Ontwikkelen 'vanuit de database' in het J2EE-tijdperk

Nieuw is niet altijd beter

Toon Koppelaars

In dit artikel wordt het achterliggende gedachtegoed toegelicht dat een rol heeft gespeeld bij het inrichten van de J2EE-ontwikkelstraat bij het Centraal Boekhuis. Het Centraal Boekhuis is een 'Oracle (PL/SQL) bolwerk', zaken als database vendor-onafhankelijkheid en/of open source databases zijn hier niet aan de orde.

Bij het Centraal Boekhuis ondersteunt men zo goed als alle bedrijfsprocessen met op maat gemaakte client-server applicaties. De benadering bij de client-server applicatieontwikkeling is altijd geweest om daar waar mogelijk de business-logica in de database, en niet in de frontend (Forms) modules te implementeren. Achterliggende reden hiervan is dat (centrale) PL/SQL-programmatuur in de database beter beheerbaar is. Enkele kerngetallen van deze applicatie 'software-plas', die in de jaren negentig is ontwikkeld, en momenteel in beheer is: 1.3 miljoen regels PL/SQL code in de database (ongeveer 15 duizend objecten) en rond de 1500 Forms modules. Sinds 2002 is ook J2EE in gebruik.

> Afhankelijk van het resultaat van de calls zal de user interface anders opgebouwd worden

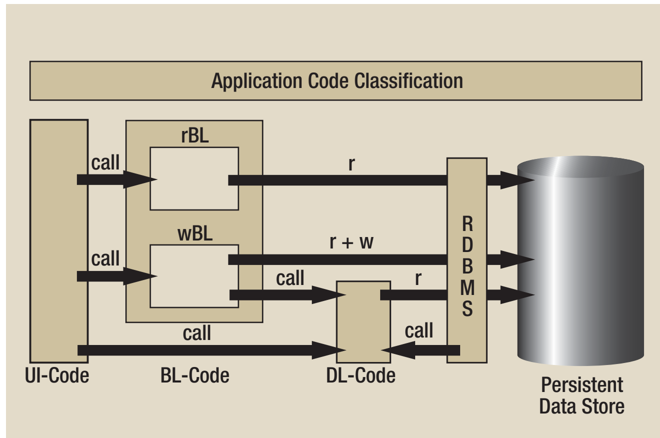
Een J2EE-applicatie ontwikkelen in Java is iets heel anders vergeleken met het toch wel relatief makkelijke en snel te leren PL/SQL. Verder zijn er ook vele nieuwe technologieën die men zich eigen moet maken alvorens met Java een applicatie te gaan ontwikkelen. Er zijn 'frameworks' die men inzet om het Model-View-Control (MVC) design pattern te implementeren; HTTP als het stateless-protocol (en de consequenties die dat heeft); HTML als de presentatietaal van de browser, inclusief Javascript; en natuurlijk een Java ontwikkel-tool, Oracle's JDeveloper in dit geval.

Overstap

Een traditionele client/server-ontwikkelaar die de overstap maakt naar Java/J2EE zal de eerste maanden, druk lerende en zich van alles eigen makende, tevens ontdekken dat het kennisgebied van wat nog onbekend is, alleen maar groeiende is. De vraag doemt op of men ooit alles onder de knie (en paraat tussen de oren) krijgt, om productief een applicatie te ontwikkelen in de Java/J2EE-wereld.
Voor iemand met een gedegen PL/SQL-achtergrond zijn er manieren om deze, voor het gevoel immense, complexiteit hanteerbaar te maken. De meest succesvolle manier is om ook de Java/J2EE-wereld, net als de client/server-wereld, te benaderen 'vanuit de database'.
Oracle's database server is de afgelopen jaren enorm verrijkt met talloze features op het gebied van applicatieontwikkeling (zowel op SQL- als op PL/SQL-gebied). Nieuwe data-typen, inline views, case-expressies, virtual private database, instead-of triggers, updateable views, ref cursoren en pipelined table functions, maken het, naast de al bekende zaken als functions, procedures, packages en triggers, mogelijk om tegenwoordig het merendeel van de business-logica in de database server te implementeren. Door veel code in PL/SQL te implementeren, is de ontwikkelinspanning die overblijft in de Java/J2EE-laag bevattelijk te maken. Zo'n databas-centrische benadering zal niet alleen het risico in een eerste Java/J2EE-project aanzienlijk verlagen, maar stelt tevens in staat om zelf op de 'driving-seat' te gaan zitten, en je zelf niet helemaal over te leveren aan externe Java-programmeurs.

Classificeren van applicatie-code

De hamvraag is: welke code implementeren we waar? In het J2EE-tijdperk is er naast de bekende database-omgeving en haar PL/SQL-taal, een middle tier-omgeving met Java als programmeertaal bijgekomen. Om deze vraag enigszins gestructuureerd te kunnen beantwoorden moeten we applicatie-code classificeren, zodat we daarna per klasse een uitspraak kunnen doen over waar dat soort code gebouwd moet worden.
Bij applicatieontwikkeling is onderscheid te maken tussen de volgende klassen code: Data retrieval- en manipulatiecode; User interface-code; Business-logicacode; Data-logicacode.
Data retrieval- en manipulatiecode is de code die een rechtstreeks interface heeft met de persistent data store. Alle query's (SQL-select) en manipulaties (SQL-update, insert, delete) vallen in deze klasse. Dit zijn dus feitelijk alle embedded SQL-statements die in code van de drie andere klassen zitten.
User interface-code (UI-code) is de code die verantwoordelijk is



Afbeelding 1: De code-klassen en hun onderlinge interactie.

voor het produceren van de 'look' van de applicatie (de user interface), inclusief code die reageert op user interface events, wat weer tot aanpassing van de user interface kan leiden. UI-code creëert de voorkant van de applicatie die we aan de eindgebruiker presenteren en waarmee deze met de applicatie kan interacteren en reageert op deze interacties.
Vaak zal dit het creëren en/of aanpassen van de user interface calls naar business-logicacode (zie hierna) vereisen. Afhankelijk van het resultaat van deze calls zal de user interface anders opgebouwd of aangepast worden. Deze embedded business-logica calls in user interface-code, vallen buiten de UI-codeklasse.
Zij behoren tot de business-logicaklasse en maken onderdeel uit van de API die deze klasse aan de UI-code klasse biedt.
Voorbeeld: in een browser-applicatie klikt de gebruiker op een tabblad dat niet actief is. UI-code bepaalt de impact hiervan op de user interface en voert de code uit die de nieuwe HTML-tekst aanmaakt voor de browser.

De API-calls die elke page van een web-applicatie nodig heeft, vormen feitelijk het contract voor die page met de BL-code

Business-logicacode (BL-code) is onder te verdelen in twee subklassen. Code die transacties samenstelt en uitvoert en code die query's samenstelt en uitvoert.
Query samenstellende/uitvoerende code is de procedurele code rondom data retrieval-code. Deze code is verantwoordelijk voor het samenstellen van uit te voeren Select statements, of het conditioneel bepalen van welke Select statements uitgevoerd moeten gaan worden. Het daadwerkelijk uitvoeren van de query wordt ook door deze code geïnitieerd, inclusief het verwerken van de records die door de query geretourneerd worden. We noemen deze codeklasse read-BL-code (rBL-code). Voorbeeld: in een browser-applicatie voert de gebruiker zoekcriteria in en drukt vervolgens op een 'zoeken'-knop. rBL-code stelt met behulp van de criteria een query samen en submit deze aan het RDBMS.
De geretourneerd records worden ergens in het geheugen gezet, waar UI-code ze vervolgens kan oppakken.
Transactie samenstellende/uitvoerende code is procedurele code rondom data-manipulatiecode. Deze code is verantwoordelijk voor het samenstellen van een (aantal) DML statement(s), of het conditioneel bepalen welke DML statements uitgevoerd moeten gaan worden. Het daadwerkelijk uitvoeren van deze statements wordt ook door deze code geïnitieerd. Afhankelijk van de return code(s) die het RDBMS hierop retourneert, zal deze code de transactie ook 'committen' danwel 'rollbacken'. Het uitvoeren van data retrieval-code zal vaak deel uitmaken van transactie samenstellende code. We noemen deze codeklasse write-BL-code (wBL-code). Voorbeeld: in een browser-applicatie heeft de gebruiker een nieuw record ingetypt, en drukt vervolgens op de 'bewaar'-knop. wBL-code zal het benodigde DML statement samenstellen om het nieuwe record in de persistent data store op te slaan, en ook uitvoeren.

Alle code die in de klasse data-logica (zie hieronder) valt, behoort expliciet niet tot de zojuist behandelde BL-codeklasse.
Data-logicacode (DL-code) is code die betrekking heeft op de validatie van data-integriteitsregels (vaak ook Business Rules genoemd). Data-logicacode is verantwoordelijk voor het bewaken van alle regels (constraints) die van toepassing zijn op de inhoud van de tabellen in de persistent data store. We beschouwen deze code expliciet als een aparte klasse, die separaat staat van de hierboven genoemde BL-code. Elke keer als wBL-code een transactie uitvoert, kunnen de data-manipulatie statements hiervan potentieel data-integriteitsregels schenden. Als dit het geval is, dan is het taak van DL-code om ervoor te zorgen dat zo'n transactie niet kan 'committen' en geforceerd wordt om te 'rollbacken'.
Om te kunnen bepalen of de data-manipulatie van een transactie toegestaan is, zal DL-code vaak op haar beurt data retrieval-code moeten uitvoeren (zie literatuur 2) voor een uitgebreide behandeling van data-integriteitsregels in het algemeen, en het ontwerpen van solide DL-code om ze te bewaken. Daar introduceren we ook een subclassificatie voor data-integriteitsregels die één-op-één toepasbaar is op de DL-code die deze regels moet valideren.
Hierbij kijken we naar de scope van data waarop DL-code data retrieval moet uitvoeren om de betreffende regel te bewaken:

1. Scope is slechts één column waarde (attribuut-regels);
2. Scope is meer dan één column waarde welliswaar binnen één record (tupel-regels);
3. Scope is meerdere records, weliswaar binnen één tabel (tabel-regels);
4. Scope is meerdere tabellen (database-regels).

Bij bovenstaande vier leest DL-code altijd alleen maar de door de transactie nieuw gecreëerde waarden. In de vijfde subklasse zitten

de regels waarvoor zowel oude als nieuwe waarden in de persistent data store gelezen moeten worden.

5. Scope is zowel waarden (column, record, tabel) uit de oude database-toestand, als waarden uit de nieuw gecreëerde database-toestand (transactie-regels).

DL-code voorbeeld: in een browser-applicatie voert de gebruiker een nieuw record in en klikt op de 'bewaar'-knop. WBL-code voert een insert statement uit. Het RDBMS voert een pre-insert table trigger uit die op haar beurt een stuk DL-code uitvoert om betrokken regels te valideren.
Van de vier hierboven geïntroduceerde codeklassen, zijn de laaste drie volledig gescheiden van elkaar, en vormen tezamen het volledige spectrum aan applicatiecode. De eerste klasse, data retrieval- en manipulatiecode, is *embedded* in (en daarmee automatisch onderdeel van) ofwel BL-code ofwel DL-code.
We veronderstellen dat UI-code nooit direct data retrieval- of manipulatiecode uitvoert, maar dat altijd delegeert aan BL-code. UI-code en BL-code kunnen DL-code aanroepen, om 'van te voren' de integriteit van gegevens die 'getransact' gaan worden, te checken. Ook het RDBMS kan DL-code aanroepen (vanuit database triggers), om 'achteraf' integriteit te checken.

Afbeelding 1 verduidelijkt een en ander visueel. Let wel dat het geheel aan code van een applicatie nooit zo volledig gescheiden is als hier gevisualiseerd. Bijvoorbeeld: in UI-code zullen embedded BL-code calls of misschien ook wel DL-code calls zitten. Tevens zal wBL-code typisch embedded DL-code calls bevatten. Deze embedded code-modules van een bepaalde klasse A in de code van een andere klasse B, kunnen we beschouwen als de API-modules van klasse A. Om het geheel aan applicatie-code beheerbaar te houden (en het bekende 'spaghetti' te voorkomen) is het belangrijk om heldere afspraken te maken over met name deze API's.

Code executie-omgevingen

Terug naar de vraag 'Welke code implementeren we waar?'
Met de gegeven classificatie zijn we nu in staat om over code te praten. Voordat deze vraag beantwoord kan worden, volgt eerst een overzicht van de executie-omgevingen die in het J2EE-tijdperk ter beschikking staan. Laten we ons beperken tot browser HTML-gebasedeerde applicaties, dan zijn de volgende omgevingen te onderscheiden:

1. Het (Oracle) RDBMS. Data retrieval- en manipulatie-executie vindt hier plaats (dat is een gegeven). Tevens kan men hier BL-code en DL-code implementeren in de PL/SQL taal.
2. De middle tier web- en/of application-containers. Hier kan men in de Java-taal UI-code (web-container), BL-code, en DL-code implementeren.
3. De browser. Hier kan men JavaScript-code uitvoeren om UI-code, BL-code, en DL-code te implementeren (alle sterk af te raden trouwens).

Nu we zicht hebben op een code-classificatie en de executie-omgevingen die ons ter beschikking staan, kunnen we de oorspronkelijke vraag gaan beantwoorden, met name worden database-centrische mappings bekeken. En aangezien we ons beperkt hebben tot browser-/HTML-applicaties zullen we niet kijken naar een UI-code mapping. Die staat vast in de eigen context: UI-code executeert in de Java-omgeving van de web-container, gebruikmakend van een View framework en Controller framework (dit is in een paper nader uitgewerkt, zie literatuur 3). Voor BL-code en DL-code zijn er verschillende mapping-mogelijkheden. Deze worden eerst toegelicht, en daarna worden de keuzen die voor de J2EE-ontwikkelstraat zijn genomen gepresenteerd.

	JavaScript	Java in Container	PL/SQL in RDBMS
1		Alle DL-code	
2			Alle DL-code
3		Deels	Deels

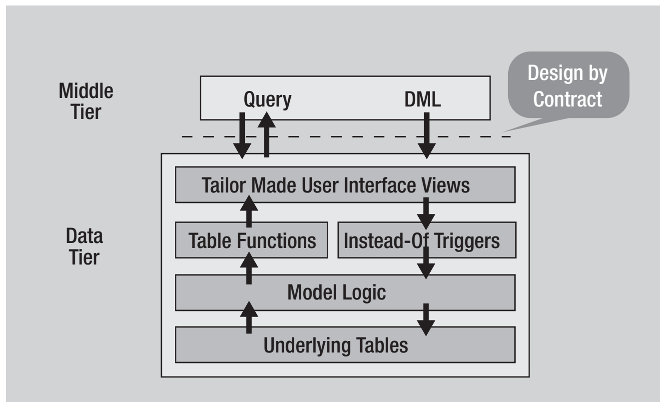
Afbeelding 2: Alternatieve DL-Code Mappings.

DL-Code Mapping

In hoofdlijnen zijn er drie opties voor het positioneren van DL-code: alles in de middle tier, alles in de data tier, of we verdelen afhankelijk van bepaalde criteria de DL-code over voornoemde tiers, zie afbeelding 2.
De J2EE-wereld biedt de Java-programmeur zogeheten model frameworks voor het implementeren van DL-code. Oracle's BC4J framework is zo'n model framework. BC4J heeft diverse features waarmee DL-code voor alle data-integriteitregels geïmplementeerd kan worden. De manier waarop dit gaat lijkt erg veel op hoe dat voorheen, in een client/server-omgeving, gefaciliteerd werd binnen de client-ontwikkelomgeving. Er is echter één groot verschil: waar in een (vroege) client/server-omgeving DL-code voor integriteitregel x, geïmplementeerd moest worden in potentieel meer dan één client-module, is in de J2EE-omgeving met een model framework dit nooit het geval.
De DL-code voor regel x zit eenmalig in het model framework en kan aangesproken worden door de diverse BL-code modules die daarvoor zitten. Latere client/server-implementaties hebben dit nadeel niet meer. Solide (PL/SQL) frameworks als RuleFrame

	JavaScript	Java in Container	PL/SQL in RDBMS
1		Alle BL-code	
2			Alle BL-code
3		Deels	Deels

Afbeelding 3: Alternatieve BL-Code Mappings.



Afbeelding 4: De 'dikke' database met alle business-logica.

faciliteren ook daar het eenmalig centraal implementeren van DL-code: weliswaar in de data tier dan. In beide alternatieven, model framework op de middle tier of rule framework in de data tier, is alle DL-code separaat van andere applicatie-code (UI en BL) geïmplementeerd. Dit heeft in de totale life cycle van de applicatie enorme voordelen op het gebied van beheer en onderhoud.

In het derde alternatief, een deel van de DL-code in de middle tier en een deel ervan in de data tier, zijn vooraf vastgestelde richtlijnen, aan de hand waarvan de programmeurs kunnen afleiden wanneer iets in Java en wanneer iets in PL/SQL gebouwd moet worden, van enorm belang. Voorbeelden van deze richtlijnen zijn: alle DL-code die declaratief binnen het RDBMS vastgelegd kan worden (primary/unique keys, foreign keys, check-clauses), om die in de data tier te doen, en de rest, waarvoor code gebouwd moet worden om die in het Java framework te doen.

Een andere strategie zou kunnen zijn om alle DL-code die data retrieval vereist (die query's moet uitvoeren om de regel te checken) om die in het PL/SQL framework te doen, en de rest (die geen retrieval nodig hebben; code voor attribuut- en tupel-regels) om die in het Java framework te doen. Nog een ander alternatief zou kunnen zijn om DL-code voor bepaalde regels op meer dan één tier te implementeren (dubbel dus).

Mapping BL-Code

Het is een trend om alle BL-code in Java te implementeren. Nieuwe modelling tools gebaseerd op UML en het 'by default' object oriented ontwerpen van de applicatie-logica (waarbij persistent data storage een afgeleide wordt), zijn daarbij sterke drijfveren. Het onderscheid tussen DL- en BL-code wordt vaak ook niet ontworpen. "Een relationeel datamodel; wat is dat?" luidt de repliek op een vraag of een onderliggend data-model en de bijbehorende integriteitregels toegelicht kunnen worden. Wat hierbij geheel genegeerd wordt is dat het merendeel van de applicaties die we bouwen, feitelijk 'window-on-data' type applicaties zijn. Voor dit soort type applicaties blijft het ontwerpen van een solide relationeel data-model, en vanuit daar bottum-up ontwerpen van business-logica, de meest geëigende route. Blijven we ook opteren om business-logica in het RDBMS te implementeren, dan hebben we precies dezelfde opties voor het mappen van BL-code over de beschikbare tiers als bij DL-code, zie afbeelding 3. Model frameworks zoals BC4J assisteren in het samenstellen van de concreet uit te voeren data retrieval- en/of manipulatie-statements. De conditionele logica die hier omheen zit zal natuurlijk zelf geschreven moeten worden, wat resulteert in Java classes buiten zo'n framework (die komen 'ervoor' te zitten). Getriggerd door events in de user interface zal UI-code deze classes aanroepen. Uiteindelijk resulteert dat dan, via het model framework, in de query's en DML-statements voor uitvoering in het RDBMS. Er zit in dit scenario geen BL-code in het RDBMS. Er zijn echter ook manieren om zo goed als alle BL-code in het RDBMS te implementeren. Hoe men zoiets doet hangt vooral af van de inrichting van de API tussen UI-code en BL-code. Meer concreet: is die gebaseerd op SQL calls, of op procedure calls?

Database-centrische benadering: DL-Code

Het RDBMS is de 'final frontier' als het op data-integriteit aankomt. Centrale DL-code in het RDBMS (en wel los van BL-code) waarborgt de integriteit van de data, ongeacht welke applicaties ontwikkeld worden bovenop de tabellen, en in wat voor een omgeving (Oracle*Forms gisteren, J2EE vandaag, en .Net morgen). Het zal geen verrassing zijn dat dit ook de keuze is die bij het Centraal Boekhuis genomen is. Momenteel zit men nog met veel legacy wBL (PL/SQL) code met daarin embedded de nodige DL-code. Bij alle verbouw en nieuwbouw echter stoppen we de DL-code achter database triggers (hiervoor is een eigen framework ontwikkeld: RuleGen), en is er de visie dat BL-code geen embedded DL-code meer zal bevatten. Voor slechts twee typen regels mag nog DL-code elders geïmplementeerd worden: attribuut- en tupel-regels. DL-code hiervoor hoeft immers geen data retrieval uit te voeren, en is vrij simpel in de client tier (door middel van Javascript) te dupliceren. Wij implementeren dus geen DL-code in de middle tier. Naast het eerder genoemde 'final frontier' argument, is de rationale hierachter dat DL-code ofwel data retrieval uitvoert, ofwel niet (attribuut- en tupel-regels). In het eerste geval is het efficiënter (lees: performt het beter) om die DL-code in het RDBMS te positioneren, in het tweede geval is het gebruiksvriendelijker om het in de client tier te positioneren (eigenlijk: kopiëren). Ergo, er blijft geen DL-code over voor de middle tier.

Database-centrische benadering: BL-Code

Wij implementeren alle BL-code in het RDBMS. Eerder is vermeld dat de API tussen de UI-code en de BL-code op twee fundamenteel verschillende manieren ingevuld kan worden:

- API van UI-Code naar BL-Code is gebaseerd op *PL/SQL procedure calls*. In dit scenario is alle BL-code geïmplementeerd in de betreffende stored procedure die door de UI-code wordt aangeroepen. Context van de user interface kan rechtstreeks door middel van parametrisering van deze procedure

meegegeven worden. De PL/SQL code componeert dan op basis hiervan de transactie/query en voert deze uit;
- API van UI-Code naar BL-Code is gebaseerd op *SQL-statements*. In dit scenario moeten we een API-laag van views introduceren om business-logica in het RDBMS te krijgen. Vanuit de gedachte dat de applicatie een 'window-on-data'-achtige applicatie is, creëren we nu een view per applicatie-page. De data retrieval en/of manipulatie die de betreffende page aan de gebruiker biedt, wordt dan helemaal door de view afgehandeld. Door de view één-op-één te baseren op een *pipelined table function* kan alle rBL-code als PL/SQL in deze table function uitgecodeerd worden. Door vervolgens de view te voorzien van instead-of insert/update/delete triggers kan alle wBL-code in deze triggers uitgecodeerd worden.
  Om context van de user-interface naar de pipelined table function en triggers te krijgen, kunnen we gebruik maken van de applicatie context feature (dit is in een paper nader uitgewerkt, zie literatuur 3).

Procedure calls worden ook ondersteund, maar daarvoor moet zelf in het framework geprogrammeerd worden

Wij hebben gekozen voor SQL-statements als API tussen UI-code en BL-code. De belangrijkste reden hiervoor is geweest dat een model framework (zoals BC4J) 'out-of-the-box' via SQL-statements communiceert met het RDBMS. Procedure calls worden ook ondersteund, maar daarvoor moet zelf in het framework geprogrammeerd worden.

De BL-code calls die embedded in de UI-code zitten, zijn nu feitelijk de enige business-logica die zich buiten het RDBMS bevindt. In ons geval zijn dit dus concrete SQL query- of DML statement-teksten, die als API-calls dienen naar de BL-code. Nu kan een query- of DML statement-tekst op zich heel complex zijn. Een query kan bijvoorbeeld een join betreffen met subquery's in de where-clause en function calls in de select-list. Zo'n constructie betreft feitelijk veel business-logica, die embedded in UI-code, en dus buiten het RDBMS, zit. Wij zijn nog een stap verder gegaan en leggen restricties op deze embedded SQL-teksten. Deze restricties lijken heel ver te gaan, maar zijn in de praktijk geenszins beperkend in de mogelijk te bouwen functionaliteit in een applicatie.

Query's:
- zijn altijd gebaseerd op één view (FROM-clause): alle additionele logica, joins, subquery's etcetera zijn uitgecodeerd in de onderliggende table function van de view;
- selecteren (SELECT-clause) altijd kale kolommen van de view: formatteren (upper, lower, initcap etcetera) van waarden in de API SQL-tekst is niet toegestaan, dit dient in de view-text (of liever nog, in de table function) te gebeuren;
- hebben nooit een WHERE-clause (!): door middel van de applicatie context feature wordt user interface context beschikbaar gesteld aan de table function, die vervolgens gebruikmakend van ref cursoren en/of dynamic SQL at runtime where-clauses kan berekenen en toepassen.

Hiermee verkrijgen we het volgende template voor een rBL-code API call in UI-code:

```
SELECT <kale kolommen>
FROM <view>;
```

DML-statements:
- zijn altijd gebaseerd op een view;
- manipuleren altijd slechts één record (via een primary key predicate in de WHERE-clause).

Hiermee verkrijgen we de volgende templates voor wBL-code API calls in UI-code:

```
UPDATE <view> SET <column>=<value>,
<column>=<value>, ...
WHERE <primary key>=<value>;
DELETE FROM <view>
WHERE <primary key>=<value>;
INSERT INTO <view> VALUES(<value>,<value>,...);
```

De instead-of triggers op de API views interpreteren het single record DML statement en vertalen dit naar potentieel complexe DML statements over mogelijk meerdere tabellen.

De API-calls die elke page van een web-applicatie nodig heeft, vormen feitelijk het contract voor die page met de BL-code.
Het ontwerp van deze contracten verzorgt automatisch de taakverdeling tussen de PL/SQL- en de Java-programmeurs gedurende de bouwfase. Tevens kan door zo'n 'design by contract' de BL-interface geheel los van enige Java-code al doorgetest worden, zie afbeelding 4.

UI-Code

Zoals eerder vermeld wordt alle UI-code in de Java-executie-omgeving van de web-container geïmplementeerd met gebruikmaking van het UNIX framework en onze eigen controller.
Deze code is nu wel 'ultradun' geworden. Er wordt HTML-code gegenereerd met daarin embedded query-resultaten van een rBL API-object, en/of HTML-code aangepast naar aanleiding van het resultaat van een wBL API object call. DL- of BL-code-executie vindt niet meer plaats in de middle tier.

Conclusies

Als men, zittend in een PL/SQL-omgeving en solide expertise op dat gebied in huis hebbend, van plan is om de Java/J2EE-wereld

te betreden, dan is de hier beschreven ‘vanuit de database’ ontwikkelgedachte er een die deze stap minder risicovol maakt. Let wel, het zal nog steeds niet geheel zonder risico zijn, want ook met de hierboven beschreven ontwikkelmethode is er nog steeds een hele nieuwe wereld die ontdekt en geleerd moet worden. Een database-centrische benadering kan de risico’s wel een stuk kleiner maken.

Het grootste voordeel echter zit bij de TCO van de Java/J2EE-applicatie. Het verleden leert dat slechts 20 procent van de TCO van een applicatie toegewezen kan worden aan het initieel bouwen ervan. Het restant van 80 procent zit in de beheer- en onderhoudfase van de applicatie. Als men veel PL/SQL-expertise in huis heeft, zal het beheren van een J2EE-applicatie die ‘vanuit de database’ is ontwikkeld, veel goedkoper zijn, dan het beheren van een pure Java-applicatie.

Literatuur

1. *Koppelaars, 2002, A First JDeveloper Project: Choices Made. Lessons Learned. Proceedings van het Oracle Openworld congres 2002, San Francisco.*
2. *Koppelaars, 2003, Data Integrity Rules: Theory and Implementation (What and How). Proceedings van het Business Rules Symposium ODTUG-2003, Miami.*
3. *Koppelaars, 2004, A Database Centric Approach to J2EE Application Development (Where should code go: Java or PL/SQL?). Proceedings van de ODTUG-NOW! conference 2004, Scottsdale.*

Op http://web.inter.nl.net/users/t.koppelaars zijn deze papers beschikbaar.

Toon Koppelaars

Ir. Toon Koppelaars (t.koppelaars@centraal.boekhuis.nl) is Senior IT-architect bij het Centraal Boekhuis te Culemborg.