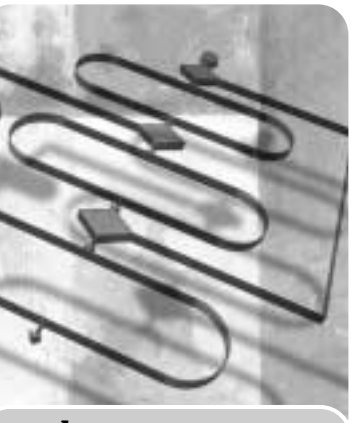

thema

Alle dure woorden ten spijt: "Object georiënteerde analyse en ontwerp", "Aspect Oriented Programming", "Separation of Concerns"... Als de objecten echt het magnetisch veld van de harde schijf raken gaat dat nog steeds via 'the good old' RDBMS. Aan de dure slogans komt dan snel het staartje van de dure licenties: "Object Relational Mapping Framework", of in de J2EE context "Container Managed Persistency". Marc Portier, mede-oprichter van Outerthought, toont in dit artikel hoe de combinatie van enkele vrij beschikbare tools en raamwerken een developer toelaat het veelkoppige beest zelf te temmen.

Je eigen O/R mapping onder controle

Inzet van open source frameworks

"Tijdens de cursus klopte het toch allemaal." Zo voelt het OOAD gebeuren wel meer aan als we weer 'thuis' aan de tekentafel van het volgende project plaatsnemen. Use Cases, Class Diagrams en natuurlijk: Patterns alstublieft. Van de gezonde ontkoppeling en reuse achter factory's en interfaces blijft vaak weinig over zodra er ook werkelijk een GUI en een relationele database aan te pas komt. Dan is het moeilijk vechten tegen de automatismen van onze client-server historie: inputvelden knippen en plakken in sql statements[1] enerzijds en anderzijds invulschermen die er niet in slagen de echo van de onderliggende relationele opslag achter zich te laten. De resulterende omgevingen doen met recht en reden vragen of die objectgeoriënteerde beweging nu wel heeft opgebracht wat men ervan verwachtte?

PRAGMATISCHE WEG Objectoriëntatie is dan ook veel meer dan de theorie en de talloze eye-openers die men in de workshops weet door te geven. Het is ook *echt* bruikbaar, én het werkt in de praktijk. De onderliggende voorwaarde 'het gewoon doen' hoeft dan ook geen onmogelijke drempels op te werpen, of in de praktijk van bestaande systemen niet overeind te blijven. In die praktijk merken we dat bestaande relationele databases in veel organisaties het struikelblok bij uitstek vormt. Zonder de pretentie te hebben alle problemen in detail aan te pakken introduceert dit artikel een pragmatische weg om de Object en RDBMS werelden te verenigen op een nette en onderhoudbare manier.

Om de gedachten te vestigen nemen we het volgende eenvoudige domeinmodel (zie figuur 1) zoals het na het initiële analysewerk naar boven komt. De eenvoudige use cases die hiervan de grondslag zijn, kunnen we snel opnieuw ontdekken: (UC-1) Personen kunnen zich registreren voor een Event, maar uiteraard ook naar hun lijst van registraties teruggaan om een (UC-2) eventuele registratie ongedaan te maken. Tenslotte wil de organisator van de events ook in staat zijn (bijvoorbeeld om een nieuwsbrief uit te sturen) om een (UC-3) lijst te kunnen krijgen van alle ingeschreven bezoekers van een gegeven gebeurtenis.

DATABASE BESLOMMERING Zelfs een eenvoudig model als dit brengt al meteen een aantal scherpe kantjes naar boven in het raakvlak van objecten en relationele databases. Het opgebouwde analysemodel trekt zich niets aan van de praktische database beslommering om ook een primary key te voorzien. Toegeven aan de spontane neiging om één van de beschikbare velden (met een domein-betekenis, zoals 'e-mail' voor Persoon) ook te laten fungeren als primary key wordt in database design doorgaans afgeraden. Die druk vertalen naar

1 Vooral in beveiligde web applicaties is deze techniek sterk af te raden, een recente golf van 'sql spoofing' heeft op pijnlijke wijze de gevolgen aangetoond van authenticatie-testen van de vorm: [select 1 from usr_table where user=' + user + ' and passwd=' + passwd + '] (vul in: user=[ikke] passwd=[' or 'A'='A])

een motivatie om de primary key dan maar ineens in het object model op te nemen is helaas even bevreemdend. Vervolgens verbergt de relatie tussen de Persoon en Event objecten de klassieke tussentabel die de database implementatie zal aanwenden om de relaties te bewaren. Ook al slagen we er in dit geval netjes in de relatietabel een zinvolle naam te geven - 'registration' klinkt mooier dan het obligate 'persoon_event_linktable' - voor de OO ontwikkelaar is het een vreemde druk die hem zonder use cases omtrent bijvoorbeeld op te volgen betalingen per registratie ook een Registratie object laat voorzien. Tenslotte kijkt de OO ontwikkelaar met enige weerzin naar de verwachte bidirectionele relatie tussen de Persoon en Event. De beperkingen in het éénrichtingsverkeer aangeboden door een objectpointer dwingen hem dan vermoedelijk tijdens implementatie toch op het pad van de Registratie-class. Een oplossing die hem laat profiteren van het gemak waarop de wederzijdse foreign-key relatie op het relationele niveau wordt gerealiseerd zou hem ten zeerste welkom zijn.

OPEN RAAMWERKEN Tot zover de voetangels en schietgeweren. Enig bewustzijn omtrent de aangehaalde valkuilen is uiteraard nuttig als algemene leidraad, de concrete oplossing zoeken we in de aanwending van de volgende open raamwerken: Hibernate, Apache Ant en xDoclet. Dit combo laat ons toe op een vlot werkbare manier de nodige implementatie details in te vullen door zo min mogelijk af te wijken van de gezonde (pure) OO praktijk.

Die laatste start met een duidelijke aflijning van de interfaces(1), die in dit geval rechtstreeks volgt uit het domeinmodel. Het lijkt er in dit geval een beetje om gedaan natuurlijk, maar de weg omkerend is het meestal een goede lakmoesproef voor de helderheid van de interfaces: "Blijven ze overeind wanneer je ze terugkoppelt naar domeinspecialisten, of zijn ze al zo vergeven van implementatiedetails dat die terugkoppeling enkel leidt tot verbazing en totale verwarring?"

Technisch slaagt een Java interface er trouwens zeer goed in door middel van gedeclareerde getters and setters of 'properties' de realisatie van verwachte businessvelden of 'attributen' af te dwingen. Door de dualiteit tussen de get- en de set- variant slaagt ze er zelfs in een stuk algemene lees-schrijf rechten op een logische manier in het model in te brengen (iets wat het bewaken van nodige invarianten zeker ten goede kan komen). Het eenvoudige voorbeeld van de Persoon eruit lichtend krijgen we dan (zie codevoorbeeld 1).

JAVABEAN Voor de besproken use cases moet dit als interface volstaan. Alle uitbreidingen hierop zijn typisch ingegeven door implementatiedetails en vaak van het type 'just because we can'. Elke toevoeging is een uitnodiging tot grotere afhankelijkheid tussen de verschillende implementaties en als regel dan ook stellig te vermijden. Ook het opvolgen van de zogenaamde 'te voorziene toekomstige use cases' is een reflex die men in het huidige OO denken eerder overlaat aan toekomstige refactoring sessies eerder dan de huidige design en realisatie ermee te bezwaren.

De meest voor de hand liggende implementatie van deze interface wordt uiteraard de zogenaamde 'JavaBean' die eenvoudige (private) velden voorziet voor het bewaren van de gedeclareerde properties. Het Hibernate O/R mapping framework laat toe dergelijk naïeve implementatie vlot te mappen op de tabel – kolom –datatype details van een geselecteerde database. Het vereist daartoe slechts een minimale mapping file die per te 'persisteren object' de details van de object-

```
public interface Person {
      String getEmail();
      String getName();
      void setEmail(String email);
      void setName(String name);
    void registerFor(Event evt);
    void unRegisterFor(Event evt);
    public Set getRegisteredEvents();
}
```

Codevoorbeeld 1

```
<hibernate-mapping>
  <class name="org.outerj.regis.PersonPO"
table="person">
    <id name="id"    length="32"    unsaved-
value="null">
      <generator class="uuid.hex" />
    </id>

    <property name="email" />

    <property name="name" column="fullname"
/>

    <set name="registeredEvents" table="regi-
stration"
         lazy="false" inverse="false" casca-
de="none"
         sort="unsorted" >
      <key column="person_id" />
      <many-to-many
class="org.outerj.regis.EventPO"
              column="event_id" />
    </set>

  </class>
</hibernate-mapping>
```

Codevoorbeeld 2

Object		Database
class: org.outerj.regis.PersonPO	↔	tabel: “person”
methode: get/setEmail()	↔	kolom: “email”
methode: get/setName()	↔	kolom: “fullname”
methode: get/setId()	↔	kolom: “id”
methode: get/setRegisteredEvents()	↔	many-to-many relatie via tabel ‘registration’ waarvan de velden ‘person_id’ en ‘event_id’ wijzen naar het huidige Person-object resp. de op te laden set van Events

Tabel 1

record transities beschrijft. In het geval van de persoon zou die er kunnen uitzien zoals codevoorbeeld 2

(De bean realisatie van de interface krijgt de -PO (persistent object) suffix mee)

Hierin worden verwijzingen gemaakt naar verschillende properties (i.e. get/setPropertyName-methodes) van de class en hoe die rechttoe-rechtaan mappen op welke velden in welke tabel.

Zoals voorspeld komt de in te voeren primary key hierin om de hoek kijken. Aangezien het O/R mapping framework uiteraard die sleutel gaat gebruiken om het verband tussen de gegenereerde objecten en de op te pikken records te ontdekken dwingt het ons die toe te voegen als property aan de bean. Die toevoeging kan evenwel zonder de publieke interface van de bean zelf

Het Hibernate framework laat toe een implementatie te mappen op de details van een geselecteerde database

ermee te verzwaren: Door het exploiteren van dezelfde techniek die de standaard Java toelaat private velden van objecten te serialiseren slaagt het Hibernate framework er ook in de private properties van een bean te lezen of in te stellen bij ‘save’ of ‘load’ uit de corresponderende record. Binnen de PersonPO bean volstaat dus de toevoeging van een private get/setId().

Enige varianten op het primary-key gebeuren zien er dan zo uit:

- Voor hen die genoeg argumentatie vinden om toch de email-property op te waarderen tot primary key is er:

```
<id name="email">
  <generator class="assigned" />
</id>
```

- Voor hen die de generatie van de unieke key willen overlaten aan de onderliggende database (zoals geweten: afhankelijk van wat die ondersteund) kan dan weer:

```
    <id name="id"    length="32"    unsaved-
value="null">
       <generator class="native" />
    </id>
```

- Voor nog wildere plannen: laat u vooral leiden door de zeer volledige documentatie van het Hibernate project. (zie link op pagina 14)

De vergelijking met de Java Serializatie gaat trouwens nog net iets verder: de te persisteren bean dient ook een default constructor te hebben (namelijk zonder argumenten). Toegegeven, helemaal los komen van de database doen we niet, maar verder dan een (optioneel) toe te voegen private property voor de primary key en de eis voor een default constructor reiken de opgedrongen inbreuken niet.

KOPPELING Het nauwgezet onderhouden van de mapping file blijft dan de grootste inspanning vragen. Het saaie (en dus foutgevoelige) karakter hiervan mapt evenwel rechtstreeks op de doelstellingen van het xDoclet project. xDoclet realiseert naar eigen zeggen een concept van ‘attribute oriented programming’[2] en heeft vooral zijn sporen verdiend in de EJB wereld, waar behoorlijk wat gelijkaardige bewerkelijkheden bestaan in het onderhoud van de verschillende deployment descriptors. Het verhaal ent zich op de doclet interface die door de javadoc utility in de JDK wordt aangeboden. Een javadoc-achtig soort commentaar als */** @veldnaam */* kan zo door het systeem worden opgepikt en gekoppeld worden aan de Java class elementen die het in de source file vooraf gaat. Die kennis wordt in de xDoclet- Hibernate ondersteuning

2 Het belang van een zinvol gebruik van interfaces valt moeilijk te overschatten. Het grote geheim achter object re-use zit namelijk in het isoleren van de afhankelijkheden van andere componenten (classes) ten opzichte van uw component. De interface biedt een formeel mechanisme dat die andere classes toelaat om binnen hun applicatie uw component schaamteloos te vervangen door een andere. De deemoedige opstelling (achter een interface) van uw plotseling waardeloos geworden component realiseert zo een aanzienlijke intrinsieke meerwaarde voor de applicatie zelf: daar werden namelijk a*lle andere componenten hergebruikt*. De ogenschijnlijke contradictie in het realiseren van hergebruik door ‘weg te gooien’ wordt bijzonder poëtisch beeld gegeven in een citaat van Antoine de Saint-Exupéry: “Perfectie bereikt men niet wanneer er niets meer toe te voegen valt, maar wanneer er niets meer te verwijderen valt.”

```
/**
 * @return String
 * @hibernate.id   generator-class="uuid.hex"
 *             length="32" unsaved-
value="null"
 */
private String getId(){
    return this.id;
}

/**
 * @return String
 * @hibernate.property  length="128" uni-
que="true" not-null="true"
 */
public String getEmail() {
    return this.email;
}

/**
 * @return String
 * @hibernate.property  length="128"
column="fullname"
 */
public String getName() {
    return this.name;
}

/**
 * @return Set containig all events this
person is registered to
 * @hibernate.set table="registration"
 * @hibernate.collection-key
column="person_id"
 * @hibernate.collection-many-to-many
column="event_id"
 *   class="org.outerj.regis.EventPO"
 */
public Set getRegisteredEvents() {
  if (this.registeredEvents == null) {
    this.registeredEvents = new HashSet();
  }
  return this.registeredEvents;
```

Codevoorbeeld 3

aangewend om de besproken mapping files te genereren. De developer focust zich dan enkel op code, die wordt verrijkt aan de hand van zogenaamde 'attributering' (zie codevoorbeeld 3).

Om de te genereren code netjes mee op te nemen in het build-proces is er uiteraard het wereldvermaarde Ant. Binnen de xDoclet en Hibernate projecten vindt u meteen alle nodige tips en trucs om alles gestroomlijnd in het buildproces te integreren. (Gezien de wijde ondersteuning van Ant door alle toonaangevende Java IDE's is deze optie dus snel door iedereen aan te wenden.)[3]

WAT OVERBLIJFT De winst van de hele oefening zit in de overblijvende rechtdoor-implementatie van de process-classes. Die moeten met de besproken entiteiten aan de slag gaan, via de uitgeschreven interfaces Person en Event. De daar beschreven methodes zijn via de gevolgde weg beschikbaar op de persistente objecten.

De enige problematiek die overblijft is hoe het raamwerk gevraagd kan worden de betrokken persistente objecten op te hoesten en weg te schrijven. Wat de initialisatie-code betreft is er enkel het volgende:

```
import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.cfg.Configuration;

Configuration cfg = new Configuration();
cfg.addClass(PersonPO.class);
cfg.addClass(EventPO.class);
sessions = cfg.buildSessionFactory();
session = sessions.openSession();
```

Het resulterende session object hiervan geeft toegang tot het obligate load(), save(), update(), delete().

RANZIGE KANTJES Met deze minimale introductie kunt u meteen op pad, maar de nog te ontdekken wereld is er helaas niet kleiner door geworden[4]. Dat blijkt wel met de greep uit wat we van het zeer volledige Hibernate framework niet hebben besproken in het functionele domein:

- mapping van meerdere fijnere objecten op soms meerdere kolommen uit eenzelfde tabel (het Location object voorbeeld)
- mapping van inheritence relaties (en niet alleen associaties) tussen objecten naar mogelijke implementaties van het relationeel schema
- de ingebouwde 'object' query language
- cascade van updates en deletes

en in het administratie/infrastructuur-domein:

- ondersteuning van JDBC 2.0 connection pools (Datasources)
- gerealiseerde 'location transparency' van Hibernate sessions via JNDI services
- transactioneel beheer
- foutdetectie en afhandeling
- concurrency instellingen en isolation levels

3 De .NET en C# wind heeft de term trouwens een nieuw leven ingeblazen op een manier die sommigen verbaasd doet opkijken wanneer ze vernemen dat het ervoor ook al bestond.

4 Marc Portier kunt u bereiken via e-mail mocht u code en build-setup van zijn test wensen te ontvangen: mpo@outerthought.org.

Vooral die laatste voorbeelden zeggen het nog eens duidelijk: 'there is no such thing as a free lunch'. Wie zich met relationele databases inlaat (of elke andere persistency techniek wat dat betreft), voegt een nieuw aspect toe aan zijn oplossing of problematiek. Dat zoiets gevoeligheden en details met zich meebrengt weet u al een tijdje. De voordelen kent u ook. Eveneens is het nut van de aangehaalde tools in de praktische aanpak van dagelijkse projecten reëel. De aandacht van de OO ontwikkelaar wordt naar de essentie van de zaak teruggebracht, en de mapping-laag doet als bij wonder de rest. Dat de 'ranzige kantjes' van die onderliggende 'wonderen van het operationeel databasebeheer' via formele wegen toch door de abstractielagen lekken mag niemand verbazen.[5]

> 'There is no such thing as a free lunch': wie zich met relationele databases inlaat, voegt een nieuw aspect toe aan zijn oplossing

MIDDLE OUT AANPAK De voorgestelde werkvolgorde (objecten eerst) neemt een zogenaamde top-down aanpak die uiteindelijk niet helemaal vreemd is voor een overtuigd OO-gelovige die zijn OOAD workshop in de praktijk wil inzetten om herbruikbare componenten te schrijven op de middle-tier. Binnen de Hibernate-distributie bevinden zich evenwel alle wenselijke tools om ook vanuit andere richtingen - bottom-up (vertrekkend van een bestaand database schema) en zelfs middle-out (meteen de mapping file aanpakken) - het probleem aan te vallen en de nodige code, mapping files of ddl statements te genereren uitgaande van uw toevallige bestaande situatie.

Voor de volledigheid verwijzen we graag naar de project homepages van de betrokken tools:

- Hibernate, O/R mapping: http://hibernate.sf.net
- xDoclet, attribute oriented programming: http://xdoclet.sourceforge.net
- Apache Ant, Java oriented build: http://ant.apache.org

Hibernate neemt in huidige open source zowat de rol op van 'leading O/R implementation'. Het heeft zich dan ook onder meer tot doel gesteld eerder functioneel, bruikbaar en volledig te zijn dan de door de Java 'standaardisatie' organisatie (http://jcp.org) zelf opgeworpen JDO standaard te implementeren. Voor wie dat laatste een belangrijk punt vindt is het jonge ObJectRelationalBridge van Apache (http://db.apache.org/ojb) vermoedelijk een werkbaar alternatief.[6]

GEWOON DOEN Niet te vergeten: het hele ding is open source. Voor hen die zich hier alsnog ongemakkelijk bij voelen blijft het volgende citaat de enige troost: "Open Source is zoals bungy jumpen en stijldansen. Je kunt er uren vanuit je luie stoel naar kijken, en er veel meningen over ventileren." Je kunt het ook gewoon doen. Veel succes.

Marc Portier <mpo@outerthought.org> is mede-oprichter van Outerthought, een technisch Java & XML kenniscentrum in België. Marc is sinds jaar en dag ambassadeur en gebruiker van Java, XML en open source technologieën en actief in verscheidene Apache projecten. Marc onderhoudt een weblog op http://radio.weblogs.com/0116284/

5 Hoe meer u weet, hoe meer u dreigt te vergeten, maar ook hoe dieper het besef van alles wat u nog dient te weten. Een grafische voorstelling van deze stelling vindt u op: http://blogs.cocoondev.org/tomk/archives/000745.html

6 Lees ook: 'Leaky Abstractions' op http://www.joelonsoftware.com/articles/LeakyAbstrac-tions.html