Coderen en modelleren gaan hand in hand

Visual Studio 2010 Team Architect Edition

Marcel de Vries

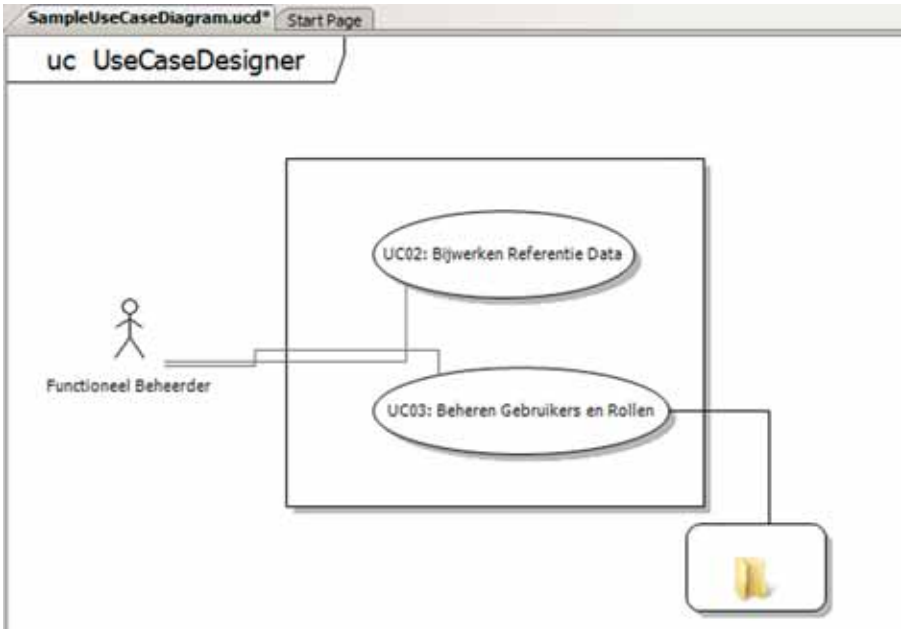
Hoe vaak kom je het niet tegen in de praktijk: je wordt aan een ontwikkelteam toegewezen en om je in te werken vraag je naar de documentatie van het systeem. Meestal krijg je een verontschuldigende glimlach en de opmerking dat er helaas geen tijd is geweest deze bij te werken. Vervolgens krijg je een beknopt document en - met wat geluk - een aantal UML diagrammen die in het verleden zijn gemaakt. Na deze te hebben bestudeerd denk je bij jezelf: “Dat snap ik, aan de slag”. Je opent Visual Studio en vol goede moed duik je in de code en gaat op zoek naar de onderdelen, die je zojuist in de diagrammen hebt gezien. Helaas blijkt de ‘documentatie’ dusdanig te zijn verouderd dat deze zo goed als geen hulp meer biedt om snel in de bestaande code-base thuis te raken.

De meeste programmeurs zien wel de noodzaak van het gebruik van modellen en diagrammen, maar vaak zijn de modellen nogal onderhevig aan de tijdgeest en geven deze niet meer weer dan de intentie die we ooit hebben gehad. Ze lijken niet meer op hetgeen we aan code onderhanden hebben. De vraag is waardoor dit fenomeen ontstaat. In mijn beleving heeft dit alles te maken met het doel dat we met de modellen willen bereiken. Primair worden ze gebruikt voor het communiceren met de medeprojectleden. Naarmate iedereen meer in de materie thuis is, wordt het belang van modellen voor het team minder groot. Het up-to-date houden van modellen kost veel tijd en daarom wordt het een sluitpost van de begroting. Voor de nieuwkomers in het team is het daarentegen zeer belangrijk dat de modellen up-to-date zijn, dus hoe kun je dit probleem doorbreken?

Je kunt modellen en diagrammen voor meer doeleinden gebruiken dan puur communicatie. Bijvoorbeeld voor het controleren of de software die wordt gemaakt, voldoet aan de eisen die je eerder hebt uitgedrukt in een diagram. Je kunt diagrammen gebruiken als alternatief om door de software te navigeren. Visual Studio 2010 Team Architect Edition gaat hiervoor goede handvatten bieden.

Design First

Modellen en diagrammen worden primair gebruikt om elkaar duidelijk te maken hoe we een softwaresysteem willen bouwen of om te laten zien wat er al is gebouwd. Visual Studio 2010 Team Architect Edition ondersteunt beide werkwijzen voor het gebruik van modellen. Deze twee werkwijzen zijn beter bekend als ‘Design First’ en de ‘Code First’-aanpak. Bij de ‘Design First’-aanpak worden eerst modellen gemaakt die alle informatie bevatten over het te realiseren softwaresysteem. Vervolgens worden diagrammen aan de modellen toegevoegd die een bepaalde view geven op een specifiek deel van het systeem. Ieder diagram dat je toevoegt heeft tot doel op een specifieke informatie-



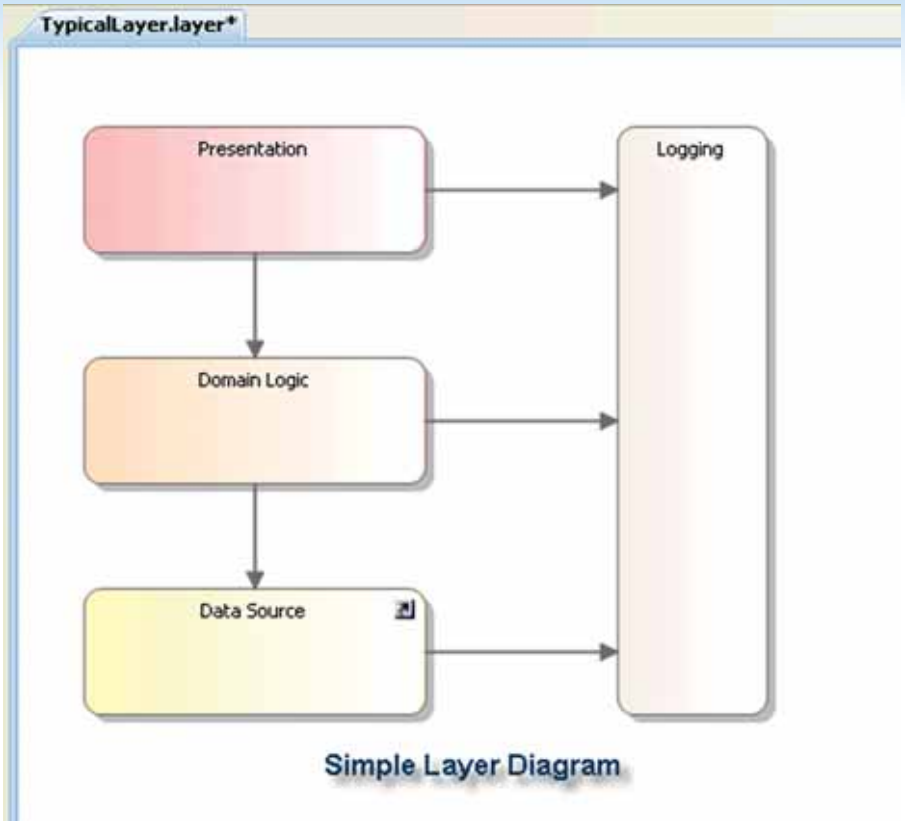
FIGUUR 1: USE CASE DESIGN

vraag antwoord te geven. Veelal wordt er in de softwareindustrie gebruik gemaakt van de standaardnotatie op basis van de Unified Modelling Language (UML). UML biedt een aantal standaarddiagramtypen die zijn bedoeld op een specifieke manier naar het systeem te kijken (view). Zo is er een weergave voor het vastleggen van requirements door middel van een Use Case Diagram. Hierbij wordt gevisualiseerd wie er interactie hebben met het systeem (Actors) en wat het systeem aan functionaliteit biedt (Use case). Daarnaast heb je bijvoorbeeld een Activity Diagram. In een dergelijk diagram wordt uitgedrukt wat de stappen zijn voor de realisatie van bijvoorbeeld een Use Case. Activity Diagrams kunnen ook heel goed worden gebruikt om op een hoger niveau aan te geven welke bedrijfsprocessen het systeem gaat ondersteunen.

In Visual Studio 2010 Team Architect Edition gaat Microsoft ondersteuning bieden voor het maken van modellen op basis van de UML 2.1 specificatie. Microsoft is sinds medio 2008 lid van de OMG (verantwoordelijk voor de UML-specificatie) en werkt nu zelf actief mee aan het verder ontwikkelen van deze standaard. In de Team Architect Edition gaat Microsoft ondersteuning bieden voor tenminste de volgende diagrammen: Use Case, Class (Conceptual en Physical), Activity, Sequence (Conceptual & Physiscal) en Component Diagram.

In Figuur 1 is een voorbeeld te zien van een Use Case Diagram. Dit diagram is gemaakt met de Community Technical Preview van Visual Studio 2010 (CTP).

Een onderdeel van de UML-standaard is het concept van 'profiles'. Een UML-profile biedt de mogelijkheid UML aan te passen voor gebruik in een specifiek oplossingsdomein. Zaken die je onder andere kunt vastleggen in een profile zijn standaardtypen die gebruikt kunnen worden uit bijvoorbeeld een bedrijfseigen bibliotheek, of een set aan stereotypen die vaak worden toegepast. Zelfs specifieke symbolen die in een specifiek verticaal segment wordt toegepast kunnen onderdeel zijn van het UML-profile dat je beschikbaar maakt voor je eigen organisatie. Een interessant aspect is dat je een dergelijk profile kunt ontwikkelen voor je eigen organisatie. Stel dat je als softwareafdeling vaak gelijksoortige software moet realiseren en je hebt daarvoor een werkwijze ontwikkeld die je wilt vastleggen ten aanzien van de modellen die je wilt gebruiken. Het is mogelijk hiervoor je eigen profile op te stellen en deze te delen in je organisatie. Sterker nog, het is mogelijk deze diagrammen te controleren op het feit dat een bepaald profile is gebruikt, zodat uniformiteit eventueel zelfs af te dwingen is. Het gebruik van profiles is ook ondersteund in Visual studio 2010.



FIGUUR 2: TIER LAYER DIAGRAM

Naast het gebruik van UML-profiles is er ook een andere optie die zorgt voor uniformiteit in een organisatie. Dit betreft het maken van een zogenaamde modelling template, die als basis kan worden gebruikt voor nieuwe projecten. Een template bevat een set aan modellen, die na het aanmaken van de modelling solution direct vastliggen in het startmodel. Wanneer je bijvoorbeeld gebruik maakt van het Rational Unified Process (RUP) is het heel gebruikelijk dat de architect een model conform de zogenaamde '4 + 1 Views of software architecture' hanteert. Dit '4+1 model' definieert een vijftal gezichtspunten van waaruit je de architectuur wilt bezien voor een bepaalde oplossing. Als je wilt dat dit voortaan standaard wordt toegepast binnen de organisatie, is het mogelijk zelf een dergelijke set aan modellen vast te leggen in een Visual Studio modelling project en deze te exporteren als een template. De template kun je vervolgens importeren in Visual Studio, waarna deze als standaard modeltemplate beschikbaar is voor anderen.

Traceability en Workitems

Een van de belangrijkste voordelen van het gebruik van modellen en diagrammen is het vastleggen van zogenaamde traceability. Traceability wil zeggen dat de gerealiseerde code herleidbaar is naar wensen en eisen, die zijn vastgelegd voor de realisatie van het product. In tegenstelling tot het eenmalig maken van modellen en diagrammen kunnen deze gedurende de gehele levenscyclus van het product inzicht geven in de vraag welke onderdelen van de software zijn gemaakt voor een bepaald functioneel doel als beschreven in de requirements. In een RUP-gebaseerde aanpak is dit bijvoorbeeld de traceability naar Use Cases en voor bijvoorbeeld een MSF-aanpak betreft dit de herleidbaarheid naar Scenario's. De reden dat dit belangrijk is, heeft alles te maken met de onderhoudsfase van een systeem. Traceability zorgt er namelijk voor dat veranderingen die worden ingediend ook door middel van impactanalyse vooraf in te schatten zijn ten aanzien van de impact op het systeem.

Tot op heden is het zo dat Team System 2008 het al mogelijk maakt bij het inchecken van code in version control aan te geven welk workitem daarbij geassocieerd moet worden. Dit levert een traceability op

Dat Layer Diagrammen niet alleen voor communicatie, maar ook voor verificatie te gebruiken zijn, maakt ze erg krachtig

tussen de broncode en de workitems. Stel dat we nu workitems aanmaken die bijvoorbeeld een Use Case weergeven en een relatie hebben met de use cases in het diagram, dan is daarmee de code herleidbaar te maken op een specifieke use case.

In Team Foundation Server 2010 wordt een nieuwe feature geïntroduceerd onder de naam hiërarchische workitems. Met behulp van hiërarchische workitems is het veel eenvoudiger mogelijk geworden de requirements van een systeem weer te geven met workitems die ook een onderlinge relatie hebben. Denk daarbij aan Use Case en Feature, waarbij een feature wordt geïmplementeerd door meerdere use cases. Als je nu gestructureerd relaties aanbrengt tussen workitems en deze workitems koppelt aan diagrammen en code, wordt het mogelijk bijvoorbeeld een use case te koppelen aan onder andere non functional requirements, aan features, aan broncode, etcetera. Indien er dan een verzoek tot wijziging binnenkomt, kan door een analist worden bepaald op welk feature of requirement dit verzoek betrekking heeft en is met een simpele workitem query vast te stellen op welke broncode de wijziging impact kan hebben.

In Team Architect is het mogelijk om ieder willekeurig modelelement (dat kan zijn een class, Use Case, Activity, etcetera) te koppelen aan een workitem. Op die manier is het dus ook mogelijk geworden om op basis van workitem queries te achterhalen hoe de relatie tussen verschillende elementen terug te leiden zijn op vastgelegde workitems als Use case, Test Case, Story Board, Feature of elk ander workitem welke onderdeel is van de gebruikte proces template.

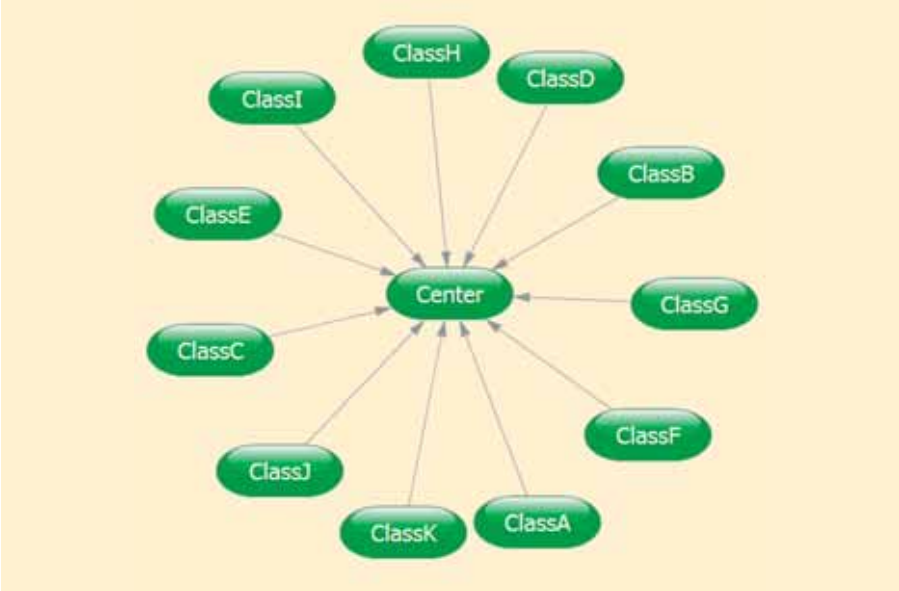
Layer diagram

Naast de ondersteuning voor UML-diagrammen heeft Microsoft een nieuw diagramtype ontwikkeld onder de naam 'Layer Diagram'. Een Layer Diagram is het best te vergelijken met een diagram dat iedereen op het bord tekent als de architectuur van een systeem. Dit is bijvoorbeeld een klassiek drie lagenmodel of een model waarbij men een service bus hanteert en services daaromheen. Figuur 2 geeft een Layer Diagram weer voor een simpele '3-tier applicatie' met scheiding van presentatie, domein logica en data access. Verder is op alle lagen dezelfde vorm van logging van toepassing.

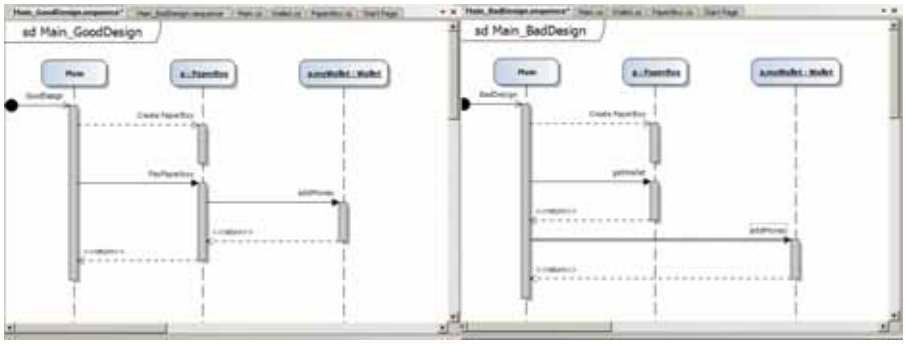
Je kunt je afvragen waarom Microsoft dit diagram introduceert. Het is immers geen standaarddiagram binnen de UML-specificatie. De reden van dit diagram is het feit dat we modellen gebruiken om te communiceren met elkaar. Microsoft heeft een aantal studies uitgevoerd om te achterhalen welke diagrammen veel worden gebruikt in de softwareindustrie. Daaruit bleek dat iedereen naast een aantal standaard UML-diagrammen altijd een diagram tekent met vakjes die een bepaald doel hebben in een betreffende architectuur. Of een vakje nu betekent dat het gaat om user interface of bedrijfslogica of iets anders, iedereen tekent ze om één andere reden of op een andere manier.
Een gemeenschappelijk element in al deze diagrammen is echter dat ieder team een diagram tekent, waar vlakken in staan die feitelijk een aantal layers in de architectuur weergeven. Het Layer Diagram is dus ontstaan uit het bewijs dat dit diagram overal terugkomt, maar geen weerslag kent in een van de standaard onderkende diagrammen in bijvoorbeeld UML. Microsoft is nu een stap verder gegaan. Naast het kunnen tekenen van het Layer Diagram kun je er veel meer doen. Het is namelijk mogelijk om een bepaald stuk code te associëren met een betreffend deel van het diagram. Deze associatie zorgt ervoor dat bepaalde code kan worden aangewezen als een onderdeel van een getekende layer in het diagram. Vervolgens is het dan mogelijk om de bestaande code te verifiëren met het getekende model. Als we terugkijken naar Figuur 2, dan is bijvoorbeeld te verifiëren dat de data access laag geen rechtstreekse communicatie heeft met de presentatielaag. Nu is dit wel een simpel voorbeeld, maar veel complexere regels zijn ook mogelijk.

Dit feature is volledig integreerbaar met de Team Foundation Server Build omgeving. Het is namelijk mogelijk om de software te verifiëren aan een opgesteld Layer Diagram tijdens een build. Vervolgens levert dit een set aan errors en warnings op die in de buildrapportages terugkomen. Op deze manier is het dus mogelijk geworden om de diagrammen naast communicatie tevens te gebruiken voor verificatie en dat is natuurlijk erg krachtig.

Ook is in het ontwerp van het Layer Diagram meegenomen dat Microsoft nooit alle mogelijke richtlijnen kan verzinnen die een architect aan zijn sofwareproduct wil



FIGUUR 3: DEATH START PATROON



FIGUUR 4: VISUALISATIE VAN GOED EN SLECHT DESIGN VAN CLASS INTERACTIE

opleggen. Daarom is dit mechanisme uitbreidbaar gemaakt en is het mogelijk zelf metadata toe te voegen aan het Layer Diagram. Een verifier kan controleren of de broncode voldoet aan de gestelde eisen, uitgedrukt in de metadata.

Code first en models

Iedere ontwikkelaar wordt in zijn dagelijks werk regelmatig geconfronteerd met code die niet zelf is gemaakt of ontworpen. Veelal richt men zich tot je met de vraag of je even een aanpassing wilt maken op de bestaande applicatie. Hoe ga je nu te werk? Hoe krijg je inzicht in de structuur van de applicatie? Hoe zorg je ervoor dat helder is, voordat je een aanpassing maakt, welk deel van de applicatie daarvan afhankelijk is in het kader van risico's van de change?

Een hulpmiddel voor deze problematiek is terug te vinden in Visual Studio 2010 in de vorm van de zogenaamde Architecture Explorer. De Architecture Explorer biedt de mogelijkheid om op verschillende manieren de bestaande applicatie te visualiseren door middel van onder andere een Directed Graph Diagram (DGML). Dit diagramtype maakt het mogelijk voor een bestaande applicatie afhankelijkheden te visualiseren tussen onder andere namespaces, classes en assemblies. Deze afhankelijkheden geven al vrij snel inzicht in de basisstructuur van een applicatie. Daarnaast kan het perfect inzicht geven in een aantal standaard architectuurpatronen, die bekend zijn om hun onderhoudbaarheidsproblemen. Denk daarbij aan een soort Uberclass in het product, waar vervolgens bijna alle andere classes een dependency mee blijken te hebben. Aardig is dat dit soort Architectuur (anti)Patterns een bepaalde visuele structuur blijken te hebben in de Directed Graph Models. In voorgaand voorbeeld zal een 'Uberclass' waar veel anderen een dependency mee hebben zich presenteren als een 'death start' patroon, waarbij de 'death start' het middelpunt blijkt te zijn van alle afhankelijkheidspijlen die op hem zijn gericht. Figuur 3 geeft een voorbeeld van zo'n 'death start' patroon. Leuk hieraan is dat veel voorkomende patronen nu direct visueel herkenbaar worden en makkelijker kunnen worden gevonden in een bestaand systeem. Vanuit de DGML-diagrammen is het ook zeer eenvoudig door te klikken naar de onderliggende code. In de code is het vervolgens ook weer mogelijk daar direct een UML-sequence diagram te genereren. Ook deze sequence diagrammen geven vrij snel een visualisatie van patronen die we graag willen zien (een V- Model) of een patroon dat duidt op een ontwerp waarin bijvoorbeeld de wetten van Demeter niet zo best zijn toegepast. Dit is te zien in een patroon waarbij de initiërende class communiceert met alle classes op het diagram en dit niet doet via classes die ertussen liggen. Het wordt dus mogelijk door middel van modellen relatief eenvoudig op basis van visualisatie een goed beeld te geven hoe een systeem is opgebouwd. In Figuur 4 is een voorbeeld te zien van een sequence diagram waarbij de wetten van Demeter wel en niet goed zijn toegepast.

Test driven design en modelling

De opties in Team Architect voor deze modellen zijn bijzonder nuttig bij het schrijven van unit testen en het vroeg detecteren van designfouten in het onderhanden systeem. Stel je voor dat je volgens de test driven aanpak eerst een unit test class schrijft, vervolgens de implementatie, en je draait dan de test. Deze faalt, maar waarom? Naast dat je natuurlijk zelf door de code heen kan lopen, is het wel erg handig dat je gewoon op dat moment van de code een Sequence Diagram kunt genereren en op die manier kunt visualiseren wat de test allemaal aanroept. Daaruit kun je tevens direct het design visualiseren en kijken of het voldoet aan de figuren, die horen bij goed design. Je ziet ook snel of een anti-pattern in je code is geslopen.
De manier waarop Team Architect de modellen voor je genereert, maakt dat het snel uitnodigt de diagrammen te gebruiken als alternatieve view op de code die je onder handen hebt. Hiermee heb je dus naast designtools eigenlijk ook tools in handen gekregen die meer inzicht verschaffen in bestaande code. Daarmee worden diagrammen als alternatief gebruikt om door je code te navigeren.

Conclusie

Met behulp van de in Team Architect 2010 geïntroduceerde tools wordt het veel makkelijker gemaakt modellen te gebruiken voor de communicatie in het team. Doordat deze modellen zowel vooraf als achteraf kunnen worden gemaakt, is het niet alleen mogelijk om aan te geven wat je wilt bouwen, maar ook te visualiseren wat daadwerkelijk is gerealiseerd. Erg handig is dat je zelfs modellen kunt gaan inzetten waarbij je formaliseert hoe je systeem vanuit architectuurperspectief in elkaar moet zitten en dat je dit gedurende de build kunt verifiëren.

Links

Community Technical Preview van Visual Studio 2010: (https://connect.microsoft.com/VisualStudio/content/content.aspx?ContentID=9790)

4 + 1 Views of software architecture, Philippe Kruchten: http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf)

De wet van Demeter is ook bekend onder de naam "Principle of Least Knowledge". Simpel gezegd geeft deze wet aan dat het gewenst is dat een class enkel en alleen kennis heeft van zijn directe buren en niet van 'de buren van de buren'.

Marcel de Vries, is Technology Manager Microsoft bij InfoSupport

BEN JIJ ONZE NIEUWE TOPPER?

Gek eigenlijk. Dat je tegenwoordig al opvalt als je voor klanten je nek uitsteekt. Bij Giraffe IT is resultaatverplichting de gewoonste zaak van de wereld. En daar gaan we heel ver in. Dat klinkt heel stoer maar als je onze mensen ontmoet, begrijp je dat het kan. Giraffe IT is 100% gericht op Microsoft oplossingen. Wij gaan voor advies en realisatie van SharePoint, Unified Communications, BizTalk en .NET oplossingen. Ben jij die specialist die Het Nieuwe Werken kan vertalen in concrete totaaloplossingen voor klanten? Dan horen wij graag van je! **Giraffe, geen twijfel mogelijk!**

NIKKELSTRAAT 3H 2984 AM RIDDERKERK T: 0180-425700 F: 0180-423300 E: INFO@GIRAFFE.NL I: WWW.GIRAFFE.NL

GIRAFFE