



## XML meets JavaBeans

## Zugriff auf Altsystemdaten

Andreas Senft

Oft werden durch Projekte Altsysteme abgelöst und es stellt sich die Aufgabe, die Altsystemdaten zu migrieren. Für den Fall dateibasierter Altsysteme wird hier ein effektiver und gleichzeitig flexibler Ansatz präsentiert, bei dem auf proprietäre Binärdateien via XML und JavaBeans zugegriffen wird. Ein vereinfachtes Beispiel mit reinen Textdateien verdeutlicht die Idee.

## Motivation

Im Rahmen eines Migrationsprojekts stellte sich die Aufgabe, ein bestehendes Verfahren auf Basis von dezentralen Einzelplatzanwendungen zu zentralisieren. Im Rahmen der Migration waren hierbei auch die Daten der bestehenden Applikationen in eine neue, zentrale, Datenbank zu überführen. Da das neue Verfahren auf der J2EE aufsetzt, bot sich eine Java-Lösung der Migrationskomponente an.

Die Ursprungsdaten liegen in verschiedenen Dateien in proprietären Binärformaten vor. Ziel muss also sein, dieses Datenformat auszulesen und in gültige Java-Objekte zu überführen. Da die Migrationsphase über einen längeren Zeitraum hinweg läuft, während das bestehende Verfahren weiter in Betrieb ist, wird außerdem eine kompatible Backup-Möglichkeit benötigt. Dies bedeutet, dass im Falle des Verlusts der Ursprungsdateien diese aus der zentralen Datenbank regeneriert werden müssen.

Insgesamt betrachtet, sieht die Ausgangslage also folgendermaßen aus:

- ▼ Eine Migrationskomponente muss proprietäre Binärdateien lesen und den Inhalt in Java-Objekte überführen können.
- ▼ Eine Backupkomponente muss proprietäre Binärdateien schreiben können, wobei der Inhalt aus Java-Objekten konvertiert werden muss.
- ▼ Die Binärdateien haben jeweils eine feste Satzlänge und beinhalten Sätze verschiedener Strukturen.

Da zwei verschiedene Komponenten auf die Dateien zugreifen müssen, bietet es sich an, die Schnittstellenfunktionalität in eine dritte, gemeinsame, Komponente auszulagern. Da die Zugriffe im Prinzip immer gleichartig sind, ist ein generischer Ansatz vielversprechend. Dies bietet folgende Vorteile:

- ▼ Die gesamte Zugriffsfunktionalität ist ausschließlich in einer Komponente gekapselt. Dies erhöht die Wartbarkeit.
- ▼ Die generischen Zugriffsfunktionen lassen sich einfach testen, wodurch eine höhere Testabdeckung erreicht wird (das Testen jedes speziellen Zugriffs ist schlicht unpraktikabel).

- ▼ Falls dennoch Fehler in der generischen Schnittstellenkomponente vorliegen, werden diese in der Entwicklung schnell bemerkt, da die Funktionalitäten häufig verwendet werden (im Gegensatz zu konkreten Implementierungen für jeden einzelnen Datenzugriff).
- ▼ Durch die deklarative Spezifikation der Dateistruktur entsteht als Nebenprodukt eine Strukturbeschreibung der Originaldateien, welche auch bei Anpassungen (notwendigerweise) stets aktuell ist.

Im Rahmen der Erstellung dieses Artikels habe ich eine vereinfachte aber funktionierende Beispielapplikation [App] erstellt, welche den generischen Ansatz auf Basis von Textdateien veranschaulicht. Eine Anwendung auf Binärdateien ist nicht ganz so anschaulich, würde allerdings genau derselben Vorgehensweise folgen, welche hier erläutert wird. Die wesentlichen Bestandteile der Beispielapplikation werden im Folgenden genauer beschrieben.

Anmerkung: Eine Implementierung (so wie die Beispielapplikation) ist nicht dafür gedacht, universell verwendbar zu sein, sondern sie ist angepasst auf einen konkreten Kontext. Das prinzipielle Verfahren als solches ist allerdings für viele Anwendungsfälle geeignet.

## Struktur der proprietären Dateien

Für die Beispielapplikation [App] kann eine Datei aussehen wie in Listing 1. Hier sind drei verschiedene Datensatzarten zu unterscheiden. Der erste Satz ist der Kopfsatz, welcher Zähler für die Datensätze enthält. Danach folgen Sätze, welche Personen oder Unternehmen beschreiben. Diese Sätze werden durch ein Kennzeichen (P oder C) unterschieden.

2	1				
PHans	Müller	19630914	Frankfurt	1	
CGärtnerei	Schmitt	Hamburg			
PFriedrich	Schulz	19570107	Bertin	2	

Listing 1: Exemplarische Dateistruktur



Das vorgestellte Verfahren kann vom Prinzip her sowohl auf Text- als auch auf Binärdateien operieren. Es wurde ursprünglich entwickelt für Dateien mit Sätzen fester Länge, wodurch ein effizienter Zugriff (über `RandomAccessFile`) realisiert werden kann. Prinzipiell ist aber auch eine Anpassung für Dateien mit variablen Satzlengthen vorstellbar. Wenn ein initialer Indizierungsschritt möglich ist, kann auch in einem solchen Fall ein wahlfreier Zugriff auf die Datensätze realisiert werden.

## Beschreibung der Dateistruktur in Form von JavaBeans

Die Beschreibung einer Datei und der Struktur ihrer Datensätze ist durch eine generische Objektstruktur darstellbar (s. Abb. 1). Die Interpretation der Elemente der oben beschriebenen Textdateien ist in den Klassen `GenericTextFile` und `TextFileRecord` gekapselt. Die Interpretation anderer Datenstrukturen kann mittels Austauschs dieser Klassen durch geeignete Alternativimplementierungen vorgenommen werden, für den Anwender bleibt dies jedoch verborgen.

Die Art der Satzbeschreibungen bleibt von der Art der Dateninterpretation unberührt. Das Spezifizieren der Datenstrukturen und der Zugriff auf die so spezifizierten Elemente ist für Text- wie für Binärdaten identisch.

`GenericFile` beinhaltet eine oder mehrere Satzbeschreibungen (`RecordDefinition`). Dieser Typ ist dafür verantwortlich, konkrete Dateien zu öffnen, welche mit den registrierten Satzstrukturen interpretiert werden können. Weiterhin bietet `GenericFile` Möglichkeiten, Sätze (`Record`) wahlfrei zu lesen, neue Sätze zu erzeugen und Sätze zu schreiben.

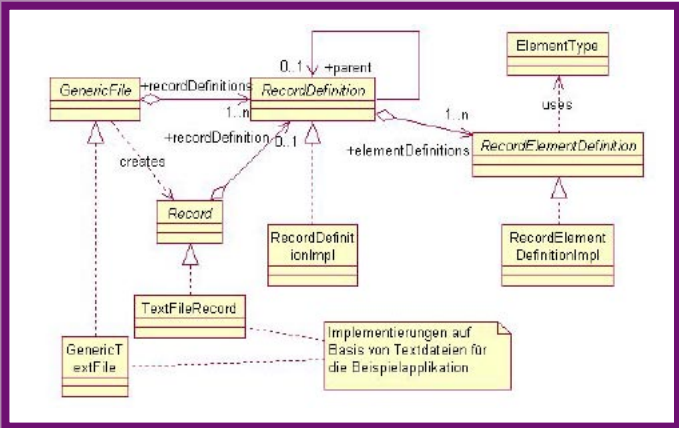


Abb. 1: Generisches Modell der Dateistruktur

Eine **RecordDefinition** hat eine (je Datei-beschreibung) eindeutige Id und eine zugeordnete Datensatzlänge. Der eigentliche Inhalt des definierten Datensatzes leitet sich aus den beinhalteten Elementdefinitionen (**RecordElementDefinition**) ab. Optional kann eine Hierarchie von Satzbeschreibungen gebildet werden. Dies bietet sich dann an, wenn mehrere Datensatzarten gleiche Elementdefinitionen aufweisen.

Die **RecordElementDefinition** beschreibt ein einzelnes Datensatzelement. Eine solche Instanz beschreibt die Lokation der Elementrepräsentation innerhalb eines Datensatzes und spezifiziert dessen Typ (**ElementType**, ein Aufzählungstyp, welcher die unterstützten Typen definiert). Außerdem weist sie diesem Element einen eindeutigen Namen zu. Die Lokation einer Elementrepräsentation innerhalb eines Datensatzes wird über Start- und Zielindizes angegeben. Im Falle von Textdateien beziehen sich die Indizes auf Zeichen der Textdatei. Bei Binärdateien wird auf die Bytefolge Bezug genommen.

**Record** repräsentiert schließlich einen logischen Datensatz, welcher entsprechend der zugeordneten Datensatzdefinition (**RecordDefinition**) Zugriff auf die definierten Elemente über deren Namen erlaubt. Der Anwender ist somit von der eigentlichen Datenquelle entkoppelt. Das Lokalisieren und Konvertieren von Elementrepräsentationen geschieht transparent innerhalb der generischen Klassen. Die Implementierung dieser Schnittstelle in der Beispielapplikation ist textbasiert und operiert auf einem **StringBuffer**, eine Implementierung für Binärdateien würde hier mit **byte[]** arbeiten. Die Implementierungsklassen der Interfaces stellen Methoden zur Konfiguration und Validierung bereit.

Abschließend sei angemerkt, dass die dargestellte Struktur trotz ihrer Leistungsfähigkeit durchaus noch ausbaufähig ist. So lässt sich beispielsweise durch einen Subtyp von **RecordElementDefinition** mit Bezug zu einer **RecordDefinition** sehr einfach das Konzept rekursiver Datenstrukturen innerhalb eines Datensatzes beschreiben.

### Deklarative Ablage der Datei-beschreibung

Zunächst wurde dargestellt, wie die Dateistruktur über Java-Klassen angemessen repräsentiert werden kann. Das nächste Ziel ist es, die deklarative Beschreibung der Dateistruktur zu definieren. Das heißt, es wird eine textuelle Beschreibung der Struktur benötigt, welche dann in das präzentrierte Java-Modell übertragen wird. Als Ablageform bietet sich hier aufgrund der guten Unterstützung, speziell auch im Java-Umfeld, XML an.

Die XML-basierte Konfiguration der JavaBeans, welche die Datei-beschreibungen bilden, wurde hier nicht manuell implementiert. Stattdessen wurde auf das Spring-Framework [Spring] zurückgegriffen. Dieses Open-source-Projekt bietet ein flexibles und mächtiges Framework, welches auf einem Lightweight Container aufsetzt. Der Vorteil eines solchen Containers ist die weitestgehende Unabhängigkeit der Applikation vom Framework selbst. Weitere Informationen zu Spring sind der Website sowie den Büchern [Joh02] und [Joh04] zu entnehmen. Selbstverständlich kann die benötigte Funktionalität auch durch den Einsatz eines anderen Lightweight Containers erzielt werden. Als Alternative sei hier z. B. PicoContainer [Pico] genannt.

Die Umsetzung der deklarativen Datei-beschreibung mit Hilfe von Spring wird im Folgenden demonstriert. Listing 2 zeigt einen Auszug aus dem XML-Dokument, welches die Dateistruktur beschreibt. Die prinzipielle Struktur ist hierbei sehr einfach. Bei der Definition einer JavaBean wird dieser ein logischer Name (Attribut **id**) zugewiesen, über den die Bean referenziert werden kann. Die Instanziierung erfolgt dann über den Standard-Konstruktor der spezifizierten Klasse (die Verwendung parametrierter Konstrukto-ren ist theoretisch auch möglich).

```

<beans>
<bean id="Demo"
  class="demo.core.impl.GenericTextFile"
  init-method="validate">

  <property name="recordDefinitions">
  <list>
  <ref local="Demo_Def_Header"/>
  <ref local="Demo_Def_AbstractData"/>
  <ref local="Demo_Def_PersonData"/>
  <ref local="Demo_Def_CompanyData"/>
  </list>
  </property>
  </bean>

  <!-- Beschreibung eines Datensatzelements -->
  <bean id="Demo_Element_PersonCount"
    class="demo.core.impl.RecordElementDefinitionImpl"
    init-method="validate">

    <property name="name"><value>PersonCount</value></property>
    <property name="type"><value>int</value></property>
    <property name="startIndex"><value>0</value></property>
    <property name="endIndex"><value>4</value></property>
  </bean>
  <!-- weitere Bean-Definitionen -->
</beans>

```

Listing 2: Auszug aus der Datei-beschreibung

Die Parametrierung der Instanz erfolgt über die spezifizierten Attributwerte. Hierbei wird z. B. erwartet, dass für das Property name eine Methode der Form **setName(...)** in der angegebenen Klasse existiert. Dies folgt der JavaBean-Konvention [JavaBeans] von Sun Microsystems. Für einige Datentypen (u. a. alle primitiven Typen) werden automatisch geeignete Konversionen durchgeführt. Für benutzerdefinierte Typen können auch sehr einfach eigene Konverter auf Basis der Klasse **java.beans.PropertyEditor** definiert werden. In der Beispielapplikation wird für den Typ **ElementType** von dieser Methode Gebrauch gemacht.

Neben der Definition von einfachen Werten können auch andere Objekte referenziert werden. Man kann also sehr einfach ein ganzes Netz von JavaBeans deklarieren und über das Spring-Framework transparent darauf zugreifen. Bei konsequenter Trennung von Interfaces und implementierenden Klassen kann man letztere sogar komplett aus dem eigentlichen Applikationscode heraushalten, wie man in der Beispielapplikation schön sehen kann.

Die optionale Spezifikation von **init-method** bei der Bean-Definition wird zum Aufruf einer Methode zur Validierung verwendet. Dies erleichtert das Aufspüren von Konfigurationsproblemen.

Der Zugriff auf die Datei-beschreibung ist sehr einfach, wie man in Listing 3 sieht. Die Bean mit dem logischen Namen „Demo“ wird über eine **BeanFactory** erzeugt. So erhält man eine Instanz, welche das Interface **GenericFile**



```

BeanFactory factory
= new ClassPathXmlApplicationContext("file-config.xml");

GenericFile genericFile = (GenericFile)
factory.getBean("Demo", GenericFile.class);

```

Listing 3: Instanzieren der Dateibeschreibung mit Spring

implementiert. Die Zuordnungen zu den definierten Instanzen von `RecordDefinition` bzw. `RecordElementDefinition` sind bereits komplett vorhanden. Der Verwendung steht also nichts mehr im Wege.

### Der Zugriff auf konkrete Dateien über die generische Schnittstelle

Nachdem die Struktur definiert ist, kann nun eine konkrete Datei geöffnet werden und die enthaltenen Datensätze können gelesen und ausgewertet werden. Listing 4 zeigt den exemplarischen Lese- und Schreibzugriff auf ein Feld des Kopfsatzes der Datei aus Listing 1. Die `Record`-Implementierung beherrscht das Lesen und Schreiben der benötigten Java-Typen und bewirkt damit die Lokalisierung der Typkonversionsfunktionalität.

Natürlich schließt auch der Datenzugriff mit der hier vorgestellten Vorgehensweise nicht alle Fehlerquellen aus. Jedoch werden die möglichen Fehlerquellen stark reduziert, da sie zur Laufzeit erkannt werden können:

- ▼ Es stehen nur die definierten Datensatzdefinitionen zur Verfügung.
- ▼ Nur auf definierte Elemente der aktuellen Datensatzdefinition kann zugegriffen werden.
- ▼ Der deklarierte Typ eines Datensatzelements wird beim Zugriff gegen den erwarteten Typ abgeglichen.
- ▼ Die Codierung und Position der Repräsentation eines Datensatzelements ist für den Anwender ohne Bedeutung. Zugriffsfehler aufgrund fehlerhafter Indizierung sind somit ausgeschlossen. Auch die Art der Herkunftsdateien (binär oder textuell) ist für den Anwender transparent.

### Typisierte Kapselung des Datenzugriffs

Der in Listing 4 dargestellte Zugriff auf die Dateien ist sehr flexibel. Als problematisch kann allerdings die mangelnde Typisierung beim Zugriff auf die Datenelemente angesehen werden. Diese erfolgt über Stringlitterale, welche in der Dateibeschreibung definiert sind. Um diesen Zustand zu entschärfen, kann eine zusätzliche Klasse zur typischeren Kapselung herangezogen werden (für ein Beispiel siehe Listing 5). Eine solche Klasse kann aus den Informationen der Dateibeschreibung leicht über einen Codegenerator erzeugt werden und verursacht dadurch keinen erhöhten Entwicklungsaufwand.

### Fazit

Die „einfache“ Möglichkeit, den Zugriff auf proprietäre Bestandsdateien

für alle Dateibestandteile explizit zu programmieren, führt für nicht-triviale Fälle zwangsläufig zu recht umfangreichem Code; dieser ist daher schwer wart- und testbar. Fehler können durch die verteilte Implementierung der Zugriffslogik unter Umständen erst spät bemerkt werden.

Als Alternative wurde im Verlauf dieses Artikels ein flexibler Ansatz vorgestellt, Datenstrukturen von Altsystemdateien deklarativ zu beschreiben. In Verbindung mit einer generischen Zugriffskomponente werden die oben genannten Nachteile deutlich reduziert. Man erhält eine überschaubare, gut test- und wartbare Codebasis. Fehler in der Dateibeschreibung können einfach und schnell korrigiert werden. Als Bonus erhält man sogar eine stets aktuelle Dateibeschreibung in Form des XML-Dokuments. Die typisierte Zugriffsschicht lässt sich darüber hinaus mit Hilfe eines Codegenerators automatisch erzeugen.

Die vorgestellte Methodik ist einfach auf verschiedene Anwendungsfälle anpassbar und hat sich auch bereits im Projekteinsatz bewährt.



**Dipl.-Inform. (FH) Andreas Senft** ist Berater bei der eMundo GmbH Unterhaching/Frankfurt a.M. Seit Ende 1996 ist er in der objektorientierten Softwareentwicklung tätig und verfügt mittlerweile über sechsjährige Erfahrung in der Java-Programmierung. Er war maßgeblich an mehreren Projekten zur Entwicklung von Enterprise-Applikationen beteiligt. E-Mail: [Andreas.Senft@e-mundo.de](mailto:Andreas.Senft@e-mundo.de).



### Weitere Informationsquellen

<http://team.e-mundo.de/eInfo/Members/andreas/downloads.html>

```

GenericFile genericFile = ...
File dataFile = ...

genericFile.open(dataFile, true);

try {
    Record headerRec = genericFile.readRecord(0, "Header");

    int personCount = record.getIntElement("PersonCount");
    record.setIntElement("PersonCount", ++personCount);

    genericFile.writeRecord(record);
} finally {
    genericFile.close();
}

```

Listing 4: Generischer Dateizugriff

```

public final class Demo {
    public static String DEF_HEADER = "Header";
    // weitere Konstanten

    private Demo() {}

    public static int getPersonCount(Record record) {
        return record.getIntElement("PersonCount");
    }

    public static void setPersonCount(Record record, int value) {
        record.setIntElement("PersonCount", value);
    }
    // weitere Methoden
}

```

Listing 5: Wrapper zur Typisierung des generischen Zugriffs

### Literatur und Links

- [App]** Beispielapplikation zum Artikel, <http://team.e-mundo.de/eInfo/Members/andreas/downloads.html>
- [JavaBeans]** <http://java.sun.com/products/javabeans/reference/>
- [Joh02]** R. Johnson, Expert One-on-One J2EE Design and Development, Wrox Press, 2002
- [Joh04]** R. Johnson, J. Höller, Expert One-on-One J2EE Development without EJB, Wrox Press, 2004
- [Pico]** PicoContainer, <http://www.picocontainer.org>
- [Spring]** Spring-Framework, <http://www.springframework.org>