

In Arbeit. Letzte Aktualisierung: 16.04.2005
JiBX: Noch ein XML Binding framework für Java!
von Sven Ehrke

Neue XML Bindung für Java

Verwenden Sie XML in Ihren Java Programmen? Und benutzen Sie dazu ein XML framework, bei dem Sie ständig das Gefühl haben, daß das doch einfacher gehen müßte? Nachdem Sie diesen Artikel gelesen haben, wissen Sie, wie die Java und XML Welt absolut einfach und komfortabel miteinander verbunden werden können. „Und wie sieht’s mit der Performance aus“ fragen Sie?. Obwohl JiBX erst in beta 3 vorliegt, zeigen die Messungen, daß JiBX auch hier viel besser abschneidet, als die meisten anderen frameworks.

Von der Java Welt noch weitgehend unbemerkt macht sich das Projekt JiBX [1] daran, die XML Java Binding Welt zu erobern. Warum das so ist, werden wir im Lauf dieses Artikels sehen.

Bisherige Ansätze

Man kann Java und XML natürlich mit DOM oder SAX direkt verbinden. Obwohl man mit dem DOM Modell einigermaßen komfortabel programmieren kann, kommt es in vielen Situationen nicht in Frage, da es einfach zuviel Speicher für den Aufbau des Baums benötigt. Das eventbasierte SAX Modell benötigt dagegen zwar nicht soviel Speicher, ist aber von der Anwendung her nicht besonders komfortabel. Und wenn man sich dann eine Weile durch den XML Dschungel geschlagen hat, fällt einem vielleicht auch wieder ein, daß man dafür eigentlich keine Zeit hat, sondern das eigene Programm endlich fertig machen sollte. Dieser Meinung waren offensichtlich auch andere Leute und fingen an, eigene frameworks zu entwickeln.

In einem seiner Artikel [2] vergleicht Dennis Sosnoski, der Autor von JiBX die gebräuchlichsten frameworks bezüglich Ansatz und Performance miteinander. JiBX schneidet in diesen Untersuchungen in allen Kategorien erstklassig ab.

Viele der bekannten frameworks generieren Java code, der auf einem DTD oder Schema basiert. JAXB von Sun ist zum Beispiel so ein framework. Als ich es vor etwa zwei Jahren entdeckte war ich sehr davon angetan, da man einfach aus einem DTD Java Klassen generieren konnte, die das zu verarbeitende XML Dokument genauestens repräsentierten. Mit Hilfe dieser Java Klassen war es dann ein Leichtes, sehr bequem XML Dokumente einzulesen und auch wieder zu generieren.

Die Nachteile dieser Technologie werden einem erst nach einiger Zeit bewußt. Zum einen generiert man pro DTD/Schema immer sehr viele Klassen. Benutzt man mehr als eine Handvoll verschiedener XML Dokumenttypen, kann das schnell sehr groß und unübersichtlich werden. Der zweite Nachteil dieser Technologie ist, daß immer das gesamte XML Dokument in Java Objekte dieser generierten Klassen umgewandelt wird, selbst wenn man sich nur für einen kleinen Ausschnitt oder ein paar Werte dieses Dokuments interessiert. Und drittens wird auch mit dieser Technologie ein Baum aufgebaut, der ziemlich viel Speicher benötigt. Damit will ich es mit der Betrachtung anderer frameworks gut sein lassen und zum eigentlichen Thema kommen.

Ein bißchen Theorie

JiBX generiert wie gesagt keinen Code, sondern arbeitet mit einem Mappingverfahren. JiBX schreibt die XML Daten in Java Objekte, die der Programmierer selbst erstellt hat. Dies hat den Vorteil, daß man sie so gestalten kann, wie sie in der jeweiligen Situation am vorteilhaftesten sind. In einer sogenannten binding Datei definiert man, wie die Daten vom XML Dokument in diese Java Objekte und umgekehrt abgebildet werden.

Ein erstes Beispiel

Nehmen wir an, unser Programm hat eine Klasse namens Person (s. Abb.)

Listing 1: Person.java

```
package organization

class Person
{
    public String firstName;
    public String lastName;
}
```

Die Daten zu unserer Person liegen nun in folgender XML Datei vor:

Listing 2: person.xml

```
<?xml version="1.0"?>
<person>
  <firstname>
    Bart
  </firstname>
  <lastname>
    Simpson
  </lastname>
</person>
```

Um unser Personenobjekt mit den Daten aus der XML Datei zu füllen, müssen wir noch irgendwie festlegen, wie das passieren soll. Wie schon gesagt, verwendet JiBX hierzu den Mappingansatz. Alles, was wir tun müssen, ist ein sogenanntes binding zu definieren. Dies wird ebenfalls in einer XML Datei gemacht. Hier die binding Definition für unser Beispiel:

Listing 3: binding.xml

```
<binding name="org_binding">
  <mapping name="person"
    class="organization.Person">
    >
    <value name="firstname"
      field="firstName"/>
    <value name="lastname"
      field="lastName"/>
  </mapping>
</binding>
```

In dieser Bindingdefinition wird ein einziges mapping definiert. Das Attribut name des mapping elements enthält den Wert person. Damit wird festgelegt, daß das Element <person> gemappt werden soll. Auf was, wird im Attribut class festgelegt: nämlich auf unsere Klasse Person. Im Klartext heißt das: taucht ein Element <person> beim Parsen des XML Dokuments auf, wird ein Objekt vom Typ Person erzeugt. Für das Abfüllen dieses Objekts mit den gewünschten Daten sorgen die beiden value Elemente in der mapping definition. Das Attribut name des elements value bestimmt, welches XML Element gemeint ist und das Attribut

In Arbeit. Letzte Aktualisierung: 16.04.2005

field gibt an, in welches Attribut des Javaobjekts der gelesene Wert geschrieben werden soll. Hier wollen wir also, daß der Wert des Elements <firstname> nachher in Person.firstName und der Wert des Elements <lastname> in Person.lastName verfügbar ist.

Bytecode enhancement

Damit das XML binding transparent abläuft und trotzdem performant ist, benutzt JiBX das sogenannte bytecode enhancement. Dazu wird die kompilierte Klasse Person um ein paar Funktionen ergänzt, die JiBX dann für das sogenannte Marshalling (Java nach XML Umwandlung) und Unmarshalling (umgekehrt) verwendet.

Als erstes muss die Klasse Person und was man sonst noch so an Java code benötigt wie gewohnt kompiliert werden. Danach wird der JiBX Compiler

angeworfen, der das erwähnte bytecode enhancement durchführt:

```
java -jar jibx-bind.jar binding.xml
```

Dann benötigen wir nur noch ein kleines main, mit dem wir das XML einlesen und den Namen der Person ausgeben (Listing 4). Und so sieht die Ausgabe aus:

Listing 5:

```
Person.firstName: Bart  
Person.lastName: Simpson
```

Zu den Beispielen dieses Artikels gibt es ein kleines Ant script mit dem man die Schritte kompilieren, enhancen und ausführen bequem durchführen kann:

```
ant compile enhance  
ant run
```

Fazit

Links & Literatur

- [1] <http://jibx.sf.net/>
- [2] <http://www-106.ibm.com/developerworks/library/x-databdopt2/>
- [3]

Listing 4: Main.java

```
public static void main(String[] args){  
    Person p;  
    IBindingFactory ibf;  
    IUnmarshallingContext uctx;  
    Object obj = null;  
  
    try {  
        // Get binding factories for the named bindings:  
        ibf = BindingDirectory.getFactory("binding1", organization.Person.class);  
        // Unmarshal document to construct bean:  
        uctx = ibf.createUnmarshallingContext();  
        obj = uctx.unmarshalDocument(new FileInputStream("person.xml"), null);  
        p = (Person) obj;  
        System.out.println("Person.firstName: " + p.firstName);  
        System.out.println("Person.lastName: " + p.lastName);  
    }  
    catch (Exception ex) {  
        ex.printStackTrace(System.out);  
    }  
}
```