

# EIN JAVA-PAKET ZUR VERARBEITUNG VON DATENSTRUKTUREN IN BELIEBIGEN DATENSPEICHERN

Torsten Richter, Berthold Firmenich, Karl Beucke

{torsten.richter | berthold.firmenich | karl.beucke}@bauing.uni-weimar.de

Bauhaus-Universität Weimar, Informatik im Bauwesen

**Kurzfassung:** Für die Planung von Bauwerken auf Grundlage eines Datenmodells ist es wichtig, dieses speichern und später Teilmodelle zur Bearbeitung bilden zu können. Im Bereich der Produktmodelle werden von der IAI (International Alliance for Interoperability) die IFC (International Foundation Classes) propagiert. Mit der in diesem Aufsatz vorgestellten Feature-Logic steht ein Konzept zur Repräsentation beliebiger Datenmodelle zur Verfügung, das die Speicherung in verschiedenen Arten von Datenspeichern und die Teilmengenbildung in einer einfachen Art und Weise ermöglicht. Hierfür wird eine Umsetzung als Java-Paket für zwei verfügbare Datenbanken vorgestellt.

## 1. Motivation für Feature-Logic

Bei der Planung eines Bauwerks am Computer wird das Planungsmaterial mit Hilfe eines Modells abgebildet. Dieses Modell muss persistent speicherbar sein. Für die spätere Bearbeitung sind die Selektion, das Laden und das Speichern von Teilmodellen wichtige Funktionalitäten. Ausgehend von einem versionierten System mit dem Konfliktmanagement nach [3] sind nur seltene Zugriffe zu erwarten.

Im Bereich des Bauwesens nutzen viele Anwendungen die von der IAI entwickelten IFC. Mit Hilfe von EXPRESS-G lassen sich aus den IFC Teilmodelle bilden. EXPRESS-G ist eine graphische Beschreibungssprache für objektorientierte Produktmodelle, die im ISO-Standard aufgenommen wurde (ISO 10303). Die Anbieter kommerzieller CAD-Software haben die Funktionalität der Teilmengenbildung noch nicht in ihre Programme aufgenommen. Auch ist derzeit noch die Teilmengenbildung in den IFC kaum dokumentiert.

Für Operationen auf dem Datenmodell ist eine geeignete Sprache auf Grundlage der Mengenalgebra erforderlich. Feature-Logic ist eine solche Sprache.

Feature-Logic ist eine von Smolka [5] entwickelte mathematische Theorie und kommt aus dem Bereich der Wissensrepräsentation. Zeller [6] zeigt die Anwendbarkeit im Bereich des Software Configuration Management. Firmenich [3] beschreibt die Anwendung auf strukturierte Objektversionsmengen für CAD im Bauplanungsprozess.

Mit dem Attributierungskonzept der Feature-Logic können beliebige Datenmodelle repräsentiert werden. So ist es auch möglich, ein Produktmodell basierend auf den IFC-Klassen [7] abzubilden. Ein weiterer Vorteil der Feature-Logic besteht in der Unabhängigkeit vom Speichermedium. So lässt sich das betrachtete Modell in relationalen (RDBMS) und objektorientierten (ODBMS) Datenbankmanagementsystemen oder in einer normalen Datei abspeichern.

Feature-Logic ist eine Mengenalgebra. Teilmengen werden über definierte Verknüpfungen gebildet [4]. Dafür sind die inneren Operationen Komplement, Vereinigung und Durchschnitt definiert. Zusätzlich besitzt Feature-Logic noch äußere Operatoren, die auf den Eigenschaften der Elemente beruhen. Die wesentlichen Operationen der Feature-Logic sind in [1, Tabelle 2] aufgeführt.

## 2. Beispiel

Zur Verdeutlichung des Lösungsansatzes wird in diesem Beitrag ein durchgängiges Beispiel verwendet, das analog zu [3] gewählt wird.

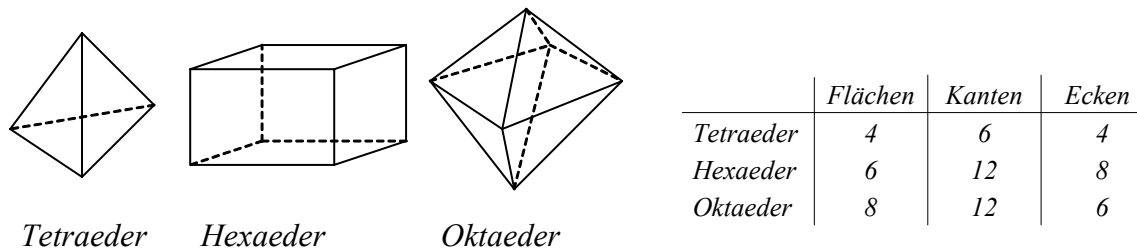


Abbildung 1: Polyeder und ihre Eigenschaften

Abbildung 1 stellt die Eigenschaften einiger Polyeder dar. Die Polyeder sollen mit der Feature-Logic modelliert werden. Ein Mengenbezeichner mit hochgestelltem  $I$  entspricht der Menge selbst, ohne den Kennzeichner  $I$  ist die Beschreibung der Menge gemeint. Alle verwendeten Elemente müssen in der so genannten Domäne  $D$  eingetragen sein. In diesem Beispiel gilt:

$$D^I = \{tetraeder, hexaeder, oktaeder, vier, sechs, acht, zwei\}$$

mit

$$D_A^I = \{vier, sechs, acht, zwei\} \quad \text{primitive Elemente (Atome)}$$

$$D_E^I = \{tetraeder, hexaeder, oktaeder\} \quad \text{nicht-primitive Elemente}$$

Die Attribute eines Objekts werden durch (Objektname, Attributwert) - Paare modelliert, die gemäß dem Attributnamen jeweils in eine Relation eingetragen werden. Die Attribute werden auch als Feature bezeichnet:

$$flaechen^I = \{(tetraeder, vier), (hexaeder, sechs), (oktaeder, acht)\}$$

$$kanten^I = \{(tetraeder, sechs), (hexaeder, zwoelf), (oktaeder, zwoelf)\}$$

$$ecken^I = \{(tetraeder, vier), (hexaeder, acht), (oktaeder, sechs)\}$$

Die Werte der Atome sind:

$$4^I = vier, \quad 6^I = sechs, \quad 8^I = acht, \quad 12^I = zwoelf$$

Das Beispiel beschränkt sich auf zwei Operationen. Dabei bezeichnet  $f$  ein Feature und  $S$  einen beliebigen Feature-Term:

$$f : S \quad \{x \in D^I \mid \exists_{s \in S^I} (x, s) \in f^I\}$$

**Operation Selektion:** Bezeichnet die Menge der Objekte, deren Feature  $f$  den Wert  $S$  haben.

$$f(S) \quad \{x \in D^I \mid \exists_{s \in S^I} (s, x) \in f^I\}$$

**Operation Extraktion:** Bezeichnet die Menge der Objekte, die Wert des Features  $f$  der Objektmenge  $S$  sind.

Mit diesen Operationen werden beispielhaft einige Feature-Terme formuliert:

$$kanten : \{sechs\} = \{tetraeder\}$$

Alle Polyeder mit 6 Kanten.

$$kanten : \{zwolf\} = \{hexaeder, oktaeder\}$$

Alle Polyeder mit 12 Kanten.

$$kanten : \{sechs, zwolf\} =$$

Alle Polyeder mit 6 oder 12 Kanten.

$$\{tetraeder, hexaeder, oktaeder\}$$

$$ecken : \{flaechen(oktaeder)\} = \{hexaeder\}$$

Alle Polyeder mit gleich viel Ecken wie der Oktaeder Flächen hat.

### 3. Schnittstelle für die Anwendung

Die Speicherung von Objekten mit der Feature-Logic ist durch ein Entity-Relationship-Modell in [1, Kapitel 3] ausführlich beschrieben.

Für die Verwendung in Java wurde das Paket *featureLogic* entwickelt, dessen Klassendiagramm in Abbildung 2 in UML-Notation dargestellt ist. Das Interface *FeatureLogic* definiert die zu implementierenden Methoden mit den zugehörigen Übergabeparamete-



Instanz der Klasse *Interpreter* zur Interpretation der Feature-Terme und zur Berechnung der damit beschriebenen Mengen.

Das Interface *ResultSetIterator* ist von *Iterator* abgeleitet. Die Implementierung *ResultSetIteratorImp* dient zur Traverse der berechneten Elementmenge.

```
interface ResultSetIterator extends Iterator:  
    public void close()  
    public String nextElement()
```

Das Interface *SetDsc* selbst ist leer und kann für jedes Datenmodell individuell gestaltet werden: Ein *SetDsc* (Set Deskriptor) dient der jeweiligen Implementierung zur Beschreibung einer Elementmenge.

```
interface SetDsc:  
    // Enthält keine Methoden
```

Die Klasse *FeatureLogicMyDatabase* erweitert *FeatureLogicAdapter* für die Anbindung an die Datenbasis *MyDatabase*, für die ein Zugriff mit dem Standard-Paket *java.sql* verfügbar sei.

```
class FeatureLogicMyDatabase extends FeatureLogicAdapter:  
    // Stellt im Konstruktor die Verbindung zu einer Datenbank her  
    public FeatureLogicMyDatabase(java.sql.Connection con)  
    ...
```

Nachfolgend wird die Anwendung des Pakets anhand eines Java-Fragments gezeigt: Es werden Feature-Terme über die Standard-Eingabe eingegeben, die korrespondierende Elementmenge mit Hilfe des Interpretierers berechnet, die Elemente der Menge mit Hilfe eines Iterators ermittelt und auf der Standard-Ausgabe ausgegeben:

```
import java.sql.Connection  
import java.sql.DriverManager  
  
class ParseFeatureTerm {  
    public static void main(String[] args) {  
        Connection con = DriverManager.getConnection(args[0], args[1], args[2]);  
        FeatureLogic featureLogic = new FeatureLogicMyDatabase(con);  
        while (true) {  
            System.out.print("Enter Feature Term: ");  
            ResultSetIterator rsi = featureLogic.interpret(System.in);  
            while (rsi.hasNext()) {  
                System.out.println("\t" + rsi.next());  
            }  
        }  
    }  
}
```

Bei der Berechnung der in Kapitel 2 vorgestellten Feature-Terme erhält man folgende Ausgabe:

```
java ParseFeatureTerm databaseURL username password  
Enter Feature Term: kanten:{6}  
    tetraeder  
Enter Feature Term: kanten:{12}  
    hexaeder  
    oktaeder
```

```

Enter Feature Term: kanten:{6,12}
    tetraeder
    hexaeder
    oktaeder
Enter Feature Term: ecken:{oktaeder.flaechen}
    hexaeder
Enter Feature Term:

```

## 4. Umsetzung

Es werden Umsetzungen für zwei verfügbare relationale Datenbanken gezeigt. Für das Beispiel erhält man folgende Tabellenbelegung:

Domain

Element
Tetraeder
Hexaeder
Oktaeder
Vier
Sechs
Acht
Zwoelf

Atom

element	value
vier	4
sechs	6
acht	8
zwoelf	12

Feature

feature
flaechen
kanten
ecken

RelSlot

Object	feature	value
Tetraeder	flaechen	vier
Hexaeder	flaechen	sechs
Oktaeder	flaechen	acht
tetraeder	Kanten	sechs
Hexaeder	Kanten	zwoelf
Oktaeder	Kanten	zwoelf
tetraeder	Ecken	vier
Hexaeder	Ecken	acht
Oktaeder	Ecken	sechs

Die Methoden der Feature-Logic müssen gemäß den Möglichkeiten des gewählten Datenspeichers umgesetzt werden. Im konkreten Fall bietet sich die SQL-Sprache an. Obwohl SQL unter dem Namen SQL/92 in die ISO aufgenommen wurde, besitzen die SQL-Implementierungen der verfügbaren RDBMS zum Teil erhebliche Abweichungen und Ergänzungen zum Standard.

## Oracle

Oracle besitzt eine außerordentlich leistungsfähige SQL-Implementierung. Wesentlich für die konkrete Umsetzung ist die Möglichkeit der Schachtelung von SELECT-Abfragen. Damit gelingt es, beliebig komplexe Feature-Terme mit einer einzigen SQL-Abfrage zu berechnen. Die Implementierung des Interface SetDsc besitzt ein String-Attribut mit der SQL-Abfrage:

```
class SetDscOracle implements SetDsc {
    private String m_query;
    public SetDscOracle(String query) {
        m_query = query;
    }
    public String toString() {
        return m_query;
    }
}
```

Abbildung 3 zeigt das Klassendiagramm für die Oracle-Implementierung:

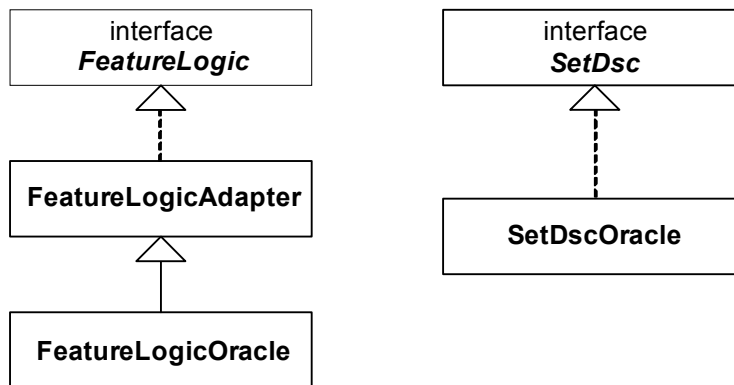


Abbildung 3: Klassendiagramm für die Datenbank Oracle

Nachfolgend wird beispielhaft die Implementierung einiger wesentlicher Grundoperationen der Feature-Logic gezeigt. Ergebnis einer jeden Methode ist ein Objekt vom Typ *SetDscOracle*.

- Elemente der Domäne werden über ihren Namen identifiziert. Die Methode *element()* liefert eine Beschreibung für die Teilmenge mit genau diesem einen Element oder die leere Menge, falls das Element nicht existiert:

```
SetDsc element(String x) {
    return new SetDscOracle(
        "SELECT element FROM Domain WHERE element = '" + x + "'");
}
```

- Primitive Elemente können über den ihnen zugewiesenen Wert ermittelt werden. Die Methode *primitiveElement()* liefert eine Beschreibung für die Teilmenge mit genau diesem einen Element oder die leere Menge, falls der Wert nicht existiert:

```
SetDsc primitiveElement(int atom) {
    return new SetDscOracle(
        "SELECT element FROM Atom WHERE value = '" + atom + "'"
    );
}
```

- Mit *extract()* wird die Grundoperation  $f(S)$  umgesetzt. Die Operation ist ein Beispiel für die Schachtelung der Abfragen:

```
SetDsc extract(String f, SetDsc s) {
    return new SetDscOracle(
        "SELECT DISTINCT RelSlot.value AS element" +
        " FROM (" + s + ") S, RelSlot" +
        " WHERE RelSlot.feature = '" + f + "' AND" +
        " Relslot.object = S.element"
    );
}
```

- Mit *selection()* wird die Grundoperation  $f:S$  umgesetzt. Die Operation ist ein weiteres Beispiel für die Schachtelung der Abfragen:

```
SetDsc selection(String f, SetDsc s) {
    return new SetDscOracle(
        "SELECT DISTINCT Relslot.object AS element" +
        " FROM (" + s + ") S, RelSlot" +
        " WHERE Relslot.feature = '" + f + "' AND" +
        " Relslot.value = S.element"
    );
}
```

Das Zusammenwirken der Methoden wird beispielhaft erläutert, indem für den Feature-Term *ecken:{flaechen(oktaeder)}* die resultierende SQL-Abfrage ermittelt wird. Nach dem Aufruf von *element()* ist die resultierende Abfrage:

```
SELECT element
FROM Domain
WHERE element = 'oktaeder'
```

Auf dieser Grundlage ist die resultierende SQL-Abfrage der Methode *extract()*:

```
SELECT DISTINCT RelSlot.value AS element
FROM (
    SELECT element
    FROM Domain
    WHERE element = 'oktaeder'
) S, RelSlot
WHERE RelSlot.feature = 'flaechen' AND Relslot.object = S.element
```

Schließlich liefert *selection()* auf dieser Basis die endgültige SQL-Abfrage:

```
SELECT DISTINCT Relslot.object AS element
FROM (
    SELECT DISTINCT RelSlot.value AS element
    FROM (
        SELECT element
        FROM Domain
        WHERE element = 'oktaeder'
```

```

) S, RelSlot
WHERE RelSlot.feature = 'flaechen' AND Relslot.object = S.element
) S, RelSlot
WHERE Relslot.feature = 'ecken' AND" Relslot.value = S.element"

```

Die resultierende Abfrage wird ausgeführt. Für die Traverse wird ein ResultSetIterator-Objekt geliefert.

## Access

Access besitzt derzeit keine Möglichkeit einer Schachtelung von SELECT-Abfragen: Daher muss ein vollkommen anderer Lösungsansatz gewählt werden. Dies führt nicht nur zu einer anderen Implementierung der Java-Klassen, sondern auch zu einer Erweiterung des Schemas der Datenbank.

Bei der Auswertung eines Feature-Terms müssen die Ergebnismengen der Grundoperationen in der Datenbank zwischengespeichert werden. Hierzu dient die Relation ResultSet, in der (Objekt, Mengenindex) - Paare eingetragen sind: Der Mengenindex iSet ordnet das Objekt einer bestimmten Teilmenge der Domäne zu. Mögliche Werte der Relation ResultSet sind nachfolgend beispielhaft dargestellt:

Object	iSet
Tetraeder	0
Hexaeder	0
Oktaeder	0
Oktaeder	1

Diese Relation beschreibt zwei Mengen:

Teilmenge mit dem Index 0:  $M_0 = \{tetraeder, hexaeder, oktaeder\}$

Teilmenge mit dem Index 1:  $M_1 = \{oktaeder\}$

Nachfolgend wird ein Fragment der Implementierung des SetDsc beschrieben, dessen Objekte eine Folge von SQL-Abfragen und einen Mengenindex enthalten. Werden die SQL-Abfragen der Reihe nach ausgeführt, so ist die resultierende Ergebnismenge unter dem gespeicherten Mengenindex verfügbar:

```

class SetDscAccess implements SetDsc {
    int m_iSet;
    List m_sqlCmds = new ArrayList();

    SetDscAccess(int iSet);
    SetDscAccess(int iSet, String sql);
    int getIndex();
    Iterator sqlIterator();
    void add(int iSet, String sql);
    void add(int iSet, SetDscAccess setDsc);
}

```

Abbildung 4 zeigt das Klassendiagramm für die Datenbank Access.

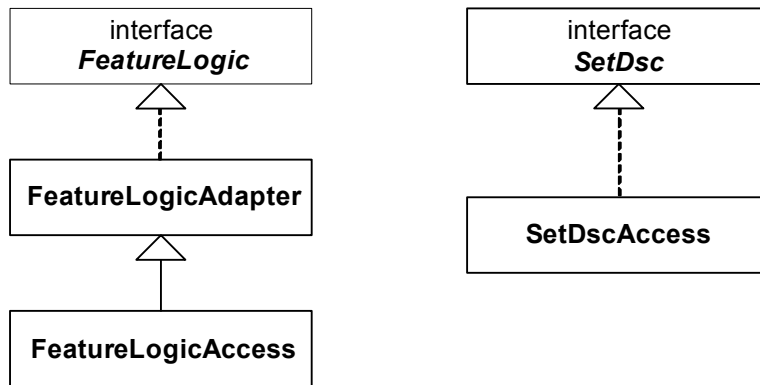


Abbildung 4: Klassendiagramm für die Datenbank Access.

Für den bekannten Feature-Term *ecken: {flaechen(oktaeder)}* wird die Folge der SQL-Abfragen und der Mengenindex ermittelt. Der Aufruf von *element()* liefert die erste auszuführende Abfrage. Das Element *Oktaeder* wird in Menge 0 eingetragen:

```

INSERT INTO ResultSet (iSet, object)
  SELECT 0 AS iSet, Domain.element AS object
  FROM Domain
  WHERE element= 'oktaeder';
  
```

Die Methode *extract()* liefert den Wert des Features *flaechen* des Oktaeders. Als Ergebnis wird das Element *acht* in die Menge 1 eingetragen:

```

INSERT INTO ResultSet (object, iSet)
  SELECT DISTINCT RelSlot.value AS object, 1 AS iSet
  FROM RelSlot INNER JOIN ResultSet
  ON RelSlot.object=ResultSet.object
  WHERE ResultSet.iSet= 0 AND RelSlot.feature='flaechen';
  
```

Schließlich wird mit der *selection()* Methode die Menge der Objekte ermittelt, die acht Ecken haben: Die Ergebnismenge 2 enthält den Hexaeder:

```

INSERT INTO ResultSet (iSet, object)
  SELECT DISTINCT 2 AS iSet, RelSlot.object AS object
  FROM RelSlot, ResultSet
  WHERE RelSlot.feature = 'ecken'
  AND RelSlot.value=ResultSet.object
  AND ResultSet.iSet = 1;
  
```

Die resultierende Abfrage wird ausgeführt. Für die Traverse wird ein *ResultSetIterator*-Objekt geliefert.

## Fazit

Die Programmierschnittstellen verfügbarer Datenspeicher sind von unterschiedlicher Konzeption und Leistungsfähigkeit. Im Kontext des vorliegenden Beitrags erfordert diese Situation einen möglichst flexiblen Lösungsansatz, dessen generelle Tauglichkeit bei der Umsetzung für zwei verfügbare Datenbanksysteme nachgewiesen wurde.

## 5. Ausblick

Ein wesentliches Ziel besteht in der Einbindung der Feature-Logic in das von der DFG geförderte Projekt interCAD. Praxistauglichkeit und Performance sind noch für real zu erwartende Datenmengen zu testen. Dabei sollen unterschiedliche Datenspeicher verglichen werden. Ein weiterer Forschungsschwerpunkt ist der Übergang von der transienten zur persistenten Speicherung der Objekte.

## Danksagung

Die Ergebnisse dieses Aufsatzes entstanden im Rahmen des DFG-Schwerpunktprogramms 1103 „Vernetzt-kooperative Planungsprozesse im konstruktiven Ingenieurbau“ im Projekt interCAD. Das Projekt wird von der DFG gefördert.

## Literatur

- [1] BEER, G.; FIRMENICH, B.; BEUCKE, K.: *Motivation für eine Sprache zur Handhabung strukturierter Objektversionsmengen*. Hannover: 2003
- [2] DEHNHARDT, W.: *Anwendungsprogrammierung mit JDBC*. München: Hanser, 1999
- [3] FIRMENICH, B.: *CAD im Bauplanungsprozess: Verteilte Bearbeitung einer strukturierten Menge von Objektversionen*. Aachen: Shaker, 2002
- [4] PAHL, P. J.; DAMRATH, R.: *Mathematische Grundlagen der Ingenieurinformatik*. Berlin: Springer, 2000
- [5] SMOLKA, G.: *Feature constraint logics for unification grammars*. In: *The journal of logic programming*. New York, 1992
- [6] ZELLER, A.: *Configuration Management with Version Sets*. Dissertation am Fachbereich Mathematik und Informatik der Technischen Universität Braunschweig, 1997
- [7] Webseite der IAI. URL: <http://www.iai-ev.de/spezifikation/lfc2x/index.htm>. Stand 14.07.2003