Objektorientierte Datenbanken

Vorlesung 5 (JDOQL, Bausteine)

Sebastian Iwanowski
FH Wedel

Datenbankanfragemöglichkeiten ohne Anfragesprache

a) **Extraktion eines einzelnen Objects (spezifiziert durch Name / ID)**

b) **Extraktion des Extents einer Klasse**

Ungeeignet für das Problem:

Finde Objekte mit bestimmten Eigenschaften !

a) Hoher Programmieraufwand

b) Hoher Hauptspeicherbedarf

Ziele einer Objektanfragesprache

Beispiele: OQL, JDOQL, HQL, VQL

1) **gezielte Extraktion von Objekten mit bestimmten Eigenschaften**

   mehr Programmierkomfort, mehr Effizienz

2) **SQL-ähnliche Abfragemöglichkeiten ohne detaillierte Java-Kenntnisse**

   politischer und sozialer Grund

   mehr Abfragekomfort, mehr Effizienz

Für JDOQL in Version 1.0.1 galt nur das 1. Ziel

In Version 2.0 wurde auch das 2. Ziel realisiert

Wesentliche Merkmale von JDOQL-Queries

- **Jede Query bezieht sich meistens auf genau eine Klasse**

- **Jede Query sucht aus einer Menge von Elementen dieser Klasse diejenigen Elemente, die eine bestimmte Eigenschaft erfüllen**

2 Anfragetypen:

1) Filter-Queries (aus Version 1.0.1, auch in 2.0 verfügbar)

- **Die gesuchte Eigenschaft der Objekte wird durch einen Booleschen Ausdruck bestimmt (*Filter* genannt)**

`2) Single-String-Queries (nur in 2.0 verfügbar)`

- **Die gesuchte Eigenschaft der Objekte wird durch eine SQL-ähnliche Frage bestimmt (eingeleitet durch `SELECT`)**

Einstiegsbeispiel für JDOQL, Single-String-Query

```
Properties pmfProps = new java.util.Properties();
pmfProps.put ("javax.jdo.PersistenceManagerFactoryClass",
        "com.versant.core.jdo.BootstrapPMF" );
pmfProps.put ("javax.jdo.option.ConnectionURL", "versant:myDatabase@host" );
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory( pmfProps );
PersistenceManager pm = pmf.getPersistenceManager();
Transaction txn = pm.currentTransaction();

txn.begin();

// Frage formulieren:
String queryString = " SELECT FROM Professoren WHERE Raum == `Ü11` ";
// Frageobjekt erzeugen:
Query query = pm.newQuery (queryString );
// Frage stellen:
Collection result = (Collection) query.execute();
// Ergebnis auswerten:
Iteration iter = result.iterator();
Professoren professor;
while (iter.hasNext()) {
      professor = (Professoren) iter.next();
      print (professor.name);}

txn.commit();
```

Einstiegsbeispiel für JDOQL, *Filter-Query*

```
Properties pmfProps = new java.util.Properties();
pmfProps.put ("javax.jdo.PersistenceManagerFactoryClass",
        "com.versant.core.jdo.BootstrapPMF" );
pmfProps.put ("javax.jdo.option.ConnectionURL", "versant:myDatabase@host" );
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory( pmfProps );
PersistenceManager pm = pmf.getPersistenceManager();
Transaction txn = pm.currentTransaction();

txn.begin();
Extent AlleProfessoren = pm.getExtent (Professoren.class, true);
// Frage formulieren:
String queryString = " Raum ==  `Ü11` ";
// Frageobjekt erzeugen:
Query query = pm.newQuery (AlleProfessoren, queryString );
// Frage stellen:
Collection result = (Collection) query.execute();
// Ergebnis auswerten:
Iteration iter = result.iterator();
Professoren professor;
while (iter.hasNext()) {
      professor = (Professoren) iter.next();
      print (professor.name);}

txn.commit();
```

Funktionalität von JDOQL-Queries

Grundfunktionalität:

Interface PersistenceManager

public Query newQuery ();

public Query newQuery (Class class, Collection elements, String filter);

public Query newQuery (Extent elements, String filter);

```
public Query newQuery (String single-string-query);
public Query newQuery (String language, Object language-specific-query);
```

Interface Query

public void setClass (Class class);

public void setCandidates (Collection elements);

public void setCandidates (Extent elements);

public void setFilter (String booleanExpression);

public Object execute (); gibt per default eine Collection zurück

Funktionalität von JDOQL-Queries

Syntax von Single-String-Queries:

```
select [unique] [ <result> ] [into <result-class-name>]
[from <candidate-class-name> [exclude subclasses] ]
[where <filter>]
[variables <variable -list> ]
[parameters <parameter-list>]
[imports <import-list>]
[group by <grouping-clause> ]
[order by <ordering-clause>]
[range <from-range> ,<to-range>]
```

aus: http://www.solarmetric.com/resources/jdoql-quickref.pdf

Funktionalität von JDOQL-Queries

Merkmale von Filter-Queries:

Filterdefinition:

- *Der Filter entspricht einer Java-Methode, die* ***`boolean`*** *zurückgibt.*
- *Die Syntax des Filters ist gleich normaler Java-Syntax.*
- *Es sind fast alle Java-Operationen zulässig (Ausnahmen später).*

Filterauswertung:

- *Der Filter wird auf jedes Element der Kandidatenmenge angewandt.*
- *Die Attribute der Kandidatenelemente können im Filtercode direkt angesprochen werden. Das angesprochene Attribut bezieht sich auf das Element, das gerade ausgewertet wird. Alle Elemente, für die der Filter* ***`true`*** *ergibt, werden im* ***`execute`*** *zurückgegeben.*
- *Wenn kein Filter explizit definiert wurde, wird die gesamte Kandidatenmenge im* ***`execute`*** *zurückgegeben.*

JDOQL-Queries: Einfaches Beispiel

Single-String-Query:

- Finde alle Studenten, die von Iwanowski geprüft wurden:

select s **from** Studenten

where s.wurdeGeprüft.Prüfer.Name == „Iwanowski“

Filter-Query ?

Achtung:
Diese Lösung setzt voraus,
dass wurdeGeprüft ein einwertiges Attribut ist,
d.h. jeder Student nur eine Prüfung macht.

Für die Abfrage nach mehrwertigen Attributen
siehe Folie OODB 5-14.
Weitere Beispiele dazu in OODB 6.

Wie kann man erreichen, dass dieselbe Frage für beliebige Professoren gestellt werden kann ?

Single-String-Query:

- Finde alle Studenten, die von einem bestimmten Professor geprüft wurden:

```
select s from Studenten
where s.wurdeGeprüft.Prüfer.Name == name
```

Filter-Query ?

Achtung:
Diese Lösung setzt eine Parameterdeklaration voraus und eine Verwendung des Parameters im execute!

Funktionalität von JDOQL-Queries

Query mit Parametern:

Interface Query

public void declareParameters (String declarationOfParams);

Deklaration der formalen Parameter für diese Query

public Object execute (Object param1);

public Object execute (Object param1, Object param2);

public Object execute (Object param1, Object param, Object param3);

Ausführung der Query mit aktuellen Parametern

- **declarationOfParams: normale Java-Syntax**
- **Typen und Anzahl der Parameter müssen zueinander passen**
  Ausnahme: primitive Datentypen als formale Parameter werden mit zugehörigen Objektdatentypen als aktuelle Parameter aufgerufen, Beispiel: `int` mit `Integer`
- **im Filter dürfen die Parameter wie lokale Variablen benutzt werden**
  (mit den Namen, die in declarationOfParams deklariert wurden)

Funktionalität von JDOQL-Queries

Query mit mehr als 3 Parametern:

Interface Query

public void declareParameters (String declarationOfParams);

public Object executeWithMap (Map mapOfParams);

public Object executeWithArray (Object[] arrayOfParams);

- **Deklaration der formalen Parameter und Verwendung im Filter wie zuvor**
- **Ausführung mit aktuellen Parametern:**

  Vor dem Aufruf mit `execute` muss die aktuelle Parameterliste als `Map` oder `Array` erzeugt werden:

  - **Maps** — **key: Parametername (als `String`)**
    value: eingesetzter Wert vom Typ wie in declarationOfParams angegeben
  - **Arrays** — **Elemente: eingesetzte Werte in derselben Reihenfolge wie in declarationOfParams**

  Typen und Anzahl der Parameter müssen zueinander passen
  (mit derselben Ausnahme für einfache Datentypen wie zuvor)

JDOQL: Bsp. für Attribut, das selbst Collection ist

SQL-ähnliche Frage:

- Finde alle Studenten, die bei Iwanowski Vorlesung haben:

select s **from** Studenten, v in s.hört

where v.gelesenVon.Name = „Iwanowski“

Wie geht das mit Single-String-Queries ?

Filter-Query ?

Funktionalität von JDOQL-Queries

Zugriff auf Attribute, die selbst Collections sind:

Zugriff auf Collection-wertige Attribute im Filtercode mit den Methoden:

- **`Collection.isEmpty ()`**
- **`Collection.contains (varname)`**

zur Abfrage der Existenz von Elementen mit bestimmten Eigenschaften

`varname` muss als Variable deklariert werden.
`varname` bindet ein Element der `Collection`
und sollte in einem zweiten Booleschen Ausdruck weiter eingeschränkt werden

Interface Query

public void declareVariables (String declarationOfVariables);

- **declarationOfVariables: normale Java-Syntax**
- **im Filter dürfen die Variablen als lokale Variablen benutzt werden**
  (mit den Namen, die in declarationOfVariables deklariert wurden)

Deklarationen in JDOQL

„Normaler“ Weg (JDO 1.0.1):

Interface Query

public void declareParameters (String declarationOfParams);

public void declareVariables (String declarationOfVariables);

Manchmal sind Importe notwendig:

public void declareImports (String declarationOfImports);

- **declarationOfImports: normale Java-Syntax**
- **Alle Klassen aus dem Package java.lang sind per default bekannt.**
- **Alle Klassen aus dem Package der zur Query gehörenden Kandidatenklasse sind per default bekannt.**
- **Alle anderen Klassen müssen hier importiert werden (normale Java-Syntax).**

Weitere Wege (nur für JDO 2.0):

- **optional in single strings der Select-Queries**
- **durch implizite Vereinbarungen (siehe Spec. 2.0, Kap. 14.6.3, S. 153)**

Unterschiede der Filtersprache zu Java

Aus der Java-Dokumentation der Methode `Query.setFilter (String filter):`

Rules for constructing valid expressions follow the Java language, except for these differences:

- Equality and ordering comparisons between primitives and instances of wrapper classes are valid.
- Equality and ordering comparisons of **`Date`** fields and **`Date`** parameters are valid.
- White space (non-printing characters space, tab, carriage return, and line feed) is a separator and is otherwise ignored.
- The assignment operators =, +=, etc. and pre- and post-increment and -decrement are not supported. Therefore, there are no side effects from evaluation of any expressions.
- Methods, including object construction, are not supported, except for **`Collection.contains(Object o)`**, **`Collection.isEmpty()`**, **`String.startsWith(String s)`**, and **`String.endsWith(String e)`**. Implementations might choose to support non-mutating method calls as non-standard extensions.
- Navigation through a **`null`**-valued field, which would throw **`NullPointerException`**, is treated as if the filter expression returned **`false`** for the evaluation of the current set of variable values. Other values for variables might still qualify the candidate instance for inclusion in the result set.
- Navigation through multi-valued fields (**`Collection`** types) is specified using a variable declaration and the **`Collection.contains(Object o)`** method.

Unterschiede der Filtersprache zu Java

Zusammenfassung der wichtigsten Unterschiede:

- **keine Veränderungen der abgefragten Objekte möglich**
- **keine Methodenaufrufe möglich (mit wenigen Ausnahmen)**
- **Wenn Abfragen nicht möglich sind (Exceptions erfordern würden), wird `false` zurückgegeben.**

Unterschiede der Filtersprache zu Java

In Version 2.0 gibt es mehr Möglichkeiten (siehe Spec. 2.0, Kap. 14.6.2, S. 150 ff.)***:***

Table 4: Query Operators

Operator	Description
==	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
&	boolean logical AND (not bitwise)
&&	conditional AND
|	boolean logical OR (not bitwise)
||	conditional OR
~	integral unary bitwise complement
+	binary addition, unary plus, or String concatenation
-	binary subtraction or unary numeric sign inversion
*	times
/	divide by
!	logical complement
%	modulo operator
instanceof	instanceof operator

Table 5: Query Methods

Method	Description
contains(Object)	applies to Collection types
get(Object)	applies to Map types
containsKey(Object)	applies to Map types
containsValue(Object)	applies to Map types
isEmpty()	applies to Map and Collection types
size()	applies to Map and Collection types
toLowerCase()	applies to String type
toUpperCase()	applies to String type
indexOf(String)	applies to String type; 0-indexing is used
indexOf(String, int)	applies to String type; 0-indexing is used
matches(String)	applies to String type; only the following regular expression patterns are required to be supported and are portable: global "(?i)" for case-insensitive matches; and "." and ".*" for wild card matches. The pattern passed to matches must be a literal or parameter.
substring(int)	applies to String type
substring(int, int)	applies to String type
startsWith(String)	applies to String type
endsWith(String)	applies to String type
Math.abs(numeric)	static method in java.lang.Math, applies to types of float, double, int, and long
Math.sqrt(numeric)	static method in java.lang.Math, applies to double type
JDOHelper.getObjectId(Object)	static method in JDOHelper, allows using the object identity of an instance directly in a query.