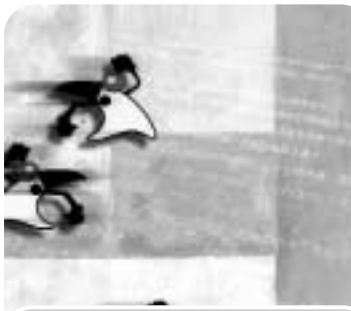
Java Data Objects (JDO) is één van de Java specificaties van SUN. Het voornaamste doel van JDO is het voorzien in een framework voor transparante persistentie van Java objecten. Persistentie betekent het opslaan van data op een fysieke locatie zodat applicaties deze data op een later tijdstip weer kunnen terughalen, zelfs na een herstart van het systeem. In dit artikel worden de alternatieven voor Java persistentie tegen elkaar afgezet en wordt vervolgens in detail ingegaan op de werking en implementatie van JDO.



Java Data Objects

Alternatieven voor Java-persistentie

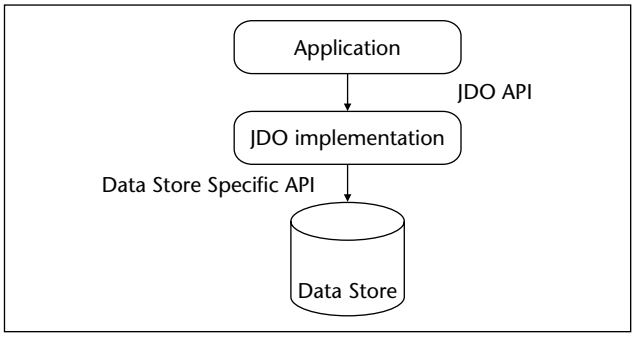
Voor datapersistentie zijn in een Java omgeving verschillende alternatieven voorhanden. Van oudsher is er de mogelijkheid de data op te slaan in files en Java I/O operaties gebruiken voor het benaderen van deze files. Een objectgeoriënteerd alternatief is persistentie door serialisatie van Java objecten. In de praktijk het meest toegepast is het gebruik van de JDBC API voor het benaderen van een relationele database. En de nieuwste objectgeoriënteerde loot aan de boom is JDO. Deze jongste techniek is veelbelovend maar is nog niet geheel de kinderschoenen ontgroeid. In dit artikel worden de alternatieven voor Java persistentie tegen elkaar afgezet en wordt vervolgens in detail ingegaan op de werking en implementatie van JDO.

Het opslaan van data in files is een eenvoudige, lichtgewicht oplossing. Deze optie vergt weinig van het onderliggende operating system en is geschikt als de systeemresources schaars zijn, zoals bij embedded systems. Ieder door de developer bepaald dataformaat is in principe toepasbaar. Stream I/O en het java.io package wordt gebruikt voor het benaderen van de files. Er is geen ondersteuning voor het opslaan van objecten en ook transacties en query's zijn problematisch.

SERIALISATIE Serialisatie zit ingebouwd in de Java API en is een standaard onderdeel van de iedere Java Virtual Machine(JVM). Serialisatie gebruikt Java classes als datamodel en maakt directe persistentie van het objectmodel mogelijk. Door serialisatie kun je een hele objectgraph, een complete set van objecten met hun waarden van attributen en referenties naar elkaar, associëren met een ObjectOutputStream, en vervolgens wegschrijven naar een file. Serialisatie is heel eenvoudig in het gebruik. Een class die geserialiseerd wil worden verklaart eenvoudig dat hij het Serializable interface implementeert. Serialisatie biedt weliswaar directe objectpersistentie, maar ontbeert belangrijke eigenschappen als transacties en query's. Er is geen bescherming tegen het tegelijkertijd schrijven van data in de met de geserialiseerde objecten geassocieerde file. Serialisatie biedt dan ook geen serieus alternatief ten opzichte van een robuuste database-omgeving. Het is wel een heel nuttige techniek die op de achtergrond veel gebruikt wordt bij communicatie tussen objecten over het netwerk (o.a. RMI).

JDBC Java Database Connectivity (JDBC) is de meest gebruikte techniek voor het opslaan van data in een Java omgeving. Het is een gestandaardiseerde API, in verschillende versies beschikbaar, waarbij SQL statements gebruikt worden als parameters bij JDBC functies. De JDBC aanroepen worden door een JDBC driver vertaald naar de commando's voor een specifieke database. JDBC heeft als groot voordeel dat het transacties en SQL query's direct ondersteunt. Maar JDBC heeft een relationeel datamodel waarbij de entiteiten uit de applicatie worden gerepresenteerd door tabellen met rijen, kolommen en relaties. Een nadeel van JDBC is dat het Java objectmodel er niet direct mee kan worden opgeslagen. Eerst moet het objectmodel in code vertaald worden in een relationeel model. De applicatie is zo gedwongen twee verschillende datamodellen te gebruiken. Andere nadelen van JDBC zijn verder nog dat de SQL dialecten van verschillende databases zelden compatibel zijn en dat van developers zowel een goede kennis van Java als van SQL wordt verwacht.

JDO JDO kan gezien worden als een integratie van serialisatie en JDBC, waarbij het beste uit beide werel-



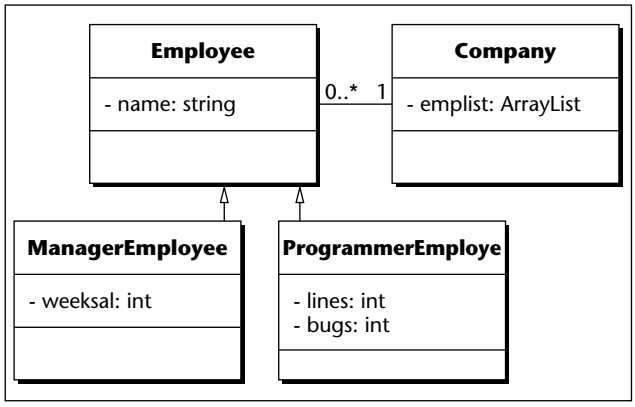
FIGUUR 1. JDO Overview

den is gecombineerd. JDO maakt, net als serialisatie, directe persistentie van het Java objectmodel mogelijk, maar ondersteunt tevens, net als JDBC, transacties en query's. De JDO benadering van persistentie vergt een minimale inspanning van de developer. Objectpersistentie is bij JDO geïntegreerd in de Java taal en wordt uitgevoerd door het aanroepen van methoden uit standaard JDO interfaces, de JDO API. Marktpartijen leveren JDO implementaties van deze JDO API en zo kan het object model in relationele databases, object databases of andere datastores worden opgeslagen. Dit model vertoont een grote overeenkomst met het JDBC model, waarbij marktpartijen JDBC drivers, een implementatie van de JDBC API, kunnen leveren voor het benaderen van mogelijk dezelfde databases. De diverse JDO implementaties kunnen zo met elkaar concurreren, hetgeen de kwaliteit ten goede komt. JDO heeft net als JDBC een pluggable architectuur. De applicaties die op JDO zijn gebaseerd kunnen verschillende JDO implementaties gebruiken aangezien deze dezelfde API ondersteunen. JDO is daarmee portable. JDO is verder geschikt voor stand-alone Java applicaties en embedded omgevingen, maar kan ook in een managed omgeving in de context van een applicatieserver met EJB's worden gebruikt.

PERSISTENT-CAPABLE CLASSES Niet alle Java classes in een applicatie behoeven persistentie. Beschouwen we als voorbeeld een applicatie, waarin de gegevens van de werknemers van een bedrijf worden bijgehouden. De applicatie zal bestaan uit een reeks van schermen met user-interface elementen die typisch niet persistent hoeven te zijn.

Businessconcepten uit dit applicatiedomein zijn de Company class en diverse Employee classes in een inheritance hierarchy. Deze classes zijn typisch wel persis-

FIGUUR 2. Niet persistent GUI Employee Management



FIGUUR 3. UML Diagram Employee Management

tent zijn en worden opgeslagen in bijvoorbeeld tabellen van een relationele database.

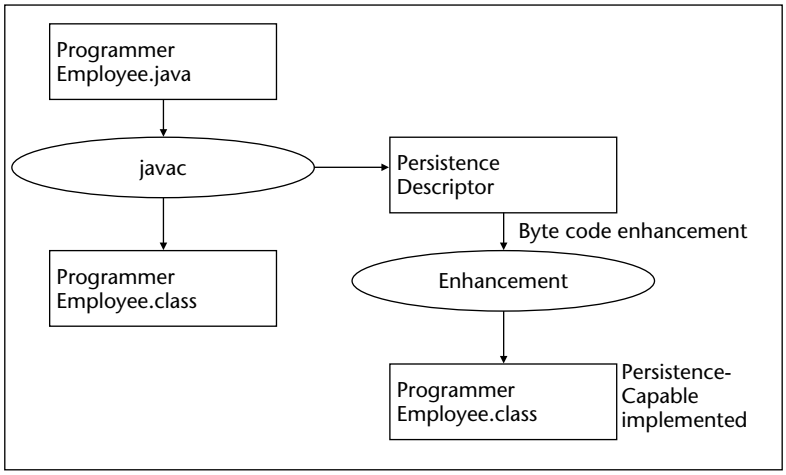
De developer kan bij JDO zelf bepalen welke classes persistent-capable of kortweg persistent zijn. Persistent classes moeten het PersistenceCapable interface implementeren. Gelukkig hoeft de developer deze interface niet zelf te implementeren, maar is dit geautomatiseerd door het enhancementprocess. De developer kan werken met gewone Java-objecten. De Java-classes hoeven geen speciale JDO types te gebruiken of speciale methoden van JDO te implementeren. Vrijwel alle Java-types, encapsulatie en overerving zijn toegestaan. Details over de wijze van persistentie worden beschreven in een deployment descriptor.

CLASS ENHANCEMENT Het enhancement process vormt een gewone Java class om tot een persistent class

```
package Employee;
public class ProgrammerEmployee extends
Employee
{
  int lines;
  int bugs;
  public ProgrammerEmployee (String s) {
     super(s);
     lines = 0;
     bugs = 0;
  }
  public void setLines(int l){
     lines = l;
  }
  public void setBugs(int b){
     bugs = b;
  }
  public int calcSal() {
     return (super.calcSal() + (lines -
     bugs));
  }
}
```

KADER 1. Persistent class ProgrammerEmployee

en maakt dat developers zich niet hoeven te bekommeren om de details van JDO. De gewone Java class wordt eerst gecompileerd tot byte code. Vervolgens wordt de enhancement facility, een onderdeel van iedere JDO implementatie, toegepast op de gecompileerde byte code. Het resultaat is de toevoeging van attributen en methodes van het PersistenceCapable interface aan de gecompileerde class. De JDO implementatie kan hiermee de toestand van de objecten in het geheugen met de datastore synchroniseren. Vrijwel iedere door de developer gedefinieerde class kan persistent worden gemaakt. Uitzonderingen zijn sommige systeemclasses, zoals Thread, Socket en File. Class enhancement is één van de voornaamste faciliteiten die JDO transparant voor de developer maken.



FIGUUR 4. Class Enhancement

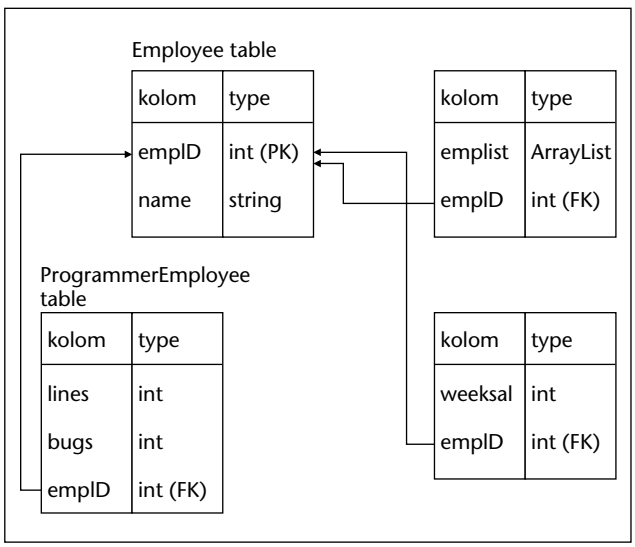
PERSISTANCE DESCRIPTOR Ook is nog een persistentie descriptor nodig. Dit XML-bestand met de extensie .jdo bevat JDO meta-data over de gewenste persistentie. Minimaal staat er in welke classes persistent zijn, maar ook wordt erin aangegeven wat de super class van een persistent class is, welke attributen persistent zijn en welk JDO identity type wordt gebruikt. De descriptor wordt gelezen tijdens het class enhancement process. Via de persistentie-descriptor heeft de developer volledige controle over welke classes en velden van classes worden opgeslagen. Leveranciersspecifieke extensies worden wel gebruikt om aan te geven hoe persistente classes mappen op een relationele database. Voorbeelden zijn de naam van de tabel waarin een type object wordt opgeslagen en de kolom namen voor persistent attributen.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
    <package name="Employees>
    <class name="ProgrammerEmployee"
          identity-type="datastore"
          persistence-capable-
          superclass="Employees.Employee">
       <!— Here are some vendor-specific JDO
       extensions —>
       <extension vendor-name="tcc"
       key="table"
       value="ProgrammerEmployee"/>
       <extension vendor-name="tcc" key="key-
       column" value="id"/>
    </class>
  </package>
</jdo>
```

KADER 2. Persistence Descriptor

OBJECT RELATIONAL MAPPING Vaak zal de datastore in een JDO applicatie een relationele database zijn. Alvorens de applicatie kan draaien moeten in dat geval de met de objecten corresponderende tabellen aanwezig zijn. De persistent classes worden daartoe gemapt op een database schema. Classes en attributen worden vertaald naar tabellen en kolommen en de relaties tussen de classes, en naar relaties tussen tabellen. JDO implementaties leveren over het algemeen tools die dit proces automatiseren. Er zijn verschillende mappings mogelijk, met ieder zo hun consequenties voor query's en wijzigingen. De gebruikte mapping wordt aangegeven in de deployment descriptor. Het resultaat van een zogeheten vertical mapping van het UML diagram uit figuur 3 is te zien in figuur 5.

JDO IDENTITY Applicaties hebben de behoefte objecten te kunnen onderscheiden op grond van een identiteit. De identiteit van persistent objects wordt bepaald door welke data ze in de data store representeren. JDO objecten zijn identiek als ze verwijzen naar dezelfde data. JDO borduurt in deze voort op het concept van Java identiteit. De identiteit van Java objecten



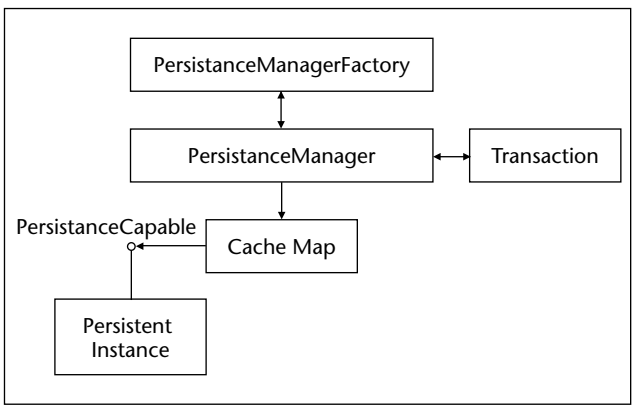
FIGUUR 5. Object Relational Mapping

wordt immers bepaald door de geheugen locatie die zij innemen in de Java Virtual Machine (JVM). Objecten zijn daarbij identiek als zij dezelfde geheugenlocatie innemen. Maar de JDO identiteit verschilt dus van de Java identiteit. De JDO specificatie onderscheidt een drietal mogelijke vormen van identiteit, die worden aangegeven in de deployment descriptor:

1) Datastore Identity : In dit geval wordt de identiteit gegenereerd door de JDO implementatie of de database. Alle persistent objects krijgen een unieke identiteit. Een extra kolom in de tabel voor de persistent class is nodig om deze identiteit op te slaan.
2) Application Identity : In dit geval wordt de identiteit bepaald door één of meer velden uit de persistent class die een primary key vormen en het object uniek identificeren.
3) Non-data Store Identity : Uniciteit wordt gegarandeerd in de JVM, maar wordt niet ondersteund in de data store. Deze vorm van identiteit wordt wel toegepast bij log files of als performance van belang is.

> Veel analisten hebben al opgemerkt dat JDO is wat entity beans hadden moeten zijn

JDO Architectuur Centrale element in de JDO applicaties is de PersistenceManager met een reeks van methoden voor het benaderen van persistente objecten. Een PersistenceManager is verantwoordelijk voor een eigen persistent objectcache. Deze cache kan veranderen als de applicatie, via methoden van PersistenceManager, de objecten benadert of transparant als de applicatie een attribuut benadert dat nog niet geladen is. In de cache worden ook zaken bijgehouden die betrekking hebben op de identiteit en toestand van de objecten, maar dit is niet zichtbaar voor de applicatie. Een PersistanceManager wordt aangemaakt door een PersistanceManagerFactory die op zijn beurt volgens een standaard JDO mechanisme wordt geconfigureerd op basis van een property's file:



Figuur 6. JDO Architectuur

```
PersistenceManagerFactory pmf = PersistenceManager-
Factories.getFactory();
PersistenceManager pm = pmf.getPersistenceManager();
```

Iedere PersistenceManager heeft een één-op-één relatie met een Transaction object voor het starten of beëindigen van een transactie.

JDO's Caching mechanisme Eén van de karakteristieke eigenschappen van JDO is dat het de illusie geeft dat persistente objecten zich in het geheugen bevinden terwijl ze feitelijk mogelijk uit de data store moeten worden opgehaald. Zodra de objecten worden benaderd, zorgt JDO ervoor dat ze in de cache van de PersistenceManager komen. Het expliciet laden van objecten is niet nodig. Ook zorgt JDO ervoor dat er slechts één kopie van het object in een transactie aanwezig is. Elke poging om een persistent object via welke methode dan ook te benaderen, geeft steeds hetzelfde object in de cache terug. Deze wijze van werken verschilt nogal van het mechanisme in andere data access applicaties en wordt door ontwikkelaars erg gewaardeerd.

Persistent versus Transient Als een persistent-capable object met de new operator wordt aangemaakt is het aanvankelijk transient. Het object wordt pas persistent gemaakt door de makePersistent methode van PersistenceManager aan te roepen. De aanroep van makePersistent moet volgens de JDO specificatie gedaan worden binnen een transactie. Een transient object gedraagt zich precies als een gewoon Java-object. Persistente objecten hebben een referentie naar een lokaal PersistanceManager object. JDO ondersteunt verder 'persistence-by-reacheability'. Alle objecten waarnaar verwezen wordt in een persistent object worden ook zelf persistent. Dit wordt bereikt zonder expliciete aanroepen in de applicatie.

Transacties Operaties op persistente objecten zijn onderdeel van transacties. De transactie wordt gestart door de begin methode van het Transaction interface dat wordt verkregen uit de currentTransaction method van een PersistanceManager. Bij een commit van een transactie worden alle wijzigingen van de attribuut waarden in het persistent object weggeschreven naar de data store en worden mogelijk ook persistente objecten

toegevoegd of verwijderd. Het feit dat dit automatisch gebeurt wordt wel 'transparante persistentie' genoemd. De operaties op de datastore kunnen ook worden teruggedraaid door een rollback.

```
Transaction t = null;
t = pm.currentTransaction(); // get transac-
tion
t.begin(); // start transaction
ProgrammerEmployee pro = new Programmer-
Employee(); // transient pro object
pm.makePersistent( pro ); // making pro per-
sistent
pro.setName("Bill"); // changing data
t.commit(); // commit transaction
```

OPVRAGEN VAN JDO OBJECTEN Applicaties kunnen persistente objecten vanzelfsprekend pas zinvol gebruiken als ze deze objecten kunnen ophalen, wijzigen en verwijderen. JDO kent de volgende methoden voor opvragen van objecten:

1) Via de getExtent() methode van PersistenceManager kan de Extent van een class worden opgevraagd. Een Extent representeert alle objecten van een bepaalde class, en optioneel zijn subclassen, in de data store.
2) Met een valide ObjectId kan de getObjectById methode van PersistenceManager worden aangeroepen. Het resultaat is een persistent object.
3) Door query's in JDO Query Language (JDOQL) kunnen objecten die voldoen aan de gewenste zoekcriteria worden opgevraagd.

JDO QUERY LANGUAGE Query's in JDO applicaties worden geformuleerd in Java Data Objects Query Language (JDOQL). Deze op Java gebaseerde query taal voorziet in een uniforme query syntax die de applicatie isoleert van de query-taal van de onderliggende datastore. Een query-object wordt gecreëerd door de newQuery method van de PersistenceManager. Het uitvoeren van een query in JDOQL komt neer op het toepassen van een booleaans filter op een verzameling van objecten en het retourneren van de objecten die aan de filter conditie voldoen:

```
Extent ext =
pm.getExtent(ProgrammerEmployee.class, true);
Query qry = pm.newQuery();
qry.declareParameters("int nrLines, int
nrBugs");
qry.setFilter("lines == nrLines | bugs ==
nrBugs");
qry.setCandidates(ext);
Collection res = (Collection) qry.execu-
te("1000", "0");
```

J2EE INTEGRATIE JDO kan geïntegreerd worden in een managed J2EE omgeving. In een dergelijke omgeving draaien Enterprise Java Beans (EJB) componenten (session, message-driven en entity beans) in de context van een EJB container. De EJB container is verantwoordelijk voor transactiemanagement, concurrency en security. JDO maakt in deze context gebruik van de transactiemechanismen van de EJB container. Het meest voor hand ligt de JDO integratie in J2EE session beans. In essentie komt de integratie erop neer dat vroeg in de levenscyclus van de beans, een PersistenceManagerFactory wordt aangemaakt. Deze wordt vervolgens bewaard als attribuut van de bean class. Methoden die persistente objecten benaderen, openen dan eerst een PersistenceManager. Deze PersistanceManager wordt weer gesloten voor de methode retourneert om te voorkomen dat de transactiemechanismen van EJB container en JDO conflicteren. Andere opties zijn overigens JDO te gebruiken vanuit de BMP (Bean Managed Persistence) entity beans in plaats van JDBC of vanuit de non-managed web tier, in servlets en JSP's.

TOT BESLUIT JDO is een nog jonge veelbelovende nieuwe technologie voor persistentie in Java applicaties.

> Serialisatie biedt geen serieus alternatief ten opzichte van een robuuste database-omgeving

JDO ondersteunt de transparante persistentie van het domein-object model waarin alle objectgeoriënteerde zaken als encapsulatie en overerving zijn toegestaan. Veel analisten hebben al opgemerkt dat JDO is wat entity beans hadden moeten zijn. Het is niet onwaarschijnlijk dat JDO de komende jaren een grote verspreiding gaat krijgen in J2EE omgevingen. In ieder geval doen bedrijven er verstandig geen nieuwe projecten te starten die gebaseerd zijn op entity beans met alvorens eerst eens goed naar JDO gekeken te hebben.

drs. Willem Koppenol is senior trainer en product manager bij Twice IT Training.