Flexible Erzeugung von ValueObjects in Entity Java Beans

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

Einleitung

Ein Problem bei der Verwendung von entfernten liegenden Komponenten (wie EJBs) ist, dass ein Methodenaufruf recht lange dauert, weil eine Netzwerkverbindung beteiligt ist. Dieses Problem kommt besonders dann deutlich zum tragen, wenn man Entity Beans verwendet, die dem typischen JavaBeans-Muster mit *getProperty() / setProperty()* Methoden folgen. Eine typische EntityBean dieser Art wäre z.B. die folgende:

```
package de.mathema.valueobjectfactory.example;

import java.rmi.*;
import javax.ejb.*;

public interface Person extends EJBObject {
    // liefert die ID der Person
  public String getID() throws RemoteException;
    // liefert den Namen
  public String getName() throws RemoteException;
    // liefert den Vornamen
  public String getFirstName() throws RemoteException;
    // liefert den Middle Initial
  public String getMiddleInitial() throws RemoteException;
    // liefert das Alter
  public int getAge() throws RemoteException;
    // liefert die Adresse
  public String getAdress() throws RemoteException;
}
```

Entsprechende setXXX-Methoden wären ergänzend dazu üblicherweise auch vorhanden. Wenn ein Client mit der Bean arbeitet, so ist der naheliegendste Ansatz der folgende:

```
  // Referenz auf Person-Bean besorgen
Person p = ...
  // Daten lesen
String name = p.getName();
String firstName = p.getFirstName();
  // irgendwas mit den Daten tun…
  // und später vielleicht wieder updaten
p.setName( name );
p.setFirstName( firstName );
```

In diesem Stückchen Code sind bereits vier Remote-Aufrufe enthalten. Man stelle sich vor, wie die Performanz in den Keller geht, wenn man nicht nur zwei, sondern vielleicht zehn Attribute auf diese Art und Weise liest und ändert...

Offensichtlich ist dies also keine gute Möglichkeit, performante, verteilte Anwendungen zu entwickeln. Um dem Problem der „vielen kleinen Remoteaufrufe“ zu begegnen, gibt es nun im Wesentlichen zwei Möglichkeiten:

- **Verlagerung der Prozesse auf den Server.** Wenn die Aufgaben des Clients ohne weitere Interaktionen mit dem Benutzer (also als ein abgeschlossener Prozess) erledigt werden können, bietet es sich an, diese Verarbeitungsschritte in Session Beans zu kapseln. Der Client ruft auf dieser Session Bean eine Methode auf, der er per Parameter alle Informationen mitgibt, die diese zur Ausführung des Verarbeitungsschrittes benötigt. Dies hat auch den angenehmen Nebeneffekt, dass die Session Bean eine Transaktionsklammer setzen kann.
- **Verwendung von ValueObjects:** Wenn es wirklich nötig ist, dass der Client die Daten „in Händen hält“, verändert und wieder in die Bean zurückschreibt (z.B. weil dazwischen der Benutzer per Editor die Werte ändern kann), kann man ValueObjects verwenden um mit einem Methodenaufruf mehrere Attribute zu übertragen.

Ich möchte im Folgenden auf den zweiten Fall eingehen. Zunächst eine Erklärung des Begriffs *ValueObject*. Dies sind kleine Datenobjekte, die „by value“ zum Client übertragen werden. Sie müssen daher serialisierbar sein. Sie werden auf dem Server von der Bean mit Daten gefüllt, und dann zum Client kopiert. ValueObjects sind oft auch dafür zuständig, die Korrektheit von zu setzenden Daten bereits auf dem Client zu überprüfen, und nur im Falle einer erfolgreichen Gültigkeitsprüfung Kontakt mit dem Server aufzunehmen. Die Anzahl „unnötiger“ Remoteaufrufe kann dadurch weiter reduziert werden.

Bei der Verwendung von ValueObjects ergeben sich aber einige Probleme. Im Falle des Beispiels (die Person-Bean) ist es sicherlich kein Problem, alle Attribute der Bean in ein ValueObject zu packen und dem Client zurückzusenden. Ist der Zustand allerdings umfangreicher, ist es nicht besonders sinnvoll, immer den gesamten Zustand zu übertragen, wenn der Client nur einen bestimmten Teil benötigt.

Daher verwendet man zu diesem Zweck üblicherweise mehrere ValueObjects pro Entity Bean. In unserem Beispiel wäre das möglicherweise eine Klasse *NamesVO* und eine Klasse *AgeVO*:

```
package de.mathema.valueobjectfactory.example;
import java.rmi.*;
public class NamesVO implements Serializable {
 private String name;
 private String firstName;
 private String middleInitial;
 // getter und setter für die Attribute...
}

package de.mathema.valueobjectfactory.example;
```

```
import java.rmi.*;
public class AgeVO implements Serializable {
 private int age;
 // getter und setter für die Attribute...
}
```

Damit der Client eine Instanz einer solchen Klassen von der Bean bekommen kann, muss diese eine entsprechende Fabrikmethode besitzen:

```
public interface Person extends EJBObject {
    // wie bisher…

    // Fabrikmethode für die ValueObjects
  public NamesVO getNamesVO() throws RemoteException;
  public AgeVO getAgeVO() throws RemoteException;
}
```

Das heißt, für jeden ValueObject Typ muss

- eine entsprechende Klasse erzeugt werden
- und eine passende Fabrikmethode zum Interface der Bean hinzugefügt werden.

Dies sieht auf den ersten Blick nicht besonders bedenklich aus. Dieses Vorgehen hat allerdings einen gravierenden Haken: Üblicherweise bestimmt nicht die Bean selbst welche Art von ValueObjects die Client benötigen, da dies von den Use-Cases der entsprechenden Client-Anwendung abhängt. Als Folge dessen ist also die Schnittstelle der Bean zum Teil von den Anforderungen der Clients bestimmt. Wenn man nun davon ausgeht, dass eine Bean in verschiedenen Anwendungen wiederverwendet wird, so muss die Schnittstelle der Bean regelmäßig um neue Fabrikmethoden für ValueObjects für neue Clients ergänzt werden. Dies hat jedes Mal ein Redeployment zur Folge.

Dieser Artikel stellt eine Möglichkeit vor, wie dieser Nachteil umgangen werden kann, ohne die Vorteile von ValueObjects zu verspielen.

Das Prinzip

Eine ideale Lösung für das oben geschilderte Problem sollte die folgenden Charakteristika aufweisen:

- Es sollten beliebige neue ValueObject-Typen definierbar sein, ohne dass die Bean selbst verändert (oder neu deployt) werden muss
- Es sollten keine Performanznachteile gegenüber der traditionellen Lösung entstehen.
- Die Validierung soll weiterhin auf dem Client möglich sein.
- Alle Operationen sollten typsicher sein.

Wir werden am Ende des Artikels überprüfen, inwiefern die vorgeschlagene Lösung den Anforderungen gerecht wird. Die vorgeschlagene Lösung basiert auf dem folgenden Vorgehen:

- Die Entity Bean bekommt eine Operation *createVO(String id)* die mittels dynamischen Klassenladens eine Instanz der durch *id* identifizierten ValueObject Klasse erzeugt. Dabei verwendet die Bean ihr Environment dazu, die zur ID gehörende Klasse herauszufinden.
- Da alle derartigen Klassen von *ValueObject* erben, kann die Bean auf der erzeugten Instanz die Operation *fill(this)* aufrufen (die Operation *fill(Object source)* ist in ValueObject abstrakt deklariert).
- Die Operation *fill()* ruft die entsprechenden *getXXX()* Operationen auf der Bean auf und setzt die eigenen Attribute.
- Nachdem *fill()* beendet ist, wird das entsprechende ValueObject als Rückgabewert von *createVO()* an den Client zurückgegeben.

Basis des Mechanismus ist die abstrakte Klasse *ValueObject*:

```
package de.mathema.valueobjectfactory;
public abstract class ValueObject
       implements java.io.Serializable {
 public abstract void fill( Object source )
        throws VOFillingError;
}
```

Diese Klasse stellt die abstrakte Basis für alle ValueObject Klassen dar. Sie implementiert *Serializable*, weil sie mit Wert-Semantik zum Client übertragen werden soll. Die Operation *fill()* dient dazu, das neu instanziierte ValueObject mit Daten von der Quell-Bean zu füllen. Jede ValueObject Klasse implementiert diese Methode so wie sie es benötigt.

Zweiter wichtiger Punkt bei der vorgeschlagenen Lösung stellt das Interface *ValueObjectCreator* dar. Jede Bean, die den vorgeschlagenen Mechanismus verwenden können soll (also ValueObjects erzeugen) muss dieses Interface implementieren. Es bietet eine einheitliche Schnittstelle für den Client, um ValueObjects zu erzeugen.

```
package de.mathema.valueobjectfactory;
import java.rmi.*;
public interface ValueObjectCreator {
 public     ValueObject     createVO(     String     voID     )
          throws VOClassNotFound,
          VOFillingError, RemoteException;
}
```

Um ein ValueObject zu erhalten, ruft der Client die Operation *createVO()* auf und liefert als Parameter die ID des zu erzeugenden ValueObjects.

Intern schaut diese Operation in den Environment-Properties nach, wie die konkrete Klasse für dieses ValueObject-ID heißt, instanziiert diese dynamisch, und ruft darauf die Operation *fill()* auf, sich selbst als Parameter übergebend. Dann liefert die Operation das soeben erzeugte und gefüllte ValueObject zurück.

Die Operation *fill()* der konkreten ValueObject-Klasse ruft die get-Operationen auf der übergebenen EntityBean auf. Es sei explizit darauf hingewiesen, dass der Parameter in *fill()* nicht die Remote-Referenz auf die Bean ist, sondern direkt die Instanz der Beanklasse. Damit ist es nicht nötig, die setter/getter im Remote-Interface zu deklarieren, damit sind sie vor den (schlecht performanten) Aufrufen des Clients versteckt.

Beispiel fortgesetzt

Wir wollen nun die Person-Bean so ergänzen, dass sie ValueObjects nach dem oben beschriebenen Schema erzeugen kann. Dazu muss das Remote-Interface erweitert werden, es muss *ValueObjectCreator* implementieren.

```
package de.mathema.valueobjectfactory.example;
import java.rmi.*;
import javax.ejb.*;
import de.mathema.valueobjectfactory.*;
public interface Person extends EJBObject, ValueObjectCreator {
 // wie bisher
}
```

Die Operation muss dann in der Klasse *PersonBean* natürlich auch implementiert werden. Da der Ablauf praktisch immer derselbe ist, lässt er sich in eine Hilfsklasse herausfaktorisieren.

```
package de.mathema.valueobjectfactory.example;
// imports …
public class PersonBean implements EntityBean {
  // verschiedenes…

  public ValueObject createVO( String voID )
                  throws VOClassNotFound, VOFillingError {
    ValueObject o = null;
    try {
      o = VOHelper.createVO( voID, this, new InitialContext() );
    } catch ( NamingException ex ) {
      ex.printStackTrace();
    }
    return o;
 }
}
```

Die Operation *createVO()* der Hilfsklasse erwartet drei Parameter:

- Die ID des zu erzeugenden ValueObjects

- Die Instanz der Beanklasse, von der das ValueObject seine Daten bekommen soll
- Der Namenskontext dieser Bean zum Zugriff auf die Umgebungsparameter (wegen ID/Klasse Zuordnung).

Die Umgebungsparameter müssen für jede ID den Eintrag VO_<*ID*>=<*VO-Klassenname*> enthalten, beispielsweise *VO_Names=de.mathema.valueobjectfactory.example.NamesVO*, wobei *Names* die vom Client beim Aufruf zu verwendende ID darstellt.

Die Implementierung der Operation der Hilfsklasse ist naheliegend und soll hier kurz skizziert werden:

```
package de.mathema.valueobjectfactory;
// imports…
public class VOHelper {
  public static ValueObject createVO( String voID,
              Object o, Context c )
              throws VOClassNotFound, VOFillingError {
    String voClassName = null;
    try {
      voClassName = (String)c.
                  lookup( "java:comp/env/VO_"+voID );
      if ( voClassName == null ) {
        throw ( new VOClassNotFound(
                "Environment entry not found: VO_"+voID ) );
      }
      Class cl = Class.forName( voClassName );
      ValueObject vo = (ValueObject)cl.newInstance();
      vo.fill( o );
      return vo;
    } // verschiedene Exceptions fangen und in behandeln
    }
  }
}
```

Der Client kann nun ein solches ValueObject sehr einfach bekommen und verwenden:

```
package de.mathema.valueobjectfactory.example;
// imports…
public class PersonClient {
  public static void main( String[] args ) {
    Person p = // Referenz auf eine Person besorgen...
    try {
      NamesVO v = (NamesVO)p.createVO( "Names" );
      System.out.println( “Name ist “+v.getName() );
    } catch ( Exception ex ) {
      ex.printStackTrace() ;
    }
  }
}
```

Ändern von Attributen der Bean

Der lesende Zugriff auf Attribute mittels ValueObjects ist eine Sache – eine sehr wichtige zwar, aber üblicherweise müssen die Attribute auch aktualisiert werden können. Dies ist im Rahmen der vorgestellten Lösung kein Problem, mit den folgenden Schritten lässt sich eine einfache Lösung erreichen:

- Erweiterung des Interfaces *ValueObjectCreator* um eine Operation *updateVO(ValueObject)*. Dies erlaubt es dem Client, ein von ihm modifiziertes ValueObject der Bean zurückzugeben, mit dem Ziel, die Werte im ValueObject in die Bean zurückzuschreiben.
- Damit dies geschehen kann, sollte die abstrakte Basisklasse *ValueObject* einer Operation *writeBack(Object destination)* besitzen. Jede ValueObject-Unterklasse überschreibe die Operation so, dass die entsprechenden Attribute auf der Bean aktualisiert werden.

Der Mechanismus ist derselbe wie beim lesenden Zugriff: *updateVO()* ruft eben mittels der Hilfsklasse die Operation *writeBack()* auf, die wiederum das konkrete Zurückschreiben der Attribute übernimmt.

Die eigentlichen Probleme liegen hier etwas anders, und zwar haben die mit der Locking-Strategie zu tun. Nehmen wir an, Client A holt sich ein ValueObject, ändert es, und schreibt es zurück. In der Zwischenzeit hat aber nun ein anderer Client B ein ValueObject geholt und zuckgeschrieben. Im einfachsten Fall, also ohne spezielle Vorkehrungen, würden die Änderungen von Client B einfach von den Änderungen von Client A überschrieben. Dieses Verhalten ist sicherlich nicht immer gewünscht.

Das hier geschilderte Problem ist ein alt bekanntes in der Datenbankwelt, und es gibt verschiedene Lösungsansätze, die auf verschiedenen Sperrstrategien beruhen. Im Zusammenhang mit den hier geschilderten ValueObjects gibt es eine recht allgemeingültige und flexible Strategie, um dieses Problem anzugehen. Das erzielte Verhalten stellt sich folgendermaßen dar: Ein Client schreibt ein ValueObject zurück, dabei kann er ein Flag *overwrite* angeben. Ist dieses Flag *false* und hat in der Zwischenzeit ein anderer Client die Entity Bean aktualisiert, so wirft *updateVO()* eine Exception, andernfalls wird das Update durchgeführt. Ist das Flag *true*, so wird auf jeden Fall überschrieben.

Es gibt nun verschiedene Möglichkeiten, zu erkennen, ob in der Zwischenzeit ein anderer Client die Bean upgedatet hat. Die präziseste ist sicherlich die, dass sich das ValueObject die alten Werte aller Attribute merkt, und vor dem Update jedes einzelne auf Veränderungen überprüft. Nur wenn alle ungeändert sind, kann das Update erfolgen. Dies hat aber recht erheblichen Programmieraufwand zur Folge – auch zur Laufzeit sind Performanzeinbussen durch die Überprüfung nicht zu vermeiden.

Unter der Annahme, dass die Attribute einer Entity Bean ausschließlich über ValueObjects aktualisiert werden können, bietet sich eine andere, wesentlich elegantere Möglichkeit:

- Die EntityBean führt für jedes Art von ValueObject einen (persistenten) Zähler mit.
- Bei der Erzeugung eines *ValueObjects* wird in dem ValueObject der aktuelle Zählerstand gespeichert.
- Beim Update kann nun überprüft werden, ob der Zählerstand noch der gleiche ist. Wenn ja, wird aktualisiert und der Zähler erhöht. Andernfalls wird eine Exception geworfen.

Das Schöne an dieser Lösung ist, dass sie vollkommen generisch implementiert werden kann – der Programmierer von ValueObjects sieht davon nichts, die Logik steckt komplett in der Hilfsklasse und der ValueObject-Basisklasse. Übrigens kann dieser Zähler auch verwendet werden, um im Falle von Bean-Managed Persistence unnötige Updates in der Datenbank zu verhindern.

Eine weitere, schöne Erweiterung des Konzepts stellt die folgende dar. Nehmen wir an, ein ValueObject soll Attribute mehrerer EntityBeans enthalten. Dann lässt sich leicht eine Session Bean bauen, die nur dazu dient, ValueObjects zu erzeugen. Die entsprechenden *ValueObject*-Klasse kann dann die zugehörigen EntityBeans nach ihren Attributen befragen und auch beim Update wieder in die richtigen EntityBeans zurückschreiben. Natürlich muss dann beim Anfordern des ValueObject angegeben werden, aus welchen EntityBeans die Daten kommen sollen (Primärschlüssel).

Resümee

Am Anfang des Artikels haben wir bestimmte Anforderungen an die Lösung gestellt. Es soll nun diskutiert werden, ob und inwiefern die Lösung diesen Anforderungen gerecht wird.

Es war gefordert, dass beliebige neue ValueObject-Typen definiert werden können, ohne dass die Bean selbst verändert (oder neu deployt) werden muss. Dieses Ziel wurde erreicht. Das Verwenden eines neuen *ValueObjects* bedarf ausschließlich der Programmierung eines neuen *ValueObject*, und dessen Eintragung im Environment der Bean.

Die nächste Forderung war, dass keine Performanznachteile gegenüber der traditionellen Lösung entstehen sollten. Dieses Ziel wurde sicherlich nicht ganz erreicht, da einige zusätzliche Indirektionsschritte sowie das Lookup und das dynamische Erzeugen der *ValueObject*-Klassen hinzukommen. Wie üblich fordert Flexibilität ihren Preis bezüglich Performanz. Jedoch bleibt festzuhalten, dass diese Performanznachteile bei verteilten Anwendungen nicht ins Gewicht fallen werden.

Es sollte, wie bei der traditionellen Lösung, weiterhin möglich sein, dass Validierungen bereits auf dem Client erfolgen. Dies ist uneingeschränkt der Fall, da ValueObjects beliebige Klassen – incl. Validierungslogik – sein können. Übrigens steht dies im Gegensatz

zur bekannten „HashMap-Lösung“, bei der die Attribute in Name-Wert-Paaren in einer generischen ValueObject-Klasse gespeichert und dem Client zurückgegeben werden.

Die Typsicherheit der Operationen kann im großen und ganzen sichergestellt bleiben, es ist nur ein einziger Downcast nötig – nämlich beim Anfordern des *ValueObjects* mittels *createVO()*. Alle weiteren set/get Attributoperationen sind aber weiterhin typsicher.