Dr. Hartmut Schorrig, www.vishia.org

Lesen, Analysieren und Verarbeiten von XML-Files mit XmlJzReader

Abstrakt

Zum Lesen von XML-Files ist traditionell das DOM- und SAX-Modell bekannt. DOM liest den gesamten XML-Baum ein und präsentiert ihn als interne Daten, hier in Java. SAX benötigt für jedes Element eine programmierte Entscheidung. Zweiteres ist aufwändig, ersteres braucht viel Speicher und produziert viele Daten, die im Konkretfall gar nicht benötigt werden. Systeme mit Annotations an User-Sourcefiles zum Einlesen von XML-Daten sind ebenfalls recht aufwändig, wenn zunächst noch in der Struktur unbekannte XML-Daten gesichtet und verarbeitet werden sollen. Zudem sind mit den jeweiligen Systemen umfangreiche Libraries verbunden, mit entsprechenden Dependencies.

Die hier vorgestellte Java-Module XmlJzReader filtert XML-Daten über einen Konfigurationsfile bereits beim Einlesen. Der Konfigurationsfile enthält die Anweisungen zum Ablegen der Daten in Textform, über Reflection in Java realisiert. Zudem existieren Module für die Analyse der XML-Datenstruktur, um automatisch die Vorlage für den passenden config.xml-File zu erstellen und um automatisch aus einem gegebenen config.xml-File die Java-Sources der Datenablage zu erstellen.

Damit braucht es insbesondere für die Analyse von noch in der Stuktur unbekannten XML-Files zunächst wenig Aufwand, um die Daten im Java-Kontext einzulesen und zu sichten. Die XML-Daten können dann mit der Leistungsfähigkeit von Java-Algorithmen verarbeitet werden, als Gesamtdaten, ohne den/die Input-XML-Files weiter zu benötigen, um letztlich entsprechende Ausgabefiles zu erstellen, als XML. Text oder anderes bzw. die Daten intern spezifisch zu verwerten.

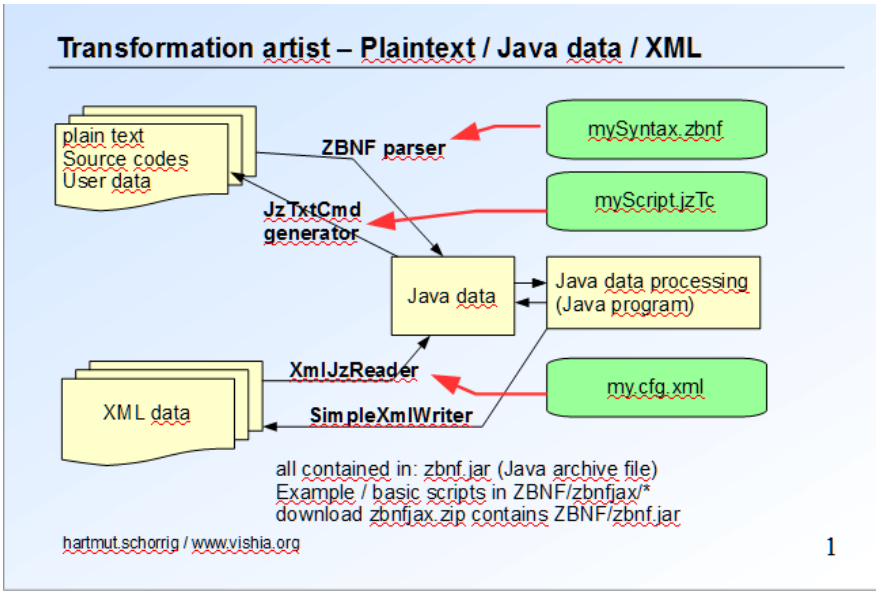
Inhaltsverzeichnis

1. Einordnung

Das folgende Bild zeigt die Einordnung des XmlJzReaders in die Möglichkeiten der Text- und XML-Konvertierungen mit dem vishia zbnf.jar-File. Die Namensgebung des Files kommt vom ZBNF-Parser, der in diesem Zusammenhang allerdings nicht benutzt wird.

Dieser Artikel bezieht sich auf das Einlesen von XML-Daten zum Zweck der Darstellung innerhalb Java zur Weiterverarbeitung.

Darauf aufbauend können die Daten entweder konvertiert wieder ausgegeben werden, als Text ausgegeben werden, oder intern zur Steuerung innerhalb Java anderweitig verwendet werden oder über Kommunikationsschnittstellen mit anderen Softwareteilen zusammenarbeiten.



Für eine Gesamtanwendung könnte einerseits die Verwendung des JzTxtCmd-Interpreters oder Text-Erzeugers interessant werden. Andererseits kann eine Textrepräsentation neben der standardgemäßen Textausgabefunktionalitäten im Java über die class org.vishia.util.OutTextPreparer

2. Das Konfigurationsfile config.xml

Der XmlJzReader benötigt einen Konfigurationsfile für das Einlesen, der nachfolgend als config.xml bezeichnet wird. Es ist ein xml-File. Die Benennung sollte im allgemeinen `MyApplication.cfg.xml` sein, das File hat eine zentrale Bedeutung in der selbst erstellten Tool-Umgebung.

Das config.xml kann automatisch einmalig oder wiederholt zum Vergleich erstellt, werden, siehe Kapitel *3 Automatische Generierung des config.xml aus gegebenen XML-Daten* Seite *3.* Es kann und sollte dann per Hand an die konkreten Belange angepasst werden.

2.1 Kopf, Namespace

Folgend wird das config.xml am Beispiel erläutert:

```
<?xml version="1.0" encoding="windows-1252"?>
<xmlinput:root xmlns:xmlinput="www.vishia.org/XmlReader-xmlinput" >
```

Die NameSpace-Deklaration im Kopf sorgt für eine eindeutige Unterscheidung von User-Namespaces und der hier notwendigen Control-Informationen, die also mit Attributen des `xmlinput`-Namespace gekennzeichnet werden.

2.2 Subtree

```
  <xmlinput:subtree xmlinput:name="user_tag"  xmlinput:class="user_tag"
          xmlinput:data="!new_user_tag()" >
    <subtag>!set_subtag(text)</subtag>
  </xmlinput:subtree>
```

Die `xmlinput:subtree`-Elemente sind dann in Benutzung, wenn ein Element in verschiedenen Node-Ebenen gleich verwendet wird, also mit gleichem Tagnamen auch tatsächlich die vergleichbare Information enthält. Das ist insbesondere bei rekursiven Node-Strukturen der Fall, dann geht es nur so. Aber auch bei faktisch gleichen Node-Typen in unterschiedlichen Node-Strukturen kann dies eine gute Vereinheitlichung sein. Dem selben `xmlinput:subtree` entspricht bei der Datenspeicherung die Verwendung der selben class-Instanz.

Im `xmlinput:subtree` ist eine Substruktur wie auch direkt unterhalb der Root enthalten.

```
<xmlinput:cfg xmlinput:data="!new_root()"  xmlinput:class="root" >
    <tag1 xmlinput:data="!new_tag1()" xmlinput:list="" xmlinput:class="tag1" >
      <tag2>!set_tag2(text)</tag2>
      ......
    </tag1>
  </xmlinput:cfg>
</xmlinput:root>
```

Das `xmlinput:cfg` - Element ist dasjenige, dass dem User-Xml-File ab der root-Ebene entspricht. Das `<tag1...` entspricht also dem ersten Tag im User-XML-File.

2.3 Nicht erwähnte Element-Tags und Attribute: Keine Datenspeicherung

Grundsätzlich gilt: Wenn ein tag-Name im config.xml nicht erwähnt wird, aber im Input-XML steht, dann wird dieses Input-Element insgesamt (mit Sub-Elementen) nicht ausgewertet. Es werden keine Daten gespeichert. Das Element muss selbstverständlich im User-XML-File korrekt, well-formed sein. Es wird geparst, um dessen richtiges Ende zu erkennen. Dies ist die im Abstrakt erwähnte Filterwirkung, um nicht zu viele Daten zu erzeugen. Adäquates gilt für die Attribute. Löscht man also `<tag2...>` und weitere aus einem automatisch generierten config.xml, weil im Konkretfall nicht benötigt, dann ist dies die richtige Herangehensweise.

2.4 Speicher-Operation für Elemente auch als Create-Operation

Unter `xmlinput:data="!.."` wird die Speicherroutine oder das Speicherfeld für das Element beschrieben, wobei das ! Pflicht ist. Die Speicherroutine / das Speicherfeld wird über Reflection in der aktuellen Java-class, also anfänglich in der unmittelbar angegeben Speicherinstanz, aufgesucht. Ist es nicht vorhanden, dann erfolgt eine Fehlermeldung. Die Bezeichnung der Speicherroutine / des Speicherfeldes ist beliebig. Für die automatische Generierung des Ablage-Java-Files wird allerdings das new_tagname verwendet, (Stand 2019-08-19).

Eine Speicherroutine hat (...) mit ggf. Argumenten, ein Speicherfeld (variable, Field in Reflection) hat keine (). Der Typ des Speicherfeldes in der Java-Ablage sollte `String` sein, oder ein numerischer Wert wird automatisch konvertiert.

Die Angabe `xmlinput:class="tag1"` ist nur von Bedeutung für die automatische Generierung des Ablage-java-Files, es wird in diesem Fall in der Speicherroutine eine new Tag1() erzeugt und die entsprechende class (mit uppercase als erstes Zeichen) angelegt. Für einen handkorrigierten config.xml und einer handkorrigierten Ablageclass hat dieser Eintrag dann keine Bedeutung mehr, sollte aber für eine Nachgenerierung (mit merge) konsistent gehalten werden.

Adäquat verhält es sich mit dem angegebenen Attribute `xmlinput:list=""`. Dann wird bei der Generierung der Ablage-Java-source ein List<Type>-Container vorgesehen um ein mehrfaches Auftreten des gleichen Elementes im eingelesenen XML-Input richtig zu speichern.

2.5 Attribute als Argumente für Speicher/Create-Operation

```
    <ref idref="!@idref"  xmlinput:data="!new_ref(idref)"  xmlinput:subtree="ref" />
```

Dies ist ein Beispiel, bei dem der Inhalt des Attributes `idref` als Argument für die Speicheroperation benutzt wird. Die Ablage ist damit einfacher und eindeutig programmierbar bzw. wird auch so automatisch generiert:

* Der Attributwert muss nicht mit einer extra Routine gespeichert werden, das ist aber ein unwichtiger Unterschied

* Der Attributwert ist in Java als `final` Variable speicherbar weil der Wert immer beim Constructor feststeht (noch nicht relalisiert im automatischen Generator Stand 2019-08-18, aber manuell so schreibbar).

* Es ist klar ersichtlich, welche Attribute-Werte immer besetzt sind.

Kennzeichen ist "!@..." nach dem @ der Argumentname. der Attributwert selbst wird nur lokal gespeichert um in der Speicheroperation übergeben zu werden, nicht extra abgelegt.

2.6 Attributewerte gespeichert

Ist ein Attribute mit `name="!operation(name, value)"` gekennzeichnet, dann wird nach der Anlage der zum Element zugehörigen Instanz die angegebene Operation aufgerufen. name und value sind Variable, die name und value des Attributes enthalten. Beides kann entfallen.

Wenn `name="!variable"` geschrieben wird, dann wird der Attribute-Wert in der genannten String-Variable gespeichert.

2.7 Attribute als Element-Speicher-Auswahl

Häufig wird in XML-Files ein sehr allgemeines Tag verwendet, um mit einem spezifischen Attributwert dann zu bestimmen, um was es sich handelt. Dazu ein Auszug aus einem Simulink-xml-File:

```
      <Block BlockType="Constant" Name="Constant1" SID="36">
        <P Name="Position">[805, 190, 890, 210]</P>
        <P Name="ZOrder">5820</P>
        <P Name="Value">2</P>
        <P Name="OutDataTypeStr">single</P>
        <Port>
          <P Name="PortNumber">1</P>
          <P Name="Name">x2</P>
        </Port>
      </Block>
```

Alle mit `<P...>` bezeichneten Elemente enthalten ganz unterschiedliche Informationen, die in Name="..." bestimmt werden.

Der zugehörige config-xml-Ausschnitt sieht wie folgt aus (gekürzt):

```
<Block BlockType="!@blockType" Name="!@name" SID="!@sid"
    xmlinput:data="!addBlock(blockType, sid, name)">!text(text)
  <P Name="!CHECK"/>  <!--XmlReader: the attribute NAME is used as key-->
  <P Name="Ports">!portInfo(text)</P>   <!--Info on some blocks how many ports-->
  <P Name="Port">!portNr</P>   <!--Used for ports, the port number in model form 1 -->
  <P Name="Commented">!commented</P>
  <P Name="ZOrder">!zorder</P>
  <P Name="Value">!value(text)</P>
  <P Name="Gain">!gain(text)</P>   <!--only for Gain block -->
  <P Name="OutDataTypeStr">!outType(text)</P>
```

Der Schlüssel hierfür ist `<P Name="!CHECK"/>`. Der Inhalt des Attributes, in dessen Wert im Config.xml `!CHECK` steht, wird als Erweiterung zum Tagnamen des Elements benutzt, um die entsprechende Speicherklasse zu ermitteln. Im Beispiel wird das Element `<P Name="Value">2</P>` erkannt. Aufgrund

```
  <P Name="Value">!value(text)</P>
```

wird dann der Text des Elementes, hier 2, als Argument der Operation `value(String text)` übergeben. und somit gespeichert. An dieser Stelle hätte auch eine Variable (ein Field) stehen können.

3. Automatische Generierung des config.xml aus gegebenen XML-Daten

Für die Generierung eines ersten oder zu vergleichenden config.xml werden reale XML-Daten verwendet. Diese sollten möglichst vielfältig und ausführlich sein, damit alle Datenkonstellationen erfasst werden.

Die andere Möglichkeit wäre, ein Schema der XML-Files heranzuziehen. Eine XML-Schema-File enthält die genaue Vorschrift für den Aufbau der Anwender-XML-Daten, es wäre also insoweit korrekter, auf den vorhandenen Schema-File aufzubauen. In der Praxis ist aber ein XML-Schema oft nicht vorhanden oder in seiner Ausführung nochmals komplexer (da alle theoretischen Fälle berücksichtigt werden) als in der Praxis nötig.

Daher wird von XML-Files als Repräsentanten von Beispielen ausgegangen.

3.1 Wie funktioniert die automatische Generierung

Das XML-File wird mit allen Elementen eingelesen. Der Text-Inhalt von Elementen und Attributen wird nicht beachtet. Es wird ermittelt, ob ein Element-tag in der gegebenen Node-Struktur bereits vorhanden ist, jedes Element-tag wird nur einmal aufgeführt. Dabei wird festgestellt, ob das Element-Tag nur einmal oder mehrfach vorkommt. Das wird im Attribute `xmlinput:list=""` vermerkt und in der Datenspeicherung verwendet.

Wird ein gleiches Element-Tag in mehreren Node-Ebenen verwendet, dann muss entschieden werden, ob es sich um den tatsächlich gleichen Element-Typ handelt, der insbesondere rekursiv auftritt, oder ob die Tagbezeichnung nur zufällig gleich ist. Grundsätzlich ist bei gleicher Tag-Bezeichnung in einer anderen Node-Struktur **nicht** davon auszugehen, dass es sich um den selben Element-Typ handelt. In der Praxis ist es aber oft so, dass in verschiedenen Node-Ebenen die adäquate (vergleichbare) Information mit dem selben Tagnamen gespeichert wird und faktisch einem selben Typ zugeordnet werden kann. Bei der Datenspeicherung kann und sollte dann die selbe class-Definition verwendet werden.

Die Entscheidung, ob ein Element in anderer Strukturebene mit der selben Tagbezeichnung der selben class bzw. im config.xml dem selben xmlinput:subtree zugeordnet wird, erfolgt aus dem Vergleich der Attribute und Sub-Nodes der ersten Stufe. Die Entscheidung basiert allerdings auf einem score, ist nicht eindeutig feststellbar. Es kann sein, dass der Anwender, unter Kenntnis der Datenstuktur und ggf. des XML-Schema, in der config.xml noch korrigierend eingreifen sollte.

Gleiche tag, den verschiedene Typisierungen zuerkannt werden, werden in der

```
<xmlinput:subtree xmlinput:name="tag_A" ...
```

mit suffix _A usw. bezeichnet, dies ist vom Anwender anpassbar. Auffällig ist dies in der Aufrufzeile:

```
<tag xmlinput:list=""  xmlinput:data="!new_tag()"  xmlinput:subtree="tag_A" />
```

Es unterscheiden sich *tag* und der subtree-Name.

Die Namen der Set- und new-Routinen weden entsprechend dem Tagnamen als default vergeben.

3.2 Aufruf

Mit einem gegebenen config.xml-File kann der Aufruf wie folgt als cmdline erfolgen (als eine Zeile):

```
java  -cp ../../exe/zbnf.jar
  org.vishia.xmlReader.XmlJzCfgAnalyzer
  path/to/input.xml
  path/to/config.xml
```

Es wird empfohlen, den config.xml aus verschiedenen input.xml zu generieren, diese zu vergleichen und manuell zu mergen. Wenn ein xml-File weniger Strukturelemente enthält, dann sollte er eine Teilmenge erzeugen, die direkt über ein textulles diff-view erkennbar ist.

3.3 Pflege des config.xml

Der config.xml wird durch diese Aktion nur einmalig erzeugt. Er kann und sollte danach angepasst werden, insbesondere um nicht notwendige XML-Strukturelemente auszuschließen.

Eine nachfolgende Genereirung, beispielsweise mit einem anderen umfänglicheren XML-File kann dann per Merge eingepflegt werden.

Datenelemente, die Ablage- und Create-Operations, sollten wegen der Mergefähigkeit nicht extrem geändert werden.

4. Automatische Generierung der Datenablage-Java-Files aus dem config.xml

Die Datenablage muss zum config.xml passen. Grundsätzlich kann dies manuell erfolgen. Günstiger ist jedoch, direkt die Java-Files für die Datenablage direkt aus dem cfg.xml zu erstellen.

4.1 Aufruf

Mit einem gegebenen config.xml-File kann der Aufruf wie folgt als cmdline erfolgen (als eine Zeile):

```
java  -cp ../../exe/zbnf.jar
  org.vishia.xmlReader.GenXmlCfgJavaData
  -cfg:myUser.cfg.xml
  -dirJava:../srcJava_Path
  -pkg:pck.path.for.genClass
  -class:MyUser
```

Die Argumente können auch in einem Textfile stehen, dass mit

```
java  -cp ../../exe/zbnf.jar org.vishia.xmlReader.GenXmlCfgJavaData --@argfile.arg
```

aufgerufen werden kann. Das ist übersichtlicher.

* `-cfg:` benennt den Filepath zum cfg-file

* `-dirJava:` benennt den Filepath zum Verzeichnis, in dem mit dem Packagepath die Java-Sources geschrieben werden

* `-pkg:` benennt den package-Path und bestimmt damit den Speicherort des Files innerhalb dirJava

* `-class:` Benennt den Classname der Datenklasse. Es werden zwei classes erzeugt. Die weitere `MyUser_Zbnf` ist die abgeleitete Klasse, die zum Einschreiben der Daten vom XmlJzReader benutzt wird. Sie enthält nur Operationen, keine Daten.

Es wird empfohlen, die generierte Datenclasses nicht nachzubearbeiten. Einzig die Gestaltung der toString-Operations fehlt derzeit noch (2019-08-18), diese muss dort hinein. Dies als TODO und alle anderen Dinge können im config.xml bestimmt werden. Die Datenclass hat noch keine Beziehung zur Verarbeitung, die in jedem Fall ein andererm manueller Programmierschritt ist. Die Nachverarbeitung verwendet lediglich die Daten, sie wird nicht dort gespeichert.

Folglich ist der config.xml-File die Quelle, wie die Daten in der Ablage strukturiert sind. Wird diese Quelle geändert, dann kann entweder ohne Änderung der Datenablage eine Anpassung an eine andere XML-Stuktur erfolgen, oder es wird die Datenablage in Details mit geändert. Die Weiterverarbeitung muss dann entsprechend angepasst werden (Refactoring). Wobei Syntaxfehleranzeigen in Java sehr hilfreich sind und das Refactoring erleichtern.

Die Bezeichnung `_Zbnf` für die class zum Einschreiben rührt daher, dass dieses Tool zuerst für den Zbnf-Parser geschrieben wurde, auch dort kann vom Syntaxfile direkt die datenablage abgeleitet werden. TODO allgemeingültige Bezeichnung, Stand 2019-08-18.

5. XmlJzReader

5.1 Aufruf

Der XmlJzReader ist nur sinnvoll zu verwenden in einer Java-Umgebung, die danach die Daten verarbeitet. Daher ist ein Aufruf nur im Java-Kontext vorgesehen:

```
import java.io.File;

import org.vishia.xmlReader.XmlJzReader;
import org.vishia.xmlReader.XmlCfg;

public class MyClass {

  XmlJzReader xmlReader = new XmlJzReader();



  private void readXml() {
    XmlCfg cfg = xmlReader.readCfg(new File(pathToCfg));
    if(cfg !=null) { //else: Error in config
      MyUserXmlData dst = new MyUserXmlData_Zbnf(); //create derived class
      File fileToRead = new File(pathToInputXml);
      String error = xmlReader.readXml(fileToRead, dst);
      if(error !=null) {
        System.err.println(error);
      }
      .... evaluate the dst file
    }
  }
```

Das Template sollte das Wesentliche zeigen. Die class MyUserXmlData ist diejenige, die mit Kapitel 4 Automatische Generierung der Datenablage-Java-Files aus dem config.xml, Seite 4 generiert wurde.

Einige Fehlermeldungen werden direkt in `System.err` geschrieben und gelangen auf die Console-Ausgabe. TODO nicht verwenden, keine Exception, entweder Log-Ausgabe oder error-String-Rückgabe. Es ist günstiger keine Exception zu erzeugen wenn eine Speicheroperation nicht aufgefunden wurde. Eine eindeutige Bezeichnung der fehlenden Operation wird angegeben, mit class und Argumenten. Exception nur bei Fehlern im XML-File, dieser muss well-formed sein.

5.2 Verarbeitung von Texten im XML

Selbstverständlich werden die Umschreibungssequenzen `&`, `<` usw. sowie die Erkennung von Spezialzeichen `
` für Linefeed usw. und alle UTF16-Character mit `` richtig übertragen.

Ein Zeilenumbruch im Text wird mit nachfolgenden Indent-Leerzeichen als ein Leerzeichen gelesen. Das ist XML-konform. Damit werden beautificated-XML-Files auch im Textbereich richtig eingelesen.

Das Encoding des XML-Inputfiles wird selbstverständlich beachtet.

5.3 Abhängigkeiten, Files

Der XmlJzReader benötigt Files außerhalb des Java-Standardlibraries nur in der Komponente srcJava_Zbnf, dort auch nur aus den Packages

* `org/vishia/util/Assert.java`
* `org/vishia/util/CalculatorExp.java` abhängig wegen DataAccess.java
* `org/vishia/util/DataAccess.java` Grundalgorithmen für Nutzung Reflectionzugriff
* `org/vishia/util/FileSystem.java`
* `org/vishia/util/FilepathFilter.java`
* `org/vishia/util/IndexMultiTable.java` kann mit java.lang.TreeMap ersetzt werden
* `org/vishia/util/StringFunctions.java`
* `org/vishia/util/StringPartScan.java` Grundfunktionen für Text parsen
* `org/vishia/util/TreeNodeBase.java` Xml Nodes abbilden
* `org/vishia/msgDispatch/LogMessage.java` allgemein nützlich, aber hier wegen `IndexMultiTable.java` indirekt abhängig
* `org/vishia/util/bridgeC/*.java` Allgemeines aus Java2C resultierende Basic functionality
* `org/vishia/xmlReader/*.java` Die wenigen Files dieses packages

Das ist eine überschaubare Anzahl von abhängigen Files.

Diese Darstellung ist deshalb wichtig, weil allgemein bei Software selbst bei kleinen Funktionalitäten häufig allgemeine und umfangreiche Libraries eingezogen werden, deren Quellen vermaschte Abhängigkeiten haben. Software wird dadurch immer unnötig groß. Ein allgemein anerkanntes großes Basispaket wie die gesamte JRE mit einigen zig MByte ist akzeptabel, wobei es für kleine Plattformen entsprechende Anpassungen gibt. Zusätzliche große Fremdlibraries verkomplizieren die Lage aber nochmals. Für dieses package kann man die Nutzung des `IndexMultiTable.java` noch abstellen, denn diese class zieht `org/vishia/bridgeC/AllocInBlock.java` ein, weil es für die Übersetzung Java2C insoweit vorbereitet ist, hier aber nicht notwendig. Es gilt, unnötige Abhängigkeiten zu vermeiden.

Der jar-File aus mit Vollständiger Compilation aller Abhängigkeiten für XmlJzReader umfasst 211 kByte. Der gesamte zbnf.jar, der alle Files der Komponente enthält (umfasst auch den ZBNF-Parser), hat 1.1 MByte.

*****