GEPUBLICEERD IN: INFORMATIE, 40(4), 28-33, APRIL, 1998.

Flexibiliteit en informatiemodellering; schieten op een bewegend doelwit.

H.A. Proper
Origin/Architecture Modelling
Postbus 12593
1100 AN Amsterdam
E.Proper@acm.org

1 Evolutie is een constante

Heden ten dage bevinden de meeste organisaties zich in een dynamische omgeving. Deze dynamiek dwingt een hoge mate van flexibiliteit van de organisaties af: “evolve or die”. Het wegvallen van protectionisme, de deregulering van het internationale handelsverkeer, de invoering van nieuwe technologiëen, de privatisering van staatsbedrijven en de invoering van de Euro zijn allemaal voorbeelden van aspecten die deze dynamiek teweeg brengen.

Om hun overlevingskansen in een dynamische omgeving te vergroten, zullen organisaties zich snel moeten kunnen aanpassen [1-4]. Dit zal zeer waarschijnlijk resulteren in een verandering van de informatiebehoefte, hetgeen weer tot gevolg heeft dat de in gebruik zijnde informatiesystemen aangepast moeten worden. Als een organisatie dus een hoge mate van flexibiliteit wil bereiken, dan zullen de onderliggende informatiesystemen eveneens flexibel moeten zijn. Met flexibiliteit bedoelen we hier overigens de mate waarin de kosten (in termen van geld, tijd, mensen, etc.) om een aanpassing uit te voeren beperkt blijven.

In de context van informatiemodelleren is het daarom relevant eens stil te staan bij de flexibiliteit van informatiemodellen. Immers, informatiemodellen worden gebruikt om de structuur van informatiesystemen vast te leggen. Wanneer we te maken hebben met een hoge mate van dynamiek, dan zal dit zo zijn effecten hebben op deze onderliggende informatiemodellen. Ontwerpers van informatiemodellen ontwerpen voor een deel hun eigen onderhoudslast, getuige de vele conversie-, jaar 2000- en Euro-problemen. Een informatiemodel dat vandaag van toepassing is, kan morgen alweer verouderd zijn. Het is als schieten op een bewegend doelwit. In dit artikel gaan we kort in op deze problematiek. Wat zijn de te verwachten knelpunten? Hoe moeten we daar in de praktijk nu mee omgaan? Dit artikel verkent een aantal van deze knelpunten en mogelijke oplossingsrichtingen.

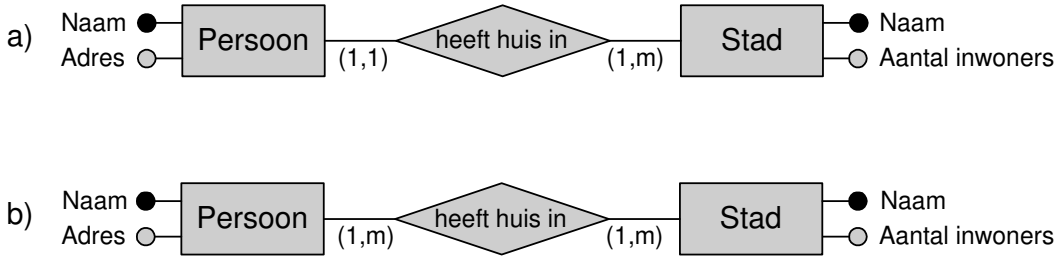
2 De knelpunten

Een informatiemodel zal zelf als model per definitie flexibel zijn. Het kost immers relatief weinig moeite om in het eigenlijke informatiemodel een verandering door te voeren. Het veranderen van een bouwtekening (model) van een huis zal ook *relatief* weinig moeite kosten. Echter, het informatiemodel vormt de basis voor het datamodel volgens welke de onderliggende datastructuren van het informatiesysteem worden ingericht, en het is natuurlijk aanzienlijk moeilijker om deze onderliggende structuren ongestraft te wijzigen

In dit artikel zullen we het begrip *datamodel* gebruiken als we refereren naar de onderliggende tabellenstructuur en consistentieregels zoals die door een DBMS gebruikt zullen worden. Wat terminologie betreft, zullen we de term *informatiemodel* gebruiken als we refereren naar het conceptuele model van de informatie die in het onderhavige informatiesysteem opgeslagen dient te worden; dus een representatie zonder implementatie-gebonden details.

De impact van veranderingen in het informatiemodel zal met name aan het licht komen wanneer we verder kijken dan de informatiemodellen alleen en de applicaties die gebruik maken van de opgeslagen gegevens in ogenschouw nemen. Veronderstellende dat er geen sprake is van een vervuilde database,

dan zal het aanpassen van een datamodel en migreren van de onderliggende gegevens nog wel een redelijk snel te volbrengen taak zijn. Echter, het aanpassen van de applicaties die bovenop het informatiesysteem gebouwd zijn, zal voor meer hoofdbrekens zorgen. Hoewel we in dit artikel informatiemodellering als uitgangspunt gebruiken, zullen we zeker verder kijken dan onze informatieneus lang is, en de applicatiewereld zeer zeker ook in ogenschouw nemen.



Figuur 1: Veranderen van constraint

We bespreken nu kort twee simpele voorbeelden die de lezer wat meer gevoel voor de problematiek zouden moeten geven. Als eerste bekijken we wat er gebeurt als we een constraint veranderen. Beschouw de twee ER-schema's uit Figuur 1[1]. Beide schema's modelleren een domein waarin het relevant is te weten in welke steden mensen een huis bezitten. In het bovenste schema mogen mensen slechts in één stad huizen bezitten, terwijl deze eis in het onderste schema is komen te vervallen. Hoewel dit wellicht een kunstmatig voorbeeld is, is het een prima illustratie van de problematiek. Schema a) zou als volgt in een (relationeel) datamodel vertaald kunnen worden:

HuizenBezit:
[PersoonsNaam, Adres, StadsNaam]

Stad:
[Naam, AantalInwoners]

Schema b) kan door het veranderen van de cardinaliteitsconstraint niet meer op deze manier gerepresenteerd worden daar er anders redundantie zou kunnen gaan optreden omdat het adres van een persoon voor elke stad waar deze persoon huist, zou worden opgeslagen. Dit laatste schema zou daarom aanleiding geven tot het datamodel bestaande uit de volgende drie tabellen:

Persoon:
[Naam, Adres]

HuizenBezitter:
[PersoonsNaam, StadsNaam]

Stad:
[Naam, AantalInwoners]

Alle aspecten van een applicatie of informatiesysteem die afhankelijk zijn van deze tabelstructuur zullen bij een dergelijke verandering eveneens aangepast moeten worden. Veel, zo niet de meeste, informatiesystemen worden op basis van relationele databasemanagementsystemen (DBMS'en) gebouwd. Dit betekent dat men doorgaans van een taal als SQL gebruik zal maken om de applicaties met de database te laten communiceren, en uitdrukking te geven aan eventuele complexe constraints. Bij een verandering van het onderliggende datamodel zullen deze aansluitingen, en mogelijkerwijze grote delen van de applicatie, ook mee moeten veranderen. Deze laatste veranderingen zijn nu juist de duurste.

Een voorbeeld waarbij niet alleen een constraint verandert, maar ook de daadwerkelijke informatiestructuur, is te vinden in Figuur 2. In schema a) valt te zien dat we te maken hebben met een bedrijf waarin mensen werken voor afdelingen. Doordat het bedrijf groeit zal men er op zeker moment

[1] De in dit artikel gebruikte ER-variant is die van Batini, Ceri en Navathe [5].

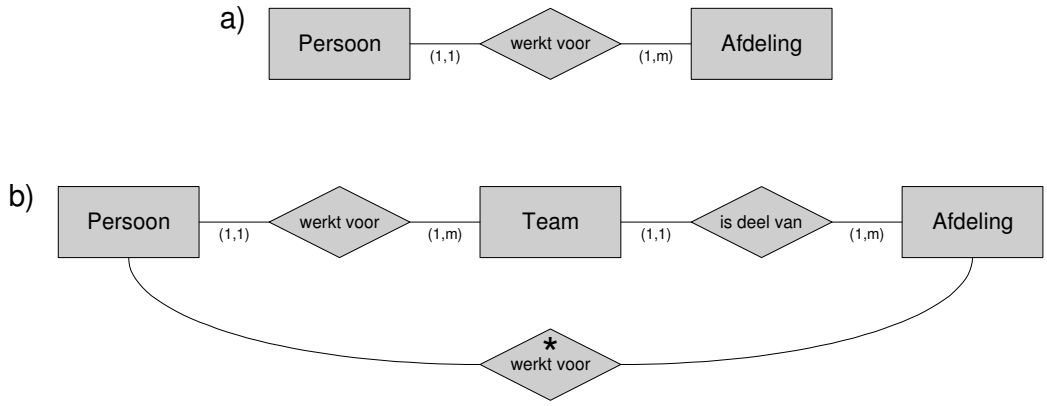
toe kunnen overgaan afdelingen onder te verdelen in teams. Dit betekent dat vanaf dat moment afdelingen bestaan uit teams, en dat mensen voor teams werken. Dit staat weergegeven in schema b). Er kan natuurlijk nog wel een afleidbare relatie (aangegeven door een *) tussen werknemer en afdeling ingevoerd worden die aangeeft voor welke (unieke) afdeling een werknemer werkt.

Ook deze verandering zal leiden tot een wijziging van het datamodel zoals dit gebruikt wordt door het DBMS. Overigens zal bovenstaand voorbeeld niet alleen vervelende consequenties voor de applicaties hebben als gebruik gemaakt wordt van een relationeel DBMS. Ook bij een object-georïenteerd DBMS zullen de onderliggende objectstructuren veranderen.

Hoewel ze bij veranderingen van legacy systemen een belangrijke rol spelen, zullen we in dit artikel niet ingaan op de problematiek van reverse engineering, of het opschonen van vervuilde data. Dit zijn problemen die op zich los gezien moeten worden van flexibiliteits problemen. Uiteraard is het zo dat het aanpassen van een vervuilde database waarvan het eigenlijke informatiemodel niet eens meer voorhanden is een nog veel groter probleem is. Echter, het is een probleem dat uitéén valt in drie deelproblemen: reverse engineering van het data model, het opschonen van data, en het aanpassen aan de nieuwe situatie (forward engineering).

3 Oplossingsrichtingen

We zullen nu, kijkende vanuit een informatiemodelleringsperspectief, een aantal mogelijke strategiëen bekijken die erop gericht zijn beter om te kunnen gaan met de flexibiliteit van een organisatie.

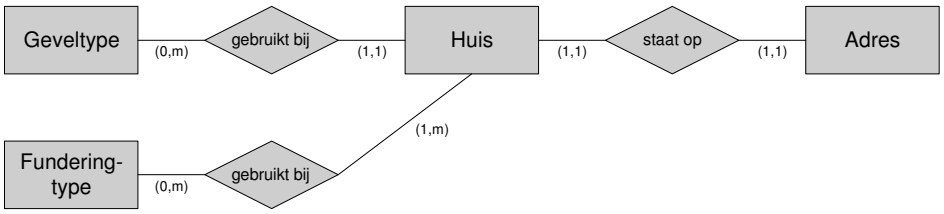


Figuur 2: Structurele verandering

We beschouwen kort drie klassen van oplossingsrichtingen. Ten eerste kunnen we proberen latere veranderingen te *voorkomen* door, voor zover mogelijk, vooruit te kijken naar eventuele te verwachten veranderingen. Ten tweede kunnen we proberen de effecten van eventuele veranderingen zoveel mogelijk te *dempen.* Laatstelijk zullen we kijken naar technische hulpmiddelen die het maken van eventuele veranderingen *vergemakkelijken.*

3.1 Modelleren is vooruitzien

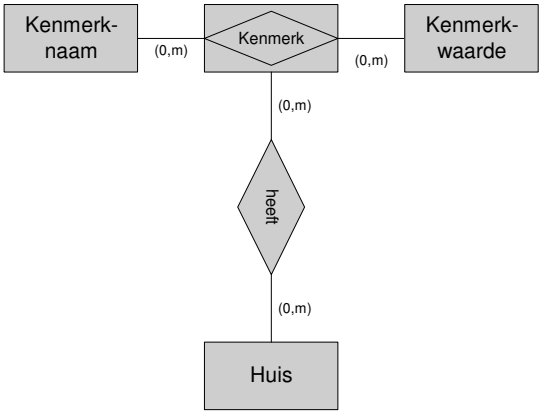
Soms zal men in staat zijn om al tijdens het informatiemodelleren in te spelen op te verwachten veranderingen. Deze aanpak is natuurlijk niet voldoende wanneer het gaat om onvoorziene of grote veranderingen, maar kan toch in veel gevallen op een eenvoudige wijze al veel ellende voorkomen. Neem bijvoorbeeld de situatie uit Figuur 1. Als van tevoren al duidelijk is dat er een redelijke waarschijnlijkheid is dat de cardinaliteitsconstraint op (korte) termijn verruimd gaat worden, is het beter meteen te opteren voor het tweede relationele schema, en initieel de strengere constraint gewoon expliciet af te dwingen. Bijvoorbeeld door middel van een SQL-trigger. Het ‘van te voren’ maken van veranderingen kan dus gebruikt worden om de ‘elasticiteit’ van een model te vergroten.



Figuur 3: Huizendatabase

Een ander voorbeeld is te vinden in Figuur 3. In deze figuur staat het informatiemodel van een fictieve huizendatabase weergegeven. Voor elk huis worden een aantal kenmerken opgeslagen. In dit geval worden het type van de gevel (klok, trap, …), en het type van de fundering opgeslagen. Het zal niet moeilijk zijn om voor te stellen dat er in de loop der tijd steeds meer kenmerken nodig zullen blijken te zijn. Dit zou betekenen dat er voor elk nieuw kenmerk, zoals 'architectuurstijl', een wijziging van het informatiemodel zou moeten plaatsvinden.

Voor een domein waarin het te verwachten is dat er voor een bepaald objecttype steeds meer kenmerken opgeslagen zullen worden, zou een generiekere oplossing zoals weergegeven in Figuur 4 wellicht bruikbaarder zijn. In plaats van voor elk kenmerk een speciale relatie in te voeren, worden deze nu als het ware 'op één hoop geveegd'. Dit schema biedt de ruimte om wanneer nodig dynamisch



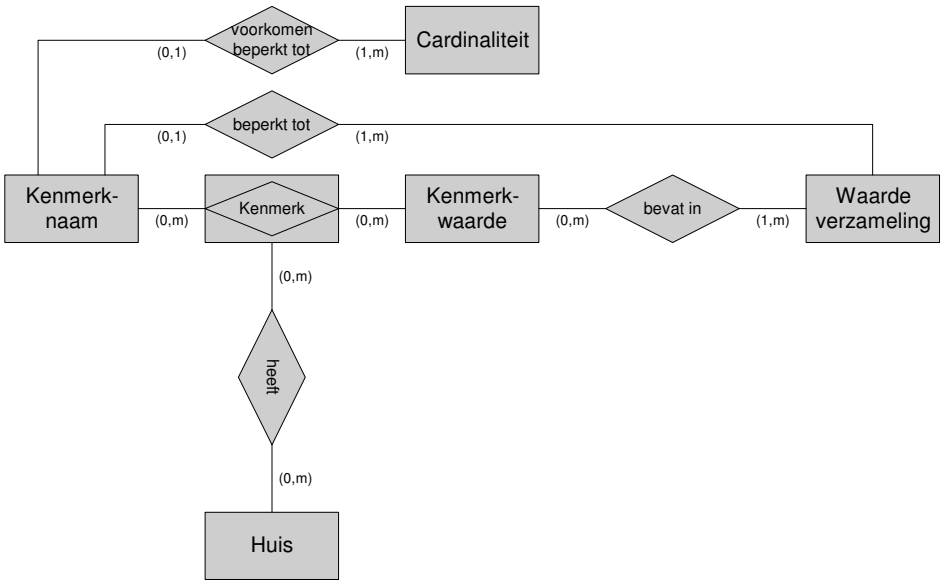
Figuur 4: Generieke huizendatabase

nieuwe kenmerktypen toe te voegen. Nadeel is natuurlijk wel dat de eventuele constraints die specifiek zijn voor kenmerktypen, bijvoorbeeld "elk huis moet een uniek geveltype hebben", expliciet in de applicaties afgedwongen moeten worden. Als er dus een nieuw kenmerktype wordt ingevoerd moeten dergelijke constraints wel in de applicaties doorgevoerd worden.

Om dit laatste probleem ook zoveel mogelijk te vermijden, kunnen we ervoor kiezen om voor de constraints een zelfde constructie toe te passen als de kenmerken zelf. Simpele constraints, zoals cardinaliteiten, beperkingen van waardendomeinen[2], etc., kunnen eveneens expliciet in de database zelf opgeslagen worden. Dit resulteert in de situatie zoals weergegeven in Figuur 5. Als er een nieuw kenmerktype ingevoerd wordt, dan kunnen de eventuele cardinaliteitsregels en beperkingen van het waardedomein eveneens gespecificeerd worden in de database. De semantiek van deze constraints moet natuurlijk wel expliciet afgedwongen worden.

Het zal duidelijk zijn dat dit soort 'trucs' maar een beperkte reikwijdte hebben. Structurele veranderingen zoals die in Figuur 2 zijn vaak nauwelijks van tevoren te voorspellen. Daarnaast vereist het van tevoren inspelen op te verwachten veranderingen ook de aanwezigheid van een langere termijnvisie op het functioneren van de organisatie. In welke richting denkt de organisatie zich te gaan ontwikkelen? Hoe betrouwbaarder en preciezer de antwoorden op deze vragen zijn, des te makkelijker wordt het om van tevoren op veranderingen in te kunnen spelen.

[2] Het kenmerktype 'geveltype' kan bijvoorbeeld worden beperkt tot de waarden 'klok' en 'trap'.



Figuur 5: Constraints voor huizendatabase

3.2 Wat niet weet, dat niet deert

Een in de software engineering al sinds jaar en dag bekend principe is het gebruik van abstractie en encapsulatie. Er zijn ten minste twee goede redenen te noemen om van deze mechanismen gebruik te maken. Ten eerste wordt het zo mogelijk gemaakt om de impact van veranderingen te localiseren, omdat hogere software-lagen niets `weten' van de details van de lager liggende software lagen. Ten tweede zijn deze mechanismen bijzonder nuttig om complexe stukken software voor mensen beter behapbaar te maken, bijvoorbeeld door middel van een `top down programming' stijl.

In de wereld van informatiemodellering zijn deze mechanismen voor dit laatste doel reeds eerder toegepast. In [6 , 7] bijvoorbeeld worden mechanismen besproken om complexe informatiemodellen beter behapbaar te maken.

Wanneer we praten over flexibiliteit, dan zullen we de abstractie- en encapsulatiemechanismen met name in willen zetten om de impact van veranderingen op applicaties te verminderen. Bijvoorbeeld in [8] staat beschreven hoe men informatiesystemen kan zien als een verzameling van business objects, waarbij deze business objects opgedeeld worden over een aantal lagen van abstractie.

Encapsulatie is met name één van de voordelen die een object georiënteerde aanpak met zich meebrengt. Het is natuurlijk wel zo dat encapsulatie sowieso als belangrijk kenmerk van een goede programmeerstijl genoemd mag worden, ook als men van een niet object-georiënteerde taal gebruik maakt.

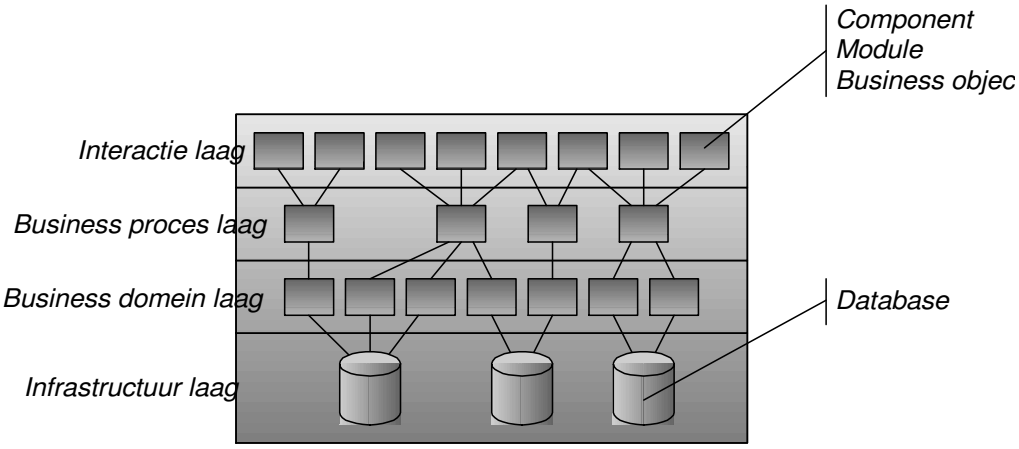
In Figuur 6 is het gebruik van abstractielagen geïllustreerd. We hebben hier onderscheid gemaakt in 4 abstractielagen. In de infrastructuurlaag komen we de onderliggende databases, en eventueel ook zaken als middleware, etc., tegen. In de implementatie van deze laag zullen we de afhankelijkheden van de eigenlijke gekozen databasestructuren kunnen vinden. In de hogere lagen is het de bedoeling dat we deze afhankelijkheden niet meer tegenkomen.

De business-domeinlaag is er op gericht om een abstractielaag te creëren welke correspondeert met het informatiemodel. Daar bovenop komt dan een business-proceslaag, gericht op de realisatie van de geautomatiseerde bedrijfsprocessen. Laatstelijk is er de interactie-laag. Deze laag richt zich met name op de gebruikersinteractie. Het is uiteraard mogelijk deze lagenstructuur nog verder te verfijnen.

De rechthoekige elementen uit Figuur 6 zullen, afhankelijk van het gekozen paradigma, overeen komen met software-modules, of (business) objecten in een object-georiënteerde stijl. Als een neutrale naam is daarom hier gekozen voor *component*.

Wanneer men een informatiesysteem en bijbehorende applicaties bouwt, dan zal de flexibiliteit van het geheel sterk bepaald worden door de keuzen die men heeft gemaakt ten aanzien van de inrichting van de componenten. Dit is een discussie die niet nieuw is. De vraag hoe software-modules in te richten is reeds bekend uit de software engineering.

Binnen Origin bestaat er een methode voor applicatieontwikkeling onder architectuur, die onder andere bovenstaande vraag probeert te beantwoorden. Een belangrijke bron van informatie voor dit



Figuur 6: Meerlagen architectuur

inrichtingsproces wordt wederom gevormd door een langere termijn visie op de ontwikkeling van de organisatie. Inmiddels is er met deze methode de nodige praktijk ervaring opgebouwd.

3.3 Conceptuele informatie manipulatie talen

Het idee van een conceptuele informatie manipulatie (CIM) taal is dat queries, updates, constraints, etc., geformuleerd zouden moeten kunnen worden in termen van het informatiemodel. Dus niet het datamodel zoals dit door de database gebruikt wordt, maar het originele informatiemodel.

Er zijn natuurlijk voor de hand liggende analogiëen tussen het gebruik van abstractielagen zoals net besproken, en het gebruik van een CIM. Een CIM zal doorgaans leiden tot natuurlijkere formuleringen die makkelijker te begrijpen zijn dan traditionele programmacode of SQL-statements [9-11].

Wat we hier concreet bedoelen met een conceptuele informatie manipulatie (CIM) taal is een taal:

1. waarin alle voor een informatiesysteem relevante queries, constraints, functies en processen gespecificeerd kunnen worden,
2. waarin expressies onafhankelijk zijn van het gebruikte implementatie platform (vandaar conceptueel),

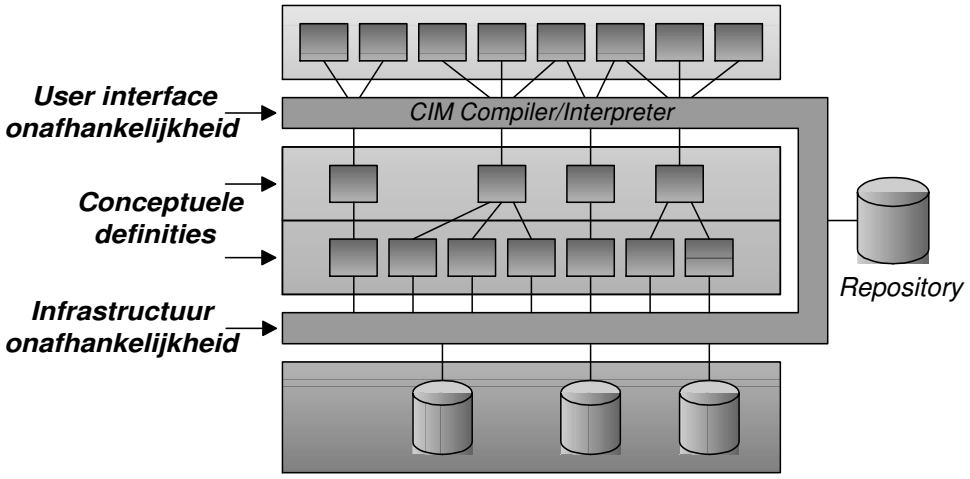
3. waarin expressies een minimale afhankelijkheid hebben van de structuur van het onderliggende informatiemodel.

Het zijn juist die laatste twee eisen die maken dat de definities van queries, constraints, functies en processen minder gevoelig zijn voor structurele veranderingen. Voorbeelden van dergelijke talen, zich beperkende tot queries en constraints, zijn RIDL [9], LISA-D [10], en ConQuer [11]. LISA-D werd sterk geïnspireerd door de ideëen achter RIDL, terwijl de taal ConQuer op zijn beurt weer geïnspireerd is door LISA-D. De ConQuer taal vormt inmiddels de basis van het commerciële product ActiveQuery [12]. Een voorbeeldformulering van een query in zo'n taal is:

LIST Persoon met een huis in de Stad: 'Arnhem'

Deze query is geformuleerd in termen van het eerste informatiemodel uit Figuur 1. Het zal duidelijk zijn dat deze query formulering niet veranderd hoeft te worden als de constraint verandert. Sterker nog, structurele veranderingen in het informatiemodel hoeven soms niet eens tot andere formuleringen te leiden.

Laten we ons als voorbeeld eens voorstellen dat een persoon niet langer via een enkelvoudige naam wordt geïdentificeerd, maar zoals we gewend zijn via een voornaam en achternaam combinatie. In termen van Figuur 1 betekent dit dat het Naam attribuut vervangen wordt door de combinatie VoorNaam en AchterNaam. Het ActiveQuery prmduct zal bij bovenstaande verandering automatisch



Figuur 7: CIM en abstractielagen

vanuit de query een andere SQL-query genereren die de verandering van de attributen zal weergeven.

Het omgaan met updates is uiteraard moeilijker in zo'n geval. Een update als:

ADD Persoon: 'Erik Proper' heeft een huis in de Stad: 'Arnhem'

zal bij de verandering van de attributen mee moeten veranderen. Het voordeel dat een CIM-taal dan met zich mee zou brengen is zijn natuurlijkheid. De taal staat immers dicht bij het informatiemodel en heeft daardoor een vrij natuurlijke gedaante.

De ideëen achter CIM talen zijn natuurlijk goed te combineren met de abstractielagen uit Figuur 6. In een complete CIM-taal zouden in ieder geval de business-domeinlaag en de business-proceslaag volledig op een conceptueel niveau gespecificeerd moeten kunnen worden. In de infrastructuurlaag en de interactielaag zullen we dan de details van de gebruikte implementatie platformen terug zien. Dit zou leiden tot de situatie uit Figuur 7, vergelijkbaar met de ANSI/SPARC 3-schema-architectuur.

Dit is natuurlijk een architectuur die zichzelf eerst in de praktijk zou moeten bewijzen. Sommige 4GL ontwikkelomgevingen, bijvoorbeeld InfoModeler [12], en USoft Developer [13] evolueren langzaam in deze richting. Maar ook ERP-paketten zoals Baan [14] lijken zich steeds meer in deze richting te bewegen in een streven naar meer business functionaliteit.

4 Tot besluit

In dit artikel hebben we kort gekeken naar flexibiliteit in de context van informatiemodellering. Het effect hiervan is pas goed merkbaar als we verder kijken naar informatiemodellering alleen. De applicaties die bovenop de ontworpen databases gebouwd worden, zullen eventuele veranderingen in het informatiemodel aan den lijve ondervinden.

We hebben kort aangegeven wat de te verwachten problemen zijn en daarna kort een aantal oplossingsrichtingen de revu laten passeren. Het omgaan met flexibiliteit van organisaties en vooral er voor zorgdragen dat applicaties organisatorische veranderingen niet in de weg zitten, is een boeiend probleemgebied dat voorlopig nog niet is uitgekristalliseerd.

Referenties

[1] P. W. G. Keen, *Shaping the Future - Business Design Through Information Technology*. Boston, Massachusetts: Harvard Business School Press, 1991.

[2] D. Tapscott and A. Caston, *Paradigm Shift - The New Promise of Information Technology*. New York, New York: McGraw-Hill, 1993.

[3] P. Magrassi, "Architecture: A Foundation for Durable Applications," Gartner Group - ADM, Strategic Analysis Report R-560-124, April 19 1995.

[4] H. A. Proper, "A Theory for Conceptual Modelling of Evolving Application Domains," . Nijmegen, The Netherlands: University of Nijmegen, 1994.

[5] C. Batini, S. Ceri, and S. B. Navathe, *Conceptual Database Design - An Entity-Relationship Approach*. Redwood City, California: Benjamin Cummings, 1992.

[6] L. J. Campbell, T. A. Halpin, and H. A. Proper, "Conceptual Schemas with Abstractions - Making flat conceptual schemas more comprehensible," *Data & Knowledge Engineering*, vol. 20, pp. 39-85, 1996.

[7] P. N. Creasy and H. A. Proper, "A Generic Model for 3-Dimensional Conceptual Modelling," *Data & Knowledge Engineering*, vol. 20, pp. 119-162, 1996.

[8] R. Prins, A. Blokdijk, and N. E. v. Oosterom, "Family traits in business objects and their applications," *IBM Systems Journal*, vol. 36, pp. 12-31, 1997.

[9] R. Meersman, *The RIDL Conceptual Language*. Brussels, Belgium: International Centre for Information Analysis Services, Control Data Belgium, Inc., 1982.

[10] A. H. M. ter Hofstede, H. A. Proper, and T. P. van der Weide, "Formal definition of a conceptual language for the description and manipulation of information models," *Information Systems*, vol. 18, pp. 489-523, 1993.

[11] A. C. Bloesch and T. A. Halpin, "ConQuer: a conceptual query language," in *Proceedings of the 15th International Conference on Conceptual Modeling (ER'96)*, vol. 1157, *Lecture Notes in Computer Science*, B. Thalheim, Ed., October ed. Cottbus, Germany: Springer-Vergal, 1996.

[12] InfoModelers, "http://www.InfoModelers.com," . Bellevue, WA.

[13] USoft, "http://www.USoft.com," . Brisbane, CA.

[14] Baan, "http://www.baan.com," . Barneveld, The Netherlands.