Remia, producent van sauzen, margarines en vetten, maakt sinds 1990 gebruik van modelgebaseerde systeemontwikkeling. In tegenstelling tot veel van haar collega-productiebedrijven geeft Remia systeemontwikkeling een belangrijke plaats bij de invulling van het informatie-beleid. Deze keuze wordt ondersteund door een model-gebaseerde benadering en het gebruik van daarbij passende tools.

Werken met patterns in de praktijk

Deze twee aspecten vormen een belangrijke voorwaarde voor het met succes hanteren van een maatwerk-praktijk voor belangrijke delen van de informatie-infrastructuur. Inmiddels is de aanpak verder doorgegroeid en wordt momenteel veelvuldig gebruik gemaakt van 'patterns' voor het abstraheren van gegevensstructuren en het daaraan gerelateerde gedrag. Hoewel het abstraheren van functionaliteit zijn eigen eisen aan het ontwikkelproces stelt, blijken de voordelen van deze manier van werken in de praktijk groot te zijn.

Advantage/Plex Na een lange traditie van programmatuur-ontwikkeling op basis van IBM Midrange-computers, met RPG als belangrijkste, werd met de selectie van Synon/2E gestart met een moderne model-gebaseerde aanpak. Synon/2E, inmiddels opgenomen in het portfolio van Computer Associates onder de naam Advantage/2E, heeft zijn wortels in de Information Engineering benadering en kent daardoor een zware rol toe aan het gegevensmodel binnen het ontwikkelproces. Na een klein decennium ervaring met dit ontwikkelhulpmiddel, koos Remia eind jaren '90 voor de inzet van een tweede product uit dezelfde 'familie': Advantage/Plex.

Plex is in belangrijke mate gebaseerd op hetzelfde gedachtegoed, met pluspunten op het gebied van platformonafhankelijkheid en daarmee de mogelijkheid van een moderne grafische gebruikersinterface. Hiermee zou dit product een logische keus zijn als uitbreiding van de mogelijkheden van Synon/2E. Toch gaf vooral de mogelijkheid om functionaliteit te abstraheren in 'patterns', en de hieraan gerelateerde winst in snelheid en kwaliteit, de doorslag. In dit artikel willen we ingaan op de manier waarop Plex het ontwikkelen op basis van 'patterns' in de praktijk ondersteunt.

Technische concepten Binnen Advantage/Plex staat het logische gegevensmodel centraal. In dit model worden entiteiten, velden en verschillende verbanden daartussen gedefinieerd, zoals 'owned by' , 'refers to' en 'known by'. Op basis van de modeldefinities genereert Plex met behulp van codegeneratoren tabellen en views voor verschillende platforms. Voor het definiëren van functionaliteit wordt gebruik gemaakt van zogenaamde 'action diagrams', waarin op basis van een krachtige, platformonafhankelijke taal kan worden geprogrammeerd. Vanuit deze action diagrams kan code worden gegenereerd naar bijvoorbeeld: Client/Server iSeries (RPG en C++), Java, C++/ODBC et cetera. Daardoor kunnen we spreken van een snelle ontwikkelomgeving. Plex voegt hier echter nog een tweetal belangrijke zaken aan toe:

- Overerving: de mogelijkheid om modelelementen op abstract niveau te definiëren en zo hergebruik te realiseren
- Meta taal: het gebruik van een abstracte programmeertaal om de generatie op basis van het model te beïnvloeden.

Deze twee onderdelen vormen de basis voor de mogelijkheid 'patterns' te ontwikkelen en toe te passen. Naast deze aspecten biedt Advantage:Plex een multi-user repository, met ondersteuning van dimensies als variant, versie en taal, de mogelijkheid om diagrammen te maken van elementen uit de repository, alsmede de mogelijkheid voor 'impact analyse' et cetera. Omdat het hier gaat om de toepassing van 'patterns' in de concrete ontwikkelpraktijk gaan we hier nu niet op in.

'Patterns' in Plex Het concept pattern is ontstaan op het gebied van software-ontwerp. Bekende auteurs op dit gebied zijn Gamma et al. (the Gang of

Four), David Hay en Martin Fowler. Ieder van hen bood een verschillende invalshoek, maar steeds ging het om ontwerpbeschrijvingen. Plex was een van de eerste commerciële tools die de mogelijkheid bood 'patterns' als constructie-element te benutten. De in de documentatie gehanteerde 'definitie': *a pattern is a solution to a problem in a context*, heeft verwantschap met de in de literatuur gevonden begripsomschrijvingen, maar maakt niet veel duidelijk over het pattern-concept zoals dat in de concrete ontwikkelomgeving wordt ondersteund. Binnen Advantage:Plex is een pattern meestal een combinatie van elementen uit het gegevensmodel en de daarbij behorende functionaliteit. Dit kan variëren van een aantal 'velden' in een entiteit, met de bijbehorende lees- en schrijf-functionaliteit tot en met een structuur waarbinnen verschillende entiteiten aan elkaar zijn gerelateerd met de functionaliteit die nodig is om die onderlinge relatie te ondersteunen.

Advantage:Plex en MDA

Wie vandaag de dag de term 'model-gebaseerd ontwikkelen' gebruikt, roept al snel associaties op met Model Driven Architecture. MDA is een standaard (in ontwikkeling) van de Object Management Group. MDA is gebaseerd op het ontwikkelen van systemen door middel van geautomatiseerde code-generatie vanuit modellen, over het algemeen beschreven in UML. Het wordt gekenmerkt door een drie-lagen opbouw:

- PIM voor Platform Independent Model,
- PSM voor Platform Specific Model en
- het Code Model.

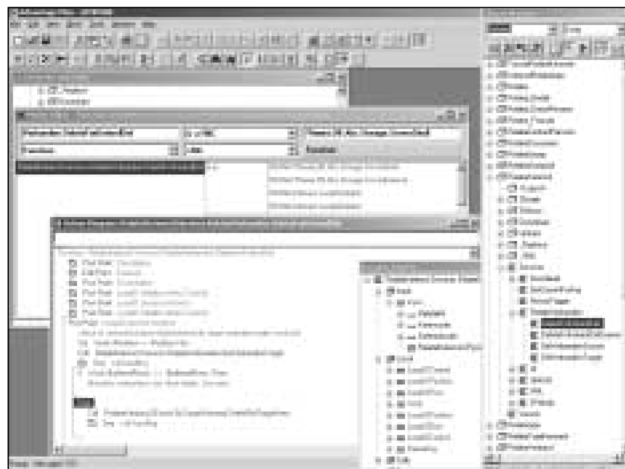
Tussen de modellen vinden transformaties plaats van PIM naar PSM en Code. Deze architectuur beoogt een omgeving te bieden waar verschillende platforms vanuit een 'conceptueel' model kunnen worden geadresseerd. Uiteindelijk moet dit leiden tot bescherming van de model-investering tegen de gevolgen van veranderingen in de onderliggende technologie.

Terwijl het object-georiënteerde UML vrij algemeen beschouwd wordt als de te gebruiken modelleertaal in een MDA-omgeving, is Advantage:Plex niet gebaseerd op een strikt OO paradigma. De concepten die gebruikt worden binnen de Plex-modellen kennen sterke wortels in information engineering. Daardoor is er een sterke nadruk op 'entiteit-modellen'. Wel is vanuit object-oriëntatie het concept overerving toegevoegd en in later instantie zelfs meervoudige overerving ter ondersteuning van het pattern-concept.

Ondanks de verschillen met MDA, kent Advantage:Plex in belangrijke mate dezelfde doelstellingen.

Technologie onafhankelijkheid is ver doorgevoerd binnen Advantage:Plex. Dit gebeurt niet zozeer door gebruik van verschillende modellen, met transformaties daartussen, maar door middel van varianten die in de repository worden vastgelegd. Het genereren van code gebeurt voor 100%, of het nu voor een traditionele omgeving als iSeries 5250 is, voor C++ Clients, ODBC-gebaseerde data-servers of het J2EE platform.

Hoewel er momenteel niet gesproken kan worden van een MDA-tool, speelt Advantage:Plex, voor wat betreft de brede platformondersteuning en de mate waarin code gegenereerd wordt, mee in de voorhoede.



FIGUUR 1. De Plex ontwikkelomgeving

- Een voorbeeld van de eerste soort is de 'audited entity'. Dit pattern voegt velden toe voor 'gebruiker', 'datum' en 'tijd' plus de specifieke routines die nodig zijn om deze gegevens vast te leggen bij mutaties.
- Een voorbeeld van een iets ingewikkelder structuur is bijvoorbeeld een 'parent' - 'child' constructie zoals die vaak in de vorm van 'order' - 'orderregel' terugkomt.

Later in dit artikel volgen nog concrete voorbeelden uit de Remia praktijk. Om patterns te kunnen combineren tot implementeerbare tabellen en functies, is er in Advantage:Plex een overervings-mechanisme ingebouwd dat multiple-inheritance ondersteunt. Door één model-object van een aantal abstracte objecten te laten overerven, zijn de verschillende aspecten van deze 'parents' te combineren. Voor Entiteiten betekent dit bijvoorbeeld dat de velden van alle parents en grandparents in de tabel aanwezig zijn. Op het gebied van functionaliteit (action diagrams) worden overerfde functies, onder voorwaarde van een 'gemeenschappelijke voorouder', in elkaar geweven. Op ieder niveau binnen de hiërarchie kan op van tevoren aangeduide 'user points' nadere detaillering van de functionaliteit plaatsvinden.

De mogelijkheid van meervoudige overerving laat toe om specifieke 'aspecten' van een informatiesysteem separaat te modelleren en naar behoefte te combineren. Bij het combineren van objecten via multiple-inheritance wordt een belangrijke rol vervuld door de meta-programmering. Hiermee is het mogelijk om op het moment van genereren de repository uit te vragen en op basis hiervan de gegenereerde code specifiek te maken voor de concreet gerealiseerde situatie.

DE PRAKTIJK BIJ REMIA Als het gaat om de 'ontdekking' van patterns zijn er steeds twee werkwijzen in het spel, die in de Remia-praktijk beiden zijn toegepast:

- een 'empirische' aanpak, waarbij de constatering dat een bepaald soort functionaliteit voor de tweede of

derde keer ontwikkeld wordt, leidt tot de afweging of een pattern de voorkeur geniet;
- een framework-benadering, waarbij van tevoren een (gelaagde) architectuur en standaard onderdelen worden uitgewerkt.

Als we naar de framework-benadering kijken, is in de eerste plaats de scheiding van database en UI-modellen van belang. Binnen de opzet van het REMIA-framework is een drie-lagen architectuur opgezet, met logisch gegevensmodel, fysiek gegevensmodel met concrete functionaliteit en een user-interface laag. Op basis van deze opzet, die op verschillende punten in belangrijke mate afwijkt van de standaard meegeleverde benadering, zijn een aantal 'technische' patterns opgezet. Te denken valt aan begrippen als 'Entiteit met Tekst' (voor memo-achtige functionaliteit), 'Entiteit met Historie' (voor logging van relevante velden).

HET SYNCHRONISATIEPATTERN Een voorbeeld van een wat groter technisch pattern is het synchronisatiepattern. Binnen de opzet van het Remia-CRM systeem is gekozen voor een replicatiemechanisme ten behoeve van 'lokaal' gebruik. Voor deze functionaliteit is dankbaar gebruik gemaakt van de mogelijkheid om, op basis van één model, zowel een centrale, iSeries gebaseerde Client/Server applicatie te kunnen genereren, als een lokale, ODBC gebaseerde variant. Voor de uitwisseling van gegevens tussen de centrale en lokale applicaties is een combinatie van patterns ontwikkeld, die de uitwisseling van gegevens tussen de twee omgevingen afhandelt.

Het synchronisatie-pattern bestaat op database niveau uit een 'download' en een 'upload'-tabel, waarin de wijzigingen worden vastgehouden. Behalve de 'logging'-functionaliteit is er ook een abstracte distributiefunctie gedefinieerd, die zorg draagt voor het maken van XML-documenten met de gegevens die gesynchroniseerd dienen te worden tussen de centrale en de lokale database. Binnen de centrale applicatievariant is specifieke validatie en concurrency-afhandeling gerealiseerd.

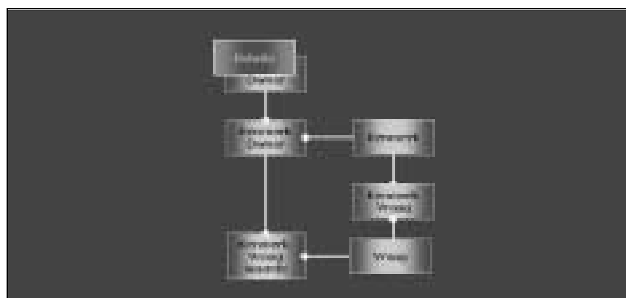
De concrete functie die synchronisatie tussen de beide omgevingen afhandelt, is gemodelleerd als een 'meta-functie' waarmee de 'verwerkingsschil' dynamisch uit het Plex-model wordt gegenereerd. Tijdens het genereren wordt het Plex-model doorlopen om alle entiteiten te bepalen die overerven van de 'Entiteit met Upload'. Voor ieder van die entiteiten wordt vervolgens een 'call' naar de 'eigen' verwerkingsfunctie uitgeschreven. Gezien het grote aantal entiteiten dat voorzien is van deze functionaliteit, blijkt het gebruik van dit pattern een enorm winstpunt.

Patterns met een technisch karakter zijn in de praktijk redelijk gemakkelijk aan te wijzen. In de beschreven situatie gaat het om functionaliteit waarvan vanuit de ontwerpfase is vast te stellen dat de mate van hergebruik hoog zal zijn. Andere voorbeelden van vroegtijdig herkenbare patterns zijn bijvoorbeeld constructies op het gebied van autorisatie of het automatisch bijwerken van cumulatieven.

HET KENMERKENPATTERN Behalve technische patterns zijn er binnen de Remia-praktijk ook verschillende patterns met een meer functioneel karakter ontwikkeld. Een voorbeeld van zo'n pattern uit de Remia-praktijk is de 'Entiteit met kenmerken'.

Voor Remia is het belangrijk dat de informatieverwerking snel kan inspelen op nieuwe ontwikkelingen die binnen of buiten het bedrijf plaatsvinden. Onderdeel van deze aanpak is dat gebruikers in bepaalde situaties zelf kunnen bepalen welke gegevens over een bepaalde entiteit worden vastgelegd. Het flexibel vastleggen van gegevens geldt binnen de relatiebeheersomgeving als algemene strategie en moet op verschillende niveaus en voor verschillende entiteiten kunnen worden ingevoerd. Dit is een uitstekende reden om hiervoor een pattern te ontwikkelen.

GESCHEIDEN LAGEN Voor dit kenmerken-pattern is dataopslag nodig om de kenmerken en configuratie vast te leggen en is een interface nodig die het verwerken van de kenmerken door de gebruiker mogelijk maakt. Binnen het door Remia gedefinieerde 'framework' is de databaselaag gescheiden van de 'user interface'-laag. Dat betekent dus dat het kenmerken-pattern voor beide



FIGUUR 2. Data-opslag in de databaselaag vindt plaats in een aantal aan elkaar gerelateerde tabellen

FIGUUR 3. Voor de user interface is een aantal functies en panels ontworpen

lagen een oplossing moeten bieden. Data opslag in de database-laag vindt plaatst in een aantal aan elkaar gerelateerde tabellen (zie figuur 2). De user interface maakt het configureren van de kenmerken en het invoeren van de uiteindelijke data mogelijk. Voor deze user interface is een aantal functies en panels ontworpen (zie figuur 3). Zo zijn op abstract niveau de data opslag en het gedrag van de entiteit-gerelateerde kenmerken geregeld. Het toepassen van dit abstracte pattern in de realisatie-omgeving bijvoorbeeld voor de entiteit 'Relatie' in de database-laag wordt uitgevoerd door in het model het statement *Relatie is a Entiteit met Kenmerken* in te voeren. Hierdoor worden de kenmerktabellen voor de entiteit 'Relatie' geïnstantieerd en kunnen we voor elke relatie een onbeperkt aantal kenmerken vastleggen.

Voor de user interface dient een aantal functies te worden gecreëerd die overerven van de abstracte functies en die verwijzen naar de relatie-kenmerken. Dit gebeurt met enkele statements waarin de functies worden overerfd en de abstracte database-elementen worden vervangen door de database-elementen van het relatie-kenmerk (zie figuur 4). Het kenmerken-pattern wordt binnen één applicatie op deze manier voor vijf entiteiten gebruikt waaronder ook Document en Persoon. De gebruikersorganisatie kan zo zelf bepalen welke kenmerken voor een bepaalde Relatie, Document of Persoon geregistreerd kunnen worden. De functionaliteit wordt 'aangevuld' door een abstract query-pattern dat de gebruiker in staat stelt op eenvoudige wijze selecties te definiëren over de betreffende entiteiten. Het implementeren van het pattern is in de praktijk eenvoudig. Een aantal model statements volstaat, waarna de Plex-generator de database- en interface-elementen creëert.

FIGUUR 4. Enkele statements waarin de functies worden overerfd en de abstracte database-elementen worden vervangen door die van het relatie-kenmerk

Bij het ontwerpen van abstracte functionaliteit is het denken in 'aspecten' van groot belang

HET COMBINEREN VAN PATTERNS De relatiekenmerken worden veelal door de verkoopmedewerkers off-line in de remote client ingevoerd en dienen vervolgens naar de centrale database te worden verzonden. Hiervoor combineren we binnen de Relatie-entiteit het Kenmerken-pattern met het Synchronisatie-pattern. Door een model statement op te nemen als *Relatie.Kenmerk.KenmerkVraagWaarde is a Entiteit met Upload* is dit eenvoudig te realiseren.

HET COMBOBOX PATTERN Bovenstaande patterns zijn typisch voorbeelden van patterns die in de ontwerpfase zijn herkend. Een voorbeeld van een pattern dat 'empirisch' is ontstaan, vermelden we het combobox pattern. Tijdens de applicatie ontwikkeling ontstond de behoefte om een combobox met data uit de database te vullen. Met wat handwerk is dat binnen de standaardprogrammatuur wel te realiseren. Tijdens verdere ontwikkeling kwam deze behoefte opnieuw aan de orde, nu bovendien met de noodzaak om meervoudige sleutelwaarden in de box op te nemen. Tijd voor een pattern! Het combobox pattern wordt inmiddels veelvuldig toegepast en heeft een grote tijdwinst opgeleverd.

AFSLUITING Bij het ontwerpen van abstracte functionaliteit is het denken in 'aspecten' van groot belang om te komen tot combineerbare structuur en functionaliteit. Hiertoe is het van belang op verstandige wijze om te gaan met de 'aanhechtingspunten' in de patterns, zodat weefsels uiteindelijk gecombineerd kunnen worden en het gewenste effect hebben.

Hoewel (kleinere) technische patterns vaak relatief eenvoudige functionaliteit vertegenwoordigen, biedt de aanwezigheid van een hoeveelheid van deze patterns een prima handvat om te komen tot snelle en kwalitatieve systeemontwikkeling. Daarnaast kunnen ze, zoals het een pattern betaamt, een leidraad bieden bij het ontwerpen van functionaliteit. Het gebruik van grotere, functionele patterns ligt binnen een eindgebruikersomgeving minder voor de hand, maar komt in de praktijk wel degelijk voor. De initiële investering bij dergelijke structuren is uiteraard groter dan die bij kleine patterns, maar dat geldt ook voor de winst die geboekt wordt bij het toepassen ervan.

Willem de Vries, IT Manager Remia C.V.
Simon Jasperse, Senior Software Engineer, Kiboko