ECONSTOR
Make Your Publications Visible.

A Service of

ZBW
Leibniz-Informationszentrum Wirtschaft
Leibniz Information Centre for Economics

Brauer, Johannes

Working Paper

Objektmutation: Nützlich aber teuer?

Arbeitspapiere der Nordakademie, No. 2008-04

Provided in Cooperation with:
Nordakademie - Hochschule der Wirtschaft, Elmshorn

Suggested Citation: Brauer, Johannes (2008) : Objektmutation: Nützlich aber teuer?, Arbeitspapiere der Nordakademie, No. 2008-04, Nordakademie - Hochschule der Wirtschaft, Elmshorn

This Version is available at:
http://hdl.handle.net/10419/38586

Standard-Nutzungsbedingungen:

Die Dokumente auf EconStor dürfen zu eigenen wissenschaftlichen Zwecken und zum Privatgebrauch gespeichert und kopiert werden.

Sie dürfen die Dokumente nicht für öffentliche oder kommerzielle Zwecke vervielfältigen, öffentlich ausstellen, öffentlich zugänglich machen, vertreiben oder anderweitig nutzen.

Sofern die Verfasser die Dokumente unter Open-Content-Lizenzen (insbesondere CC-Lizenzen) zur Verfügung gestellt haben sollten, gelten abweichend von diesen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Terms of use:

Documents in EconStor may be saved and copied for your personal and scholarly purposes.

You are not to copy documents for public or commercial purposes, to exhibit the documents publicly, to make them publicly available on the internet, or to distribute or otherwise use the documents in public.

If the documents have been made available under an Open Content Licence (especially Creative Commons Licences), you may exercise further usage rights as specified in the indicated licence.

WWW.ECONSTOR.EU

Mitglied der Leibniz-Gemeinschaft

ARBEITSPAPIERE DER NORDAKADEMIE

ISSN 1860-0360

Nr. 2008-04

Objektmutation – nützlich aber teuer?

Prof. Dr.-Ing. Johannes Brauer

April 2008

Eine elektronische Version dieses Arbeitspapiers ist verfügbar unter:
http://www.nordakademie.de/index.php?id=ap

Köllner Chaussee 11
25337 Elmshorn
http://www.nordakademie.de

Objektmutation – nützlich aber teuer?

Johannes Brauer
18. April 2008

Es werden verschiedene Varianten betrachtet, objektorientierte Implementierungen für Binärbäume vorzunehmen. Dabei soll zum einen beleuchtet werden, wie Polymorphie benutzt werden kann, um Fallunterscheidungen zu vermeiden. Zum anderen soll untersucht werden, ob die Technik der Objektmutation (Wechsel der Klassenzugehörigkeit eines Objektes) einen Vorteil bringen kann. Die Technik ist aber nicht in allen objektorientierten Programmiersprachen einsetzbar. Dies stellt eine willkürliche Beschränkung der Ausdrucksmöglichkeiten des Programmierers dar.

1 Einführung

In der Zweckbeschreibung des Zustandsmusters[1] wird sinngemäß formuliert, dass einem Objekt eine Verhaltensänderung ermöglicht wird, wenn sein interner Zustand sich ändert. Das Objekt *scheint* dabei seine Klassenzugehörigkeit geändert zu haben. Diese Formulierung könnte so interpretiert werden, dass es sich beim Zustandsmuster um eine „Umgehungslösung“ für den Fall handelt, dass die verwendete Programmiersprache einem Objekt den Wechsel der Klassenzugehörigkeit nicht erlaubt. Man kann auch für andere Entwurfsmuster argumentieren, dass sie Defizite von objektorientierten Programmiersprachen beschreiben. Als Beispiel sei hier nur das Singleton-Muster[2] genannt, mit dem man umständlich gegen das Grundkonzept klassenbasierter Sprachen „anprogrammieren“ muss, dass von einer Klasse beliebig viele Exemplare erzeugt werden können. Damit sollen nun nicht pauschal alle Entwurfsmuster als überflüssig bezeichnet werden, wenn denn nur die Programmiersprachen über entsprechende Ausdrucksmittel verfügten. In den folgenden Betrachtungen soll auch nur der Nutzen des Wechsel der Klassenzugehörigkeit eines Objektes betrachtet werden. Steht diese Möglichkeit zur Verfügung, lassen sich manche Probleme elegant ohne die Verwendung von Entwurfsmustern lösen oder die Implementierung von Entwurfsmustern kann stark vereinfacht werden. Dies gilt z. B. für das Proxy-Muster[3].

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Der Wechsel der Klassenzugehörigkeit eines Objektes wird in der Literatur vielfach mit dem Begriff *Objektmutation* bezeichnet. Es sind aber auch andere Bezeichnungen, wie z. B. *Objektmigration*, *Re-Klassifikation* (engl. re-classification) oder dynamische Klassifikation (engl. dynamic classification) anzutreffen. Objektmutation wird nur von wenigen objektorientierten Programmiersprachen direkt unterstützt. Es handelt sich dabei meist um dynamische Sprachen mit ausgeprägten Möglichkeiten der Metaprogrammierung, wie z. B. Smalltalk. Es gibt aber auch Bemühungen, „populäre" Sprachen wie Java um entsprechende Fähigkeiten zu erweitern[4]. Aber auch die Arbeiten die Programmiersprachen *Fickle*[5,6] und *Gilgul*[7] betreffend gehen in diese Richtung. Diese Anstrengungen sind zumindest ein Anzeichen dafür, dass es einen Bedarf für derartige Ausdrucksmöglichkeiten in objektorientierten Programmiersprachen gibt.

[4]Li, L. Extending the java language with dynamic classification. *Journal of Object Technology 3*, 7 (Juli-August 2004), 101–120.

[5]Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. More dynamic object reclassification: Fickle∥. *ACM Trans. Program. Lang. Syst. 24*, 2 (2002), 153–191.

[6]Damiani, F., Drossopoulou, S., and Giannini, P. Refined effects for unanticipated object re-classification: Fickle3 (extended abstract). In *ICTCS'03* (2003), LNCS 2841, Springer, pp. 97–110.

[7]Costanza, P. Dynamic replacement of active objects in the gilgul programming language. In *Component Deployment* (2002), J. M. Bishop, Ed., vol. 2370 of *Lecture Notes in Computer Science*, Springer, pp. 125–140.

Die Anwendungsmöglichkeiten sind vielfältig. Hier seien einige Beispiele für Objekte genannt, die während ihrer Lebensdauer ihr Verhalten durch Wechsel der Klassenzugehörigkeit ändern könnten:

- Aus einem Angebot wird, wenn der Kunde es angenommen hat, ein Vertrag.
- Aus einer Auszubildenden oder einer Werkstudentin wird eine Mitarbeiterin.
- Aus einem Frosch wird durch Küssen ein Prinz.

Im Folgenden wird für eine objektorientierte Implementierung von binären Suchbäumen dargestellt, wie hier Objektmutation sinnvoll eingesetzt werden könnte.

2 *Binäre Suchbäume*

Für die folgenden Betrachtungen benutzen wir Binärbäume, deren Einträge[8] einer linearen Ordnung folgen und die in jedem Knoten zu vermerken erlauben, wie oft ein Wert in dem Baum vorkommt. Ein Knoten des Baums hat dann den in Abbildung 1 gezeigten Aufbau.

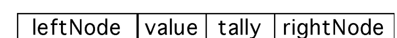

Abbildung 1: Aufbau eines Knotens

[8]Der Einfachheit der Darstellung halber werden in dem Baum Zahlen aufsteigend geordnet gespeichert.

Abbildung 2 zeigt einen kleinen Baum bestehend aus Knoten dieses Formats.

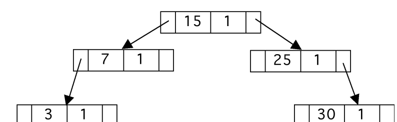

Abbildung 2: Ein Binärbaum

Für eine „naive" Implementierung dieser Datenstruktur in einer klassenbasierten, objektorientierten Programmiersprache könnte für die Knoten eine Klasse[9] mit vier Exemplarvariablen (`left`, `value`, `tally` und `right`) angelegt werden. Falls ein Knoten keinen linken oder rechten Nachfolger besitzt, wird in die entsprechende Exemplarvariable

[9]nennen wir sie `Node`

„nichts“ eingetragen. Dieses „Nichts“ wird dann je nach Programmiersprache durch „Etwas“ repräsentiert.[10]

[10] z. B. in Smalltalk durch das undefinierte Objekt `nil` oder in Java durch `null`

Unter dieser Voraussetzung sähe dann der „klassische“ Algorithmus für das Einfügen eines neuen Wertes in den Baum so aus:

```
algorithm insert(node, newval)11

if node=nil then
   return new Node(null, newval, 1 null)
elsif node.value = newval then
   return node.withTallyIncremented
elsif node.value < newval then
   return node.withleft(insert(left, newval))
else node.value = newval then
   return node.withright(insert(right, newval))
endif
```

[11] die Formulierung des Algorithmus erfolgt hier in einem hybriden funktional-objektorientiertem Pseudocode

Dieser Algorithmus ist sozusagen „kongruent“ zur rekursiven Struktur von Binärbäumen und damit einfach nachvollziehbar und übersichtlich. Als weitere Vorteile dieser Lösung könnte man ansehen:

- Es ist nur eine Klasse für Baumknoten notwendig und damit sind auch alle Methoden für die Manipulation von Bäumen dort konzentriert.
- Solange nur Einträge hinzugefügt werden, müssen ggf. neue Knoten angelegt werden. Die schon vorhandene Struktur des Baums muss nicht verändert werden. Auf diesen Punkt werden wir noch im Abschnitt 3 zurückkommen.

Aus objektorientierter Sicht ist die Lösung unter mehreren Aspekten zu kritisieren. Neben der Tatsache, dass komplexe Fallunterscheidungen möglichst vermieden werden sollten, hat die Technik, für alle Knoten des Baums die gleiche Knotenstruktur zu verwenden, noch weitere Nachteile:

Es gilt das „Dogma“ der objektorientierten Programmierung: Fallunterscheidungen sind durch Polymorphie zur ersetzen

- Für die Speicherung von Knoten ohne Nachfolger wird unnötig Speicherplatz für die nicht belegten[12] Verweise verschwendet. Da in einem vollständigen, ausgeglichenem Binärbaum mehr als die Hälfte der Knoten Blätter sind, ist dies nicht zu vernachlässigen.
- Verweise auf nicht vorhandene Nachfolger mit `nil` zu belegen, führt dazu, dass in nahezu allen Methoden, die den Baum bearbeiten, Prüfungen auf `nil` erforderlich

[12] bzw. mit `nil` oder `null` belegten

werden. Das ist – wie bereits erwähnt – nicht im Sinne der objektorientierten Programmierung.

Den zweiten Nachteil kann man leicht dadurch beheben, dass man die `nil`-Verweise durch Einführung eines zweiten Knotentyps[13] vermeidet. Exemplare der Klasse `EmptyNode` als Unterklasse der Klasse `Node` verstehen dann alle Node-Nachrichten, die Fallunterscheidungen mit den Prüfungen auf `nil` entfallen und werden durch Polymorphie ersetzt.

[13]Nennen wir die Klasse `EmptyNode`

Um Speicherplatz zu sparen, ist es ratsam, die Klasse `EmptyNode` als Singleton anzulegen.

Der Baum aus Abbildung 2 zeigt dann den in Abbildung 3 gezeigten Aufbau.

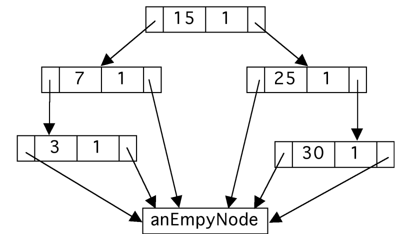

Abbildung 3: Binärbaum mit „Leerknoten“

Eine weiter gehende objektorientierte Analyse des Binärbaum-Problems könnte aber auch zu dem Ergebnis führen, dass es vier Arten von Knoten gibt:

- Knoten mit zwei Nachfolgern
- Knoten mit einem linken Nachfolger
- Knoten mit einem rechten Nachfolger
- Knoten mit keinem Nachfolger

Führt man für diese Knotenarten jeweils eine eigene Klasse ein, kann man auch den ersten o. g. Nachteil beseitigen. Die Klasse `EmptyNode` erübrigt sich dann, unser Baum erhält die in Abbildung 4 gezeigte Gestalt.

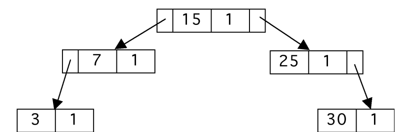

Abbildung 4: Ein Baum mit vier Knotentypen

Diese Struktur weist keine unnötigen Verweise und damit auch keine Verschwendung von Speicherplatz auf. Allerdings müssen die Knoten nun z. B. auf das Einfügen eines neuen Wertes in der jeweils ihrem Knotentyp angepassten Art und Weise reagieren. Soll in unseren Baum beispielsweise der Wert 10 eingefügt werden, muss der Knoten mit dem Wert 7, der nur einen linken Nachfolger besitzt, durch einen mit zwei Nachfolgern ersetzt werden. Das Ergebnis zeigt Abbildung 5.

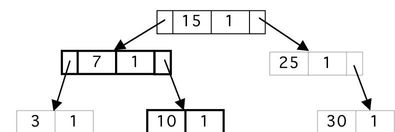

Abbildung 5: Baum nach Einfügen des Wertes 10

Die Abbildung 6 zeigt die dazu gehörige Klassenhierarchie.

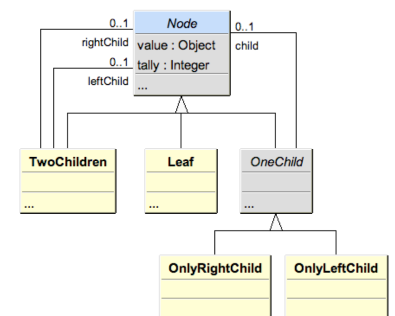

Abbildung 6: Klassenhierarchie für Binärbäume

3 *Mutation von Knoten*

Das Ersetzen eines Knotens durch ein Exemplar einer anderen Klasse, wie es bei der am Ende von Abschnitt 2 geschilderten Einfügeoperation notwendig ist, kann unter der Voraussetzung, dass ein Objekt seine Klassenzugehörigkeit *nicht* ändern kann, nur dadurch geschehen, dass

a) ein Exemplar der Zielklasse erzeugt wird (im Beispiel ein Knoten mit zwei Nachfolgern, d. h. ein Exemplar der Klasse `TwoChildren`),

b) die Werte der Exemplarvariablen (`value` und

`tally`) des zu ersetzenden Knotens in den Zielknoten kopiert werden,
c) der alte Knoten verworfen und
d) im Vorgängerknoten den Verweis auf den neu erzeugten Knoten gesetzt wird.

Insbesondere die beiden letzten Schritte sind notwendig, wenn Objektmutation nicht erlaubt ist.

Die Schritte b) und c) machen noch einmal den zweit genannten Vorteil eines Baumaufbaus aus einheitlichen Knotentypen, wie in Abbildung 3 gezeigt, deutlich: Am Vaterknoten müsste infolge der Einfügeoperation keine Veränderung vorgenommen werden.

Die geschilderte Einfügeoperation vereinfacht sich jedoch erheblich, wenn es möglich ist, dass ein Knoten[14] seine Klassenzugehörigkeit unter Wahrung seiner Objektidentität wechseln[15] kann. Die vier o. g. Schritte schrumpfen zu einem einzigen zusammen:

[14] in unserem Beispiel der mit dem Wert 7

[15] hier von `OnlyLeftChild` nach `TwoChildren`

- Der Knoten mit dem Wert 7 (vgl. Abbildungen 4 bzw. 5) mutiert in einen Knoten mit zwei Nachfolgern.

Der Vorgängerknoten (mit dem Wert 10) bleibt von dieser Einfügeoperation unberührt.

Wie bereits eingangs erwähnt, wird das Zustandsmuster[16] häufig verwendet, um Objektmutation zu umgehen bzw. nachzubilden. Obwohl es hier keineswegs infrage gestellt werden soll, dass es sinnvolle Anwendungen dieses Entwurfsmusters gibt, kann man jedoch sagen, dass es häufig eine Umgehungslösung darstellt, weil die Programmiersprache Objektmutation nicht erlaubt. Für den Binärbaum ist die Anwendung des Zustandsmusters aus Gründen des Aufwands sicher keine vernünftige Lösung.

[16] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

4 *Implementierungsaspekte*

Die Forderung, dass eine objektorientierte Programmiersprache Objektmutation ermöglichen sollte ist leicht erhoben. Eine andere Frage ist, ob das auch effizient implementiert werden kann, insbesondere dann, wenn sich die Speicherrepräsentation des zu mutierenden Objekts ändern muss.

In Smalltalk steht für den Klassenwechsel eines Objekts die Nachricht `changeClassTo:` zur Verfügung. Diese Art des Klassenwechsel darf allerdings nicht mit einer Größenänderung des Objektes einhergehen. Auf unser Beispiel übertragen müssten alle vier Knotentypen alle vier Exemplarvariablen beinhalten. Das machte zwar den Vorteil der Speicherplatzersparnis zunichte. Die übrigen Vorteile blieben jedoch erhalten.

Mit dem Ausdruck `anInstanceOfClassA changeClassTo: ClassB` wird die Klasse des Empfängerobjekts auf `ClassB` geändert.

Die Ausführung der `changeClassTo:`-Operation ist sehr effizient, weil nur der Verweis des Objekts auf die Klasse umzuhängen ist.

Wenn sich die Speicherrepräsentation bei der Objektmutation ändern soll, was das Binärbaumbeispiel ja nahe legt, ergibt sich das Problem, dass das „Mutat" an anderer Stelle im Speicher errichtet werden muss[17]. Da in den meisten objektorientierten Programmiersprachen die Objektidentität an die Speicheradresse gebunden ist, müssten alle Verweise auf das ursprüngliche Objekt auf die neue Speicheradresse des Mutats geändert werden. Costanza[18] beschreibt eine effizientere Implementierungstechnik.

[17] zumindest dann, wenn der Speicherbedarf wächst

[18] COSTANZA, P. Dynamic replacement of active objects in the gilgul programming language. In *Component Deployment* (2002), J. M. Bishop, Ed., vol. 2370 of *Lecture Notes in Computer Science*, Springer, pp. 125–140.

In Smalltalk steht für diesen Zweck die Nachricht `become:` zur Verfügung. Damit kann Objektmutation in einfacher Weise simuliert werden:

a) Zuerst wird ein Exemplar der Zielklasse erzeugt.
b) Dann werden die Exemplarvariablen des „alten" Objekts in das neu erzeugte Exemplar kopiert.
c) Schließlich tauschen die Objekte unter Verwendung von `become:` ihre Identitäten aus.

Im letzten Schritt werden durch die virtuelle Maschine automatisch alle Verweise vom alten auf das neue Objekt geändert. Das macht `become:` zu einer recht teuren Operation. Die tatsächlichen Kosten hängen möglicherweise – je nach Implementierung – davon ab, wie viele Verweise existieren.

Mit dem Ausdruck `anInstanceOfClassA become: anInstanceOfClassB` tauschen die beteiligten Objekte ihre Identität aus.

WENN SICH DER BINÄRBAUM sehr „dynamisch" verändert, kann durch die Verwendung der `become:`-Operation auch noch das Problem der Speicherfragmentierung hinzu kommen. Wenn dieses Problem als das größere im Vergleich zum absoluten Speicherbedarf angesehen werden muss, kann sich die Lösung, alle Knotentypen gleich groß zu machen, als günstig erweisen.

Wie sich die beiden Lösungen[19] in der Praxis auswirken, kann nur durch Messungen herausgefunden werden. Und dabei sind die dynamischen Änderungsmuster von großer Bedeutung. Insbesondere sind hier Fragen wichtig wie:

[19] also Klassenwechsel mit und ohne Größenänderung

- Wie hoch ist die Wahrscheinlichkeit, mit der ein Knoten mutiert?
- Wie hoch ist die Wahrscheinlichkeit, mit der ein Knoten mutiert?
- Wie viel Prozent der Knoten haben zwei Kinder, ein Kind, gar kein Kind?

Theoretisch können

- höchsten 50% zwei Kinder haben (voller ausgeglichener Baum), mindestens 0% (völlig entarteter Baum);

- 100% -1 ein Kind haben (völlig entarteter Baum), mindestens 0% (voller ausgeglichener Baum).
- höchsten 50% null Kinder haben (voller ausgeglichener Baum), mindestens aber 1 (völlig entarteter Baum).

Das heißt, für weitergehende Beurteilungen wären Messungen erforderlich, theoretische Überlegungen allein helfen nicht weiter.

5 *„Ketzerische“ Nachbemerkung*

Der in Abschnitt 2 angegebene Einfügealgorithmus ist sehr gut überschau- und ohne Probleme nachvollziehbar. Dieser Algorithmus wird in einer strikt objektorientierten Implementierung[20] sehr stark „zerfleddert“. Wenn man durch Studium des Programms versuchte, den Einfügealgorithmus zu verstehen, müsste man, wenn die in Abbildung 6 dargestellte Klassenhierarchie zugrunde gelegt wird, fünf Methoden in fünf Klassen anschauen. Das gilt natürlich für alle anderen Baum-Algorithmen ebenso. Von Lesbarkeit des Programms kann dann eigentlich keine Rede mehr sein. Die Anwendung des objektorientierten Paradigmas scheint hier nicht unbedingt angebracht. Bezogen auf unser Beispiel könnte man einwenden, dass man die Implementierung von Binärbäumen – einmal als elementare Datenstruktur in einer Klassenbibliothek angelegt – kaum je wieder betrachten wird. Aber trotzdem scheint es fraglich, ob es sinnvoll ist, dass man mit der Wahl einer Programmiersprache von vorne herein auf ein bestimmtes Paradigma festgelegt wird. Es ist durchaus nicht abwegig, auch bei der Entwicklung einer einzigen Anwendung verschiedene Programmierstile einsetzen zu wollen. Neuere Entwicklungen auch bei „populären“ objektorientierten Sprachen deuten in diese Richtung.

[20] in dem Sinne, dass Fallunterscheidungen so weit wie möglich durch Polymorphie zu ersetzen sind

Dass Objektmutation von einer Programmiersprache ausgeschlossen wird, ist eine willkürliche Beschränkung der Ausdrucksmöglichkeiten des Programmierers, die außer mit der „Bequemlichkeit“ der Implementierung kaum sachlich begründet werden kann.

Literatur

[1] Costanza, P. Dynamic replacement of active objects in the gilgul programming language. In *Component Deployment* (2002), J. M. Bishop, Ed., vol. 2370 of *Lecture Notes in Computer Science*, Springer, pp. 125–140.

[2] Damiani, F., Drossopoulou, S., and Giannini, P. Refined effects for unanticipated object re-classification: Fickle3 (extended abstract). In *ICTCS'03* (2003), LNCS 2841, Springer, pp. 97–110.

[3] Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. More dynamic object reclassification: Fickle∥. *ACM Trans. Program. Lang. Syst. 24*, 2 (2002), 153–191.

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] Li, L. Extending the java language with dynamic classification. *Journal of Object Technology 3*, 7 (Juli-August 2004), 101–120.