VELDWIJK

Modulair verknipt

In elk tijdvak heb je op databasegebied één of twee ideeën die hoogst populair zijn, maar die bij nadere beschouwing schadelijk zijn als leidraad voor database- en systeemontwerp. Medio jaren tachtig was er bijvoorbeeld een breed gedragen idee dat relationele databases leuk waren voor universiteitspractica, maar in de praktijk nooit zouden werken bij bestanden met meer dan een paar duizend records en meer dan een handvol gebruikers. Het liep anders. Een paar jaar later was relationeel een *hype* en circuleerden er opeens allerlei zweverige ideeën over relationele concepten. Zo werden er aan relationele sleutels haast magische eigenschappen toegekend. De waarde van een primaire sleutel was bijvoorbeeld een representatie van de identiteit van een ding in de reële wereld. En daaruit volgde dan dat je de waarde van een primaire sleutel niet mocht wijzigen, omdat een ding in de reële wereld ook niet zomaar een andere identiteit kan aannemen. Een paar jaar later vond niemand een *cascaded update* nog iets geks. Daarna kwam een tijd dat zeer velen vonden dat je een relationele database-toepassing niet moest ontwerpen in watervalstijl maar moest 'uitprototypen', daarna 'RADden' en tenslotte 'JADden'. In die tijd ontwikkelde systemen hebben de *legacy* status vaak niet bereikt. Nu besteden we zelfs van alles uit naar India en vinden we het maken van een ontwerp vooraf weer heel gewoon. Daarna was relationeel passé en moesten we objectgeoriënteerd gaan denken – natuurlijk wel in RDBMS-omgevingen. Het gevolg was – en is – een berg toepassingen met voor elke tabel een 'onzichtbare' *object identifier*. In de echte wereld heb je natuurlijk wel te maken met identificerende eigenschappen en dus krijgen veel tabellen minimaal twee sleutels. Meer programmeerwerk en meer database overhead, waar niets anders tegenover staat dan het fijne gevoel objectgeoriënteerd bezig te zijn. Daarna kwam een periode waarin we erkenden dat grote, geïntegreerde systemen vaak niet te bouwen waren door gewone stervelingen en we zouden moeten kiezen voor een potpourri van oude en nieuwe systemen, deels ontsloten door een datawarehouse. En omdat een *read-only* warehouse-systeem nou eenmaal iets anders is dan een compleet *read/update* systeem moest het ontwerp gebaseerd worden op heel andere, stervormige uitgangspunten. Geeft niets dat de raadpleegwereld niet in één ster is te vangen. We maken gewoon meer sterretjes – data*marts*. De versplintering die van dit denken het gevolg is, is ondertussen op zijn natuurlijke grenzen gestoten. Hoe verder je kijkt in het verleden, des te eenvoudiger is het om te zien welke denkfouten er werden gemaakt. Omgekeerd is het des te moeilijker om de meer recente dwaalwegen te doorgronden of ze zelfs maar te herkennen. Uiteindelijk is het per definitie onmogelijk dat meer dan een enkeling zijn tijd vooruit is. En toch kunnen we volgens mij leren van de *march of folly* die we met z'n allen in het verleden hebben gemaakt door te kijken naar het *patroon* in de denkfouten. Als ik dat patroon probeer te doorgronden dan zie ik twee constanten. Een constante is verwarring tussen structurele oplossingen en *patches*. Je ziet dat heel mooi bij datawarehouses. Er zijn legio omstandigheden die het ontwikkelen van een warehouse rechtvaardigen, maar het blijven patches. Nodeloos complexe datalogistiek en dito versplintering van systemen ontstaan pas wanneer we geloven dat de patch gelijk is aan de oplossing. Omdat de wereld niet ideaal is zullen er altijd en overal patches nodig blijven. Grootschalige prototyping is bijvoorbeeld meestal een slecht idee, maar er zijn wel degelijk situaties waarin het nog slechter is om eerst een onbegrepen ontwerp te maken.
Het ging pas fout toen we geloofden dat we nooit meer een ontwerp hoefden te maken. Achteraf bezien ongelooflijk, maar een logisch gevolg van het opwaarderen van *patch*-oplossingen – de eerste contante in onze geschiedenis van verkeerde gedachten.
De tweede constante is ideologische verblinding, oftewel het niet toetsen van courante ideeën op nuttigheid. Wat is het concrete nut van redundante abstracte *object identifiers*? Niemand die het kan uitleggen terwijl de extra programmeerinspanningen wel goed te berekenen zijn. Wat is het nut om een systeemgebruiker te verbieden om een artikelnummer te wijzigen? Oke, het scheelt programmeerwerk doordat we niet in *cascade updates* hoeven te voorzien en geen historie van sleutels hoeven te ondersteunen, maar we zijn er niet voor onszelf maar voor de gebruikers, toch? En zo kunnen we doorgaan. Met deze twee constanten als houvast kun je als ICT'er of als leidinggevende een eind komen in het jezelf beschermen tegen het nemen van verkeerde beslissingen. Wie het onderscheid kan maken tussen een *patch* en een structurele oplossing, voorkomt in elk geval dat er gezonde benen worden afgezet en vervangen door kekke protheses. Wie steeds vraagt naar het nut van een idee en de noodzaak van een voorgestelde oplossing, voorkomt nutteloos werk en schadelijke systeem*overhead*. Er blijven dan nog meer dan genoeg echte uitdagingen over om onze tanden in te zetten.

René Veldwijk
Dr. R.J. Veldwijk (rene.veldwijk@faapartners.com) is partner bij FAA Partners, een onderdeel van de Ockham Groep.