Java-Implementierung eines Korpus-Konverters für die Formate TigerXML, FrameNetXML und CoNLL

Hausarbeit im Rahmen des Seminars

„Korpora natürlicher Sprache“

an der Universität Trier *im* Sommersemester 2010 *am Lehrstuhl für Linguistische Datenverarbeitung*

Leitung: Dr. Sven Naumann

Betreuung: Dipl.-Ling. Andrei Beliankou

Verfasser: Sven Oos
(Matrikelnummer 842655)

Verfasst am: 04.07.2012

Inhaltsverzeichnis

Einleitung

In der Korpuslinguistik finden sich verschiedene Formate, in denen Korpora abgespeichert und vor allem annotiert werden können. Ein allgemeines und verbreitetes Problem bei unterschiedlichen Formaten verschiedener Datenhaltungs- und Informationssystemen ist die eingeschränkte Menge an Daten, auf die man mit einem speziellen Werkzeug (z.B. zur Analyse) Zugriff hat, wenn dieses nur eines der möglichen Formate beherrscht. In diesen Fällen sind Konverter interessant, die Datenbestände von einem Format in ein anderes überführen.

Der hier beschriebene Konverter arbeitet mit den in der Korpuslinguistik verbreiteten Formaten *TigerXML* des Projektes TIGER, *FrameNetXML* des Projektes FrameNet und *CoNLL*.

Diese Arbeit befasst sich mit den technischen Merkmalen der o.g. Formate. Auf die wissenschaftliche Theorie, wie z.B. Frame-Sematik, wird nicht näher eingegangen.

Aufgabenstellung

Aufgabe war es, einen Konvertierer zu schreiben, der die oben genannten Korpusformate in den folgend abgebildeten Richtungen konvertieren kann:

Formate

Die betrachteten Formate werden hier nur grob und auf die für die Architektur des Konverters wichtigsten Merkmale reduziert vorgestellt.

TigerXML

Wie schon aus dem Namen ersichtlich, handelt es sich bei TigerXML um eine Annotation mittels Extensible Markup Language (XML). Eine Korpusdatei besteht hierbei aus einem Kopf (`<head>`) und einem Rumpf (`<body>`), die von einem `corpus`-Tag eingefasst sind. Der Kopf beinhaltet Metainformationen zum Autor und Korpus selbst, z.B. Erstellungsdatum oder Format. Weiterhin werden hierin alle zur Annotierung verwendeten Merkmale (`<feature>`) aufgelistet. Im Rumpf werden Sätze als Syntaxbäume beschrieben, wobei die Wörter, oder auch Terminale (`<t>`), zusätzliche Merkmale als XML-Attribute erhalten können. Die Nichtterminale (`<nt>`) spannen den Syntaxbaum mithilfe von Kanten (`<edge>`) auf, indem diese über ein `idref`-Attribut auf das `id`-Attribut ihres Zielknotens verweisen. Die Kanten selbst sind dabei eingefasste Elemente des hierarchisch übergeordneten Knotens. Kanten können des Weiteren in einem `label`-Attribut zusätzliche Metainformation aufnehmen.

FrameNet

Bei FrameNet werden Korpora ebenfalls durch XML-Dateien repräsentiert. Anders als bei TigerXML werden Sätze hier nicht in Terminale für einen Syntaxbaum aufgeteilt, sondern als Ganzes abgelegt (`<text>`). Dieses `text`-Tag bildet zusammen mit dazugehörigen Annotierungen (`<annotationSet>`) einen Satz (`<sentence>`). Die Annotierung erfolgt mittels eines Labels (`<label>`), welches ein `name`-Attribut zur Beschreibung enthält. Zusätzliche besitzt es noch ein `start`- und ein `end`-Tag, mit denen der betreffende Bereich im Text mittels Zeichenindex angegeben wird. Labels, die thematisch zusammengehören, z.B. die Beschreibung der grammatikalischen Funktion eines Satzteils, werden in Ebenen (`<layer>`) zusammengefasst.

CoNLL

Korpora im CoNLL-Format sind Tabellen in Klartextdateien. Dabei wird steht jede Zeile für ein Wort. Die Tabelle besteht aus bis zu zehn Spalten, die durch Tabstops getrennt werden. Die erste Spalte hält einen Index fortlaufender Nummern, die zweit das Wort selbst. In den Spalten drei bis zehn können Annotationen, beispielsweise Part-Of-Speech in Spalte vier, für das entsprechende Wort hinterlegt werden. Sätze werden durch eine Leerzeile voneinander getrennt.

Architektur

Modell

Für ein unkompliziertes Arbeiten mit XML-Dateien in Java bietet dieses die Programmierschnittstelle Java Architecture for XML Binding (JAXB). Mit ihr werden XML-Dateien als Instanzen von Java-Klassen in den Arbeitsspeicher geladen, welche auf dem zugrundeliegenden XML-Schema basieren. Dabei wird jedes XML-Tag durch eine eigene Klasse repräsentiert und Attribute eines Tags als Felder in der entsprechenden Klasse. Ebenfalls als Feld realisiert werden Tag-Schachtelungen. JAXB liefert eine Kommandozeilenanwendung, den Binding Compiler (xjc)[1], mit, die zu einem XML-Schema (DTD oder XSD) die passenden Java-Klassen generiert. Dies wurde hier für TigerXML[2] und FrameNet[3] durchgeführt, wobei die resultierenden Klassen unter den Java-Paketen `tigerxml` bzw. `framenet.corpus` zusammengefasst wurden.

Ähnlich wurde das CoNLL-Modell umgesetzt. Es existieren vier Klassen `Corpus`, `Sentence`, `Token` und `Field`[4].

Mapping

Um mit diesem Modell arbeiten zu können, enthält jedes Package einen domänenspezifischen Mapper, der das Laden und Speichern übernimmt:

- `conll.ConllMapper`
- `framenet.FrameNetMapper`
- `tigerxml.TigerXmlMapper`

Alle werden über folgende statische Schnittstelle genutzt, wobei `Corpus` die Klasse des gleichen Packages bezeichnet, in dem sich der jeweilige Mapper befindet:

- `public static Corpus load(String path)`
- `public static void save(Corpus corpus, String path)`

Laden

Die `load`-Methode liefert jeweils ein `Corpus`-Objekt, das entsprechend dem zugrundeliegenden Modell den Inhalt der mit `path` angegebenen Korpus-Datei enthält.

---

[1] http://docs.oracle.com/javase/6/docs/technotes/tools/share/xjc.html

[2] http://www.ims.uni-stuttgart.de/projekte/TIGER/public/TigerXML.xsd, (Version vom 01.04.2003)
http://www.ims.uni-stuttgart.de/projekte/TIGER/public/TigerXMLHeader.xsd, (Version vom 01.04.2003)
http://www.ims.uni-stuttgart.de/projekte/TIGER/public/TigerXMLSubcorpus.xsd (Version vom 01.04.2003)

[3] corpusV1_1.xsd

[4] *Anhang: CoNLL-Modell-Strukturdiagramm*

Der CoNLL-Mapper parst die Korpus-Datei und setzt dabei das Java-Pendant stückweise zusammen, indem er pro neuem Absatz ein `Sentence`-Objekt erzeugt, welchem er pro Zeile ein neues `Token`-Objekt hinzufügt, usw.

Die TigerXML- und FrameNet-Mapper übergeben die Aufgabe an den JAXB-Unmarshaller. Dieser verwendet zur Erzeugung der Java-Objekte Fabrikmethoden[5]. Diese sind gesammelt in der Klasse `ObjectFactory`, die vom Binding Compiler während der Erzeugung der Klassen aus dem XML-Schema mit generiert wurde.

Speichern

Will man ein `Corpus`-Objekt aus dem Arbeitsspeicher persistieren, übergibt man es zusammen mit dem Zielpfad an die `save`-Methode des zugehörigen Mappers.

Der CoNLL-Mapper überträgt die Formatierung in das Dateiformat an die Modellklassen selbst und erhält ein fertig formatiertes `String`-Objekt, das er mithilfe eines `FileOutputStreams`[6] in eine Datei schreibt. Die beiden anderen Mapper bedienen sich des JAXB-Marshallers, analog zum Laden.

Der TigerXML-Mapper prüft noch, ob mehr als ein Satz im Korpus enthalten ist. Ist dies der Fall, so erzwingt er ein Abspeichern des `body`-Inhalts in einer separaten XML-Datei `subcorpus.xml`.

Konvertieren

Ähnlich den Mappern existieren auch zu jedem Format eine eigene Konverter-Klasse, die jedoch alle in einem eigenem Package zusammengefasst sind:

- `converter.ConllConverter`
- `converter.FrameNetConverter`
- `converter.TigerXmlConverter`

Prinzip

Jeder Konverter kann nur Korpora desjenigen Formats in ein anderes übertragen, dessen Domäne er selbst entspringt — erkennbar an dem Präfix des Klassennamens. Dazu instanziiert man ein Objekt seiner Klasse, indem man dem Konstruktor eine Referenz auf das zu konvertierende `Corpus`-Objekt übergibt. Für jedes verfügbare Zielformat hält die Konverter-Klasse nun eine `to`*`Zielformat`*-Methode bereit, die ein entsprechendes Corpus-Objekt als Rückgabewert liefert.

---

[5] Gang of Four: *Factory Method*

[6] http://docs.oracle.com/javase/6/docs/api/java/io/FileOutputStream.html

		nach		
		CoNLL	**TigerXML**	**FrameNet**
	CoNLL		`toTigerXML()`	
von	**TigerXML**	`toConll()`		`toFrameNet()`
	FrameNet		`toTigerXML()`	

Implementierung

Im Detail wird bei der Konstruktion eines Konverter-Objekts dessen innerer Zustand mit dem des erhaltenen `Corpus`-Objekts initialisiert. Die `tigerXmlConverter`-Klasse hält dafür ein Feld bereit, dem das Quellobjekt zugewiesen wird. Die beiden anderen Konverter-Klassen stellen selbst Erweiterungen ihrer jeweiligen `Corpus`-Klassen dar[7]. Dadurch überführen sie sich bei einer Konvertierung sozusagen selbst in das Zielformat.

CoNLL nach TigerXML

Da in CoNLL keine Syntaxstrukturen definiert sind, werden für den TigerXML-Satz alle Token als Terminale an ein Nichtterminal gehängt, welches ein direktes Kind eines Wurzelknotens ist. Die Nichtterminale erhalten das `POS`-Feld als `pos`-Attribut. Die Terminale erhalten das `FORM`-Feld als `word`-Attribut. Alle im `FEATS`-Feld enthaltenen Werte werden als Terminal-Feature-Attribute angefügt. Sollte eines ein Gleichheitszeichen enthalten, wird der linke Teil dieser Definition als Terminal-Feature-Name angesehen, der rechte Teil als -Wert. Alle Features, die bei einem Terminal nicht belegt sind, werden mit einem Doppelminus (`--`) besetzt[8].

TigerXML nach CoNLL

Hier wird für jedes Terminal ein Token erzeugt. Dabei werden Features der Terminale und der direkt übergeordneten Nichtterminale nach ihrem Namen auf die Token-Felder nach folgendem Muster verteilt:

Feature-Name	Feld
`word`	`FORM`
`lemma`	`LEMMA`
`pos`	`CPOSTAG`
sonstige	`FEATS`

[7] Siehe *Anhang: Konverter-Strukturdiagramm*

[8] Entsprechend der Empfehlung: *TIGERSearch 2.1 User's Manual, Chapter V.2.3*

Im letzten Fall wird ein String dem `FEATS`-Feld hinzugefügt, der aus dem Attribut-Namen, einem Gleichheitszeichen und dem Attribut-Wert gebildet wird. Mehrere Strings in diesem Feld werden durch ein Pipe-Symbol (`|`) voneinander getrennt.

TigerXML nach FrameNet

Das FrameNet-Schema besitzt eine größere Anzahl an Entitäten, als TigerXML, von denen die meisten ein obligatorisches `id`-Attribut besitzen. Bei der Konvertierung nach FrameNet werden daher all jene IDs neu erstellt, zu denen es kein TigerXML-Pendant gibt, z.B. `Annotation`. Dies geschieht nach folgendem Muster:

Element	id-Attribut-Wert
`Document`	*d1*
`Paragraph`	*p1*
`AnnotationSet`	*as_**
`Layer`	*lr_**
`Label`	*lb_**

Die ersten beiden Elemente werden nach der Konvertierung nur einmal existieren und daher fest auf die angegebenen Werte gesetzt. Von den Anderen können mehrere vorkommen, weshalb die Tabellenwerte durch eine fortlaufende Nummer erweitert werden, daher auch das Sternchen. Zusätzlich „erben“ die tieferliegenden Elemente den Pfad ihrer Vorfahren, d.h. alle `AnnotationSet`s des dritten Satzes besitzen die Form `as_3_*`, das zweite Layer-Element (des ersten `AnnotationSet`s des dritten Satzes) erhält die ID `lr_3_1_2`, usw. Die Sätze selbst erhalten ihre ID aus dem TigerXML-`sentence`-Element.

Eine Konkatenation der Terminale, durch Leerzeichen getrennt, ergibt den Wert des `text`-Elements. Weiterhin wird folgendes Zuordnungsschema angewandt[9]:

[9] Als Beispiel siehe *Anhang: Konvertierungsbeispiel von TigerXML nach FrameNet*

TigerXML		FrameNet		
Element	**Attribut**	**Layer**	**Label**	
			start	**end**
`edge`	`label`	`GF`	*erstes Zeichen unterhalb von* `edge`	*letztes Zeichen unterhalb von* `edge`
`nt`	*verwendetes Feature*	`PT`	*erstes Zeichen unterhalb von* `nt`	*letztes Zeichen unterhalb von* `nt`
`t`	*für jedes verwendete Feature (außer* `word`*)*	`Other`	*Startindex von* `t` *in* `text`	*Endindex von* `t` *in* `text`

FrameNet nach TigerXML

Diese Konvertierungsrichtung stellt das genaue Gegenteil des vorangehenden Abschnitts dar. Zum Aufbau des Syntaxbaums bedient sich der Konverter einer Hilfsklasse, der `FrameNetComponentsCollection`. Diese verwaltet alle Wörter und Labels in TreeMaps[10]. Dafür werden diese in eine Wrapperklasse[11] verpackt, die das Interface `FrameNetComponent` implementiert. Dieses stellt folgende Methoden bereit:

- `Object getObject()`
- `int getStart()`
- `int getEnd()`
- `int getKey()`

Je nach Implementierung dieses Interfaces (`FrameNetWordComponent` oder `FrameNetLabelComponent`) liefert `getObject()` das adaptierte Wort (`String`) bzw. `Label`. Die beiden Methoden `getStart()` und `getEnd()` liefern bei Wörtern den Index des ersten bzw. letzen Buchstabens und bei `Label`s die entsprechenden Attribute. Um die Objekte in die `TreeMap` einzusortieren, wird eine Comparator[12]-Implementierung (`FrameNetComponentComparator`) verwendet, die sich der `getKey()`-Methode bedient. Da der Comparator nur zwei Werte miteinander vergleichen kann, liefert die Methode `getKey()` einen Integer-Wert, der sich mithilfe der Cantorschen Paarungsfunktion[13] aus den Attributen `start` und `end` zusammensetzt.

---

[10] http://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html

[11] Gang of Four: *Adapter*

[12] http://docs.oracle.com/javase/6/docs/api/java/util/Comparator.html

[13] http://de.wikipedia.org/wiki/Cantorsche_Paarungsfunktion (Stand: 01.02.2012)

Applikation

Die API der bereitgestellten Klassen kann natürlich innerhalb eigener Programme verwendet werden[14]. Zusätzlich wird im `converter`-Package das ausführbare Archiv `ConverterApp.jar` mitgeliefert, das die Ausführung auf Konsolenebene ermöglicht. Dies geschieht auf folgende Art:

```
# java -jar ConverterApp.jar \
    -<Quellformat> <Quellpfad> \
    -<Zielformat> <Zielpfad> [Korpusname]
```

Dabei kann für das obligatorische Format aus den Kürzeln `cn` (CoNLL), `tg` (TigerXML) und `fn` (FrameNet) gewählt werden. Die Angabe eines Korpusnamen für das Ziel ist nur für FrameNet nötig. So sieht z.B. ein Befehl zur Konvertierung eines TigerXML-Korpus in das CoNLL-Format folgendermaßen aus:

```
# java -jar ConverterApp.jar -tg corpus.xml -cn corpus.conll
```

Will der Anwender ein Korpus von CoNLL nach FrameNet konvertieren, oder umgekehrt, so geschieht das automatisch über den Zwischenschritt des TigerXML-Formats. Es können also direkt die Parameter `cn` und `fn` in einer Anweisung verwendet werden.

[14] Siehe *Anhang: Sample.java*

Abschließende Betrachtung

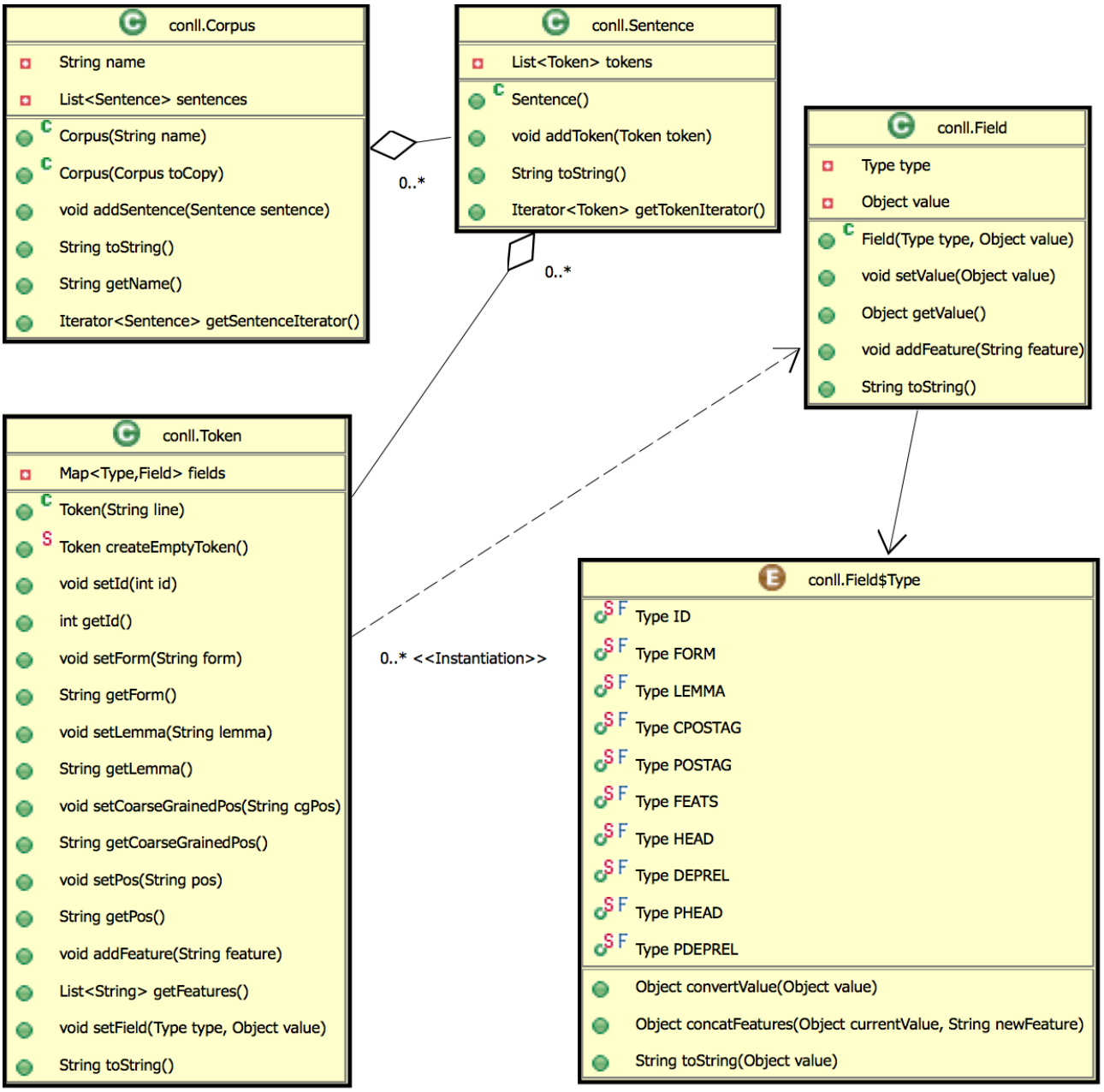
Zuletzt soll noch einmal darauf hingewiesen werden, dass jede Konvertierung zwischen unterschiedlichen Formaten verlustbehaftet ist. Die Entwickler der jeweiligen Formate haben sich auf besondere Merkmale spezialisiert, die nicht formatübergreifend gleich gut abgedeckt sind. Oftmals entsteht ein neues Format gerade aus der Problematik, dass eine gewünschte Eigenschaft von keinem bis dahin vorhandenen Format erfüllt wird.

Die bei dieser Arbeit entstandenen Klassen erleichtern Programmierern dafür das Arbeiten mit Korpora innerhalb eigener Java-Anwendungen aufgrund der entstandenen Objektorientierung, sodass mögliche Verluste während des Konvertierens durchaus in Kauf genommen werden können.

Quellen

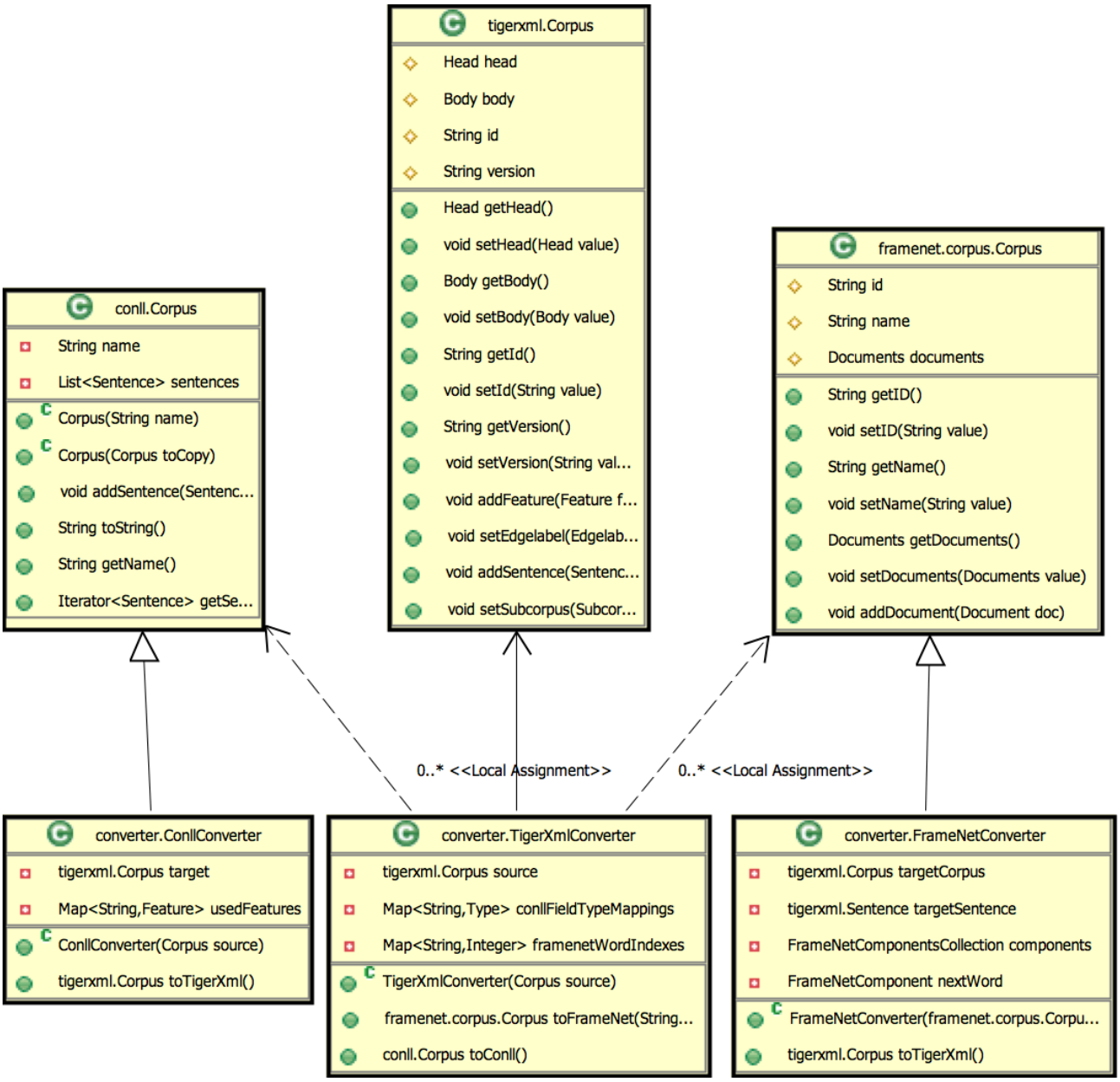
CoNLL:	http://ilk.uvt.nl/conll/index.html/
Eclipse:	http://www.eclipse.org/
FrameNet:	http://framenet.icsi.berkeley.edu/
Gang of Four:	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995): *Design Patterns*, Addison Wesley
Green UML:	http://green.sourceforge.net/
Java-API:	http://docs.oracle.com/
JAXB:	http://jaxb.java.net/
TIGER:	www.ims.uni-stuttgart.de/projekte/TIGER/
	König, Lezius, Voormann (2003): *TIGERSearch 2.1 User's Manual*
Wikipedia:	http://de.wikipedia.org

Anhang A: CoNLL-Modell-Strukturdiagramm[15]



[15] Erstellt mit *Green UML 3.5.0* in *Eclipse 3.7.2*

Anhang B: Konverter-Strukturdiagramm[16]

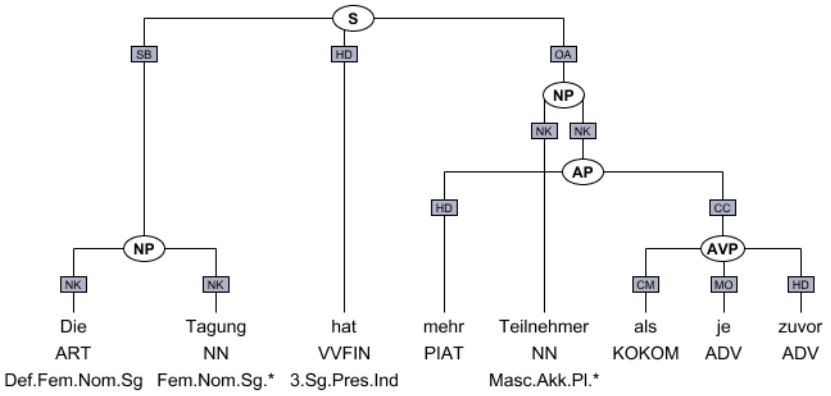


[16] Erstellt mit *Green UML 3.5.0* in *Eclipse 3.7.2*

Anhang C:

Konvertierungsbeispiel von TigerXML nach FrameNet

TigerXML-Diagramm[17]



FrameNet-Ausgabe

```
[...]
<sentence ID="s5">
  <text>Die Tagung hat mehr Teilnehmer als je zuvor</text>
  <annotationSets>
    <annotationSet status="" ID="as1_1">
      <layers>
        <layer name="GF" ID="lr1_1_1">
          <labels>
            <label end="2" start="0" name="NK" ID="lb1_1_1_1"/>
            <label end="9" start="4" name="NK" ID="lb1_1_1_2"/>
            <label end="33" start="31" name="CM" ID="lb1_1_1_3"/>
            <label end="36" start="35" name="MO" ID="lb1_1_1_4"/>
            <label end="42" start="38" name="HD" ID="lb1_1_1_5"/>
            <label end="18" start="15" name="HD" ID="lb1_1_1_6"/>
            <label end="42" start="31" name="CC" ID="lb1_1_1_7"/>
            <label end="42" start="15" name="NK" ID="lb1_1_1_8"/>
            <label end="29" start="20" name="NK" ID="lb1_1_1_9"/>
            <label end="9" start="0" name="SB" ID="lb1_1_1_10"/>
            <label end="13" start="11" name="HD" ID="lb1_1_1_11"/>
            <label end="42" start="15" name="OA" ID="lb1_1_1_12"/>
          </labels>
        </layer>
        <layer name="Other" ID="lr1_1_2">
          <labels>
            <label end="2" start="0" name="ART" ID="lb1_1_2_1"/>
            <label end="9" start="4" name="NN" ID="lb1_1_2_2"/>
```

---

[17] Entnommen aus *TIGERSearch 2.1 User's Manual: Chapter V.2.3*

```
            <label end="13" start="11" name="VVFIN" ID="lb1_1_2_3"/>
            <label end="18" start="15" name="PIAT" ID="lb1_1_2_4"/>
            <label end="29" start="20" name="NN" ID="lb1_1_2_5"/>
            <label end="33" start="31" name="KOKOM" ID="lb1_1_2_6"/>
            <label end="36" start="35" name="ADV" ID="lb1_1_2_7"/>
            <label end="42" start="38" name="ADV" ID="lb1_1_2_8"/>
          </labels>
        </layer>
        <layer name="PT" ID="lr1_2_1">
          <labels>
            <label end="9" start="0" name="NP" ID="lb1_2_1_1"/>
            <label end="42" start="31" name="AVP" ID="lb1_2_1_2"/>
            <label end="42" start="15" name="AP" ID="lb1_2_1_3"/>
            <label end="42" start="15" name="NP" ID="lb1_2_1_4"/>
            <label end="42" start="0" name="S" ID="lb1_2_1_5"/>
          </labels>
        </layer>
      </layers>
    </annotationSet>
    <annotationSet status="" ID="as1_2">
      <layers>
        <layer name="Other" ID="lr1_2_1">
          <labels>
            <label end="2" start="0" name="Def.Fem.Nom.Sg" ID="lb1_2_1_1"/>
            <label end="9" start="4" name="Fem.Nom.Sg.*" ID="lb1_2_1_2"/>
            <label end="13" start="11" name="3.Sg.Pres.Ind" ID="lb1_2_1_3"/>
            <label end="18" start="15" name="--" ID="lb1_2_1_4"/>
            <label end="29" start="20" name="Masc.Akk.Pl.*" ID="lb1_2_1_5"/>
            <label end="33" start="31" name="--" ID="lb1_2_1_6"/>
            <label end="36" start="35" name="--" ID="lb1_2_1_7"/>
            <label end="42" start="38" name="--" ID="lb1_2_1_8"/>
          </labels>
        </layer>
      </layers>
    </annotationSet>
  </annotationSets>
</sentence>
[...]
```

Anhang D: Verwendungsbeispiel der Bibliothek

Quelltext aus Datei `converter.Sample.java`:

```
package converter;

import java.io.IOException;
import javax.xml.bind.JAXBException;

public class Sample {

  public void convertFromTigerXmlToFramenet() {
    // Step 1: Create a Java Object from a corpus file.
    tigerxml.Corpus tigerCorpus = null;
    try {
      tigerxml.TigerXmlMapper.load("<Path to TigerXML corpus file>");
    } catch (IOException e) {
      e.printStackTrace();
    } catch (JAXBException e) {
      e.printStackTrace();
    }
    //  You may make some changes to the corpus object here.

    // Step 2: Create a converter from the source corpus object.
    converter.TigerXmlConverter converter = new converter.TigerXmlConverter(
                                                    tigerCorpus);

    // Step 3: Export it to the source format (FrameNet here).
    framenet.corpus.Corpus framenetCorpus = converter.toFrameNet(
                                              "<corpus name>");
    //  Additional changes to the FrameNet corpus object can be made.

    // Step 4: Save the object to a file.
    try {
      framenet.FrameNetMapper.save(framenetCorpus,
                                   "<Path to FrameNet corpus file>");
    } catch (IOException e) {
      e.printStackTrace();
    } catch (JAXBException e) {
      e.printStackTrace();
    }
  }
}
```