XDE (eXtended Development Environment), probeert de spraakverwarring al wat te beperken door de focus te verleggen naar UML. Ontwerp en code zouden zo dichter tegen elkaar kruipen en er is veel mogelijk zonder kennis van de uiteindelijke taal van de applicatie. Alleen is het pakket dan wel geïntegreerd met Microsoft's .Net dan wel IBM's WebSphere. Systeemvereisten: 400 Megabyte voor XDE zelf en een minimaal vereiste workspace van maar liefst twee Gigabyte (aanbevolen wordt vier Gigabyte). Bovendien moet dan ook nog eerst respectievelijk Visual Studio .Net of IBM WebSphere Studio Application Developer geïnstalleerd worden om het geheel te laten werken. Hoezo taalonafhankelijk? En dan nog: waarom zou UML dan die ene taal worden waarin we alles gaan ontwerpen en daarmee ook deels coderen?

Lang leve de nerds Het uiteindelijke probleem ligt duidelijk niet alleen bij Rose of XDE, maar meer nog bij de programmeertalen zelf. Binnen het oerwoud van bestaande talen zijn duidelijk wat hoofdgroepen aan te wijzen die het best geschikt zijn voor een bepaald soort systeem. Denk aan een groep met daarin Visual Basic en Delphi, zeer geschikt om razendsnel schermapplicaties op te zetten. Denk aan een groep met C++ en Java, veel beter geschikt voor onder andere embedded software. Binnen zulke groepen lijken de talen echter zo veel op elkaar, dat je je afvraagt waarom het nodig is dat deze naast elkaar blijven bestaan. Helaas is dat net alsof je vraagt waarom niet alle mensen over de hele wereld dezelfde taal gaan spreken. Leuk idee, maar onhaalbaar. We gaan niet met z'n allen Esperanto spreken, dus gaan we ook niet met z'n allen in dezelfde taal code genereren. Taal leeft en iedere taal die zichzelf presenteert als universeel en allesomvattend - Microsoft .Net's Intermediate Language, Java, maar dus ook UML – zal worden ingehaald door de werkelijkheid. Taalspecifieke technische kennis blijft dus in ieder geval de komende jaren nog onmisbaar om een systeem op te leveren dat doet wat de klant wil. Lang leve de nerds!

Pascal van Alphen
Van Alphen is software architect bij de Webtechnology practice Warp11 van Cap Gemini Ernst & Young;
E-mail: pascal.van.alphen@cgey.nl