Generalisatie als specialiteit

Objectrelationeel, maar dan anders

Frido van Orden

In onze serie verkenningstochten langs de grenzen van het databasemanagementsysteem hebben we tot nu toe uitgebreid stilgestaan bij verschillende aspecten. Bij validatie en autorisatie kwamen we tot de conclusie dat de huidige generatie dbms'en hier een redelijke, maar ook niet meer dan dat, mate van ondersteuning bieden. Ook beschreven we hier de vertaling van regels in applicatiegedrag, een gebied dat dbms-bouwers laten liggen en toolboeren maar halfslachtig oppakken. Nu dan: super- en subtypering.

Het omgaan met super- en subtyping is, evenals omgaan met tijdaspecten, een van de klassieke onderwerpen bij leergangen over gegevensmodellering. Het is tevens een van de weinige punten waarop objectoriëntatie in de wereld van gegevensmodellering potten heeft kunnen breken. Hoewel OO-technieken, hulpmiddelen en programmeertalen in de software-engineering inmiddels net zo gewoon zijn als Cobol dat twintig jaar geleden was, is het in gegevensmodellen en dbms'en nog relationeel wat de klok slaat, niet in de laatste plaats omdat standaardisering van OO-en OR- (objectrelationele) features zo goed als ontbreekt. Bovendien blijken de beloften van OO in de dagelijkse praktijk ook via bekende relationele constructies en uitbreidingen te kunnen worden gerealiseerd (zij het minder elegant) en worden sommige features als niet interessant of zelfs onjuist beschouwd, zoals blijkt uit onderstaand overzicht:

OO feature	(Relationele) praktijkoplossing
Ondersteuning voor beeld, geluid enz.	Gebruik van BLOB-velden
Object-identiteit	Primary key, autonumber-velden
Eenheid van gegevens en logica (attributen en methoden)	In architecturen juist scheiding van gegevens en logica. Dbms alleen voor data-opslag
Encapsulatie van gegevens	Wordt niet als interessant ervaren
Ondersteuning van typehiërarchieën	Hard uitmodelleren
Goede integratie met OO-applicaties	Applicatieservers en mapping-tools

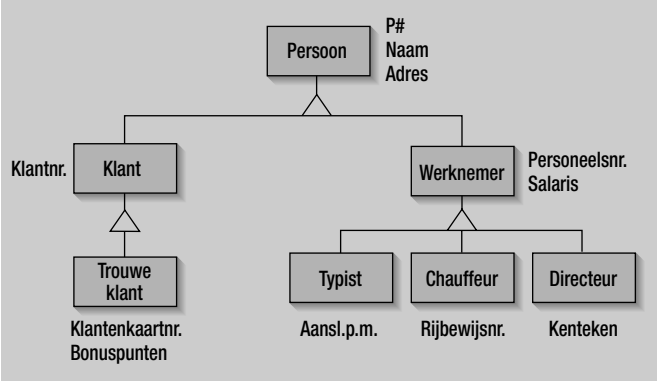
Het dbms voorbij (8)

In de artikelen die tot dusverre in deze serie verschenen, is ingegaan op twee belangrijke aspecten van gegevensverwerkende systemen: validatie en autorisatie. In het vorige nummer hebben we aandacht besteed aan presentatieaspecten die weliswaar niet meer tot het strikte domein van het dbms kunnen worden gerekend maar toch meer raakvlakken hebben met applicatiegedrag dan op het eerste gezicht wellicht lijkt. In dit nummer komen we toe aan weer een heel ander type dbms-functionaliteit: het omgaan met generalisatie- en specialisatie-aspecten, ofwel super- en subtypering.

Over het moeilijke huwelijk tussen relationeel en OO komen we later in deze artikelserie nog uitbreid terug. Voor dit artikel richten we ons zoals beloofd op super- en subtyping, of met een mooi woord ondersteuning van typehiërarchieën. Zoals hierboven al aangeduid, betekent dit in de dagelijkse praktijk: uitmodelleren. We zullen dit doen aan de hand van een casus.

CASUS

Een bedrijf houdt gegevens bij van zijn klanten en personeel. Van zowel klanten als personeel worden de naam- en adresgegevens geregistreerd. Klanten krijgen een klantnummer toegedeeld en kunnen, indien ze trouwe klant zijn, de beschikking krijgen over een speciale klantenkaart waarop bonuspunten kunnen worden gespaard. Het klantenkaartnummer en het gespaarde aantal punten worden geregistreerd. Van de werknemers van het bedrijf worden het personeelsnummer en het salaris bijgehouden. Daarnaast wordt voor typisten het aantal aanslagen per minuut geregistreerd, van chauffeurs het nummer van het rijbewijs en van directieleden het kenteken van de ter beschikking gestelde auto. Indien we dit uitmodelleren in een klassenhiërarchie ontstaat het model in figuur 1.

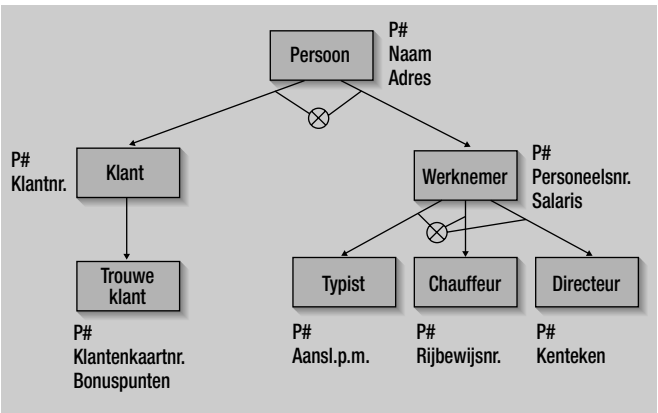


FIGUUR 1: KLASSENHIËRARCHIE

Hoe vertalen we dit model naar een relationele structuur? Hiervoor zijn twee basisvarianten mogelijk, die onderling weer kunnen worden gecombineerd. Variant 1 is het opslaan van alle gegevens uit de persoonshiërarchie in één relationele tabel. Hiermee ontstaat het relationele gegevensmodel van figuur 2.

FIGUUR 2: RELATIONELE IMPLEMENTATIE MET 1 TABEL PER HIËRARCHIE

In de enige tabel 'T_Persoon' (de logica achter deze naamgeving komt later aan bod) vinden we alle attributen terug van de basisklasse 'Persoon' en al zijn subklassen (Klant, Werknemer, Chauffeur enzovoort). Om vast te stellen of een record in de tabel 'Persoon' betrekking heeft op een Klant, een Chauffeur enzovoort is een kolom 'Type' toegevoegd waarin het type Persoon (Klant, Chauffeur enzovoort) wordt vastgelegd. Het type bepaalt welke kolommen mogen worden ingevuld en welke niet relevant zijn. Er moeten derhalve constraints worden opgenomen als 'indien Type = 'Typist' dan dienen de kolommen Klantnr, Klantenkaartnr, Bonuspunten, Rijbewijsnr en Kenteken leeg (null) te zijn'.



FIGUUR 3: RELATIONELE IMPLEMENTATIE MET 1 TABEL PER KLASSE

In variant 2 wordt per (sub)klasse een tabel gemaakt met als kolommen de attributen van de betreffende klasse. Daarnaast bevat elke tabel als primaire sleutel de kolom P#, die voor elke subklasse-tabel eveneens fungeert als verwijzende sleutel naar de tabel van de superklasse. Tot slot zijn uitsluitingsconstraints opgenomen die voorkomen dat bijvoorbeeld een Klant en een Werknemer naar dezelfde Persoon verwijzen.

Wat zijn nu de voor- en nadelen van deze beide implementatievarianten?

- Variant 1 is minder efficiënt qua ruimtebeslag, omdat elk record altijd alle attributen heeft van alle klassen die ergens in de hiërarchie voorkomen. Het effect hiervan is overigens beperkt indien het dbms gebruik maakt van variabele recordlengtes, bijvoorbeeld via het VARCHAR gegevenstype en grote objecten, met name BLOB's, niet in het record zelf worden opgeslagen.
- Variant 1 is efficiënter voor de meeste raadpleeg- en mutatieoperaties, omdat minder tabellen en indexen hoeven te worden geraadpleegd of bijgewerkt.
- Variant 1 is gevoeliger voor wijzigingen in het gegevensmodel, bijvoorbeeld toevoeging van een nieuw attribuut, omdat in dat geval de hele tabel dient te worden gereorganiseerd (dit is overigens slechts een kwestie van wachttijd, het dbms regelt alles via een simpel ALTER TABLE commando). Bovendien dienen telkens de constraints die aangeven voor welk type een kolom mag zijn ingevuld te worden aangepast.

ONDERSTEUNING IN DE DATABASE

Tot dusverre hebben we bij de implementatie van de Persoon hiërarchie in een relationele omgeving alleen gebruik gemaakt van standaard database-features. Ook hebben we geen enkele poging gedaan het resulterende relationele model zoveel mogelijk te laten lijken op het originele klassemodel. Anders gezegd: de keuze voor variant 1 of variant 2 heeft aanzienlijke invloed op de query's die op de database worden uitgevoerd. Enkele voorbeelden:

Het opvragen van de gegevens van de klant met klantnummer 12345 gaat in variant 1 met de query

```
SELECT P#, Naam, Adres
FROM T_Persoon
WHERE (Type = 'Klant' OR Type = 'Trouwe klant')
  AND Klantnr = 12345;
```

terwijl de query in variant 2 luidt:

```
SELECT T_Persoon.P#, T_Persoon.Naam,
T_Persoon.Adres
FROM T_Persoon, T_Klant
WHERE T_Klant.P# = T_Persoon.P#
  AND T_Klant.Klantnr = 12345;
```

Het toevoegen van een nieuwe klant gaat in variant 1 met

```
INSERT INTO T_Persoon(P#, Type, Naam, Adres,
Klantnr)
VALUES (123, 'Klant', 'De Groot', 'Plein 13', 134);
```

en in variant 2 met

```
INSERT INTO T_Persoon(P#, Naam, Adres)
VALUES (123, 'De Groot', 'Plein 13');

INSERT INTO T_Klant(P#, Klantnr)
VALUES (123, 134);
```

Dit heeft een aantal belangrijke nadelen:

- De keuze voor variant 1 of variant 2 is naderhand niet ongedaan te maken zonder onderhoudsinspanning aan alle reeds geschreven programmatuur.
- Bij gebruik van variant 2 dient bij het ontwikkelen van de programmatuur rekening te worden gehouden met de precieze klasse waarin een attribuut is gedefinieerd. 'Verhuizen' naar een lager niveau (bijvoorbeeld het vastleggen van adresinformatie op Klant- in plaats van Persoonniveau) leidt tot aanpassingen in de programmatuur, terwijl aan de definitie van de klasse 'Klant' *op zich* niets verandert.

Een manier om dit te voorkomen is het afschermen van de implementatie met views. Per klasse maken we een view. Zo ziet de view voor de klasse 'Klant' er in variant 1 aldus uit:

```
CREATE VIEW Klant AS
SELECT P#, Naam, Adres, Klantnr
FROM T_Persoon
WHERE (Type = 'Klant' OR Type = 'Trouwe klant')
```

terwijl in variant 2 die view er als volgt uitziet:

```
CREATE VIEW Klant AS
SELECT T_Persoon.P#, T_Persoon.Naam,
T_Persoon.Adres, T_Klant.Klantnr
FROM T_Persoon, T_Klant
WHERE T_Klant.P# = T_Persoon.P#
```

Moeilijker wordt het indien we ook de insert-, update- en delete-operaties willen afschermen. In vrijwel ieder rdbms is de hierboven gespecificeerde view niet updatable (in variant 1 omdat de waarde voor kolom 'Type' niet afleidbaar is, in variant 2 omdat het dbms geen kennis heeft van het feit dat tussen T_Persoon en T_Klant feitelijk een 1:1 relatie ligt en niet slechts een 1:n relatie in de vorm van de verwijzende sleutel vanuit T_Klant naar T_Persoon). Nu heeft Oracle ons enkele jaren geleden verblijd met het fenomeen 'instead of' triggers. Deze triggers worden gedefinieerd op views en zijn bedoeld om te kunnen muteren via views die niet standaard updatable zijn. Zo kunnen we de volgende triggers definiëren voor variant 1 respectievelijk variant 2:

```
CREATE TRIGGER I_Klant
INSTEAD OF INSERT ON Klant
BEGIN
        INSERT INTO T_Persoon(P#, Type, Naam, Adres,
Klantnr)
        VALUES (:new.P#, 'Klant', :new.Naam,
:new.Adres, :new.Klantnr);
END;
```

```
CREATE TRIGGER I_Klant
INSTEAD OF INSERT ON Klant
BEGIN
        INSERT INTO T_Persoon(P#, Naam, Adres)
        VALUES (:new.P#, :new.Naam, :new.Adres);
        INSERT INTO T_Klant(P#, Klantnr)
        VALUES (:new.P# :new.Klantnr);
END;
```

Zoals gezegd komt Oracle de eer toe deze functionaliteit als eerste te hebben geïmplementeerd. Microsoft biedt vergelijkbare functionaliteit sinds SQL Server 2000 en IBM heeft het aangekondigd voor versie 8 van DB/2.

EEN STAP VERDER

Met views en 'instead of' triggers kunnen we de generalisatiehiërarchie vergaand simuleren in de relationele omgeving. Nog mooier zou het zijn als we de implementatielogica (de SQL in de views en de triggers) niet zelf zouden hoeven bedenken, maar konden laten genereren door het dbms zelf. Elk zichzelf respecterend rdbms heeft zich de laatste jaren verrijkt met 'objectrelationele' features. Helaas blijkt dit in de praktijk vaak neer te komen op ondersteuning voor complexe gegevenstypen als afbeeldingen en geluid, de mogelijkheid tot het definiëren van (record)typen en het rechtstreeks implementeren van verwijzingen tussen objecten in het gegevensmodel zelf (REF gegevenstypen), in plaats van gebruik te maken van de standaard relationele constructie, de verwijzende sleutel. Het implementeren van het OO-kernprincipe *overerving* stond echter bij weinig leveranciers op de agenda. Informix, dat Illustra overnam en inmiddels zelf weer is overgenomen door IBM om op te gaan in de DB/2-familie was een van de weinigen, naast IBM zelf dat in bepaalde DB/2 UDB-versies de mogelijkheid biedt om tabellen op te nemen in een hiërarchie. De onderliggende implementatie hierbij is in essentie die van variant 1, waarbij alle gegevens in één tabel worden opgeslagen (de DB/2-term voor deze tabel is de 'hierarchy table'. (Een deel van) de hiërarchie uit onze casus kan dan worden aangemaakt met de volgende SQL-statements:

```
CREATE TYPE T_Persoon AS
(P# INTEGER,
 Naam VARCHAR(255),
 Adres VARCHAR(255)) MODE DB2SQL;

CREATE TYPE T_Klant UNDER T_Persoon AS ( Klantnr IN-
TEGER) MODE DB2SQL;

CREATE TABLE Persoon OF T_Persoon HIERARCHY H_Per-
soon REF IS P# USER GENERATED;

CREATE TABLE Klant OF T_Klant UNDER Persoon INHERIT
SELECT PRIVILEGES;
```

De tabellen Persoon en Klant kunnen hierna worden benaderd als waren het 'normale' tabellen. Het enige gekunstelde zit in de behandeling van de primary key van de tabellen in de hiërarchie. DB/2 dwingt af dat deze primary key uit één attribuut bestaat van het type REF(Klassenaam) en bij insert door de programmeur moet worden opgegeven. Het toevoegen van een klant gaat dan bijvoorbeeld via

```
INSERT INTO Klant(P#, Naam, Adres, Klantnr)
VALUES(Klant('001'), 'De Groot', 'Plein 13', 134)
```

Indien we de primary key uit meerdere attributen willen laten bestaan en/of een non-REF gegevenstype, dan moeten de attributen worden gedefinieerd in het type van de basistabel in de hiërarchie (in het voorbeeld T_Persoon) en moet op de basistabel zelf (in het voorbeeld Persoon) een unieke sleutel te worden gedefinieerd. Op zich niet onoverkomelijk, al ware het netter geweest als we de hiërarchie ook zonder gedwongen gebruik van een OID-kolom met REF gegevenstype zouden kunnen gebruiken.

BLIK VOORUIT

We hebben gezien hoe we generalisatiehiërarchieën kunnen implementeren in een relationele omgeving en welke features ons in het rdbms hiervoor ter beschikking staan. Niettemin blijft er een hoop programmeerwerk te doen, zoals het programmeren van views en triggers. In het volgende nummer zullen we een gegevensmodel ontwikkelen dat aansluit op de gegevensmodellen die eerder in deze artikelenserie zijn besproken en van waaruit we de database-objecten (views, triggers enzovoort) automatisch kunnen laten genereren. Ook kijken we naar de impact die generalisatie in het gegevensmodel heeft op het gedrag van de applicatie. ●

Ir. F.G.W. van Orden (fridoo@faapartners.com) is managing partner van FAA Partners BV