John Lakos is de auteur van het boek *Large-scale C++ software design*, dat handelt - zoals de titel al doet vermoeden - over het gebruik van C++ in grootschalige softwareprojecten. Lakos put voor die kennis uit zijn ervaring met het ontwikkelen van IC-ontwerpprogrammatuur, waar hij voor Mentor Graphics bij was betrokken.

HANS VAN THIEL

Interview John Lakos

'C++ is niet bedoeld voor

onervaren mensen'

PT Embedded Systems: 'In uw boek *Large-Scale C++ Design* schrijft u dat de lessen met betrekking tot de bouw van zeer grote C++-programma's met vallen en opstaan geleerd moesten worden. Toen u bij Mentor Graphics met deze speciale problematiek in aanraking kwam was hierover nog niets bekend. Wat voor project was dit?
John Lakos: 'Het bedrijf waar ik werkte werd na enkele eerdere fusies in 1990 onder de naam Silicon Design Labs overgenomen door Mentor Graphics. Wij ontwikkelden cad/cam-programmatuur voor IC-ontwerp die bekend staat als Generator Development Tools (GDT). Rond die tijd kregen we belangstelling voor de voordelen die object-oriëntatie zou kunnen bieden. Ik werd belast met *quality assurance* (QA) en in die functie kwam ik tot de overtuiging dat er een heroverweging van de architectuur nodig was. In de grond van de zaak komt het op twee zaken aan: geen achterdeuren en geen cyclische afhankelijkheden.

PT Embedded Systems: 'Wat betekent dat: geen achterdeuren?'
John Lakos: 'Alleen publieke toegang, met gebruik van de daarvoor bestemde interfaces.'

PT Embedded Systems: 'In uw boek onderscheidt u het fysieke ontwerp van het logische ontwerp. Het fysieke ontwerp betreft het onderverdelen van het programma in bronbestanden, in het bijzonder *header files* en *body files*. Een *dot-h file* en een *.dot-c file* vormen samen een compileerbare eenheid. In het kader van het fysieke ontwerp is dit een *component*. Is het kenmerk van zeer grote programma's dat de compileer- en linktijden niet meer te negeren zijn?'
John Lakos: 'Nee, bij zeer grote programma's gaan de linktijden een rol spelen. Compileertijden vormen al voor middelgrote programma's een factor van betekenis.'

PT Embedded Systems: 'Wat is een zeer groot programma?'
John Lakos: 'Tien miljoen regels code. Een paar honderdduizend regels, is middelgroot.'

PT Embedded Systems: 'Tien miljoen?'
John Lakos: 'U vroeg wat een zeer groot programma is. Het cad-ontwikkelproject voor IC-ontwerp dat Mentor Graphics van 1985 tot 1995 onder constructie had omvatte uiteindelijk tien miljoen regels code. Ik schat dat er in die periode zo'n 200 miljoen dollar aan dit project is besteed. Het werd *Falcon* genoemd en werd als zodanig enigszins berucht door de problemen er omheen. Door de vele cyclische afhankelijkheden van fysieke componenten werden de linktijden langer en langer en werd het steeds tijdrovender om veranderingen in de code te implementeren. Uiteindelijk hebben we het aantal programma regels weten te verminderen tot 1 miljoen, maar het werd mij zo rond 1989/1990 wel duidelijk dat cyclische afhankelijkheden moeten worden vermeden.'

PT Embedded Systems: 'In uw boek gaat u diep in op de vraag hoe in C/C++ cyclische afhankelijkheden, waarbij A afhangt van B en B van C, maar C weer van A, vervangen kunnen worden door lineaire afhankelijkheden, bijvoorbeeld B en C beide alleen van A. U noemt dit nivellering (*levelization*, red.).'
John Lakos: 'Er bestaan twee principiële manieren om met verandering om te gaan. De eerste is om verandering te vermijden; de tweede is om veranderingen te isoleren (*insulation*, red.). Sommige entiteiten, zoals stacks, tijd en datum, of grafische punten, zijn zo elementair dat het gewoon te veel overhead kost om daar veranderingen in aan te brengen. In alle andere gevallen moet je componenten zo goed mogelijk isoleren, zodat veranderingen een minimaal effect hebben op andere componenten.'

PT Embedded Systems: 'U introduceert ook een software-metriek hiervoor, de CCD en een aantal afgeleide kengetallen.'
John Lakos: De *cumulative component dependency* is de som over alle componenten C_i in een subsysteem van alle componenten die nodig zijn om elke C_i incrementeel te testen. Zo geeft de CCD aan wat het kost om een *test driver* te linken en de CCD is daarmee een maat voor de prijs in bouwtijd van regressie testen.'

PT Embedded Systems: 'Een aantal problemen die u behandelt lijkt direct te maken te hebben met eigenschappen van C/C++. *Inline functions*, *friend functions* en *free functions* zijn bijvoorbeeld allemaal weggelaten uit Java. Anderzijds introduceert u *interfaces* en *packages*, die juist wel in Java worden ondersteund.

> 'Het cad-ontwikkelproject voor IC-ontwerp dat Mentor Graphics van 1985 tot 1995 onder constructie had omvatte uiteindelijk tien miljoen regels code.'

John Lakos: 'In Java is een file geen component; daar is een component vrijwel hetzefde als een klasse. In C++ is een component meestal een klasse, maar met wat meer er omheen. U kunt het vergelijken met het kopen van een huis. Als je in Java een huis koopt krijg je alleen het huis, maar in C++ koop je ook tegelijk een stuk land er omheen. Een Java *jar-file* is wel een component in die zin. De problematiek van cyclische afhankelijkheid heeft verder niets met C++ te maken. Die speelt net zo goed in alle andere programmeertalen, zelfs Cobol.'

PT Embedded Systems: 'Dus u ziet het mogelijk door elkaar lopen van gestructureerd programmeren in C met de objectoriëntatie van C++ niet als bron van fouten?'
John Lakos: 'Je kunt ook overijverig zijn met betrekking tot OO. In deze context ›

heb ik een collega eens de vraag horen stellen: 'Wilt U objectgeörienteerd zijn, of wilt U succesvol zijn?'
Bepaalde basistypen zijn eenvoudigweg geen objecten en vrije operatoren zijn in bepaalde gevallen gewoon handig. Friend-functies hebben evenzeer hun nut; je moet ze alleen niet al te ver van elkaar af plaatsen omdat dat cyclische afhankelijkheid introduceert. Het delen van een interface bevordert modulariteit. Packages produceren een fysieke partitie (*physical partition*, red.). C++ staat je toe om van alles te doen. Met Java is het net alsof je een chirurg een computergestuurde laser geeft. Je kunt geen fouten maken maar je kunt ook niet doen wat je wilt. C++ vereist training. Het is niet bedoeld voor onervaren mensen.'

PT Embedded Systems: 'Bjarne Stroustrup zei in een interview met ons dat er verschillende soorten programmeertalen zijn voor verschillende behoeften. C++ was volgens hem bedoeld voor professionele toepassingen.'
John Lakos: 'Het is interessant dat hij dat zei. Naar mijn mening vallen Java Beans en dergelijke onder consumenten software. C++ moet je echt leren op doctoraal niveau. (*Masters*, red.). Het is bijvoorbeeld belangrijk te weten hoe een compiler werkt. Wij nemen ook alleen maar mensen aan die doctoraal examen hebben gedaan in de informatica, geen afgestudeerden in andere disciplines.'

PT Embedded Systems: 'Vindt u dat de ISO-standaardisatie van C++ die in 1999 officiëel is geworden tot betere tools heeft geleid?'
John Lakos: 'De standaardisatie is een probleem, met name door de veranderingen. Veel aanpassingen zijn op zich niet slecht, maar ze hebben er wel voor gezorgd dat oude code niet meer overdraagbaar is. Het is wel mooi dat de scope van een in een *for loop* gedeclareerde integer nu beperkt is tot die loop, of dat *inline functions* nu een externe *linkage* krijgen, maar bestaande code draait niet meer. Ook de introductie van een expliciet *bool* type is naar mijn mening zijn doel voorbij geschoten, al begrijp ik heel goed waarom dat is gedaan. Over het algemeen echter, vind ik dat de standaardisatie te ver is doorgeschoten. Men had dichter bij CFRONT 3.0 moeten blijven. Dat was toch de de facto standaard.'

PT Embedded Systems: 'Het eerste deel van *Large-Scale C++ Design* behandelt het fysieke ontwerp, het tweede deel gaat over het logische ontwerp. Wat is de plaats van het fysieke ontwerp in de *software development life cycle* (SDLC)?'
John Lakos: 'Op het moment van de decompositie en compositie, dus wanneer je een probleem gaat onderverdelen in subproblemen en hun onafhankelijke oplossingen, en dan gaat hercombineren.'

PT Embedded Systems: 'Bij het ontwerpen van de architectuur?'
John Lakos: 'Nee, eerder al, hoewel nog niet bij het vaststellen van de requirements. Meteen daarna, zodra je begint met het analyseren van het probleem.'

PT Embedded Systems: 'Een van de nieuwigheden van C++ ten opzichte van C is het gebruik van *exceptions*, waarmee fouten in een hiërarchie kunnen worden doorgegeven. In uw boek worden die niet genoemd.'
John Lakos: '*Exception handling* is verschillend bij verschillende leveranciers. Toepassingen die echt portable zijn, gebruiken geen exceptions. Er vallen twee dingen te onderscheiden: *error reporting* en *error status*. Foutmelding is alleen van belang voor een menselijke persoon. Die wil dat de melding, het liefst zo hoog mogelijk in de hierarchie, in leesbare vorm ergens opduikt. De fout-status heeft eigenlijk maar drie toestanden: het programma slaagt, het programma faalt, of het programma faalt op ellendige wijze. Hiervoor moet je geen exceptions gebruiken; hierop moet je testen. Zelf gebruik ik zoveel mogelijk de *assert* statement om condities te testen. Die gebruik je dan alleen in de debug fase. Ik heb geen exceptions nodig. Ik zeg niet dat ze slecht zijn, maar ik heb ze niet nodig.'

PT Embedded Systems: 'In appendix B van uw boek geeft u een aantal utilities om component-afhankelijkheden in een programma te bepalen en hanteren. Vindt u dat dergelijke utilities onderdeel zouden moeten zijn van tools?'
John Lakos: 'Zeker. Het extraheren van componenten vergemakkelijkt het testen in hoge mate en het is zeker niet voldoende dit alleen op lokaal niveau te doen. Het moet op package niveau. De bestaande mogelijkheden van commerciële tools schieten wat dat betreft tekort. Ik moet er wel op wijzen dat de code die bij appendix B hoort en die op te halen is bij ftp://ftp.aw.com/cp/lakos niet helemaal compatible is met de nieuwe standaard. Gebruikers zullen dus wat kleine aanpassingen moeten maken.'

PT Embedded Systems: 'Een vraag afkomstig vanuit de ontwikkelaars, de groep potentiële deelnemers aan uw workshop vandaag. Wat gaat er het meest mis aan grootschalige softwareprojecten en wat is daar aan te doen?'
John Lakos: 'Het grootste probleem is een gebrek aan besef van cyclische en excessieve link- en compileertijd afhankelijkheden.'

'Met Java is het net alsof je een chirurg een computergestuurde laser geeft. Je kunt geen fouten maken maar je kunt ook niet doen wat je wilt.'

PT Embedded Systems: 'Dat is duidelijk. Wat zijn uw toekomstplannen? Werkt u bijvoorbeeld op dit moment weer aan een boek?'
John Lakos: Dit boek gaat in zekere zin alleen over testbaarheid en niet over testen. De bedoeling is eigenlijk componenten te implementeren, het interface te ontwerpen, te implementeren, te documenteren en te testen. Dit boek is een *design* boek en wat we nodig hebben is een *development* boek dat zich richt op de professionele ontwikkelaar. Ik zou het boek willen omwerken tot twee delen, waarbij in het eerste het zwaartepunt komt te liggen bij het ontwerp en in het tweede bij de implementatie. Ik zou het boek ook wat korter maken (glimlachend).'

PT Embedded Systems: 'Waar werkt u op dit ogenblik aan?'
John Lakos: 'Sinds 1997 ben ik bij Bear Stearns (Amerikaanse zakenbank, red.) betrokken bij het ontwikkelen van hoogst betrouwbare, herbruikbare software-componenten voor een variëteit van *high performance* financiële toepassingen.' ◻

Literatuur:
Lakos, J. Large-Scale C++ Design. Addison-Wesley. (1996). ISBN 0-201-63362-0