Realisierung rekursiver Datenstrukturen durch generische Klassen

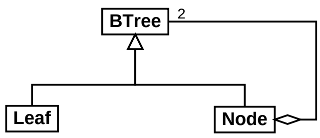
Andreas Vox

Institut für Softwaretechnik und Programmiersprachen
Medizinische Universität zu Lübeck
http://www.isp.mu-luebeck.de
vox@isp.mu-luebeck.de

Die Modellierung rekursiver Datenstrukturen, d.h. durch Tupel, Alternativen und Rekursion aufgebauter Datenstrukturen, ist ein häufiger Anwendungsfall in der Informatik. Beispiele sind Hierarchien von GUI-Komponenten, durch XML beschriebene Datenstrukturen oder abstrakte Syntaxbäume. Zur Realisierung empfehlen Gamma et al. [4] das Composite-Entwurfsmuster, ergänzt durch das Visitor-Muster und das Factory-Muster. Der Nachteil des Composite-Muster ist, dass die Definition einer rekursiven Datenstruktur auf viele Klassen aufgesplittet wird. So entspricht der kompakten SML-Deklaration

```
datatype BTree = Leaf | Node of BTree * BTree
```

nach dem Composite-Muster diese Klassenstruktur:



Man erkennt, dass für jede Alternative eine zusätzliche Klasse benötigt wird. Komplexe Datenstrukturen wie abstrakte Syntaxbäume oder XML-Dokumentstrukturen benötigen daher eine große Anzahl von Klassen – sofern die Datenstruktur typisiert implementiert wird – und lassen sich kaum noch von Hand pflegen. Der vorliegende Artikel stellt das neue Muster „Generic Factory“ vor, bei dem die gesamte Struktur durch ein einziges Interface beschrieben wird. Zum Beispiel beschreibt das Interface

```
interface GenericBTreeFactory <X> {
    X caseLeaf();
    X caseNode(X l, X r);
}
```

präzise die Datenstruktur `BTree` (das Muster benutzt generische Klassen und Methoden, die z.B. in Generic Java(GJ) [3], [5] vorhanden sind).

Die Implementierungsklassen der Datenstruktur (`BTree`, `Leaf`, `Node`) können automatisch generiert werden, so dass die Mehrzahl der benötigten Klassen nicht mehr von Hand gewartet werden muss.

Zu einer rekursiven Datenstruktur gehören auch entsprechende Algorithmen zur Erzeugung, Traversierung, Suche und Umformung. Diese Algorithmen lassen sich als Implementierungen einer Generic Factory oder eines Visitors formulieren. Standard-Algorithmen wie Erzeugung oder Traversierung können auch automatisch generiert werden.

Das Generic-Factory-Muster sieht eine `reduce`-Methode in den Datenklassen vor, die es erlaubt, die Datenstruktur anhand einer übergebenen Factory-Klasse in eine andere Struktur umzuformen. Dabei wird bottom-up das Ergebnis generiert:

```
abstract class BTree {
    <R> R reduce (GenericBTreeFactory<R> fab);
...}
class Leaf extends BTree {
    <R> R reduce (GenericBTreeFactory<R> fab) {
        return fab.caseLeaf();
}...}
class Node extends BTree {
    BTree left; BTree right;
    <R> R reduce (GenericBTreeFactory<R> fab) {
        return fab.caseNode(left.reduce(fab), right.reduce(fab));
}...}
```

Für einen `BTree` *bt* entspricht der Aufruf *bt*`.reduce(new BTreeFactory())` dem tiefen Klonen von *bt*. Diese Vorgehensweise ist eng mit Catamorphismen [2] und dem algebraischen Ansatz zu Generischer Programmierung [1] verwandt.

Das Generic-Factory-Muster bietet den methodischen Vorteil, dass die Implementierungsklassen einer rekursiven Datenstruktur sicher und in kontrollierter Weise generiert werden. Die Kapselung der Datenstruktur im Generic-Factory-Interface erhöht die Übersichtlichkeit und macht das System besser wartbar. Die Kapselung von Algorithmen in Visitors oder Generic Factories eröffnet neue Möglichkeiten der Komposition, Transformation und Wiederverwendung, die zur Zeit untersucht werden. Anhand einer größeren Anwendung soll das Generic-Factory-Muster in Hinblick auf inkrementelle Softwareentwicklung erprobt werden. Des Weiteren ist ein Werkzeug zur Codegenerierung in Entwicklung.

Literatur

[1] Backhouse, Roland, Patrik Jansson, Johan Jeuring und Lambert Meertens: *Generic Programming –An Introduction–*. In: Swierstra, S. Doaitse, Pedro R. Henriques und José N. Oliveira (Herausgeber): *Revised Lectures 3rd Int. School on Advanced Functional Programming, AFP'98, Braga, Portugal, 12–19 Sept. 1998*, Band 1608 der Reihe *Lecture Notes in Computer Science*, Seiten 28–115. Springer-Verlag, Berlin, 1999.

[2] Bird, Richard S.: *An Introduction to the Theory of Lists.* In: Broy, M. (Herausgeber): *Logic of Programming and Calculi of Discrete Design*, Band 36 der Reihe *NATO ASI Series F*, Seiten 3–42. Springer-Verlag, Berlin, 1987.

[3] Bracha, Gilad, Martin Odersky, David Stoutamire und Philip Wadler: *Making the future safe for the past: Adding Genericity to the Java Programming Language.* In: *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM.

[4] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster.* Addison-Wesley, 1996.

[5] Odersky, Martin und Philip Wadler: *Pizza into Java: Translating theory into practice.* In: *24'th ACM Symposium on Principles of Programming Languages*, Paris, January 1997.