Object Modellering en Object Relational Mapping (3)

De implementatie van een klassediagram

Toon Loonen

In dit derde en laatste deel wordt de implementatie van het klassediagram in een fysieke database behandeld.

De meeste administratieve informatiesystemen slaan hun gegevens nog steeds op in een relationele database. Alhoewel OO databases in opkomst zijn en zich daaronder ook goede producten bevinden (bijvoorbeeld Caché, zie [Ref 9]) zijn de relationele databases nog steeds ver in de meerderheid.
Als voor het systeem een klassediagram is getekend om de structuur van gegevens vast te leggen, zal dit diagram dus vertaald moeten worden naar (c.q. gemapped moeten worden op) een fysiek relationeel model. In feite komt dit sterk overeen met het mappen van een niet (of gedeeltelijk) genormaliseerd conceptueel of logisch ER model naar een fysiek model. We hebben hierbij de alternatieven:

- Geheel handmatig een (logisch en) fysiek model opstellen;
- Het fysiek model laten genereren door de designtool waarin het klassediagram is vastgelegd, zoals Enterprise Architect [Ref 8];
- Het fysiek model laten genereren door een Object Relational mapping tool zoals Hibernate (zie [Ref 10]) of TopLink van Oracle [Ref 5].

Handmatige mapping

Als het klassediagram nog erg globaal gehouden is en er nog veel aanvullingen en normaliseringen nodig zijn, dan is het zeker aan te bevelen om eerst een logisch ER model op te stellen en dit vervolgens over te zetten naar een fysiek model. Als hiervoor een tool wordt gebruikt zoals Power Designer [Ref 7] dan kunnen hieruit grotendeels de (tabel)definities voor de database gegenereerd worden. De belangrijkste mapping-aspecten waarbij niet rechtstreeks een een-op-een vertaling mogelijk is tussen de klassen enerzijds en de entiteiten of tabellen anderzijds zijn:

- Repeterende groepen en meer-op-meer relaties;
- Definitie van primary en foreign keys;
- Definitie van de fysieke datatypes, vaak afhankelijk van de mogelijkheden en typedefinities van het te gebruiken RDBMS;
- Het definiëren van constraints (validaties) en de NULL/NOT NULL constraint [Ref 15];
- Het mappen van verschillende relatietypen (associatie, aggregaties en composities) op gewone een-op-meer relaties in het relationele model;
- Inheritance relaties;
- Afgeleide gegevens;
- Naamgeving.

Repeterende groepen

Als het klassediagram nog repeterende attributen of meer-op-meer relaties bevat, dan zullen die uitgenormaliseerd moeten worden. In ons voorbeelddiagram bijvoorbeeld:

- Voor de repeterende groep van telefoontype en telefoonnummer zal een nieuwe entiteit/tabel worden geïntroduceerd.
- Voor de meer-op-meer relatie tussen order en artikel uit afbeelding 3 in DB/M 6 (met de associatieklasse ORDERREGEL in afbeelding 4 in DB/M 6) zal een nieuwe tabel orderregel met twee bijbehorende relaties worden geïntroduceerd.

Primary en foreign keys

Voor elke klasse moet bekeken worden of er een attribuut (of combinatie van attributen) is dat geschikt is als primaire sleutel. De eisen aan een primaire sleutel zijn in het algemeen: geen lange tekststrings, geen gegevens die (eventueel) kunnen wijzigen, geen combinatie van veel attributen. Meestal is een betekenisloos volgnummer (al of niet met een checkdigit of 11-proef als bij een banknummer, zie [Ref 17]) een ideale primary key, zoals in ons voorbeeld het klantnummer of ordernummer. Een acceptabel alternatief is:

- Een eenvoudige alfanumerieke code (bijvoorbeeld landcode: NL, B, D) of het twaalfcijferig artikelnummer;
- Een samenstelling van Ordernummer plus orderregelnummer voor de orderregels.

Als er geen (combinatie van) attributen is die aan deze eis voldoet, dan moet hiervoor een attribuut toegevoegd worden aan de entiteit. Dit is dan een betekenisloos volgnummer. Dit volgnummer wordt automatisch door het systeem toegekend bij het toevoegen van een nieuw record in de tabel [Ref 16].
De standaardnaam hiervoor is bijvoorbeeld ID of tabelcode_ID. Er zijn ook ontwerpers die aan elke tabel zo'n ID toekennen, of deze nu nuttig is of niet.

Domeinen en datatypes

In het klassediagram is het type van een attribuut beperkt tot de types string, integer, real en boolean plus de zelf gedefinieerde

types zoals in ons voorbeeld BedragType, DatumType, PostcodeType en SexType, enzovoort. Deze zullen gematched moeten worden op de datatypes die in het te gebruiken RDBMS mogelijk zijn. Verder is er in het klassediagram in het algemeen geen (maximale of vaste) lengte van het attribuut vastgelegd. In het fysiek model is dit wel nodig.
Bekijk nu eerst of het zinvol of mogelijk is om voor *alle* attributen een domein (is een zelf gedefinieerd datatype of user-defined datatype) te definiëren. De meeste CASE tools voor het vastleggen van een (logisch en) fysiek model ondersteunen dit begrip. De ondersteuning van deze domeinen is in de verschillende RDBMS producten nogal verschillend. In PostgreSQL is dit (in overeenstemming met de ANSI SQL standaards) mogelijk inclusief definitie van constraints, zie [Ref 24, 25]. In MS SQL Server en Sybase is dit beperkt (wel het type, geen constraint) mogelijk en ook gebruikelijk. In Oracle is het niet echt mogelijk.
Uitgaande van het voorbeeld zouden we de domeinen in afbeelding 1 kunnen definiëren voor de attributen uit het voorbeeld, in elk geval in de CASE tool, zo mogelijk ook in de database.
Het type BOOLEAN wordt niet door alle database producten als standaard type ondersteund. In dat geval moet ook hiervoor een domein gedefinieerd worden plus de beslissing hoe de waarden waar/onwaar gerepresenteerd worden (bijvoorbeeld met 1/0, True/False, Ja/Nee of Y/N).
Verder worden er voor alle in het klassediagram gedefinieerde types nog domeinen gedefinieerd, dit is te zien in afbeelding 2. Met deze beperkte set aan domeinen kunnen alle attributen in het voorbeeldmodel worden gedefinieerd en ook bij grotere systemen is het aantal domeinen meestal redelijk beperkt. Bij elk attribuut wordt dus verwezen naar een domein in plaats van naar een standaard datatype van het betreffende RDBMS. Het voordeel van deze domeinen (of user-defined datatypes) is hergebruik, consistentie en uniformiteit en vaak een hogere kwaliteit van de gegevens. Andere veel gebruikte domeintypen zijn:

- Namen waarin ook diakritische (en andere UTF) tekens toegestaan zijn;
- Datum met tijd, Tijd (zonder datum) en Timestamp (computer gegenereerde tijd met 3 of 6 decimalen voor de seconde) [Ref 12];
- Periode: het verschil tussen twee datums of twee datum/tijd-attributen;
- GEO datatype zoals in Oracle Spatial;
- Character large object en Binary large object, bijvoorbeeld een pasfoto;
- XML datatype;
- Directory naam of bestandsnaam voor verwijzing naar een bestand.

Voor het type adres hebben we in ons voorbeeld een (samengesteld) datatype gedefinieerd. In het fysiek model zijn hiervoor nu twee alternatieven:

- Het overnemen van de afzonderlijke attributen (straat, huisnummer, postcode en woonplaats) van het adres in de tabellen waarin het adres gebruikt wordt (bij de orders hebben we twee adressen, dus twee keer vier attributen);
- Het definiëren van een adressentabel met een verwijzing vanuit klant en (twee keer vanuit) order.

Omdat het (aflever- en factuur)adres op de order vaak (afhankelijk van het gebruik) gelijk zal zijn aan het adres van de klant kan overwogen worden om deze leeg (NULL) te laten:

- Factuuradres is NULL, dan geldt hiervoor het afleveradres;
- Afleveradres is NULL, dan geldt hiervoor het adres van de klant.

Enumeraties

Bij Klantstatus, Artikelstatus en Sextype is hier een domein van 1 positie gedefinieerd met een constraint dat het betreffende teken altijd een hoofdletter is en alleen een beperkte set waarden is toegestaan, die elk ook een specifieke betekenis hebben. Deze betekenis is nu gedefinieerd buiten de database. Alternatieven zijn:

- Een lange code (Man/Vrouw in plaats van M/V);
- Een referentietabel waarin de korte code met de betreffende omschrijving wordt opgenomen.

Welke van de alternatieven het beste is, is een kwestie van gewoonte en smaak. Overwegingen zijn hierbij ook performance, onderhoudbaarheid enzovoort.

Domeinnaam	Doel	Type in Sybase, MS SQL Server	Type in Oracle
Naam	Alle namen en korte omschrijvingen	varchar(80)	varchar2(80)
Volgnummer	Voor bijvoorbeeld klantnummer, ordernummer	integer	number(1)
Huisnummer		integer	number(5)
BTW code		numeric(1)	number(1)
Artikel nummer		char(12)	varchar2(12)
Percentage	Bijvoorbeeld BTW percentage	numeric(5,2)	number(5,2)
Aantal	Bijvoorbeeld voorraad of aantal besteld	integer	number(6)

Afbeelding 1: Domeinen.

Domein	Doel of constraint	Type in Sybase, SQL Server	Type in Oracle
Boolean	Waarde is altijd Y (Ja) of N (Nee)	char(1)	varchar2(1)
	Of eventueel 0 (Nee) of 1 (Ja)	Small integer	Number(1)
Postcode	Altijd 4 cijfers en twee hoofdletters	char(6)	varchar2(6)
Bedrag	Voor alle bedragen	money	number(10,2)
Datum	Een datum zonder tijd (tijd = 00:00:00)	datetime	date
Emailtype	String met 1 '@' teken, geen spaties	varchar(255)	varchar2(255)
Telefoon nummer type	String met alleen cijfers en '-' tekens	varchar(24)	varchar2(24)
Sextype	Waarde is altijd M (Man) of V (Vrouw)	char(1)	varchar2(1)
Klantstatus	Waarde is altijd P, A of C	char(1)	varchar2(1)
Artikelstatus	Waarde is altijd T, A of C	char(1)	varchar2(1)

Afbeelding 2: Extra domeinen.

Constraints (validaties of business rules)

Op de meeste van de hiervoor genoemde domeinen kan een constraint of rule gedefinieerd worden. Bijvoorbeeld het domein:

- Sex mag alleen de waarde M of V bevatten;
- Postcode moet altijd vier cijfers en twee hoofdletters hebben en de eerste positie mag geen '0' zijn;
- Datum mag geen tijd bevatten;
- Bedrag mag niet kleiner dan 0 zijn;
- Artikelnummer moet altijd 12 posities zijn en mag alleen cijfers, hoofdletters en een '-' teken bevatten.

Overal waar het betreffende domein als attribuut in een tabel wordt gebruikt zal deze constraint (rule) van toepassing zijn. Bij een complete implementatie van het begrip domein (als in de ANSI standaard vastgelegd) behoeft deze constraint niet meer bij elke tabel/attribuut herhaald te worden.
Behalve op domeinniveau kunnen er nog constraints gedefinieerd worden op attribuut, record, entiteit/tabel of databaseniveau. Deze zullen vaak als tekst in de beschrijving van het klassediagram zijn opgenomen. Bijvoorbeeld:

- Attribuut: Het BTW percentage in tabel BTW percentage mag niet groter zijn dan 99;
- Attribuut: De datum besteld van een order mag niet in de toekomst liggen;
- Record: Datum factuur moet liggen op of na de Datum besteld (in de order);
- Tabel: het aantal contactpersonen van een zakelijke klant mag niet groter zijn dan 10;
- Database: het totaal bedrag van een nieuwe order mag niet groter zijn dan het maximum uitstaand saldo minus het huidige uitstaand saldo van de betreffende klant.

Hier moet ook bekeken worden hoe de betreffende controles fysiek worden uitgevoerd: Alleen in de database, alleen in de applicatie of in beide (met mogelijk dubbel onderhoud en inconsistenties). In de database zijn er nog de opties voor declaratieve definitie of het bouwen van een trigger. Voor de meer complexe controles is meestal een trigger de beste of enige optie.
Een bijzondere constraint is de NOT NULL constraint ofwel de indicatie dat een attribuut (of relatie) verplicht is. Deze kan niet op een domein, dus alleen op een attribuut worden gedefinieerd. 'Niet verplicht' kan in het klassediagram worden aangegeven met de indicatie [0..1]. Als dat niet gedaan is moet nog *functioneel* worden bekeken welke attributen al of niet verplicht zijn [Ref 15]. Vervolgens moet dit in het fysieke model worden vastgelegd.

Associaties naar relaties

De verschillende relatietypen (associatie, aggregatie en compositie) zullen in het fysieke model en de database als standaard foreign key relaties worden geïmplementeerd. Bij de compositie kan men hierbij overwegen om een 'Cascade delete' te definiëren, dus als een order verwijderd wordt, dan moeten meteen ook alle orderregels worden verwijderd. De andere betekenissen die deze relatietypen hebben zullen in het fysiek model vervallen. Bijvoorbeeld de restrictie van een eenzijdig navigeerbare associatie tussen artikel en BTW code komt in de database niet meer terug; deze is met SQL altijd tweezijdig navigeerbaar.

Inheritance relaties en klassehiërarchie

Voor de Inheritance relaties, zoals voor de klant met twee subtypes zijn er enkele mogelijkheden:

- Drie tabellen.Voor het supertype en elk subtype wordt een eigen tabel gedefinieerd met een een-op-een relatie tussen supertype en subtype (en waarschijnlijk ook een cascade delete). Dit ligt het meest voor de hand als elk subtype veel attributen bevat, bijvoorbeeld hier de klant met de twee subtypes particuliere en zakelijke klant;
- Een tabel. Er wordt een tabel gemaakt voor het supertype en alle subtypes; deze tabel zal ook attributen bevatten die,

afhankelijk van het subtype, niet van toepassing (dus NULL) zijn. Voor het artikel (zie voorbeeld) met de twee subtypes enkelvoudig en samengesteld artikel is dit een voor de hand liggende keuze.
- Twee tabellen. Er wordt een tabel gemaakt voor elk subtype dat daarbij ook de attributen van het supertype bevat. Bijvoorbeeld een tabel particuliere klanten en een tabel zakelijke klanten en geen tabel klanten meer. Een order verwijst dan ofwel naar de ene of naar de andere tabel, hetgeen deze relatie lastiger maakt. Ook het wijzigen van type (een probleemrapport wordt een change request) is dan lastiger. Deze oplossing ligt dan ook niet vaak voor de hand.

Hieronder volgen nog enkele voorbeelden van een meer complexe (platte of diepe) klassehiërarchie:
Medewerker met subtypes:
- Man en Vrouw;

maar tegelijk ook:
- Allochtoon en autochtoon;

en:
- Inhuur of vaste dienst.

En een wat diepere structuur:
Voertuig met subtypes:
- Lucht: Vliegtuig;
- Water: Vaartuig;
- Land: LandVoertuig met twee subsubtypes;
    1. Gemotoriseerd met subsubsubtypes;
        - Motorfiets;
        - Auto met;
            - Personenauto;
            - Bus;
            - Vrachtauto;
    2. Op spierkracht met subsubsubtypes;
        - Fiets;
        - Step.

N.B. Een amfibievoertuig erft weer de eigenschappen van zowel een Vaartuig als een Auto. Dit heet 'Meervoudige overerving'.

Hoe je dit zou kunnen implementeren? Voor dit laatste voorbeeld zou ik (afhankelijk ook van de gewenste functionaliteit) geen 13 tabellen definiëren. Maar ook niet één tabel mat alle attributen van alle types. Een wat abstractere mogelijkheid is één tabel met wat algemene attributen num01, num02, .. , char01, char02, .., datum01, datum02, .. enzovoort. Afhankelijk van het type heeft num01 een andere betekenis, bijvoorbeeld bij personenauto en bus is num01 het aantal passagiers en bij vrachtauto is num01 het laadvermogen in tonnen. Vervolgens leg je dan hierover 13 views met de voor dat type zinvolle namen en de selectie. Het supertype moet dan een typecode hebben waarin aangegeven staat welk subtype op dit object/voorkomen van toepassing is. Dit principe van een algemene entiteit en per subtype specifieke views heb ik al een enkele keer toegepast, bijvoorbeeld toen ik zelf een datadictionary gebouwd heb. Het toevoegen van nieuwe functionaliteit was daarbij minimaal werk:
- In de database: een nieuwe waarde toelaten in de typecode en een nieuwe view definitie;
- In de applicatie: Door daarna de code van een bestaande functie/scherm te kopiëren en daarin wat labeltjes te veranderen (of toevoegen, verwijderen) was ook het bouwen van de CRUD (invoeren, wijzigen, verwijderen en opvragen) functie voor dit nieuwe type heel snel klaar.

Het logisch model bestond bij dat systeem uit 50 entiteiten en het fysiek model uit 3!

Afgeleide gegevens

De attributen die met een / gemarkeerd zijn in het klassediagram zijn afgeleide attributen. Dat wil zeggen; de waarde kan berekend worden uit de waarde van andere gegevens uit het diagram. Een voorbeeld is het uitstaand saldo van een klant dat de som is van de totaalbedragen van de orders minus de per order reeds betaalde bedragen. Het totaalbedrag van de order is weer de som van de bedragen (aantal maal prijs) van elke orderregel van de betreffende orders [Ref 11].
Bij het opstellen van het fysiek model moet een beslissing genomen worden over wat te doen met deze afgeleide gegevens:
- Telkens wanneer dit gegeven nodig is de betreffende berekening uitvoeren;
- Het gegeven ook redundant (dus fysiek) opslaan in de database.

De laatste optie heeft als groot voordeel dat het gegeven veel sneller beschikbaar is. Nadeel natuurlijk is dat het attribuut telkens moet worden bijgewerkt als 1 van de onderliggende gegevens wijzigt, bijvoorbeeld bij het plaatsen van een nieuwe order. Op basis van de hoeveelheid gebruik van het berekende gegeven en de complexiteit en duur van de berekening kan hiervoor een overwogen beslissing genomen worden.
Als het gegeven niet fysiek (redundant) wordt opgeslagen kan bij sommige producten een kolom worden gedefinieerd waarvan de waarde via een functie wordt berekend op het moment van ophalen van de betreffende kolom.
Als besloten wordt om dit gegeven wel redundant op te slaan moet nog bekeken worden hoe dit gegeven kan worden bijgewerkt. De meest voor de hand liggende optie is een trigger op de tabellen waarin de gegevens staan waaruit het redundante gegeven wordt berekend. Dit geeft de beste garantie dat de berekening altijd en correct wordt uitgevoerd.

Naamgeving

Aan de namen in het klassediagram worden weinig beperkingen opgelegd. Bij de naamgeving van de fysieke objecten (tabellen, attributen, domeinen, constraints, indexen) moet men zich echter conformeren aan de beperkingen die het RDBMS hieraan oplegt:
- Bepaalde tekens, zoals een spatie, –, + of . zijn vaak verboden, omdat deze al een specifieke betekenis hebben in de database;

- De lengte van de naam is meestal beperkt tot bijvoorbeeld 30 of 32 posities;
- Sommige woorden zijn 'Reserved words' in het RDBMS en mogen niet zonder meer gebruikt worden als objectnaam. Stel we hebben een tabel 'ORDER'; De SQL query:
  `Select * from order order by ID`
  wordt dan wel erg verwarrend;
  `Select * from "order" order by ID`
  is wel toegestaan (in bijvoorbeeld Oracle) maar zal snel tot fouten leiden.

Een systeem-prefix (bijvoorbeeld ORD) voor alle entiteiten/tabellen kan handig zijn om dit probleem te voorkomen. De tabelnamen worden dan bijvoorbeeld ORD_KLANT en ORD_ORDER. Verder zullen de namen van tabellen, domeinen en attributen zoveel mogelijk de logische namen uit het klassediagram volgen.

Een code of *mnemonic* voor elke tabel (van altijd drie of vier hoofdletters of eventueel cijfers, bijvoorbeeld ART voor de tabel Artikel) kan handig zijn. Deze kan gebruikt worden voor bijvoorbeeld:

- De naam van een betekenisloze primary key kolom: ART_ID;
- De foreign key in tabel artikel naar de BTW tabel: ART_BTW_ID;
- De naam van een constraint: ART_PK, ART_BTW_FK, ART_C01;
- De naam van een index: ART_PK, ART_BTW_FK, ART_I01.

Verder kan deze gebruikt worden als tabel-alias in een SQL query en voor de namen van views, triggers enzovoort. Dit valt verder buiten de scope van dit artikel.

Andere fysieke aspecten

Na deze transformatie moeten nog alle fysieke aspecten worden bekeken en gedefinieerd die ook bij een overgang van een traditioneel logisch naar een fysiek model gedaan moeten worden. Dit betreft onder meer de indexen, partitionering, fysieke verdeling over schijven, extra redundantie, het tunen van de database (geheugen gebruikt enzovoort) en andere performance aspecten [Ref 13, 18]. Dit valt buiten de scope van dit artikel.
Bekijk hierbij ook hoe transacties afgehandeld worden en hoe concurrent update control wordt geïmplementeerd. Vaak worden er in het logisch of fysiek model aan elke tabel twee kolommen toegevoegd met de gebruiker die het laatst het record heeft gewijzigd en de computertijd (type timestamp) waarop dit is gedaan.

Optimistic concurrency control

Het probleem: de gegevens van persoon X staan in de database.
Gebruikers A en B halen beide de gegevens van persoon X naar het scherm.
Gebruiker A wijzigt bijvoorbeeld het adres.
Gebruiker B wijzigt bijvoorbeeld het telefoonnummer, het adres op zijn scherm blijft ongewijzigd.
Gebruiker A klikt op opslaan: het adres in de database wordt gewijzigd.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
         "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.capgemini.datamodelling">
        <class name="ContactPersoon" table="CONTACT_PERSOON">
        <id name="id" type="long" column="ID">
                <generator class="native" />
        </id>
        <property name="firstName" type="string" column="FIRST_NAME"
                not-null="true" length="80" />
        <property name="lastName" type="string" column="LAST_NAME"
                not-null="true" length="80" />
        <property name="sex_" type="string" column="SEX" not-null="true"
                length="1" />
        <property name="telephone" type="string" column="TELEPHONE"
                not-null="true" length="16" />
        <property name="email" type="string" column="EMAIL" not-null="false"
                length="80" />
        </class>
</hibernate-mapping>
```

Afbeelding 3: Mapping van de klasse ContactPersoon op de betreffende tabel.

Gebruiker B klikt op opslaan: de gegevens op het scherm, dus het gewijzigde telefoonnummer en het oude adres worden in de database opgeslagen.
En hiermee is de wijziging van gebruiker A verloren gegaan!

Dit kan worden voorkomen door te kijken of er tussen het ophalen en wegschrijven geen andere updates zijn gedaan door:
- Een timestamp van de laatste mutatie in het record op te nemen en bij het wegschrijven alleen deze kolom te vergelijken met de waarde hiervan bij het ophalen;
- De waarde van alle kolommen in de update te vergelijken met de waarde bij het ophalen.

Als geconstateerd wordt dat er een wijziging tussendoor is geweest, dan krijgt de tweede gebruiker een melding dat hij de gegevens opnieuw moet ophalen en de wijziging opnieuw moet aanbrengen.
Dit heet optimistic concurrency control: optimistic omdat je er vanuit gaat dat er geen gelijktijdige wijzigingen zijn, maar als het zou gebeuren wordt het wel geconstateerd. Bij pessimistic concurrency control worden de gegevens die op het scherm staan in de database gelockt, zodat een ander deze niet kan wijzigen. Als de gebruiker dan van zijn scherm weggaat (kopje koffie enzovoort) lopen andere gebruikers (of batchprocessen) mogelijk tegen deze lock aan en als die gebruikers ook gegevens gelockt hebben komt langzaam het hele systeem tot stilstand.
Hiermee moet, bij het opstellen van het fysiek model, rekening gehouden worden en er moet eventueel hiervoor een timestamp attribuut worden geïntroduceerd.

Het genereren van het fysiek model

UML Design tools, zoals Enterprise Architect, kunnen geautomatiseerd een fysiek model en de tabel- (en index- enzovoort) definities genereren. Het is zeer aan te bevelen om dit gegenereerde model op bovenstaande aspecten kritisch te bekijken en eventueel bij te stellen. Te vaak komt het voor dat klakkeloos met het gegenereerde model verder wordt gewerkt met grote performanceproblemen als gevolg in een latere (te late) fase van het project.

Object-Relational mapping tools

Vaak worden in een taal als Java de gegevens uit de database opgehaald via een gewone SQL query. Het werken met klassen en objecten, zoals dat in deze talen bedoeld was, wordt dan ten dele omzeild en er wordt weer traditioneel 'procedureel' gewerkt. Het alternatief is om geheel netjes volgens de principes van OO en Java te werken. Maar op bepaalde momenten wil je toch dat de gegevens in het geheugen naar de database worden geschreven (gepersisteerd) of uit de database opgehaald. Dit kan met een Object-Relational mapping tool zoals Hibernate [Ref 10] of TopLink [Ref 5], enzovoort.
Er zijn in eerste instantie twee mogelijkheden:
- De klassedefinities worden aan Hibernate aangeboden en Hibernate maakt zelf het fysieke schema en de databasedefinities;
- Het fysieke schema en de databasedefinities worden handmatig (dat wil zeggen met een tool als Power Designer) gemaakt en de link tussen de klasse en de tabel wordt in Hibernate vastgelegd via een definitie zoals in afbeelding 3.

Hiermee kan in talen als Java netjes gewerkt worden met de klassen en objecten. Als gegevens nodig zijn uit de database, dan zorgt Hibernate dat deze worden opgehaald en als gewijzigde gegevens naar de database geschreven moeten worden dan zorgt Hibernate daar ook voor.

Bij het genereren van het fysiek model door Hibernate moet weer goed gekeken worden naar hoe dit model uit de generator komt en of er mogelijk toch wat aan gesleuteld moet worden, met name om een goede performance te garanderen. Ook de query's die Hibernate los laat om gegevens uit de database op te halen moeten kritisch bekeken worden. Als Hibernate voor bijvoorbeeld een join van enkele tabellen voor elke tabel een query naar de database stuurt en de gegevens zelf aan elkaar plakt (of als dit in Java gedaan wordt), dan kan het efficiënter zijn om in de database een view te definiëren met deze join en de gegevens via deze view op te halen. Ook als er niet direct een performanceprobleem is met de applicatie kan het nuttig zijn om alle query's die naar de database worden afgevuurd toch eens door te nemen. Als er op enkele plaatsen, met name de veel gebruikte of lang durende query's, een performanceverbetering gevonden kan worden heeft dat niet alleen voordeel voor de betreffende functies maar ook voor de performance van het systeem in het algemeen [Ref 13].

Conclusie

Steeds vaker wordt de (logische) gegevensstructuur van een systeem niet meer als een Relationeel (Entity Relationship of ER) model vastgelegd of aangeleverd maar in de vorm van een object- of klassediagram. In deze serie artikelen hebben we een introductie gegeven in het klassediagram en aangegeven hoe de objecten uit een klassediagram in een relationele database kunnen worden opgeslagen.
De conclusie uit het bovenstaande is dat het klassediagram heel geschikt kan zijn om het model logisch vast te leggen, maar dat het ongeschikt is voor een fysiek relationeel model. Er zal dus altijd een vertaalslag nodig zijn van klasse (eventueel via logisch) naar fysiek. Dit is ook het geval bij een vertaling van een logisch relationeel model naar een fysiek model, maar bij een klassediagram als uitgangspunt zal deze vertaling groter zijn, bijvoorbeeld omdat het model nog niet volledig is genormaliseerd en de datatypen (domeinen) meer afwijken van de datatypen in de gebruikte database.

Voor de literatuurreferenties verwijzen we naar deel 1 in DB/M 5.

Toon Loonen (toon.loonen@capgemini.com) is werkzaam bij Capgemini.