Het Java Data Objects (JDO) Framework is een veelbelovende technologie voor object persistentie in een J2EE omgeving. Veel experts zijn van mening dat JDO een beter alternatief is voor object persistentie dan Entity Beans die een vast onderdeel uitmaken van de J2EE standaard. JDO is niet onomstreden maar toch zijn velen van mening dat JDO eigenlijk is wat Entity Beans hadden moeten zijn. In dit artikel gaat Willem Koppenol dieper in op het gebruik van JDO in een J2EE omgeving en maakt hij en passant de vergelijking met Entity Beans.

thema

Java Data Objects in J2EE

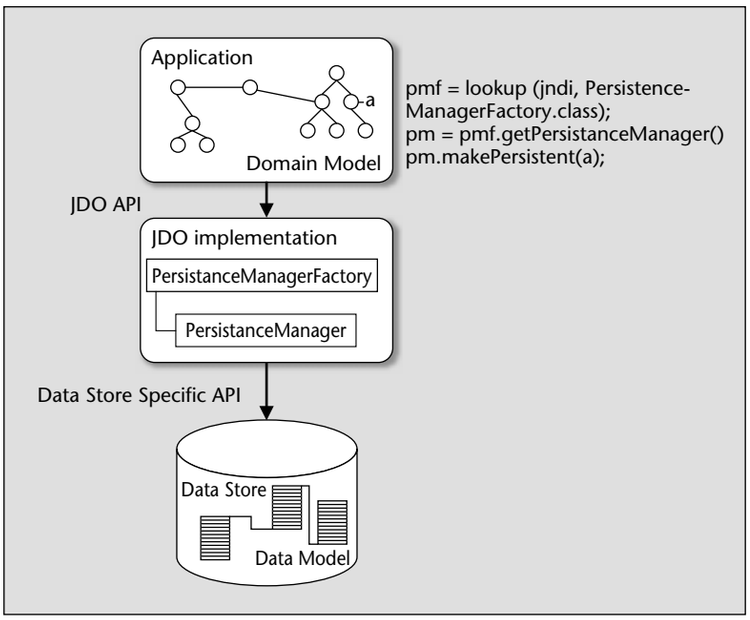
Alternatieve standaard voor object persistentie

Het Entity Beans componenten model kent een aantal problemen (complexiteit, geen ondersteuning voor inheritance en polymorphisme, beperkte mogelijkheden voor dynamische query's) die JDO niet kent. Vooralsnog echter zitten Entity Beans in de J2EE standaard en JDO niet. De consequentie hiervan is dat J2EE applicatie servers geen ingebouwde JDO ondersteuning geven en dat er dus een aparte JDO implementatie naast de J2EE server moeten draaien om JDO mogelijk te maken. En dit komt de brede acceptatie van JDO niet ten goede.

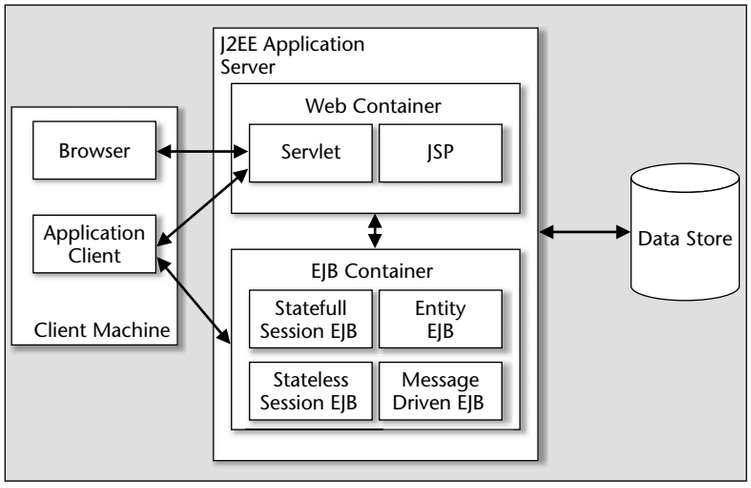
JDO BASICS JDO maakt het mogelijk een hele object graph van gewone Java objecten (POJO's, Plain Ordinary Java Objects) te persisteren. Een applicatie gebruikt daarvoor de JDO API en spreekt een JDO implementatie aan. Deze JDO implementatie vormt de laag tussen de applicatie en de persistent data store, zoals een JDBC driver dat doet bij de JDBC technologie. JDO kan in een non-managed en managed (J2EE) architecturen worden gebruikt. In een non-managed omgeving configureert een applicatie een PersistenceManagerFactory, verkrijgt daaruit een PersistenceManager en vervolgens verlopen alle persistentie operaties via dit object. De applicatie is zelf verantwoordelijk voor transactie management. De managed architectuur komt zo dadelijk aan de orde.

JDO'S TRANSPARENT PERSISTENCE Middels JDO kan het domein model inclusief mogelijke implementatie hiërarchie transparant, zonder vertaalslag, worden opgeslagen. De ontwikkelaar hoeft zich niet te bekommeren om de precieze details van de persistentie, zoals het lezen of schrijven van individuele attributen. Deze worden afgehandeld door de JDO implementatie. De persistentie wordt ook transparant genoemd omdat de applicatie de illusie heeft dat vanuit een persistent object alle gerefereerde objecten direct in memory toegankelijk zijn. In werkelijkheid bevinden ze zich mogelijk in de data store en worden ze transparant voor de applicatie door de JDO implementatie geladen.

PERSISTENT CLASSES Java classes zijn persistent voor een JDO implementatie als ze het PersistenceCapable interface implementeren. Dit interface heeft



FIGUUR 1. JDO in een non-managed omgeving



FIGUUR 2. Managed J2EE omgeving

methoden die de JDO implementatie in staat stellen de toestandsvariabelen van de objecten te synchronizeren met de data store. Meestal worden de PersistenceCapable methoden door een zogeheten byte code enhancer in de reeds gecompileerde class files aangebracht. Dit heeft het voordeel dat de standaard Java classes ongewijzigd door JDO kunnen worden gepersisteerd. Maar het is ook een bekritiseerd fenomeen van JDO omdat het onder meer het debugging process bemoeilijkt. Overigens kunnen deze methoden ook met de hand op source code niveau aangebracht worden. Een derde alternatief is gebruik te maken van een tool dat de PersistenceCapable classes genereert uit een object model of een relational database schema. De precieze persistentie voorwaarden van een class worden beschreven in een op XML gebaseerde persistentie descriptor die wordt gelezen tijdens het enhancement process.

DE MANAGED J2EE OMGEVING In een managed J2EE omgeving verzorgen J2EE applicatie servers voor gedistribueerde applicaties typische middleware services zoals transacties, concurrency, connection pooling en security. De J2EE omgeving heeft haar opmars met name te danken aan het succesvolle gedistribueerde transactie model, waardoor zaken als load-balancing over verschillende servers en fail-over effectief zijn te implementeren. Eén van de grote voordelen van J2EE is verder dat deze services declaratief kunnen worden geconfigureerd via een XML deployment descriptor te zetten. Hercompilatie van componenten is hiervoor niet nodig. De J2EE omgeving kent een non-managed Web laag met servlets en Java Server Pages (JSP) die draaien in een Web Container en een managed business componenten laag met Enterprise Java Beans (EJB) die draaien in een EJB Container. Eén van de voornaamste minpunten van de J2EE architecture is dat persistentie en distributie zijn gekoppeld. Deze concepten zijn eigenlijk orthogonaal, maar volgens de J2EE specificatie moet een applicatie dezelfde eenheid toepassen voor persistentie en distributie. JDO adresseert duidelijk dit minpunt.

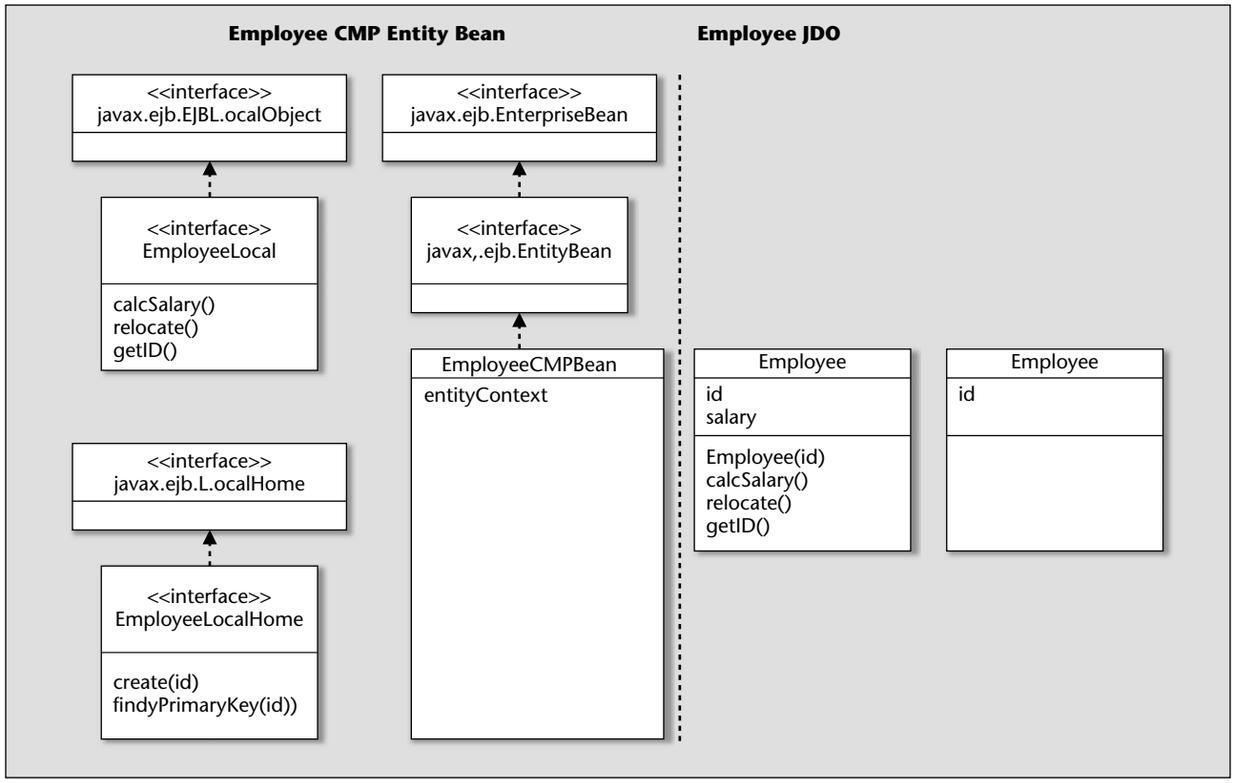
ENTERPRISE JAVA BEANS EJB's zijn er in een drietal smaken: Session, Entity en Message-Driven. JDO kan in combinatie met elk van deze EJB's worden gebruikt, maar we zullen met name Session en Entity Beans beschouwen.

Session Beans representeren een client request. Ze bestaan in twee varianten. Statefull Session Beans blijven beschikbaar zolang de client dat wenst en kunnen er over meerdere methodes heen conversational state opslaan. Stateless Session Beans blijven gedurende één methode voor de client gereserveerd. Een volgende methode aanroep door dezelfde client wordt mogelijk door een andere Session Bean uitgevoerd.

Entity Beans representeren data uit de data store en zijn de eenheid van persistentie in J2EE De EJB container zorgt door aanroep van de callback methoden `ejbLoad` en `ejbStore` in de Entity Bean voor synchronisatie met de data store. Eén van de nadelen van Entity Beans ten opzichte van JDO's is hun complexiteit. Zo moeten bij een Entity Bean minimaal twee interfaces (`LocalHome` en `Local`) en soms zelfs vier (ook `RemoteHome` en `Remote`) worden aangeboden. De Entity Bean zelf is een class waarin een tiental EJB callback methoden uit de interfaces moeten worden geïmplementeerd. Bij JDO's moet alleen een enkelvoudige Java class worden geschreven. Er zijn geen opgelegde afhankelijkheden van de JDO API en er hoeven geen callback methoden te worden geïmplementeerd.

Clients van een J2EE applicaties hebben nooit directe referenties naar de EJB's. Alle contact verloopt via de EJB Container.Clients hebben referenties aan `HomeObjects` en `EJBObjects` die bij deployment uit de EJB in de container worden gegenereerd. De container communiceert met de EJB's via een aantal voorgeschreven interfaces en callback methoden zoals het `Home` interface met creatie methoden en het `Remote` of het `Local` interface (sinds J2EE 2.0) met de business methoden.

JDO IN J2EE De integratie van JDO in J2EE wordt meestal gebaseerd op de J2EE Connector Architecture (JCA). De JDO implementatie wordt dan via een resource adapter in de J2EE server opgenomen. Connector contracts geven de JDO implementatie toegang tot connecties naar data sources en de transactie manager in de J2EE applicatie server. Componenten spreken nog steeds een `PersistenceManager` aan voor het uitvoeren van persistentie operaties, maar de `Persistance-`

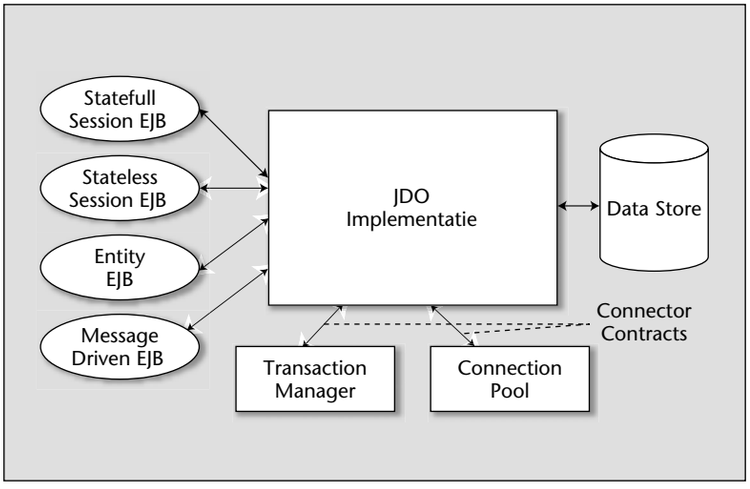


FIGUUR 3. EJB Complexity versus JDO simplicity

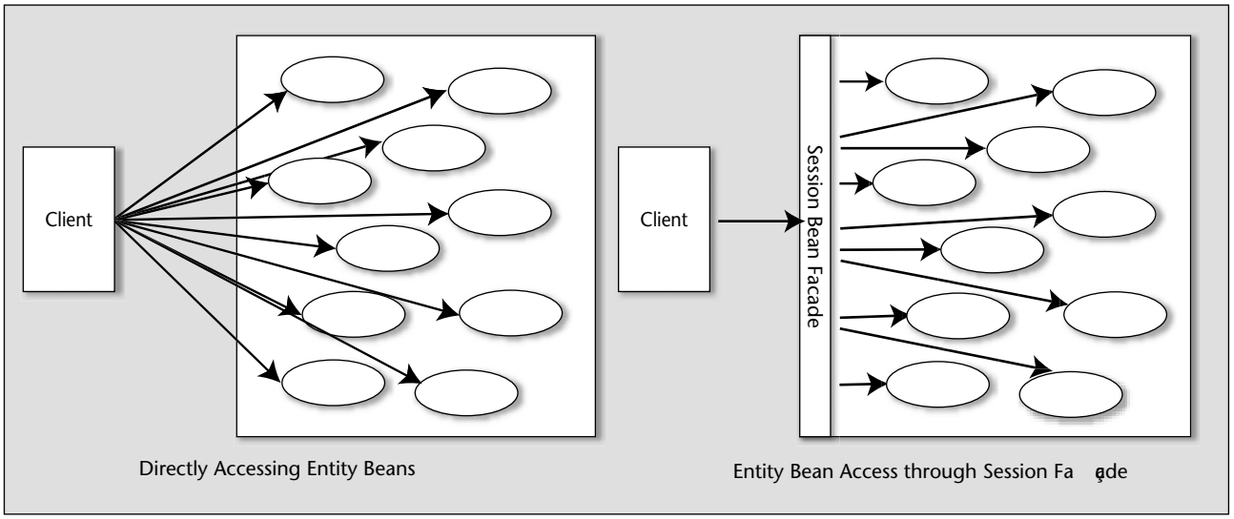
`ManagerFactory` moet nu via uit de Java Naming and Directory Interface (JNDI) van de J2EE applicatie server worden verkregen. Verder worden `PersistenceManagers` gepooled om het gebruik van resources te beperken en de schaalbaarheid te vergemakkelijken. Dit betekent dat componenten die gebruik maken van een `PersistenceManager`, deze in dezelfde methode als ze hem van de factory krijgen ook weer moeten vrijgeven. EJB componenten zijn vrij in hun keuze om het declaratieve transactie demarcatie model van J2EE gebruiken of om hun transacties zelf te regelen.

JDO IN ENTITY BEANS Entity Beans representeren data uit de data store en zijn er in twee varianten. De BMP (Bean Managed Persistence) Entity Beans zorgen zelf voor hun persistentie doordat de bean developer code schrijft waarin de data store wordt benaderd. De CMP (Container Managed Persistence) Entity Beans delegeren hun persistentie aan de container en bevatten in het geheel geen data access code. In incidentele gevallen is het mogelijk dat de EJB container intern gebruik maakt van JDO voor de realisatie van zijn CMP implementatie, maar de bean ontwikkelaar merkt hier niets van zodat we dit verder buiten beschouwing laten. De BMP Entity Beans maken voor hun data access traditioneel gebruik van JDBC. In theorie is het goed mogelijk om JDBC te vervangen door JDO. Echter daarmee verwordt JDO tot een implementatie detail van Entity Beans terwijl JDO juist een lichtgewicht alternatief daarvoor zijn. Deze benadering levert geen enkel voordeel op vanuit architectuur- of ontwikkeloogpunt. We zullen deze verder buiten beschouwing gelaten.

SESSION FAÇADE INTERMEZZO Alvorens in te gaan op het gebruik van JDO in Session Beans eerst iets over één van de meest gebruikte J2EE Design Patterns : het Session Façade design pattern. In dit design pattern worden de afhankelijkheden tussen client en server geminimaliseerd en de performance geoptimaliseerd door af te dwingen dat de Use Cases worden afgehandeld binnen één netwerk call en één transactie. Immers



FIGUUR 4. JDO integratie in J2EE



FIGUUR 5. Direct Entity Bean Access versus Session Façade pattern

vaak is het voor uitvoeren van een stuk business logica in een Use Case nodig dat een aantal attributen van verschillende server componenten (Session of Entity Beans) worden benaderd en mogelijk gewijzigd. Indien dit zou gebeuren door het uitvoeren van een aantal remote calls (en mogelijk transacties) vanuit client code zou dit een zware aanslag op de performance betekenen, omdat iedere remote call remoting overhead geeft. In het Session Façade design pattern worden deze meervoudige aanroepen gecombineerd in één aanroep van een methode van een Session Bean. De server side componenten worden vervolgens vanuit deze Session Bean methode benaderd. Dit gebeurt nu echter via locale aanroepen die veel minder overhead vergen.

JDO IN SESSION BEANS Het gebruik van JDO in de Sessions Beans van een Session Façade ligt het meest voor de hand. De Session Beans moeten daartoe de `PersistanceManagerFactory` in hun methoden kunnen benaderen. Het is gebruikelijk een referentie daarnaar via zijn geconfigureerde JNDI naam op te halen en te bewaren. Na instantiatie roept de EJB container de `setSessionContext` methode aan en de referentie wordt dan door de volgende code bewaard:

```
private PersistenceManagerFactory pmFactory =
null;

public void setSessionContext (SessionContext
ctx)
{
this.ctx = ctx;
InitialContext icx = new InitialContext();
pmFactory =
(PersistenceManagerFactory)icx.lookup("aPMFact
");
}
```

Verder moeten de deployment descriptors van de Session Beans worden geconfigureerd dat ze de JDO resource gebruiken. Een stuk van de deployment descriptor van de Session Bean ziet er dan als volgt uit :

```
<resource-ref>
   <description>JDO PersistenceManagerFactory
for Employee database
   </description>
   <res-ref-name>jdo/EmployeePMF</res-ref-
name>
   <res-
type>javax.jdo.PersistenceManagerFactory</res-
type>
   <res-auth>Container</res-auth>
   <res-sharing-scope>Shareable</res-sharing-
scope>
</resource-ref>
```

Het is gebruikelijk dat de methoden van een Session Façade overeenkomen met Use Cases en deze Use Cases moeten vaak in een transactie worden uitgevoerd. Hierbij kan gekozen worden tussen Bean-Managed Transactions (BMT) en Container-Managed Transactions (CMT).

CONTAINER-MANAGED TRANSACTIONS In het geval van CMT, worden de transactie settings in de deployment descriptor aangegeven. De transactie grenzen liggen dan bij individuele methode aanroepen van de Session Bean. Bij een setting "Requires New" begint de container een transactie aan het begin van een methode en doet een commit of rollback vlak voor het verlaten van de methode. Bij een setting "Requires" kan de container de methode aanroep ook aan laten sluiten bij een reeds lopende transactie. De consequentie van CMT voor JDO is dat iedere business methode bij de

start een `PersistenceManager` moet opvragen bij de `PersistenceManagerFactory`, zoals geschetst in het volgende code fragment:

```
void createEmployee (id)
{
PersistenceManager pm = pmFactory
.getPersistenceManager();
// get a new employee id from somewhere
Employee emp = new Employee (id);
pm.makePersistent (emp);
pm.close();
}

int calcSalary (id)
{
PersistenceManager pm = pmFactory
.getPersistenceManager();
Employee emp = pm.getObjectById ( new
EmployeeId(id) );
emp.calcSalary ();
pm.close();
}
```

In geval er al een transactie loopt wordt de `PersistanceManager` teruggegeven die hoort bij de lopende transactie context. Zoniet, dan wordt er één uit de pool van beschikbare `PersistenceManagers` gekozen en aan de huidige transactie context gebonden. Door dit automatische gedrag is het eenvoudig verschillende methoden in één container managed transactie zonder programmeerlogica te combineren.

BEAN-MANAGED TRANSACTIONS In het geval van BMT, kan de ontwikkelaar zelf bepalen wanneer transacties worden gestart en beëindigd en kan hij methoden binnen of buiten transacties aanroepen. Het nadeel van deze flexibiliteit is echter de toegenomen complexiteit. Eén mogelijkheid is een `UserTransaction` instance van de container te gebruiken om transacties te starten en te beëindigen. Ook nu moet een `PersistenceManager` in de juiste transactie context worden verkregen. Ook bij BMT is het goed gebruik om een `PersistenceManager` aan het begin van een methode op te halen en aan eind af te sluiten. Op deze wijze kunnen de verschillende methoden vrijelijk in transacties worden gecombineerd. De onderstaande code gebruikt in één transactie bijvoorbeeld de methodes uit het CMT voorbeeld:

```
void newSalaryEmployee (int empId)
{
UserTransaction
ut=icx.lookup("java:comp/UserTransaction");
ut.begin();
createEmployee(empId);
calcSalary (empId);
ut.commit();
}
```

Een andere mogelijkheid is de `PersistanceManager` en de `javax.jdo.Transaction` te gebruiken. Hierdoor kan dezelfde `PersistanceManager` voor verschillende transacties worden gebruikt zonder dat hij aan de pool wordt teruggegeven.

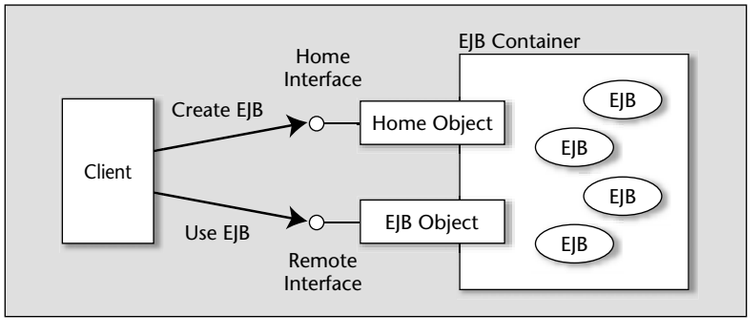
JDO'S VERSUS ENTITY BEANS Naast de al genoemde geringe complexiteit zijn er nog andere redenen waarom JDO's een beter persistentie mechanisme bieden dan Entity Beans.

Zo zijn de inheritance mogelijkheden van Entity Beans beperkt. Het delen van code is nog wel mogelijk door de Entity Bean classes van elkaar af te leiden, maar de run-time voordelen van inheritance in de vorm van polymorfisme zijn niet mogelijk. De reden hiervoor is gelegen in het feit dat een client geen directe toegang heeft tot de Entity Bean, maar een referentie heeft naar het EJB Object in de container. Up- en down casting in een inheritance tree is dan niet mogelijk. JDO's kunnen daarentegen zonder probleem gebruik maken van complexe class hiërarchieën. JDO's kunnen net als andere Java classes van elkaar erven en ook tijdens runtime zijn up- en down casts en polymorfisme mogelijk. JDO is daarmee beter geschikt om een domein model te representeren dan Entity Beans.

> JDO's kunnen zonder probleem gebruik maken van complexe class hiërarchieën

Verder kunnen alle aspecten van JDO's, behalve de persistentie, getest worden door middel van testprogramma's waarin de JDO's worden geinstantiëerd. Voor Entity Beans is dit op deze manier niet mogelijk omdat ze alleen draaien in de context van de EJB container en eerst moeten worden gedeployed.

Tenslotte zijn de mogelijkheden voor dynamische query's in EJB's minimaal. Query's worden ofwel hard gecodeerd bij BMP Entity Beans ofwel tijdens deployment aangegeven in EJB-SQL bij CMP Entity Beans. Voor JDO's zijn dynamische query's geen enkel probleem.



FIGUUR 6. EJB Clients hebben geen directe referenties naar EJB's

JDO IN SERVLETS Ook servlets kunnen JDO gebruiken voor object persistentie, maar ze missen daarbij de transactionele en gedistribueerde eigenschappen van een EJB container. Servlets draaien in een servlet container en moeten voor het gebruik van JDO na instantiatie een `PersistanceManagerFactory` ophalen en bewaren in hun `init` methode. Bij ieder request moet vervolgens een `PersistanceManager` via de factory worden verkregen en na afhandeling van het request weer worden vrijgegeven. En aangezien de servlets geen gebruik van een transactie context van een container, zullen ze zelf expliciet aan transactie demarcatie, bijvoorbeeld via JDO, moeten doen. Indien een servlet EJB componenten aanspreekt is het gebruikelijk alle JDO access te delegeren aan de EJB laag.

JDO IN JSP'S Bij gebruik van JDO in Java Server Pages (JSP) is het onwenselijk om veel Java code in een JSP op te nemen. JDO specifieke code wordt het liefst ondergebracht in een Java Bean of custom tag. Zo kan een bean een `PersistenceManagerFactory` horende bij een bepaalde JNDI naam opzoeken en een `PersistenceManager` teruggeven. Deze bean kan dan vervolgens in JSP worden geinstantiëerd met de `<jsp:useBean>` tag. Andere JDO specifieke code moet op een soortgelijke manier worden ingekapseld. De code voor een dergelijke bean class is:

```
public class EmployeePMFBean
{
   PersistenceManagerFactory pmfFactory;

   public void setJNDIName(String jndiName) {
  Context ic = new InitialContext();
  Context env = ic.lookup("java:comp/env");
  pmfFactory = (PersistenceManagerFactory)
env.lookup(jndiName);
   }

   public synchronized PersistenceManager
getPersistenceManager() {
    return
pmfFactory.getPersistenceManager();
   }
}
```

De instantiatie in de JSP pagina gaat met:

```
<jsp:useBean id="EmployeePMFBean"
scope="application"
       class="employee.ejb.EmployeePMFBean">
  <%
EmployeePMFHolder.setJNDIName("EmployeePMF");
%>
</jsp:useBean>
```

Het is goed mogelijk dat een JDO custom tag library onderdeel wordt van toekomstige versies van JDO.

SLOTWOORD De JDO standaard voor object persistentie kan een belangrijke rol spelen in de toekomst van het J2EE platform. JDO kan gebruikt worden in combinatie met alle Enterprise Beans, servlets en JSP's. JDO is een goed alternatief voor de complexe en beperkte Entity Beans. Een voor de hand liggend gebruik van JDO is in de Sessions Beans van een Session Façade. JDO is specifiek geschikt om domeinobjecten te mappen op de database. Het bestaande J2EE platform, zonder JDO, is onvoldoende voor deze taak uitgerust zoals Sun's eigen architecten hebben geconstateerd. Sun zou daarom JDO aan het J2EE platform moeten toevoegen. JDO is inmiddels opgenomen als standaard extensie voor de J2SE. Voor de opname in de J2EE standaard bestaan nog geen concrete plannen. En hiervoor zal nog heel wat politieke weerstand moeten worden overwonnen, want er is door veel partijen hevig in Entity Beans geïnvesteerd. Maar het is hoopgevend dat Sun inmiddels is toegetreden tot de JDO Specification Group.

Drs. Willem Koppenol is senior trainer en productspecialist software development voor Twice IT Training.