Extreem snelle applicatie-ontwikkeling met behulp van metadata

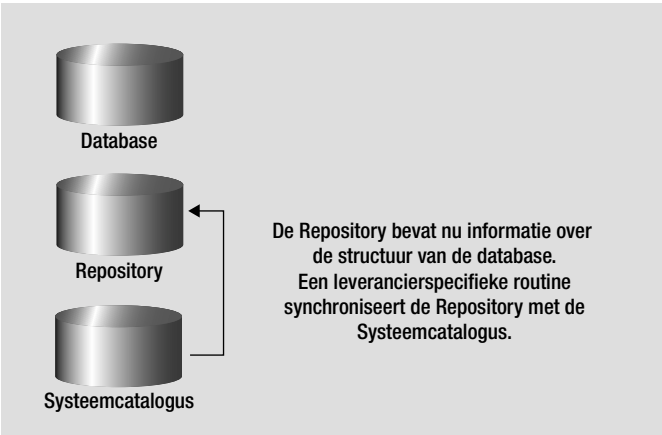
Laat de structuur ontwikkelen

Jeanot Bijpost en Marco de Groot

Iedereen die met database-ontwerp en met applicatie-ontwikkeling in aanraking komt, stelt zichzelf vroeg of laat de vraag "Kan dit niet simpeler?" Dat is bijna onvermijdelijk. Scherm na scherm wordt met de hand gemaakt en in veel gevallen volgt de opbouw van de schermen keurig de structuur van de database. Waarom zou men die structuur dan niet gebruiken om de applicatie te ontwikkelen?

In dit artikel wordt een inzicht gegeven in de belangrijkste principes achter het model-gebaseerd ontwikkelen. Ook wordt besproken welke ontwerpkeuzes er voor gezorgd hebben dat een case tool van een 'proof of concept' uitgroeide tot een zeer goed werkbare ontwikkelomgeving. In een recent project is het gelukt om in zes maanden tijd een middelgroot maatwerk ERP-systeem van de grond af op te bouwen.

Bij ontwerpen en ontwikkelen van databases en de bijbehorende applicaties gaat veel te veel tijd zitten in het schrijven van triviale zaken, zoals een simpele keuzelijst, een veld verplaatsen of een master-detail-relatie afbeelden. Het vraagt veel discipline om de schermen er uniform uit te laten zien. Erger is: het werk wordt saai, en saai werk waarbij men goed moet opletten is een basis voor het maken van fouten.
Na de ontwikkeling van een case tool voor informatiemodellering, was de overstap naar een case tool voor applicatie-ontwikkeling snel gemaakt. Toch kostte het niet minder dan vijf stevige prototypes voordat er één was gemaakt, die goed genoeg was om op verder te bouwen. De basis van het case tool dat zo ontstaan is wordt gevormd door een repository. Deze bevat zowel de structuur van de database als de structuur van de applicatie. Niet alleen de presentatie-aspecten worden vastgelegd, ook zaken als autorisatie, beperkingsregels, afleidingsregels en andere meer complexe vormen van gedrag, worden in de repository opgenomen.



Afbeelding 1: Database met repository en systeemcatalogus.

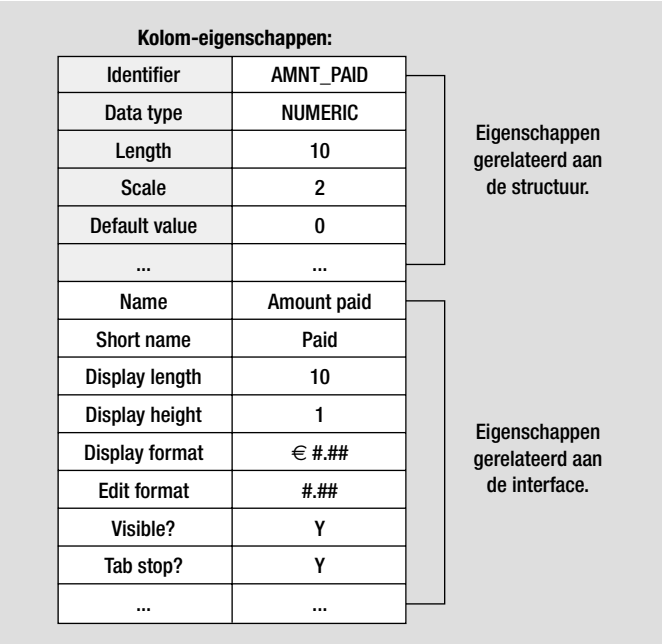
Structuurinformatie

Bij het model-gebaseerd ontwikkelen is de basis een goed gemodelleerde database. Binnen het tool wordt een nieuwe repository gemaakt voor de applicatie. Vervolgens wordt de systeemcatalogus van de database uitgelezen en in deze repository geplaatst. Dit is tot nu toe eigenlijk niets bijzonders, de ene repository (systeemcatalogus) wordt gekopieerd in de andere. Dit wordt gedaan om niet afhankelijk te zijn van de structuur van de systeemcatalogus van een bepaalde leverancier. Uiteraard is er dan wel een synchronisatie-routine nodig, die ervoor zorgt dat wijzigingen in de database gesynchroniseerd worden met de repository.

Bij de meeste tools begint men met een lege repository én een lege database. In de repository moet de structuur van de database worden ingevoerd en vervolgens wordt deze database gegenereerd. Dit is niet praktisch in de gevallen waarin de database reeds bestaat. Een reverse engineering-optie is dus in ieder geval gewenst. Verder laten de meeste interfaces om de database-structuur mee te definiëren, vaak veel te wensen over. Daarom kan men beter een volwassen modellerings-tool gebruiken.
De volledige integratie van een modellerings-tool met een ontwikkel-tool zou natuurlijk ideaal zijn.

Presentatie informatie

Nu het structuur-gedeelte van de repository is gevuld, moet de repository worden aangevuld met basis-informatie over de manier waarop men dingen gepresenteerd wil hebben. Eerst worden de presentatie-eigenschappen van domeinen, kolommen en tabellen gespecificeerd, volgens het 'Single point of definition' (SPOD) principe. De informatie per domein, kolom of tabel hoeft maar één keer ingevoerd te worden. Vervolgens zal deze informatie overal consequent gebruikt worden.



Afbeelding 2: De repository uitgebreid met presentatie-aspecten.

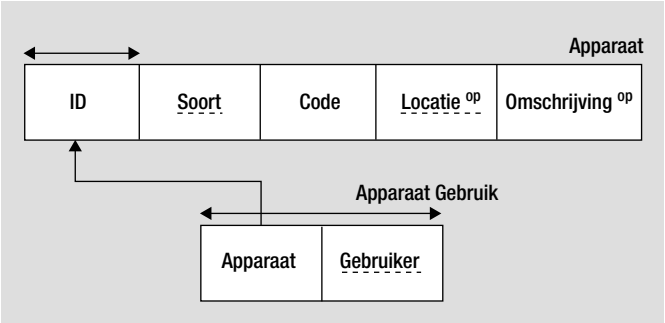
De repository kan op diverse manieren met presentatie-informatie gevuld worden. Het meest praktische is een koppeling met het modellerings-tool. Via een dergelijke koppeling kan veel informatie worden ingelezen en afgeleid. Een andere methode is het met de hand vullen van de repository. Dit lijkt misschien erg omslachtig, maar valt in de praktijk erg mee.
Omdat vrijwel geen enkele ontwikkelomgeving het SPOD-principe goed ondersteunt, moet het voor ieder veld in ieder scherm telkens opnieuw gedaan worden. Het werk wordt beloond met een consequente weergave in ieder scherm dat daarna wordt gemaakt.
De laatste methode om de repository te vullen, is het afleiden van de presentatie-eigenschappen op basis van de structuur en de data. Dit afleiden gebeurt aan de hand van vuistregels. Omdat er sprake is van 'vuistregels' en geen 'wetten', gaat het logischerwijs ook wel eens mis. Gelukkig is dit maar zelden het geval en wegen de voordelen ruimschoots op tegen het ongedaan maken van een enkele misser.

Enkele voorbeelden van vuistregels zijn:
- Zoeken naar bepaalde datatypen en daar eigenschappen aan verbinden, bijvoorbeeld de invoer- en afbeeldingsmaskers voor financiële velden;
- Het automatisch verbergen van de kolom die verwijst naar de master-tabel tijdens het afbeelden van een master-detail-relatie;
- Het herkennen van nummergeneratoren en zorgen dat deze niet te wijzigen zijn;
- Het herkennen van lookup-tabellen en verwijzingen ernaar automatisch omzetten in keuzelijsten;
- Optimale afbeeldingslengten voor velden bepalen op basis van de populatie.

Standaardschermen

Nu kunnen de eerste prototypes van de schermen gemaakt worden. In het meest simpele geval is daar geen enkele regel code voor nodig. Het intypen van de naam van de tabel is voldoende. Als men de apparaat-tabel uit het strokendiagram in afbeelding 3 wil laten afbeelden, resulteert dat in het scherm dat in afbeelding 4 te zien is.
Enige toelichting bij afbeelding 4:
- Verschillende kleuren geven aan of een veld verplicht is;
- Afhankelijk van het soort veld en het bestaan van een lookup-tabel, worden de verschillende interface-componenten voor de velden gekozen. Zo verschijnt bijvoorbeeld voor de kolom 'Omschrijving' automatisch een groot invoerveld;
- Het *plus*-knopje bij de locatie geeft aan dat het niet alleen mogelijk is om een locatie uit de lijst te kiezen, maar dat het ook mogelijk is om vanuit dit scherm een nieuwe locatie te definiëren.
- De knoppen-balk onder de velden geeft van links naar rechts de volgende standaard functionaliteit: zoeken, wisselen tussen tabel- en formulier-weergave, bewerken van gegevens, opslaan, annuleren, toevoegen, verwijderen, vorige apparaat en volgende apparaat;
- In de onderste helft zijn de tabellen afgebeeld die een verwijzing hebben naar de apparaat-tabel. De verwijzende velden zijn automatisch uit het detailscherm verwijderd.

Ook functionaliteit zoals het zoek- en het helpscherm (zie afbeelding 5) hoeven niet te worden geprogrammeerd. De helpteksten kunnen soms zelfs worden gehaald uit het modelleringstool. Door het consequent toepassen van de gegevens in de repository kunnen zelfs foutmeldingen verbeterd worden. Zo verschijnt bij het verwijderen van menig apparaat de melding; "Apparaat kan niet verwijderd worden. De tabel Apparaatgebruik bevat nog gegevens die naar het geselecteerde Apparaat verwijzen". Zowel de afbeeldingsnamen als de relaties uit de repository worden gebruikt om de melding zo duidelijk mogelijk te krijgen. Dit is zeker niet perfect, maar een stuk beter dan: "SQL error 10398. Foreign key violation …".
Tot slot hoeft de gebruiker zich geen zorgen te maken over transacties. Alles wordt volledig automatisch afgehandeld.



Afbeelding 3: Strokendiagram. De onderstreepte kolommen verwijzen naar niet-afgebeelde tabellen.

Prototype

Voor alle bovenstaande beschreven zaken was programmeren niet nodig. Het was allemaal genereerbaar op basis van de structuur, aangevuld met eenvoudige presentatie-informatie. Uiteraard is voor veel schermen meer flexibiliteit nodig. Schermen kunnen daarom eenvoudig worden aangepast aan meer specifieke wensen. Enkele voorbeelden daarvan: niet alle detail-schermen tonen, velden naast elkaar afbeelden, geen zoek mogelijkheid, niet mogen verwijderen, veld toch verplicht, namen aanpassen enzovoort. Dit alles is eenvoudig te configureren.
Hoewel het voor beginnende gebruikers en/of ontwikkelaars ongetwijfeld prettig is om het aanpassen visueel te kunnen doen, wordt er op dit moment met een eenvoudige specificatie-taal gewerkt. Dat gaat in de praktijk namelijk veel sneller dan met een visuele omgeving.

Een voorbeeld van een specificatie-script luidt:

```
<FRAME>
Dataset=Licentie
RecordView=No
Fieldlist=Aantal, Omschrijving

        <ACTION>
        Type=FORM
        Identifier=Sub_Licentie
        Caption=Detail informatie
        Linkfield=Licentie.ID
        Helptext=Toont het licentiescherm voor de
                              geselecteerde licentie
        </ACTION>

</FRAME>
```

Het specificatie-script wordt keurig opgeslagen in de repository. Er is bewust niet gekozen voor een volledige relationele uitwerking van de informatie in het script, omdat dit in de praktijk een te trage en te rigide structuur oplevert.
In tegenstelling tot wat men vaak verwacht, wordt het specificatie-script gewoon 'at runtime' geïnterpreteerd. Er wordt dus geen Java of C++ code gegenereerd. Door optimalisatie van de interpreter wordt een scherm van gemiddelde complexiteit binnen een kwart van een seconde opgebouwd (Gemeten op een Pentium II 300 Mhz machine). De ontwikkel- en testcyclus is zeer kort: script aanpassen, testen, bijwerken, direct opnieuw testen enzovoort. Het gevolg is dat Joint Application Design-sessies, zoals beschreven in RAD en DSDM, nu echt goed mogelijk worden. Een belangrijke voorwaarde hiervoor is immers rapid prototyping en ontwikkeling.

Logica

Menige poging om constraints en gedrag te formaliseren en een goede conceptuele taal te ontwikkelen is gestrand. Dat is de reden geweest dat een zeer pragmatische, maar wel zo conceptueel

Afbeelding 4: Standaardscherm.

mogelijke, oplossing is gekozen. Voor de definitie van constraints, afleidingsregels en andere vormen van gedrag is Pascal in gebruik. Omdat vanuit Pascal eenvoudig met SQL gewerkt kan worden, geeft dit het beste van twee werelden: SQL en 3GL-talen.

```
PROCEDURE Controle_van_geslacht (aField: TField);
   BEGIN
   IF aField.AsString='V'
      THEN BEGIN
         EnableField('MeisjesAchternaam',
                                    frsOptional);
         EnableField('MeisjesVoorvoegsels',
                                    frsOptional);
         END
      ELSE BEGIN
         ClearField('MeisjesAchternaam');
         ClearField('MeisjesVoorvoegsels');
         DisableField('MeisjesAchternaam');
         DisableField('MeisjesVoorvoegsels');
         END;
END;
```

Alle logica wordt ondergebracht in routines die gekoppeld worden aan bepaalde events (in SQL *trigger-momenten* genoemd).
Zo wordt bovenstaande routine gekoppeld aan het OnChange-event van het veld 'Geslacht'. Het zorgt ervoor dat bij vrouwen de mogelijkheid wordt geboden om de meisjesachternaam en het voorvoegsel in te voeren.

Men kan misschien opmerken dat er naar het veld 'MeisjesAchternaam' verwezen wordt in de vorm van een string en niet direct naar een object in de interface. Hierdoor ontstaat de gewenste scheiding tussen de logica en de interface. Het is binnen deze code niet van belang hoe de interface er uit ziet. Het is de taak van de interface generator om te bepalen welk object op het scherm hoort bij 'meisjesachternaam'. Dit zorgt ervoor dat de regel herbruikbaar is voor meerdere schermen en zelfs voor meerdere soorten clients.

Omdat het stukje voorbeeld-code wordt gekoppeld aan het OnChange-event van het Geslacht-veld van de Relatie-tabel, zal het automatisch in ieder scherm toegepast worden waar het Geslacht-veld voorkomt. In feite wordt ook hier het SPOD-principe toegepast. De code wordt zo hoog mogelijk te gedefinieerd in de hiërarchie: domein, kolom, tabel, scherm.

Er is vooral gericht op de applicatielogica en dat is goed gelukt. Het nadeel is dat SQL constraints toch nog met de hand in de database ondergebracht moeten worden. Uiteindelijk zou het wenselijk zijn om te komen tot een mechanisme, waarin alle logica werkelijk maar één keer gespecificeerd wordt. Van daaruit zou de code voor zowel de database, de applicatie-server als de client gegenereerd moeten worden. Dit is niet eenvoudig, er is onder andere een goede conceptuele taal nodig, die zowel vertaald kan worden naar bijvoorbeeld PL-SQL als bijvoorbeeld Java. Ook moet er duidelijk onderscheid gemaakt moet worden in de verschillende soorten logica. Zo zal een stukje logica dat een rapport afbeeldt niet in de database terecht moeten komen.

Een volledige applicatie

Om tot een volwaardige applicatie te komen moet de ontwikkelaar kunnen terugvallen op een compleet framework, waarin allerlei standaard functionaliteit is verwerkt. Hierbij valt te denken aan autorisatie-mechanismen, menu-structuren en rapportage-faciliteiten. Het is onmogelijk om al deze aspecten binnen dit artikel te behandelen. Wel moet gemeld worden dat ook bij deze aspecten het SPOD-principe uitstekend kan worden toegepast en tot aanzienlijke verbeteringen leidt.
In het begin was de sceptische verwachting dat veel schermen waarschijnlijk toch met de hand gemaakt en alleen de standaard onderhoudschermen met het case tool gegenereerd zouden moeten worden. Het case tool is daarom vanaf het begin opgezet als uitbreiding van een bestaande programmeertaal, zodat daarin de 'complexere zaken' ondergebracht konden worden. Uiteindelijk werden alle ruim 300 schermen van het systeem zonder hulp van de achterliggende programmeertaal ontwikkeld!

Afbeelding 5: Zoek- en helpscherm.

> Management van metadata is de kern geworden van de ontwikkeling en niet langer een sluitpost

Conclusie

Hoewel de ideeën over model based development bepaald niet nieuw zijn en er diverse tools bestaan, kan er nog flink gesleuteld worden aan de bruikbaarheid van deze tools. Ook is er nog aardig wat onderzoek te verrichten naar zinvolle uitbreidingen.
Te denken valt aan generatie van applicatie-servers en web interfaces, ondersteuning van building blocks/design patterns en meertaligheid, goede visuele interfaces voor het ontwerp, betere ondersteuning voor subtypes enzovoort.

Het is duidelijk geworden dat model based development tot een drastische vermindering en betere inschatting van de ontwikkeltijd kan leiden; snellere ontwikkeling van prototypen; significante reductie van het aantal bugs; software met een zeer consistente interface; goedkoper en eenvoudiger onderhoud en een zeer leesbare en compacte broncode.
Dit is een hele stapel superlatieven, maar de praktijk wijst uit dat dit niet overdreven is. Natuurlijk is geen enkele methode perfect maar een enorme verbetering is het zeker. Vooral het laatste aspect 'Zeer leesbare en compacte broncode' heeft plezierige consequenties. Doordat de broncode zo kernachtig is, richt deze zich eigenlijk alleen op het oplossen van het vraagstuk en wordt het nauwelijks vertroebeld door technische constructies. Hierdoor is veel minder documentatie nodig om de software te beschrijven en wordt het werk dus makkelijker overdraagbaar. Tot slot is het aardig om te melden dat management van metadata vanzelfsprekend wordt: het is immers de kern geworden van de ontwikkeling en niet langer een sluitpost.

Jeanot Bijpost en Marco de Groot
Ing. J.W Bijpost (jeanot@mattic.nl) en Ing. M.H. de Groot (marco@mattic.nl) zijn beiden werkzaam bij Mattic B.V.