Bidirektionale Assoziationen in Java

Markus Völter, voelter@acm.org, www.voelter.de

Bidirektionale Assozationen sind jedem Programmierer bekannt da sie in praktisch allen Softwareentwicklungsprojekten vorkommen. Da sie von den gängigen Programmiersprache, so auch Java, nicht unterstütz werden, werden sie in der Regel mit Hilfe von zwei einzelnen, unidirektionalen Assoziationen nachgebildet. Dies hat jedoch verschiedene Nachteile. Die in diesem Artikel vorgestellten Hilfsklassen können da Abhilfe schaffen.

Die Anwendung

In praktisch allen Anwendungen kommen bidirektionale Assoziationen vor. Eine Assoziation ist dann bidirektional, wenn sich die beteiligten Partner *gegenseitig* kennen. Sie können in verschiedenen Kardinalitäten vorliegen.

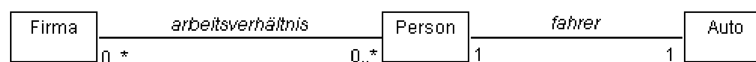

Abbildung 1: Typische Anwendung für bidirektionale Assoziationen

In Abbildung 1 sind zwei bidirektionalen Assoziationen unterschiedlicher Kardinalität abgebildet. Die Assoziation *fahrer* ist eine 1:1-Assoziation zwischen einer *Person* und einem *Auto*. Wie in der realen Welt kann ein *Auto* nur einen *Fahrer* (die *Person*) haben. Andererseits kann eine *Person* bei keiner, einer oder mehreren *Firmen* ein *arbeitsverhältnis* haben, und eine *Firma* kann mehrere Angestellte (*Personen*) haben. Man nennt dies eine n:m-Assoziation.

Das Wesentliche an bidirektionalen Assozationen ist, daß sowohl das *Auto* seinen *Fahrer*, also eine *Person,* kennt, und daß die entsprechende *Person* das *Auto* kennt mit dem sie fährt. Dies gilt sinngemäß auch für die n:m-Assoziation.

Der übliche Lösungsansatz und seine Probleme

Nachdem gängige Programmiersprachen bidirektionale Assoziationen nicht direkt unterstützen, basiert der übliche Lösungsansatz auf der Nachbildung einer bidirektionalen Assoziation durch zwei unidirektionale (Abblidung 2).

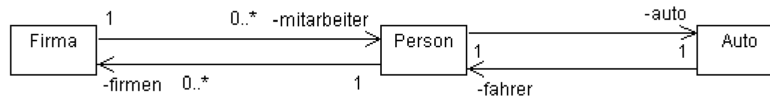

Abbildung 2: Emulation von bidirektionalen Assoziationen durch zwei unidirektionale

Implementiert wird dies in aller Regel durch entsprechende Attribute, wie aus Abbildung 3 ersichtlich.

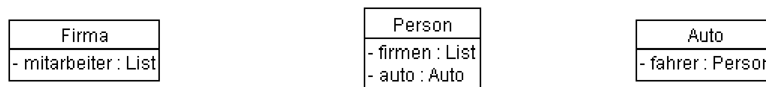

Abbildung 3: Implementation der Assoziationen durch entsprechende Attribute

Der Nachteil liegt auf der Hand: Die beiden unidirektionalen Assoziationen mit Hilfe derer die bidirektionale „emuliert" wird, müssen von Hand synchronisiert werden. Wenn also beispielsweise bei einem Auto der *fahrer* ausgetragen wird, so muss auch bei der entsprechenden Person das *auto* ausgetragen werden.

Dieses Vorgehen, also die Synchronisation der beiden Assoziationen von Hand, ist sicherlich eine legitime und praktikable Vorgehensweise. Allerdings ist sie auch sehr fehleranfällig.

Alternative Lösungsansätze

In seiner Mustersprache *Basic Relationship Patterns* diskutiert James Noble verschiedene Möglichkeiten, das Problem zu lösen [3]. Er geht dabei nicht nur auf bidirektionale Assoziationen ein, sondern auch auf komplexere Beziehungen, die mehr als zwei Partner haben. Für diesen Fall schlägt er beispielsweise vor, eine eigens dafür entworfene Klasse zu verwenden. Für den Fall der bidirektionalen Assoziation enthält das Papier ein Muster namens *Mutual Friends*. Dabei wird eine 1:n Assoziation dadurch realisiert, dass die jeweiligen *set()*/*add()*-Methoden auch die Attribute auf der Gegenseite richtig setzen. Die *add()* –Operation auf der *n*-Seite sieht also beispielsweise folgendermassen aus:

```
class Client {
   List policies;
   public void addPolicy( Policy aPolicy ) {
      policies.add( aPolicy );
      aPolicy.setClient( this );
   }
   // ...
}
```

Eine sehr detaillierte Diskussion dieses Ansatzes findet sich in Kevlin Henney's *Mutual Registration Pattern,* welches auf der EuroPLOP '99 vorgestellt wurde [2].

Eine interessante Alternative bietet der ODMG-Standard für OO-Datenbanken [1]. Dort werden in der ODL-Datendefinitionssprache bidirektionale Assoziationen definiert, und der Code um diese zu synchronisieren kann daraus automatisch generiert werden.

Wenn man nun in einer Anwendung viele bidirektionale Assoziationen verwendet, so wäre es sicherlich sinnvoll, die gegenseitige Synchronisation zu automatisieren. Im Folgenden soll eine Klasse vorgestellt werden, die diese Aufgaben (fast) automatisiert.

Lösung mittels Assoziationsobjekten

Allgemein kann eine bidirektionale 1:1-Assoziation folgendermassen dargestellt werden:

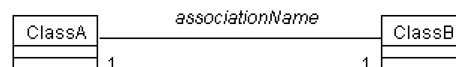

Abbildung 4: Allgemeines Beispiel für bidirektionale Assoziationen

Um die Assoziation konsistent zu halten, sind unter anderem die folgenden Aktionen nötig:

- Wenn einem ClassA-Objekt ein ClassB-Objekt zugeordnet wird, so muss auch diesem ClassB-Objekt das entsprechende ClassA-Objekt zugeordnet werden. Gleiches gilt auch in der anderen Richtung.
- Wird die Verknüpfung zu einem ClassB-Objekt auf Seite des ClassA-Objektes gelöscht, so muss auch der Verweis auf dieses ClassA-Objekt im Objekt der Klasse ClassB gelöscht werden.
- Wenn einem ClassB-Objekt ein ClassA-Objekt zugewiesen wird, während es bereits mit einem ClassA-Objekt verbunden ist, so muss bei dem alten ClassA-Objekt die Assoziation zum ClassB-Objekt gelöscht werden.

Um die Verwaltung der Referenzen zu automatisieren, kann man z.B. folgendermassen vorgehen. Statt der direkten Referenz auf den jeweiligen Partner verweist jede der beteiligten Klassen auf ein Objekt vom Typ *BidirectionalAssociation* (in Abbildung 5 abgekürzt mit *BDA*). Diese Assoziation wird niemals geändert oder gelöscht. Sie existiert während des gesamten Lebens des entsprechenden ClassA- oder ClassB-Objekts. Das *BDA*-Objekt stellt Operationen zur Verfügung, um sich mit anderen Objekten zu verbinden. Diese sorgen für die automatische Synchronisierung.

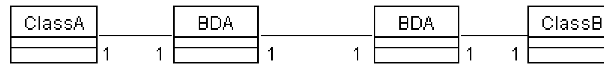

Abbildung 5: Implementierung einer bidirektionalen Assoziation mittels Zwischenobjekten, die die Synchronisation übernehmen.

Die logische Assoziation zwischen ClassA- und ClassB-Objekten wird damit in Wirklichkeit mittels der beiden BDA-Objekte realisert. Um dieses BDA-Objekt vom ClassA- oder ClassB-Objekt zu erhalten, müssen die Klassen eine entsprechende Operation bereitstellen. Diese kann z.B. so heissen, wie die Assoziation selbst. Wenn alle diese Voraussetzungen geschaffen sind, stellt sich die Situation für den Anwendungsprogrammierer folgendermassen dar:

```
  // ClassA- und ClassB-Objekte erzeugen
ClassA a = new ClassA( "a" );
ClassB b = new ClassB( "b" );
  // angenommen, die Assoziation heisst abAssoc,
  // dann kann mit der folgenden Zeile das Objekt a
  // mit dem Objekt b assoziiert werden.
a.abAssoc().connectTo( b );
  // da die Assoziation automatisch wechselseitig
  // erzeugt wird, kann nun z.B. b nach seinem
  // assoziierten Objekt befragt werden:
ClassA a2 = (ClassA)b.abAssoc().otherObject();
  // dies muss ja nun wieder a sein
if ( a != a2 ) throw ( new Exception() );
```

Implementierung

Laut Namenskonvention liefert die Methode *abAssoc()* der Klassen *ClassA* und *ClassB* das jeweilige Assoziationsobjekt zurück. Damit die hier vorgestellte Lösung funktioniert, muss die Operation auf beiden Seiten (also in beiden Klassen) denselben Namen haben. Diese Operation wird folgendermassen implementiert:

```
public class ClassA {
   private BidirectionalAssociation abAssoc;
   public BidirectionalAssociation abAssoc() {
      if ( abAssoc == null ) {
        abAssoc =
           new BidirectionalAssociation( this, "abAssoc"   );
      }
      return abAssoc;
   }
   // ...
}
```

Würde die Klasse *ClassA* noch weitere Assoziationen mit anderen Objekten besitzen, so müssten weitere Operationen hinzugefügt werden:

```
public class ClassA {
   private BidirectionalAssociation eineAndereAssoc;
   public BidirectionalAssociation eineAndereAssoc() {
      if (eineAndereAssoc == null ) {
         eineAndereAssoc = new
            BidirectionalAssociation( this, "eineAndereAssoc" );
     }
      return eineAndereAssoc;
   }
  // ...
}
```

Die eigentliche Arbeit erledigt hier die Klasse *BidirectionalAssociation*. Daher soll diese nun etwas näher beleuchtet werden.

Prinzipiell ist die Verknüpfung zwischen den beiden Assoziationsobjekten relativ einfach, es wird eben mit *java.lang.Object*s gearbeitet, um die Assoziationen generisch zu halten. Das einzige Problem bei der vorgestellten Lösung ist, daß das eine Assoziationsobjekt irgendwie eine Referenz auf das andere bekommen muss, mit dem es sich verbinden soll. Zu diesem Zweck gibt es ja laut Namenskonvention in jeder beteiligten Klasse eine Methode, die das Assoziationsobjekt zurückliefert. Diese Methode heisst genau so wie die Assoziation selbst. Da aber nicht extra ein Interface deklariert werden soll, nur um die Assoziation zu implementieren, kann diese Methode nicht regulär aufgerufen werden – schliesslich ist sie im Typ des anderen Objektes nicht definiert (ist ja ein *java.lang.Object*). Aus diesem Grunde wird Java's Reflection-Mechanismus eingesetzt, um die Methode dynamisch aufzurufen.

Nun also zu den Details der Implementierung:

```
public class BidirectionalAssociation {
```

Die Verknüpfung findet zwischen Assoziationsobjekten statt, also muss das eine Assoziationsobjekt das andere kennen:

```
    private BidirectionalAssociation other = null;
```

Desweiteren muss sich ein Assoziationsobjekt natürlich das lokale Objekt merken, mit dem es verbunden ist:

```
    private Object object;
```

Die Methode, die bei einen anderen Objekt aufgerufen werden muss um das zugehörige Assoziationsobjekt zurückzuliefern, heisst laut Namenskonvention genauso wie die Assoziation selbst – daher muss sie das Assoziationsobjekt den Namen der Assoziation merken, zu welcher es gehört:

```
    private String assocName;
```

Der Konstruktor eines Assoziationsobjektes bekommt als Parameter das lokale Objekt sowie den Namen der Assoziation:

```
    public BidirectionalAssociation( Object lo, String an ) {
```

```
        object = lo;
        assocName = an;
    }
```

otherObject() liefert das „andere" Objekt zurück, mit dem die Assoziation verbunden ist – indem das verbundene Assoziationsobjekt nach seinem lokalen Objekt befragt wird:

```
    public Object otherObject() {
        if ( other == null ) return null;
        return other.object;
    }
```

isConnected() liefert zurück, ob die Assoziation überhaupt mit einem anderen Objekt verbunden ist:

```
    public boolean isConnected() {
        return other != null;
    }
```

connectTo() dient nun dazu, das Assoziationsobjekt mit einem anderen zu verbinden:

```
    public void connectTo( Object otherObject ) {
        try {
                // wenn Objekt schon verbunden, dann die
                // aktuelle Verbindung lösen
                // release() löst Verbindung auf beiden Seiten!

            if ( isConnected() ) release();
                // anderes Assoziationsobjekt holen
            Method m = otherObject.getClass().
                      getMethod( assocName, null );
            other = (BidirectionalAssociation)
                      m.invoke( otherObject, null );
                // wenn anderes Objekt noch nicht zum lokalen
                // verbunden ist, verbinden
                // Abfrage ist wichtig, um endlose gegenseitige
                // Aufrufe zu vermeiden!
            if ( this != other.other ) {
                    // Assoziation des Anderen lösen
                other.release();
                    // zu sich verbinden
                other.connectTo( this.object );
            }
        } catch ( Exception ex ) {
            //       sinnvolles Fehlerhandling muesste dann
              // noch eingebaut werden :-)
            ex.printStackTrace();
        }
    }
```

release() trennt die Verbindung mit einem anderen Objekt. Beim anderen Objekt muss dann natürlich auch die Verbindung mit dem lokalen Objekt gelöst werden.

```
        public void release() throws AssociationException {
            boolean warn = false;
            if ( warningsEnabled && !isConnected() ) warn = true;
            if ( other != null ) {
                BidirectionalAssociation to = other;
                other = null;
                if ( to.other == this ) {
                    to.release();
                }
            }
        }

    }
```

Mit Hilfe dieser Klasse kann man nun sehr einfach bidirektionale 1:1-Assoziationen zu realisieren. Der Nachteil ist, dass zur Laufzeit zwei zusätzliche Objekte benötigt werden, und auch die Performance aufgrund der zusätzlichen Indirektion etwas schlechter wird – jeder Entwickler muss für sich selbst entscheiden, ob diese Nachteile tragbar sind.

Mittels dieses Vorgehens lassen sich prinzipiell auch 1:n- oder n:m-Assoziationen realisieren, dazu existiert dann die Klasse *BidirectionalNMAssociation*. Dies soll im folgenden Beispiel anhand des *Observer*-Patterns nochmals abschliessend gezeigt werden.

Beispiel für n:m Assoziation

Das *Observer*-Muster aus dem Buch der Gang of Four [4] ordnet einem *Subject* n *Observer*-Objekte zu, die vom Subjekt benachrichtigt werden, wenn dieses seinen Zustand ändert. Eine Erweiterung des Observer-Musters könnte dahingehend stattfinden, dass ein Observer nicht nur *ein* Subjekt beobachtet, sondern mehrere. Dies laesst sich mit dem Folgenden Klassendiagramm darstellen:

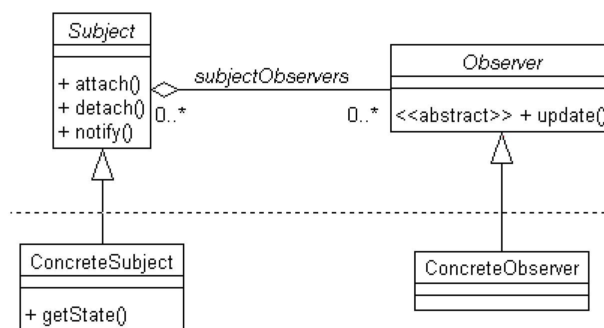

Abbildung 6: Klassendiagramm des Observer-Patterns mit mehr als einem Subjekt pro Observer

Die aus dem Diagram ersichtliche n:m-Assoziation zwischen *Subject*s und *Observer*n kann mit entsprechenden Assoziationsklassen realisiert werden.

Zunächst die Klasse *Subject*:

```
package de.voelter.utilities.assoc.test;

import de.voelter.utilities.assoc.*;
import java.util.*;

public abstract class Subject {

    BidirectionalNMAssociation subjectObservers;

    public BidirectionalNMAssociation subjectObservers() {
        if ( subjectObservers == null )
          subjectObservers = new
            BidirectionalNMAssociation( this,
                     "subjectObservers" );
        return subjectObservers;
    }

    public void notifyObservers() {
        Iterator it = subjectObservers().otherObjects();
        while ( it.hasNext() ) {
            Observer o = (Observer)it.next();
            o.update();
        }
    }

}
```

Die Klasse *Observer* sieht im Prinzip ganz ähnlich aus:

```
package de.voelter.utilities.assoc.test;

import de.voelter.utilities.assoc.*;

public abstract class Observer {

    BidirectionalNMAssociation subjectObservers;

    public BidirectionalNMAssociation subjectObservers() {
        if ( subjectObservers == null )
          subjectObservers = new
            BidirectionalNMAssociation( this,
                     "subjectObservers" );
        return subjectObservers;
    }

    public abstract void update();
}
```

Durch diese Vorgehensweise kennen nun nicht nur die Subjekte ihre Observer, sondern auch die Observer ihr beobachtetes Subjekt – und dies immer konsistent. Der einzige Schönheitsfehler an dieser Vorgehensweise ist, daß die Methoden nun nicht mehr *attach()* oder *detach()* heissen, sondern statt dessen auf dem Assoziationsobjekt *connectTo()* bzw. *release()* aufgerufen werden muss. Selbstverständlich kann dies durch entsprechende Wrapper-Methoden in der Klasse Subject und Observer umgangen werden. Beispiel:

```
  // Erzeugen des Subjektes und der Observer;
  // MySubject und MyObserver sind jeweils konkrete Unterklassen
  // von Subject bzw. Observer.
MySubject s   = new MySubject();
MyObserver o  = new MyObserver();
MyObserver o2 = new MyObserver();
  // verbinden eines Observers mit einem Subjekt
s.subjectObservers().connectTo(o);
s.subjectObservers().connectTo(o2);
  // Trennen einer Verbindung
s.subjectObservers().release(o);
```

Der Quellcode für die Assoziationsklassen und die Beispiele ist im Internet unter www.voelter.de zu finden.

Referenzen

[1] ODMG Database Standard, www.odmg.org

[2] Kevlin Henney, *Mutual Registration*, Proceedings of the EuroPLOP '99 Conference, zu finden unter http://www.argo.be/europlop

[3] James Noble, *Basic Relationship Patterns*, in Pattern Languages of Program Design 4, Addison-Wesley 2000

[4] Gamma, Helm, Johnson, Vlissides, *Entwurfsmuster*, Addison-Wesley 1995