Voordat Erik Meijer in dienst trad bij Microsoft, was hij hoogleraar aan de Universiteit van Utrecht waar hij zich met van alles bezig hield rondom - met name functionele – programmeertalen. Daarnaast was hij bestuurslid van het SERC. Hij is één van de bedenkers van de programmeertaal Haskell en is gespecialiseerd in declaratieve en high level programmeertalen. Behalve Program Manager in the Common Language Runtime Group is hij adjunct Professor in Computer Science and Engineering aan het Oregon Graduate Institute.

Diamanten verborgen in de runtime

Erik Meijer over common language runtime, ideale talen en meer

Toen u naar Microsoft ging, moet .Net al ongeveer voltooid geweest zijn.

De concepten waren inderdaad al klaar, maar ik was toen al twee jaar betrokken bij .Net. Microsoft heeft een aantal commerciële en een aantal academische talen heel vroeg benaderd om te kijken of die hun compilers wilden targetten voor de .Net runtime. Op die manier wilden ze feedback krijgen over het ontwerp. Vanuit de universiteit was ik al heel nauw betrokken bij het runtime team en ik ben eigenlijk op die manier zo langzaam erin gestroomd. Uiteindelijk ben ik overgestapt. Bij Project 7, die groep van academische en commerciële programmeertalen - waren mensen bij .Net betrokken voordat mensen bij Microsoft Nederland ook maar van het bestaan van .Net afwisten.

In feite zou je het kunnen zien als Microsoft's derde poging om Java te overvleugelen?

Daar ben ik het niet helemaal mee eens. De uitgangspunten zijn toch duidelijk anders. Bij Java is het heel duidelijk een gesloten wereldmodel, de Java virtuele machines zijn volledig geoptimaliseerd voor Java. De .Net runtime – als je op dat niveau kijkt – is juist geoptimaliseerd om zoveel mogelijk programmeertalen toe te staan. Je probeert juist jouw executieomgeving geschikt te maken voor zoveel mogelijk talen. Het is ook een heel brede verzameling talen, van ANSI-C, een low level systeemprogrammeertaal, tot talen zoals Mercury en Haskell, declaratieve talen met backtracking en unificatie enzovoorts. Ik vind dat een heel goed initiatief. Als programmeertaalonderzoeker, was Java toch een beetje een grote bedreiging in die zin, dat daarmee het onderzoek naar programmeertalen een beetje doodgemaakt werd. Het probleem van kleine talen is, dat je voor het maken van applicaties bibliotheken, programmeeromgevingen, runtime systemen, en garbage collectors nodig hebt. Daar zit heel veel werk in wat al die talen als het ware allemaal dupliceren. Door dat in één keer te doen, bespaar je heel veel werk - je kunt je concentreren op jouw taal en dan kun je gebruikmaken van die hele infrastructuur die er gewoon ligt. Dat is een enorm groot voordeel.

Theoretisch zou je dan ook heel goed een nieuwe taal kunnen ontwerpen en vervolgens daar gebruik van kunnen maken.

(Enthousiast) Precies! Dat zie je ook wel: er zijn nu mensen die allerlei kleine taaltjes gaan maken op de runtime van .Net. Ook oude talen: een tijdje geleden iemand heeft mij benaderd die GWBasic wilde targetten naar de runtime. Dus ik denk zeker dat het voor nieuwe programmeertalen een grote stimulans kan zijn.

Hoe open is dat systeem, hoeveel medewerking krijgt - laten we zeggen Borland – wanneer ze iets willen ontwikkelen dat ervan gebruik maakt?

Daar zijn twee antwoorden op. Open is natuurlijk een vrij ambigue woord Wij hebben de hele infrastructuur gestandaardiseerd bij ECMA (European Computer Manufacturers Association), en wanneer je bij ECMA standaardiseert is de bedoeling dat het ook een ISO standaard wordt. Ze noemen het wel een fast track naar ISO standaardisatie. Alle voorstellen liggen er, waarbij echt alles is gespecificeerd. Er zitten geen dubbele bodems of wat

dan ook in. Iedereen staat het vrij om met behulp van die specificaties een eigen implementatie te maken. In die zin is het open en als je kijkt naar alle andere standaarden binnen .Net zoals Webservices, dat is natuurlijk allemaal gebaseerd op open standaarden. Ik denk niet dat er op dat punt iets te klagen valt.

Terug naar het ontwikkeltraject, daar moeten een heleboel beslissingen gevallen zijn. Ooit moet er een vrij vaag concept geweest zijn, wat zich gaandeweg ontwikkeld heeft. Was het idee van de Common language Runtime er van het begin af aan, bijvoorbeeld, en zijn er veel compromissen gesloten?

Dat idee van een common language runtime was er vanaf het begin al. De vraag naar die compromissen vind ik wel grappig, want dat raakt het verschil tussen universiteit en bedrijfsleven. In de wetenschap denk je vaak na: wat zijn nu de alternatieven? Je kunt alle mogelijkheden helemaal bekijken en erover nadenken, terwijl het hier precies andersom is: je moet heel snel een beslissing nemen en dan moet je daarmee doorgaan.

Dat kan dan een niet-optimale beslissing zijn?

Ja, je kunt vaak nog niet zeggen of het een optimale beslissing is. Je hebt gewoon niet de tijd om alle mogelijkheden uit te zoeken. Dat vind ik op zich wel heel spannend. Ja, maar ik moet zeggen dat ik niet echt veel beslissingen heb moeten nemen waarvan ik achteraf denk, dat het niet goed was. En er zijn ook veel dingen bij die ik echt ongelooflijk elegant vind, waarvan de wetenschap op zich weer van zou kunnen leren.

Een voorbeeld?

Clemens Szyperski, ook werkzaam bij Microsoft, heeft in Component Software - Beyond Object-Oriented Programming een heleboel problemen bij component based programming beschreven. Eén daarvan is het probleem het versiebeheer onafhankelijk te maken van de source code. Stel, je wilt een softwarecomponent uitleveren of verspreiden, onafhankelijk van de source code. Van die component zijn verschillende versies in omloop. Ik lever hem nu uit, ontwikkel er aan door en dan komt er een tweede versie en op een gegeven moment wil iemand misschien een applicatie maken die twee verschillende versies van dezelfde softwarecomponent gebruikt. Dat probleem is nooit opgelost. In .Net heeft men een oplossing bedacht: het werken met assemblies en strong naming. Als je daar even over nadenkt: dat is van zulk een schoonheid vanuit wetenschappelijk oogpunt. Daar kun je niet omheen. Als je naar het concept kijkt dan zie je dat ze iedere keer de optimale beslissing hebben genomen bij iedere keuze. En uiteindelijk zijn ze bij dat idee uitgekomen. Om het wat te verduidelijken: als je een software component hebt en je wilt verschillende versies onderscheiden, maar je wilt ook naar die component kunnen verwijzen. Je moet die dus een naam geven. Maar als je ook nog verschillende versies wilt onderscheiden, moet je dus als onderdeel van de identificatie van de component ook de versie betrekken. Dan kan dezelfde versie en dezelfde component ook nog door verschillende leveranciers gemaakt, en in verschillende talen. Al die aspecten zijn onderdeel van een strong name.

> Het idee van versioning is ook toepasbaar op de runtime zelf

Maar je moet ook een component kunnen aanroepen zonder dat je precies weet welke versie het is.

Dat is het tweede punt, het eerste is dat je hem wilt identificeren, het tweede dat je hem wilt kunnen gebruiken. Daar zitten een aantal aspecten aan vast. Misschien wil je niet heel specifiek zijn in welke versie je wilt gebruiken, of je wilt alleen de laatste versie. Of je weet dat je systeem werkt, dus je wil juist wel die versie hebben waarmee het gebouwd is. Apart van die strong name moet je dus kunnen specificeren welke versies compatibel met elkaar zijn. Dan is er de infrastructuur waarin je de strong name geeft, volledig of niet volledig. Het systeem zoekt dan voor jou met allerlei heuristieke algoritmes en dan krijg je die component terug. Het zoekmechanisme plus het naammechansime lossen samen het probleem op. Dat is een diamant die diep verborgen zit in de runtime van .Net. Als wetenschapper kan ik daar van smullen.

Is er nog een ander voorbeeld?

Generics. In het framework zit bijvoorbeeld een lijst collection classes. Het is nu zo gespecificeerd dat ik een integer, een boolean, en een buttonobject allemaal in dezelfde lijst stop, omdat ze allemaal worden getypeerd als object. Het probleem is echter dat je het er bijvoorbeeld als een character in stopt, maar niet meer weet wat het was op het moment dat je het er weer uit haalt. Generics lijkt heel erg op C++ templates. Je kunt een schema maken en die kun je dan instantiëren. Je zegt: dit is een lijst van een bepaald type, en dan kun je zeggen nu heb ik een integer, nu een boolean, maar je kunt de code wel schrijven voor een lijst van objecten en die code specialiseren voor ieder specifiek type. Dat idee heet parametric polymorphisme. Het bestaat al een jaar of dertig en er is een aantal programmeertalen dat het al toepast. In Java is er een manier bedacht, waarbij je dat doet door transformatie op je Javaprogramma, zodat uiteindelijk niets aangepast hoeft te worden aan de JVM. Dat is een suboptimale oplossing. Het scheelt alleen een beetje schrijfwerk voor de programmeur, maar je hebt geen enkel echt voordeel. Een groep bij Microsoft Research Cambridge heeft een voor-

stel gemaakt voor generics voor de runtime, waarbij de runtime zelf generics ondersteunt op zo'n algemene manier dat verschillende talen daar weer gebruik van kunnen maken. De runtime zelf bijvoorbeeld, als hij een integeren een characterlijst heeft, maakt daar specifieke code van. Maar bij een lijst van buttons en een lijst van forms, allebei objecten, gebruikt hij dezelfde code. Dat is heel efficiënt. Met het idee dat versioning ook op de runtime zelf toepasbaar is en het idee van deversioning is gewoon ingebouwd in het raamwerk, daarmee kun je dus altijd zorgen dat je in de toekomst kunt blijven innoveren. Generics is dus ook een diamant die in de toekomst schittert.

Er zijn natuurlijk ook voorbeelden van dingen die niet optimaal opgelost zijn.

Je krijgt altijd een moeilijke situatie op het moment dat de semantieken van verschillende talen erg uit elkaar lopen. Dan moet je een beslissing nemen, een compromis maken. Het wordt dan of een beslissing die slecht is voor het een, en goed voor het andere of andersom. Een voorbeeld daarvan is het idee van constructors: als je kijkt naar een taal als Eiffel – de Eiffel.Net compiler. In Eiffel is het idee van een constructor niets speciaals, het is een methode die extra eigenschappen heeft, namelijk dat je hem als constructor kunt gebruiken, in andere talen zoals C# en VB, is een constructor juist veel gerestricteerder dan een methode. Je ziet dan dat de semantiek van de talen ver uit elkaar ligt, en daar moet je dan een beslissing nemen en we hebben dus besloten om constructoren een speciale status te geven, in plaats van zoals in Eiffel een algemene status. Op dat moment zit je in een soort dilemma. Om het op te lossen maak je dan een compromis. Van de andere kant kan Eiffel nog steeds hun semantiek implementeren, alleen moeten ze daar wat meer omheen werken.

Erik Meijer: "Op dat moment zit je in een dilemma."

Een ander voorbeeld waar je moeilijk een oplossing voor kunt vinden, is multiple inheritance. Er zijn veel talen die multiple inheritance ondersteunen, alleen het vervelende is dat ze allemaal een ander idee hebben van wat multiple inheritance nu precies is. Als je nu kijkt naar de doorsnede van al die multiple inheritance-modellen, dan is die leeg. Ik kan dus in de runtime geen multiple inheritance ondersteunen zodat dat bruikbaar is voor zoveel mogelijk talen. Maar je kunt het dus wel zelf uitprogrammeren. Het is echter onmogelijk om een gemeenschappelijk mechanisme te scheppen, omdat al die ideeën zo uit elkaar lopen. Dan moet je dus zorgen dat je de primitieven aanbiedt waarmee iedereen dat dan zelf kan opbouwen. Dat is het mooie van het ontwerpen van een executie omgeving: je kijkt eerst of je een algemene oplossing kunt aanbieden, en als dat niet kan, moet je zorgen dat je genoeg primitieven hebt zodat iedereen toch zijn taal kan implementeren.

> In programmeertalen ligt ook een stuk 'religie' besloten

De mogelijkheid om de win32 API aan te roepen is leuk, maar in het licht van toekomstige compatibiliteit toch ook weer niet zo handig.

Je kunt nog steeds de win32 API aanroepen, en er is een hoop aandacht steeds in het framework om legacy software aan te roepen. Daar hebben mensen in geïnvesteerd, je kunt niet verwachten dat mensen van de ene op de andere dag overstappen. Je kunt dus heel gemakkelijk COM-componenten aanroepen en je kunt heel gemakkelijk gewone dll's aanroepen. Maar als je vooruit kijkt is het natuurlijk niet slim als je nieuwe .Net programmatuur afhankelijk is van de win32 API. Dat het kan, wil niet zeggen dat je het ook moet doen. Als je bijvoorbeeld je eigen legacy software langzaam wilt migreren, dan kan dat, maar als je naar de toekomst kijkt, moet je natuurlijk proberen het .Net framework te gebruiken. Dat is als het ware de nieuwe Win32 API, de nieuwe verzameling van componenten waarmee je software bouwt.

Dit is nu af, maar hoe nu verder? Ik neem aan dat er al plannen zijn voor toekomstige ontwikkeling?

Zeker, ik kan daar een klein beetje van laten doorschemeren. Je bent nooit klaar, maar op een gegeven moment moet je het shippen en dan moet je een aantal dingen uitstellen voor een volgende versie. Daarnaast zijn er altijd allerlei nieuwe ideeën zijn en er is voortdurend feedback van gebruikers die binnen komt. Op de PDC hebben ze verteld dat de runtime ingebed wordt in SQLServer, zodat je de stored procedures kunt schrijven in iedere willekeurige taal in plaats van de taal die daar ingebed zit. Dus je krijgt nu allemaal van dat soort dingen: als die runtime er eenmaal is, dan is het ook mogelijk om dat in te bedden, maar als je ingebed zit in SQLServer dan heeft dat ook weer allerlei gevolgen voor de runtime zelf. Het security model van SQLServer, is dat precies het security model dat .Net biedt? Dat komt wel overeen maar in SQLServer wil je dan nog wat andere rollen definiëren. Een ander voorbeeld is BizTalk, iets waar ik trouwens ook heel erg van gecharmeerd ben als wetenschapper omdat dat ook direct gebaseerd is op de pi calculus, een theorie uit de procesalgebra. Ik vind het ongelooflijk dat een commercieel product is gebaseerd op een abstracte theorie van proces algebra waarvan de meeste echte informatici misschien nooit van gehoord hebben. Het is gebaseerd op die theorie het product is daaromheen ontwikkeld zodat de gewone programmeur ermee kan werken. Maar de volgende versie van BizTalk wordt ook vertaald naar de runtime, dus dan krijg je een veel sterkere integratie dan nu het geval is. Dat soort dingen zitten er allemaal voor de toekomst aan te komen. Mijn taak voor de volgende versie is om ervoor te zorgen dat het een goed platform blijft voor zoveel mogelijk programmeertalen en zelfs een beter platform wordt. We zijn nu aan het kijken naar de knelpunten voor bepaalde taken. Het blijft een veeltalig platform en we gaan kijken hoe we het nog kunnen verbeteren.

Hoe ziet een ideale taal er nu uit?

Daar kan ik alleen maar een persoonlijk antwoord op geven. In programmeertalen ligt ook een stuk 'religie' besloten. Mensen hebben er vaak een heel uitgesproken mening over, maar het is heel moeilijk om dat ook wetenschappelijk te meten. Maar mijn ideale talen zijn declaratieve talen, waarbij je dus zo min mogelijk vertelt hoe je iets moet doen, maar alleen wat je moet doen. De taal waarin ik gewerkt heb, Haskell is een voorbeeld van een ideale taal. Waar ik ook aan gewerkt heb, is een scripting taal. Haskell is een vrij grote taal, bedoeld om gewone applicaties mee te programmeren, maar je wilt toch ook script-achtige toepassingen ontwikkelen. Daarvoor heb ik een taaltje ontwikkelt: Mondrian - een functionele taal, maar dan voor scriptingdoeleinden. Voor de toekomst vind ik dat de programmeertaal zo abstract mogelijk moet zijn, en onnodige details moet weglaten. Daarom geloof ik ook niet dat er een ideale taal is, want voor ieder probleem zijn er andere details belangrijk. Voor ieder programmeerprobleem heb je abstracties nodig. Als ik bijvoorbeeld iets heel laag-bij-de-gronds moet doen waarbij ik controle moet hebben over het geheugen, dan moet ik niet een taal hebben die daarvan abstraheert. Aan de andere kant, wanneer ik een applicatie maak waarbij details van geheugengebruik niet belangrijk zijn, dan kan ik daar wel van abstraheren, dus daarom heb je veel verschillende talen nodig.

Wat je in de wetenschappelijke wereld ziet, maar nog niet in commerciële programmeertalen, is het idee van type-referentie. Als je in C#, in Java of in C zit, moet je als je een methode definieert, types aangeven, integer, boolean, array. Of - en dat is het extreme - je hebt een taal als Python, Perl of tcl/tk, waarbij je helemaal geen types opgeeft en types at runtime worden gecheckt. Maar de compiler kan dat ook voor je uitzoeken. Als je dat doet heb je de voordelen van statische typering. Voordat je programma draait, kan de compiler al zeggen of het wel of niet fout zal gaan. Je hebt ook het voordeel dat je die types niet hoeft op te schrijven, zoals in dynamisch getypeerde talen. Ik denk dat dat een gat in de markt is. Eerlijk gezegd verbaast het me dat er nog nooit door iemand een serieuze industriële taal is gemaakt, die daarop gebaseerd is. Terwijl het een technologie van twintig, dertig jaar oud is.

Hoe ziet u uw toekomst?

Ik ben ook adjoined professor aan het Oregon Graduate Institute, maar daar geef ik geen colleges. Ik ben verbonden aan het instituut en heb daar ook -dat is al weer vier jaar geleden- een sabbathical gedaan. Daar onderhoud ik als het ware de band met de wetenschap. Ik vind het spanningsveld tussen theorie en praktijk belangrijk. Ik heb lange tijd gewerkt aan de theoriekant, en nu zit ik aan de praktijkkant. Tussen de programmeurs en wat wij doen is ook wederzijdse feedback: wat zij willen en wat wij voor hen willen maken. Daar is een soort spanningsveld. De programmeur heeft namelijk heel specifieke problemen en zegt bijvoorbeeld: het zou mooi zijn wanneer jullie dit of dat zouden kunnen doen. Ik probeer al die dingen te bekijken, en in plaats van al die specifieke dingen probeer ik een oplossing te bedenken die al die problemen tezamen weer oplost. Anders krijg je opnieuw een allegaartje. Als ik op langere termijn kijk, is er nog een ander spanningsveld wat ik heel mooi vind: tussen het productteam, als ware het de theorie, én de echte praktijk. Ook daar zou je weer over het hek kunnen springen, door meer vanuit de kant van de gebruiker te kijken en je af te vragen: hoe kun je ervoor zorgen dat je een product krijgt dat precies doet wat je wilt. Spanningsvelden tussen twee gebieden intrigeren me.

Webservices worden heel belangrijk als we Gartner en anderen mogen geloven. Er bestaan echter ook minpunten. Security is heel voor de hand liggend voorbeeld.

Het is ook nog een heel jonge technologie, een typisch voorbeeld waar de praktijk voorloopt op de theorie. Er is duidelijk een grote behoefte aan iets als webservices maar ook hier kun je niet naar de wetenschap gaan en zeggen: 'jongens wat hebben jullie daar voor oplossingen voor?'. Er zijn nog open kwesties, maar ik verwacht dat dat zich allemaal wel zal oplossen.

Welke open kwesties bedoelt u?

Voor transacties en SOAP bijvoorbeeld zijn er wel voorstellen gedaan, maar er is nog geen echte consensus over. Aan de andere kant, op het moment dat je de runtime in de database hebt geïntegreerd is de noodzaak voor transacties veel geringer, want alle transacties vinden als het ware locaal plaats in de database. Terwijl dat bij BizzTalk weer anders is opgelost Je hebt dus óf superkort durende transacties, dat los je op door transacties in de database te draaien, of je hebt heel langlopende transacties van weken en maanden, en dat is in BizzTalk opgelost door compenserende acties te definiëren. Daar zie je andere alternatieven ontstaan, of het probleem wordt minder vervelend omdat de wereld verandert. Maar er wordt hoe dan ook hard aan gewerkt.

Een ander idee binnen de XML-wereld is dat van schema's. Daar zie je ook een ontwikkeling van DTD's en andere voorstellen voor schema's. Daar is nu langzamerhand consensus over bereikt. Ik heb goede hoop dat voor al dat soort problemen een oplossing komt. Als je wacht totdat er oplossingen voor zijn, dan kom je ook nergens. De echte problemen kom je pas tegen door het gewoon te doen. Anders heb je misschien een oplossing voor een probleem wat niet bestaat.

Het concept is enerzijds heel eenvoudig, aan de andere kant zal menig VB-ontwikkelaar er een harde dobber aan hebben. Sommige mensen vinden al dat XML heel ingewikkeld is.

Ja, dat je nu zoveel van XML ziet is ook omdat het nieuw is. In het ideale geval zie jij die XML als programmeur helemaal niet. Als je in Visual Studio een webservice gebruikt, dan zie jij als programmeur absoluut niet dat daar XML bij wordt gebruikt. Als je die webservices ergens heel diep moet implementeren, moet je het wel weten, maar dan is het ook weer een detail waarvan je kunt abstraheren. Het maakt niet zoveel uit; je weet dat het XML is en dat heeft als voordeel dat de implementatie gemakkelijk kan zijn maar als jij daarmee programmeert hoef je niet te weten dat het XML is. Voor een bepaalde groep van programmeurs zal het steeds meer naar de achtergrond verdwijnen omdat het voor hen geen relevant detail is. Ik denk juist dat het in zekere zin steeds gemakkelijker wordt. Geheugenbeheer hoef je niet meer expliciet zelf te doen, zoals in C, een heleboel onbelangrijke details worden nu voor je gedaan.

Het wordt wel lastiger voor mensen die een beetje losbandig programmeren. Het is nu allemaal wat strakker, in die zin is het meer een echte programmeertaal gewor-

De .Net runtime wordt ingebed in SQLServer

Erik Meijer: "Die kans krijg je niet iedere dag"

den. Maar we zijn er ons wel degelijk van bewust dat een aantal aspecten van Visual Basic, edit and continue bijvoorbeeld, verbetering behoeven. We proberen echt de Visual Basic experience te verbeteren in de volgende versie van .Net. Dat is zeker één van de hoofdpunten. Maar aan de andere kant, ik denk dat uiteindelijk iedereen beter af is met VB.Net.

Maar het zou in een volgende versie dus gemakkelijker worden voor VB-programmeurs. Niet iedereen stapt meteen over.

Als je niet wilt overstappen, dan hoeft dat niet, er is compatibiliteit. Aan de andere kant, de frameworks die

De doorsnede van al die multiple inheritance-modellen is leeg

er nu zijn, zijn zoveel krachtiger en gemakkelijker dan wat er was. We doen er dus alles aan om ervoor te zorgen dat de programmeurs zo productief mogelijk kunnen zijn en als er een bepaalde programmeeromgeving niet helemaal voldoen aan hun wensen, dan proberen we dat aan te passen. We praten ook met de compilerbouwers: 'wat zijn jullie ervaringen geweest, waar zaten de knelpunten, waar hadden jullie er moeite mee efficiënte code genereren?' Dan kijken wij hoe we de runtime kunnen verbeteren om tegemoet te komen aan die mensen. Het is zeker nog niet af en we proberen het nog beter te maken dan het nu al is.

Laatste vraag: uit het enthousiasme waarmee u vertelt, maakt ik op dat u het allemaal wel erg leuk vond.

Zeker, ik moet zeggen dat ik het een ongelooflijke kans vind om aan een platform te werken, waarmee de toekomstige generatie van applicaties wordt gebouwd, dat ik de programmeertalengemeenschap daarmee als het ware kan helpen. Die kans krijg je niet iedere dag.

Tekst en fotografie Dré de Man