Tussen Types en Tabellen...

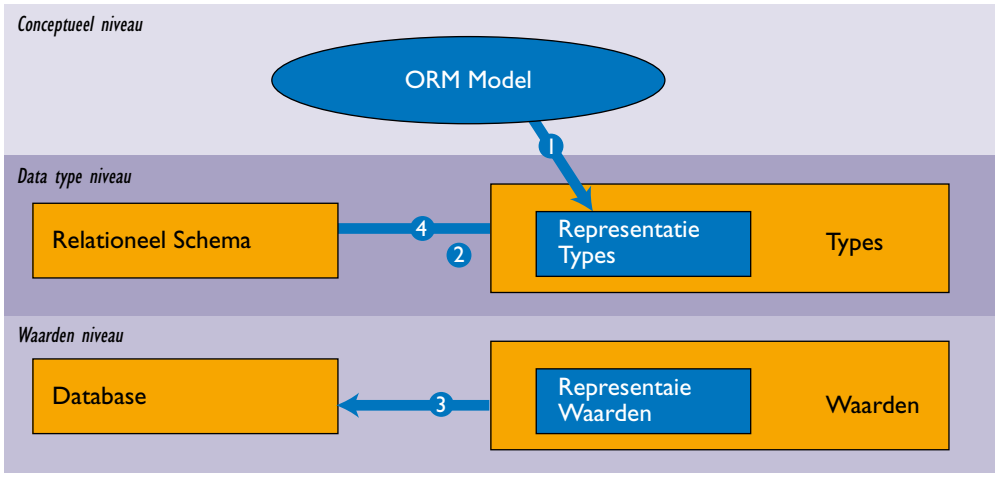
Automatisch gegevens uitwisselen

Iedereen die wel eens een informatiesysteem heeft gebouwd kent het: je wilt een simpele interface maken om wat gegevens in de database in te voeren of te bewerken en moet vervolgens hele lappen code schrijven om data uit je programma via SQL-queries in de database te krijgen. Dat moet toch automatisch kunnen. Maar hoe doe je dat dan? Deze vraag stond centraal in de met de Aia Software Masterscriptie Award 2009 bekroonde scriptie 'Between Types and Tables: Generic Mapping between Relational Databases and Data Types in Clean'.

Veel hedendaagse informatiesystemen bestaan uit twee kerncomponenten. Een centrale relationele database en één of meerdere applicaties die hiervan gebruik maken. Deze ontkoppeling van de database en applicaties heeft veel voordelen, zoals efficiënte opslag en centrale integriteitsbewaking. De keerzijde is echter dat de platte tabelstructuur, waarin gegevens in een relationele database opgeslagen worden, niet de handigste is om de data in een applicatie te manipuleren of aan eindgebruikers te presenteren. In de code van applicatiesoftware is het vaak handiger om gerelateerde data te groeperen in geneste datastructuren.

Om data uit een database in een applicatie te kunnen gebruiken, moeten er één of meerdere SQL-queries op de database gedaan worden om de verschillende stukjes informatie uit de database te halen en samen te voegen tot één structuur. Als de gegevens zijn bewerkt en weer moeten worden opgeslagen, moet opnieuw een aantal SQL-queries worden uitgevoerd. Hoewel SQL een geschikte taal is voor het doen van ingewikkelde queries op een database, is het als interface naar applicatiesoftware niet ideaal. De code die nodig is om de vertaalslag tussen de database en de applicatiedatastructuren via SQL te

Bas Lijnse (links) krijgt zijn prijs uit handen van Jeroen Huinink, CTO van AIA.



Figuur 1.

maken is vervelend om te schrijven, tijdrovend en ook nog eens een bron van bugs en securityproblemen. Omdat SQL-queries in de code voor een programmeertaal gewone strings zijn, en geen onderdeel van de taal, kan een compiler of IDE ze niet analyseren en komen fouten pas at runtime tijdens een test boven.

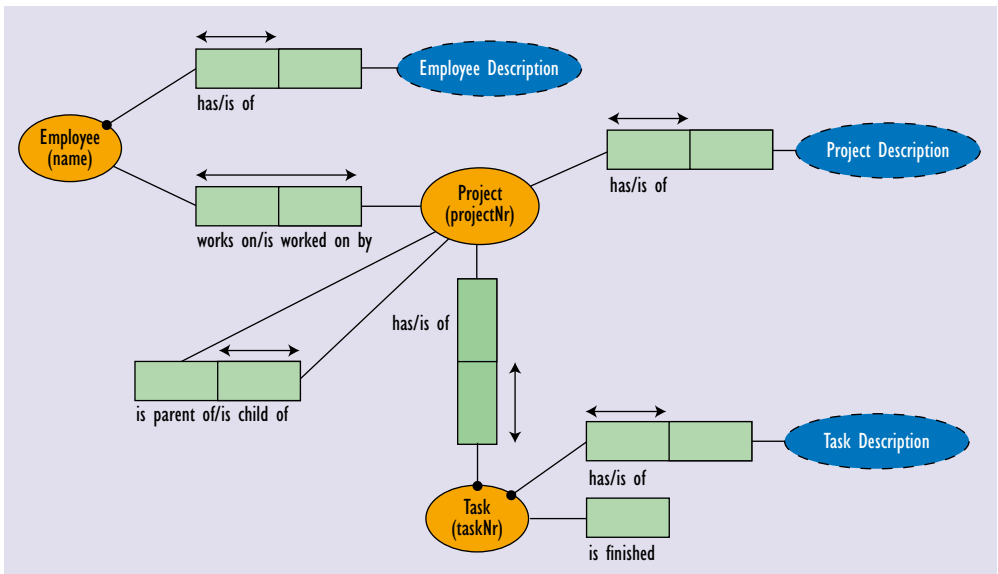
Zou het niet ideaal zijn als je dit soort code helemaal niet hoeft te schrijven? Met behulp van een recente techniek in functionele programmeertalen is het mogelijk om dit probleem in veel gevallen eens en voor altijd generiek op te lossen.

Één concept, meerdere representaties

Door de tweedeling van informatiesystemen in database- en applicatiesoftware, die informatie allebei op hun eigen manier structureren, is er een vertaalslag nodig wanneer gegevens door een applicatie uit een database gehaald, of weer weggeschreven worden. Maar op een conceptueel niveau is het natuurlijk wel dezelfde informatie. Het is hoogstwaarschijnlijk niet de bedoeling dat data uit een 'Product' tabel bij het inlezen ineens een 'Klant' datastructuur oplevert. Het lastige is dat deze conceptuele gelijkheid niet uit de code kan worden afgeleid zonder kennis van het probleemdomein en/of algemene kennis. Een programmeur weet best dat producten en klanten andere concepten zijn en dus bij het ophalen uit een database niet zomaar inwisselbaar zijn. Een compiler of bibliotheek heeft zulke achtergrondkennis niet en zal dus op een andere manier aan de benodigde informatie moeten komen.

Een manier om dit probleem op te lossen is zorgen dat er een vaste relatie bestaat tussen databases en de daarmee corresponderende datastructuren in applicaties. Om dit te bereiken moeten we op een wat hoger niveau naar de informatie kijken. We weten dat sommige selecties uit tabellen in een database conceptueel gelijk zijn aan bepaalde structuren in applicaties. Als een applicatie iets doet met klanten en producten, zullen die concepten zowel in de database als in de programmacode voorkomen. De database en applicaties worden echter los van elkaar ontworpen waardoor er geen formele relatie tussen beide bestaat.

Als we eerst een conceptueel model maken, dan kunnen we daar vervolgens zowel database tabellen als types van datastructuren uit afleiden. Door dit gestructureerd te doen ontstaat er een formele koppeling tussen de informatie in de database en applicaties waarmee de fundering is gelegd voor een generieke mapping van de informatie in de twee verschillende representaties.



Figuur 2.

In figuur 1 is dit schematisch weergegeven. Uit een conceptueel model worden de types van applicatiedatastructuren afgeleid (pijl 1). Deze worden gebruikt om de tabelstructuur van de database af te leiden (pijl 2). Vervolgens is het dan mogelijk om automatisch gegevens uit te wisselen tussen een database en een applicatie (pijl 3). Wanneer er al een database

is, kan die onder bepaalde voorwaarden ook gebruikt worden om bijpassende applicatiedatatypen af te leiden, in plaats van het conceptuele model (pijl 4).

Net genoeg informatie

De eerste stap is het maken van een conceptueel model. De modelleertaal die we daarvoor gebruiken is Object Role Modeling (ORM) waarmee objecten en de relaties ertussen beschreven worden.

ORM modellen zijn vergelijkbaar met de veel gebruikte Entity Relationship (ER) modellen, maar maken geen onderscheid tussen attributen en relaties.
Figuur 2 toont een voorbeeld ORM model voor een eenvoudige projectadministratie. Het model beschrijft de objecten 'Project', 'Task' en 'Employee' en de relaties ertussen. Projecten zijn abstracte entiteiten met een uniek projectnummer en een beschrijving. Het zijn containers voor taken en er kan door werknemers aan projecten worden gewerkt. Bovendien kunnen projecten hiërarchisch worden geordend, zodat een project mogelijk een 'ouder' project heeft en een aantal 'kind' projecten. Taken zijn werkeenheden die moeten worden gedaan voor een project. Ze worden geïdentificeerd door een uniek taaknummer en hebben een beschrijving. Verder wordt bijgehouden of taken klaar zijn of niet. Medewerkers zijn mensen met een unieke naam en omschrijving die aan projecten werken. Iedere medewerker kan aan meerdere projecten werken en meerdere medewerkers kunnen aan hetzelfde project werken.
In het vervolg van dit artikel zullen we dit model steeds als voorbeeld gebruiken.

De eerste vervolgstap die we aan de hand van het ORM-model maken, is het afleiden van een verzameling datatypen die de concepten uit het model in de applicatiecode representeren. We doen dit in dit geval voor de programmeertaal Clean. Dit is een statisch getypeerde functionele programmeertaal, zoals bijvoorbeeld Microsoft's opkomende F#, en is verwant aan Haskell. Het feit dat we een statisch getypeerde taal gebruiken is essentieel, omdat we in de programmeertaal de structuur van de data moeten kunnen definiëren. In een dynamisch getypeerde taal zoals Javascript of PHP is zoiets niet mogelijk en is het aantal mogelijke typen op een hand te tellen (number,object,array,string).

De Clean datatypes die we afleiden uit het ORM model in figuur 2 zijn de volgende:

```
:: Employee =
{ employee_name                              :: String
, employee_description                       :: String
, projectworkers_project_ofwhich_employee    :: [ProjectID]
}
:: EmployeeID =
{ employee_name                              :: String
}
:: Project =
{ project_projectNr                          :: Int
, project_description                        :: String
, project_parent                             :: (Maybe ProjectID)
, task_ofwhich_project                       :: [Task]
, project_ofwhich_parent                     :: [ProjectID]
, projectworkers_employee_ofwhich_project    :: [EmployeeID]
}
:: ProjectID =
{ project_projectNr                          :: Int
}
:: Task =
{ task_taskNr                                :: Int
, task_project                               :: ProjectID
, task_description                           :: String
, task_done                                  :: Bool
}
:: TaskID =
{ task_taskNr                                :: Int
}
```

Deze typedefinities kun je lezen als een zestal recorddefinities die als velden ofwel eenvoudige typen (Bool,Int,String,etc.), ofwel lijsten van andere records (genoteerd met blokhaken) bevatten, zodat een geneste structuur kan worden opgebouwd.
De types zijn afgeleid door voor ieder objecttype in het ORM-model twee Clean records te definiëren. Het eerste bevat alle feiten die over een object bekend zijn. Het tweede record bevat enkel de identificatie van een object en heeft daarom de suffix 'ID'. De veldnamen van de records volgen een vast patroon. Gegevens die maar op één conceptueel object van toepassing zijn, zijn geprefixt met de naam van dat concept. Gegevens die op twee concepten van toepassing zijn komen in de records van beide concepten voor. Met het 'ofwhich' tussenvoegsel in een veldnaam word een conditionele relatie tussen objecten aangegeven. Het veld 'task_ofwhich_project' in het 'Project' record wordt geïnterpreteerd als: alle taken waarvan het veld

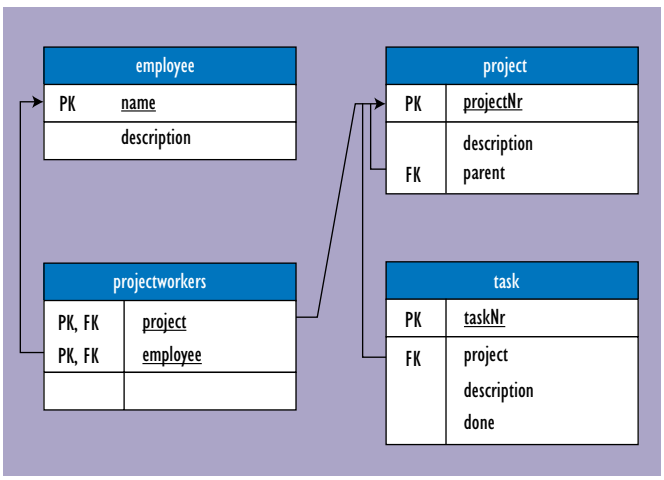
Sommen & Producten

De in Clean beschikbare vorm van datatype-generiek programmeren is ook wel bekend als "sums & products". De onderliggende theorie is dat iedere datastructuur op een vaste manier kan worden omgezet naar een structuur bestaande uit sommen en producten en omgekeerd. Dit is een representatie die de waarde en zijn type-informatie bevat. Wanneer een structuur samengesteld is uit meerdere delen is dat een som. Wanneer een type alternatieven bevat geeft dat een product. Door functies te definiëren die sommen en producten als invoer krijgen of ze als resultaat opleveren, heb je generieke functies die werken voor ieder type. Er kan immers altijd van en naar deze representatie geconverteerd worden. De Clean compiler kan deze conversies zelfs automatisch afleiden.

'task_project' gelijk is aan de identificatie van dit project. Afhankelijk van of de relaties tussen objecten one-to-one of one-to-many zijn en of deze relaties verplicht gedefinieerd zijn, bevat een veld ofwel een enkele waarde, een lijst van waardes of een optionele waarde (Maybe). De many-to-many relaties tussen twee objecten worden met een unieke relatie naam geprefixt, zoals bijvoorbeeld 'project-workers'. Zulke relaties horen namelijk evenveel bij beiden concepten en kunnen dus niet bij één van de concepten gegroepeerd worden.

Generiek default waarden produceren voor elke denkbare datastructuur

Het laatste wat nog over deze verzameling types opgemerkt moet worden is het gebruik van de identificatie records eindigend op 'ID'. Deze records worden gebruikt om aan te geven hoe 'ver' gegevens bij een object opgezocht moeten worden. Als we bijvoorbeeld gegevens over een project willen ophalen uit de database, willen we wel de namen weten van de medewerkers die aan dat project werken, maar willen we niet ook meteen alle gegevens over die personen weten. Het op deze manier greedy lezen zou in veel gevallen betekenen dat we de hele database zouden uitlezen aangezien de meeste gegevens in de database aan elkaar gerelateerd zijn. Om aan te kunnen geven dat het voldoende is om alleen identificaties te weten van gerelateerde objecten worden de "ID" records gebruikt.
De tweede stap in figuur 1 is die van types naar een databaseschema. Doordat de gegevens binnen de Clean types allemaal al gegroepeerd zijn met prefixen in de veld namen, is het afleiden van de tabelstructuur een kleine stap. Iedere groep komt in een aparte tabel met kolommen voor iedere basiswaarde die direct bij een concept gegroepeerd is. De many-to-many relaties tussen concepten komen in aparte tabellen. In figuur 3 is het database schema te zien dat uit de set voobeeldtypen is afgeleidt.



Figuur 3.

Doe meer met types

We hebben nu bijna alles wat we nodig hebben om automatisch te kunnen mappen tussen een database en applicatiedatastructuren. We hebben een databaseschema, en een verzameling types die net voldoende informatie bevatten om gegevens te kunnen vinden in de database. De enige voorwaarde waar we nog aan moeten voldoen is ervoor zorgen dat we de type informatie ook kunnen gebruiken om informatie in een database terug te vinden.
Normaal gesproken hebben types in een statisch getypeerde programmeertaal twee hoofddoelen. Ten eerste behoeden ze programmeurs voor fouten door af te dwingen dat functies de juiste parameters krijgen. Je kunt een string bijvoorbeeld niet kwadrateren en een integer niet in regels opsplitsen. Het tweede doel van types is het verschaffen van informatie aan de compiler voor optimalisatie. Als een compiler beschikt over type-informatie kan deze worden gebruikt in analyses om efficiëntere code te genereren. Clean biedt naast dit standaard gebruik van types nog een extra manier om gebruik te maken van type-informatie. Het biedt de mogelijkheid om type-generische functies te definiëren. Dit zijn functies die werken voor waarden van ieder type en die gebruik kunnen maken van de structuur van een type (zie kader Sommen & Producten). Zo is het bijvoorbeeld mogelijk om generiek een structurele gelijkheid van datastructuren te definiëren of om generieke serialisatie en deserialisatie functies te definiëren voor marshalling van data. Een interessante eigenschap van dit soort functies is dat ze, in tegenstelling tot slechts de structuur van een gegeven waarde te inspecteren, ook waardes kunnen construeren op basis van een type. Dit maakt het bijvoorbeeld ook mogelijk om generiek default waarden te produceren voor elke denkbare datastructuur.
Een extreem voorbeeld van het gebruik van generieke functies in Clean is het iTasks workflow systeem, dat naast opslag en serialisatie, ook generiek web-based GUIs genereerd.

Automatisch mappen

De derde en belangrijkste stap uit figuur 1 is het automatisch mappen tussen de gegevens in de database en structuren in applicaties. Wanneer aan de eerste twee stappen voldaan is, is deze stap te realiseren met behulp van generische functies. We doen dit door middel van een bibliotheek die de standaard CRUD (Create, Read, Update, Delete) operaties generiek aanbiedt. De API definitie van deze bibliotheek ziet er als volgt uit:

```
gsql_read   :: ref con -> (Maybe GSQLError, Maybe val, con)
gsql_create :: val con -> (Maybe GSQLError, Maybe ref, con)
```

Aia Software Awards

Aia Software heeft voor de zevende keer Awards uitgereikt aan de beste scripties van informatica en informatiekunde studenten van de Radboud Universiteit Nijmegen. De scriptie 'Between Types and Tables – Generic Mapping between Relational Databases and Data Structures in Clean' van Bas Lijnse is beloond met de Masterscriptie Award. Bas legt in dit artikel de gedachtengang achter zijn scriptie uit.Hij heeft onderzoek gedaan naar het benaderen van databases vanuit Clean. Clean is een functionele programmeertaal die ontwikkeld is aan de Radboud Universiteit. Het is Bas gelukt data te lezen en te schrijven vanuit Clean zonder dat de programmeur kennis moet hebben van SQL, met behoud van de waardevolle eigenschappen van functionele talen.

```
gsql_update :: val con -> (Maybe GSQLError, Maybe ref, con)
gsql_delete :: ref con -> (Maybe GSQLError, Maybe val, con)
```

Deze definities zijn te lezen als: Er zijn vier functies die ieder twee argumenten hebben en drie waardes opleveren. De 'gsql_read' functie krijgt een identificatiewaarde (ref), bijvoorbeeld een waarde van type 'ProjcectID', en een databaseconnectie (con). Het resultaat van de operatie bestaat uit drie delen. Een 'Error' waarde die aangeeft of de operatie goed is gegaan, als dat zo is, de datastructuur met informatie, bijvoorbeeld van type 'Project', en als laatste de database connectie. Het gaat te ver om in dit artikel de onderliggende werking van de bibliotheek helemaal uit te werken. In plaats daarvan zullen we een veel voorkomend scenario stap voor stap doorlopen: het uitlezen van gegevens uit de database, ze aanpassen en vervolgens de database updaten.

Stel dat de database in figuur 3 de volgende informatie bevat: Er is een project met projectnummer 84 en beschrijving 'Lente brochure'. Er zijn twee taken gedefinieerd voor dit project. Één met taaknummer 481 en omschrijving 'Tekst opstellen', en een tweede met taaknummer 487 en beschrijving 'Drukker bellen voor offerte'. Beide taken zijn nog niet klaar. Aan dit project werken twee medewerkers: 'john' en 'bob'.
Al deze informatie kan met de 'gsql_read' functie in één keer uit de database gelezen worden met een enkele regel code:

```
(mbError, mbProject, con) = gsql_read {ProjectID|project_projectNr =
84} con
```

Als alles goed gaat levert dit de volgende datastructuur op:

```
{ Project | project_projectNr = 84
 , project_description = "Lente brochure"
 , project_parent = Nothing
 , task_ofwhich_project =
   [{ Task | task_taskNr = 481, task_project = 84
    , task_description = "Tekst opstellen", task_done = False
    },
    { Task | task_taskNr = 487, task_project = 84
    , task_description = "Drukker bellen voor offerte", task_done =
False
    }]
 , project_ofwhich_parent = []
 , projectworkers_employee_ofwhich_project =
  [{EmployeeID | employee_name = "john" },{EmployeeID | employee_name =
"bob" }]
}
```

Het creëren van deze structuur is door de 'gsql_read' functie in een aantal stappen gedaan. Eerst is de structuur van het doeltype (Project) geanalyseerd om te bepalen welke SQL-queries nodig waren voor de informatie. Vervolgens zijn de queries uitgevoerd en is uit de set platte databaserecords de geneste structuur geconstrueerd.
Omdat deze datastructuur niet zomaar een groepering van wat data is, maar een betekenisvolle representatie van een concept uit de database, kunnen we wanneer we deze structuur manipuleren in onze applicatiecode de wijzigingen interpreteren als gewenste veranderingen op de database. We kunnen bijvoorbeeld de volgende veranderingen op de structuur aanbrengen: We veranderen het 'project_description' veld in 'Zomer brochure', we markeren de eerste taak (nr. 481) als klaar, we verwijderen de tweede taak (nr. 487) uit de lijst en verwijderen 'john' uit de lijst medewerkers. Tenslotte voegen we nog een nieuwe taak toe gedefinieerd als:

```
{ task_taskNr = 0, task_project = {ProjectID | project_projectNr = 0}
, task_description = "Online prijzen uitzoeken", task_done = False
}
```

Al deze wijzigingen kunnen we in één keer doorvoeren in de database met een enkele simpele aanroep van de bibliotheek functie 'gsql_update':

```
(mbError, mbProjectId, con) = gsql_update project con
```

Dit voert alle wijzigingen door en vult voor de nieuwe taak ook meteen een taaknummer en het projectnummer automatisch in. Een klus waarvoor wanneer het handmatig geprogrammeerd zou worden minstens zes SQL queries voor nodig geweest zouden zijn. Het programmeren van een informatiesysteem is zo ineens een stuk eenvoudiger geworden.

Referenties

- *'Between Types and Tables: Generic Mapping Between Relational Databases and Data Structures in Clean', Bas Lijnse, Master Thesis, 2008, http://www.baslijnse.nl/projects/between-types-and-tables/*
- *'Clean Language Report', Rinus Plasmeijer & Marko van Eekelen, 2002, http://clean.cs.ru.nl/*
- *'A Generic Programming Extension for Clean', Artem Alimarine & Rinus Plasmeijer, 2002*
- *'A New Approach to Generic Functional Programming', Ralf Hinze, 2000*