ESPRITSYSTEMS

Schreiben und Parsen von NDF-Dateien

Programmierung mit dem Neutral Data Format (NDF)

Rainer Büsch
2014-03

NDF Format-Version 1.6
Esprit Application Suite Version 9.6a

Inhaltsverzeichnis

1 Einführung

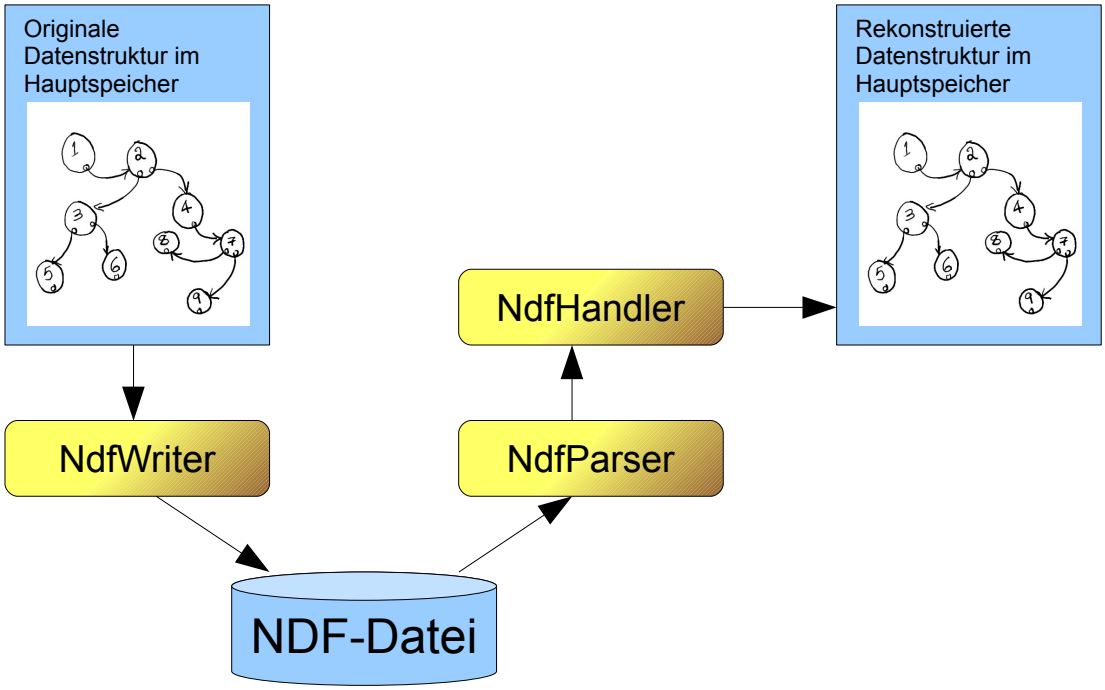
Das *Neutral Data Format* (NDF) ist ein Datenformat zur Speicherung und Langzeitarchivierung von beliebigen Informationen. Die Daten werden in einer hierarchischen Struktur gespeichert, die für Mensch und Computer leicht lesbar ist. Im Gegensatz zu XML ist NDF ein sehr kompaktes Format mit sehr wenig Overhead an Meta-Information.

Dieses Dokument beschreibt an Beispielen das Schreiben und Parsen von NDF-Dokumenten. Die Strukturen von NDF werden als bekannt vorausgesetzt - sie sind in folgendem Dokument beschrieben: http://esprit-systems.de/downloads/esprit/docu/EspritNdfDescriptionDE.pdf

1.1 Schreiben und Lesen von NDF

Das folgende Schaubild zeigt die grundsätzliche Arbeitsweise mit NDF-Dokumenten. Der *NdfWriter* speichert die im Hauptspeicher gehaltene Objekt-Struktur in einer NDF-Datei. Dabei werden alle im Speicher gehaltenen Daten im NDF-Format abgebildet.

Der *NdfParser* liest die NDF-Datei ein und informiert seinen *NdfHandler* über alle gefundenen Strukturen. Letzterer ist dafür verantwortlich, die eingelesenen Informationen auszuwerten und die betreffenden Objekt-Strukturen im Hauptspeicher wieder aufzubauen.



2 Schreiben von NDF-Dokumenten

Zum Schreiben von NDF-Dokumenten dient die Klasse *NdfWriter*. Für eine praktische Anwendung muss eine kundenspezifische Klasse erstellt werden, die von *NdfWriter* ableitet. Sie überschreibt die abstrakte Methode *performWrite()*, von der aus alle weiteren *writeXXX()* Methoden aufgerufen werden.

2.1 Ein einfaches NDF-Dokument

Das folgende „Hello World“ Beispiel zeigt das Schreiben eines einfachen NDF-Dokumentes mit einigen Properties:

```
1  public class HelloWorldNdfWriter extends NdfWriter {
2
3    public HelloWorldNdfWriter(ApplicationContext ctx, File outFile) {
4      super(ctx, outFile);
5    }
6
7    @Override
8    protected void performWrite() throws Exception {
9      writeCommentLine("My first NDF Document");
10     openDocument();  // writes document header
11     writeProperty("myStatement", "Hello World");
12     closeDocument(); // closes the output stream
13   }
14 }
```

Zur Ausführung schreiben wir eine einfache *StartNdfWriter*-Klasse, die eine *main(...)* Methode besitzt, in der die *HelloWorldNdfWriter* Instanz ausgeführt wird:

```
1  public class StartNdfWriter {
2
3    public static void main(String[] args) throws Exception {
4      ApplicationContext ctx = AppUtil.getDefaultContext();
5      File outFile = new File("hello.ndf");
6
7      new HelloWorldNdfWriter(ctx, outFile).execute();
8    }
9  }
```

Als Ergebnis erhalten wir die Datei *hello.ndf* mit folgendem Inhalt:

`Dateiausgabe: hello.ndf`

```
# My first NDF Document
<
      ndfDocVersion = "1.6"
      ndfDocEncoding = "UTF-8"
      ndfDocCreated = "2012-11-22 06:23:17"
>
<
      myStatement = "Hello World"
>
```

Von der Methode *openDocument()* wurde der obligatorische NDF-Dokument Header geschrieben. Danach folgt die von *writeProperty(...)* explizit geschriebene Property in einem eigenen Property-Block. Schließlich wird mit *closeDocument()* die Dateiausgabe geschlossen.

➔ Beachten Sie, dass *vor* dem Aufruf von *openDocument()* ausschließlich Kommentare geschrieben werden können.

2.2 Schreiben von Properties

2.2.1 Erweiterung der Header-Properties

Die Header-Properties sind leicht um weitere Properties erweiterbar. Das folgende Beispiel zeigt, wie man eigene Properties in die Header-Sektion aufnimmt:

```
1  NdfProperties props = new NdfProperties();
2  props.put(ndfDocTypeKey, "AdressBook");
3  props.put(ndfDocAuthorKey, "Mary Poppins");
4  props.put("myStatement", "Hello World");
5
6  openDocument(props);
7  closeDocument();
```

Die Ausgabe zeigt nur noch den Block der Header-Properties. Die Properties *ndfDocTypeKey* und *ndfDocAuthorKey* sind vom System bereits fest vordefiniert. Sie werden auch beim späteren Parsen erkannt und ausgewertet.

```
<
      ndfDocVersion = "1.6"
      ndfDocEncoding = "UTF-8"
      ndfDocCreated = "2012-11-22 06:23:17"
      ndfDocType = "AdressBook"
      ndfDocAuthor = "Mary Poppins"
      myStatement = "Hello World"
>
```

2.2.2 Properties mit unterschiedlichen Datentypen

Das *NdfProperties* Objekt kann mit unterschiedlichen Werte -Objekten befüllt werden. Im Folgenden schreiben wir verschiedene Datentypen. Beachten Sie dabei insbesondere die Ausgabe von Listen-Werten wie Arrays oder Collections.

```
1  NdfProperties props = new NdfProperties();
2  props.put("integerValue", 1954);
3  props.put("doubleValue", Math.PI);
4  props.put("booleanValue", true);
5  props.put("timestampValue", new DBTimestamp());
6  props.put("dateValue", new DBDate());
7  props.put("listValue", new String[] {"ASCII", "UTF-8", "UTF-16"});
8  props.put("emptyList", new ArrayList());
9  props.put("emptyValue", "");
10 props.put("nullValue", null);
11
12 setPropertiesIndented(false);  // properties have no indentation any more
13 setPropertiesAligned(true);    // values are aligned on the equal sign
14 writeProperties(props);
```

Mit Hilfe der Methoden *setPropertiesAligned(true)* und *setPropertiesIndented(false)* wird die Ausgabe der Properties gesteuert. Sie sehen die Properties linksbündig und nach dem Gleichheitszeichen ausgerichtet. Listen-Werte werden innerhalb von [eckigen Klammern] ausgegeben. Leere Listen erscheinen als ein leerer Klammernblock []. Wie Sie sehen, gibt es auch eine klare Unterscheidung zwischen einem leeren String und einem echten *null* Wert.

Dateiausgabe: properties.ndf

```
<
integerValue    = 1954
```

```
doubleValue    = 3.141592653589793
booleanValue   = true
timestampValue = "2014-02-12 21:57:16"
dateValue      = "2014-02-12"
listValue      = ["ASCII" "UTF-8" "UTF-16"]
emptyList      = []
emptyValue     = ""
nullValue      = null
>
```

➔ Listen-Werte von Properties werden grundsätzlich in eine Zeile geschrieben und sollten deshalb nicht zu lang sein. Für große Listen ist eher eine andere Darstellung der Daten z.B. als LIST Struktur zu empfehlen.

2.2.3 Format von Dezimalzahlen

Das Format von Dezimalzahlen entspricht der Standardausgabe von Java. Es kann mit Hilfe der Methode *setDecimalFormat(DecimalFormat)* anders eingestellt werden, etwa in wissenschaftlicher Exponentendarstellung, wie das folgende Beispiel zeigt:

```
1  DecimalFormatSymbols symbols = DecimalFormatSymbols.getInstance(Locale.US);
2  DecimalFormat customFormat = new DecimalFormat("0.###E0", symbols);
3
4  setDecimalFormat(customFormat);
5  writeProperty("pi", Math.PI);
```

Führt zu folgender Ausgabe:

```
pi = 3.142E0
```

➔ Hierbei ist zu beachten, dass als Dezimaltrennzeichen unbedingt der Punkt '.' beibehalten werden MUSS, andernfalls gäbe es Probleme beim Parsen der Dezimalwerte! Deshalb wurde im obigen Beispiel die *DecimalFormatSymbols*-Instanz für die amerikanische *Locale.US* verwendet.

➔ Da NDF als ein **neutrales** Datenformat konzipiert ist, sollte auf keinen Fall eine lokalisierte Darstellung von Zahlen oder Datumswerten verwendet werden! Die Lokalisierung ist vielmehr Sache der Anwendungssoftware.

2.2.4 Kompaktere Schreibweise

Ein letztes Beispiel zeigt eine etwas kompaktere Schreibweise von Properties:

```
1  setPropertiesCompacted(true);  // writes properties in a single line
2  openPropertyList();
3  writeProperty("xPos", 100);
4  writeProperty("yPos", 150);
5  closePropertyList();
```

Hierbei öffnen wir die Properties-Sektion explizit und ersparen uns dadurch die Erzeugung eines *NdfProperties* Objekts. Durch *setPropertiesCompacted(true)* erhalten wir zudem die kompaktere Ausgabe in einer einzigen Zeile.

```
< xPos=100 yPos=150 >
```

2.2.5 Properties innerhalb von Strukturen

In den vorigen Beispielen wurden Properties stets auf Top-Ebene geschrieben. Sie können aber auch jeweils Bestandteil der im Folgenden beschriebenen Strukturen OBJECT, LIST, TABLE und ARRAY sein. Alle diese Strukturen können mit einem NdfProperties Objekt „geöffnet“ und auf diese Weise mit beliebiger Zusatzinformation versehen werden.

2.3 Schreiben von Listen

Zum Schreiben von Wertelisten dient die LIST-Struktur. Im folgenden Beispiel werden die im Esprit-Framework zur Verfügung stehenden Log-Level aufgelistet:

```
1  NdfProperties props = new NdfProperties("framework", "esprit");
2
3  openList("LogLevel", props);
4  writeListValues(LogLevel.values());
5  closeList();
```

Die Listenwerte können mit *writeListValue(...)* einzeln oder mit *writeListValues(...)* in einem Zug geschrieben werden, wie im obigen Beispiel. Die Anzahl der Werte ist beliebig. Der Datentyp muss natürlich immer der Gleiche sein. Wir erhalten folgende Ausgabe:

Dateiausgabe: list.ndf

```
@LIST LogLevel {
      < framework="esprit" >
      OFF
      FATAL
      ERROR
      WARNING
      INFO
      VERBOSE
      DEBUG
      DUMP
}
```

Für Listenwerte gilt die Syntax von Property-Werten. Ein Wert kann also auch ein Listenwert sein. Das folgende Beispiel schreibt die LogLevels als einen einzigen Listen-Wert:

```
1  openList("LogLevel");
2  writeListValue(LogLevel.values());
3  closeList();
```

Die Ausgabe ändert sich dann zu:

```
@LIST LogLevel {
      [ OFF FATAL ERROR WARNING INFO VERBOSE DEBUG DUMP ]
}
```

Diese Liste hat genau einen Wert, der aber selbst eine Liste ist.

2.4 Schreiben von Tabellen

Tabellen sind die bevorzugte Art, relationale Daten kompakt abzubilden. Im folgenden Beispiel werden einfache *Course*-Objekte in Tabellenform geschrieben:

```
1  NdfTable table = new NdfTable("Courses");
2  table.addColumns("Id", "Name", "Days", "Price");
3
4  openTable(table);
5  writeTableRow("J1", "Java Fundamentals", 5, 3300.00);
6  writeTableRow("J2", "Java Performance Tuning", 5, 4500.00);
7  closeTable();
```

Das *NdfTable* Objekt wird mit den Spaltennamen gefüllt. Damit ist auch klar, wie viele Datenwerte jeweils in einer Zeile geschrieben werden müssen. Die Anzahl der Zeilen ist beliebig. Der Datentyp muss natürlich pro Spalte immer der Gleiche sein. Wir erhalten folgende Ausgabe:

Dateiausgabe: tables.ndf

```
@TABLE Courses {
      [Id Name Days Price]
       "J1" "Java Fundamentals" 5 3300.0;
       "J2" "Java Performance Tuning" 5 4500.0;
}
```

Die Tabelle wird etwas schöner formatiert, wenn wir das *NdfTable* Objekt direkt mit den Daten befüllen. Dadurch ist schon vor dem Schreiben bekannt, wie groß der maximale Datenwert ist und die Werte können sauber ausgerichtet werden. Zusätzlich wollen wir die Zahlenwerte rechtsbündig auflisten.

```
1  NdfTable table = new NdfTable("Courses");
2  table.addColumns("Id", "Name");
3  table.addColumn("Days", Alignment.RIGHT);
4  table.addColumn("Price", Alignment.RIGHT);
5
6  table.addRow("J1", "Java Fundamentals", 5, 3300.0);
7  table.addRow("J2", "Java Performance Tuning", 5, 4500.0);
8  writeTable(table);
```

```
@TABLE Courses {
      [Id   Name                        Days  Price]
       "J1" "Java Fundamentals"            5 3300.0;
       "J2" "Java Performance Tuning"      5 4500.0;
}
```

➔ Achtung, bei dieser Variante werden sämtliche Tabellendaten erst einmal im Hauptspeicher aufgesammelt, was eine natürliche Begrenzung der Datenmasse darstellt. Sie eignet sich also nur für kleinere Datenmengen.

2.5 Schreiben von Arrays

Anders als bei einer Tabelle ist bei einem Array schon vor dem Schreiben bekannt, wie viele Elemente es hat. Daher wird bereits beim Eröffnen des Arrays seine Größe mitgegeben, wie das folgende Beispiel zeigt:

```
1  setMaxArrayValuesByLine(4);
2
3  openArray("Integers", 12);
4  for (int i = 0; i < 12; i++) {
5    writeArrayValue(i);
6  }
7  closeArray();
```

Hier werden die Array-Werte in einer *for*-Schleife einzeln geschrieben. In Zeile 1 wurde definiert, dass pro Zeile 4 Werte ausgegeben werden sollen. Das Ergebnis sieht so aus:

Dateiausgabe: arrays.ndf

```
@ARRAY Integers {
      [12]
      0 1 2 3
      4 5 6 7
      8 9 10 11;
}
```

Das folgende Beispiel zeigt das Schreiben von mehrdimensionalen Arrays. Jede Dimension kann einen anderen Datentyp und eine andere Größe haben. Alle Werte einer Dimension müssen natürlich vom gleichen Datentyp sein. Die Methode *writeArrayValues(...)* schreibt alle übergebenen Werte in einem Zug. Sie akzeptiert als Argument Wertelisten, *Arrays* oder auch *Collections*.

```
1  setArrayValuesTabbed(true);
2
3  openArray("Work", 5, 7);
4  writeArrayValues(new int[] {6,8,6,8,4});
5  writeArrayValues("MO","DI","MI","DO","FR","SA","SO");
6  closeArray();
```

Die Einstellung *setArrayValuesTabbed(true)* bewirkt, dass die Werte durch einen Tabulator getrennt werden und somit das Ergebnis schöner formatiert aussieht:

```
@ARRAY Workdays {
      [7 5]
      "MO"   "DI"   "MI"   "DO"   "FR"
      "SA"   "SO";
      6      8      6      8      4;
}
```

➔ Es müssen zwingend genauso viele Werte geschrieben werden, wie im Kopf des Arrays angegeben wurde!

2.6 Schreiben von Texten

Zum Schreiben von einfachen textuellen Daten bietet NDF die TEXT-Struktur. Sie speichert im Prinzip Fließtext beliebiger Länge, wobei zwei Sonderzeichen zu definieren sind: das EOT (**E**nd**O**f**T**ext) Sonderzeichen markiert das Ende des Textes. Das LF (**L**ine**F**eed) Sonderzeichen markiert einen Abschnitt (expliziter Zeilenvorschub) im Text. Das folgende Beispiel zeigt die Erzeugung einer TEXT-Struktur mit zwei Abschnitten:

```
1  openText("Description");
2  writeText("The effect could not be reproduced");
3  writeText("There is a new project beeing set up\nfor examinating the problem.");
4  closeText();
```

Dies führt zu folgender Ausgabe:

```
@TEXT Description {
      [^|]
      The effect could not be reproduced
      ^
      There is a new project beeing set up
      for examinating the problem.
      |
}
```

Im Kopf der TEXT-Struktur sind die beiden Zeichen LF und EOT angegeben, die den Text in Abschnitte unterteilen bzw. das Ende markieren. Diese Zeichen dürfen dann nicht im zu schreibenden Text vorkommen, was natürlich von der API geprüft wird[1]. Der mehrfache Aufruf von *writeText(...)* führt automatisch zur Sektionierung in Abschnitte. Zusätzlich unterstützt die TEXT-Struktur auch Properties, wie das nächste Beispiel zeigt:

```
1  NdfProperties props = new NdfProperties();
2  props.put("format", "Header1");
3
4  openText("Message", props, '%', '~');
5  writeText("All about |TEXT| Structures");
6  closeText();
```

1 Der Leser mag erwarten, dass z.B. das EOT-Zeichen '|' im Text verwendet werden könnte, wenn man das Escape-Zeichen '\' voranstellt also: \|. Escaping wird aber hier aus technischen Gründen nicht unterstützt, da es ganz erheblich die Performance beim späteren Parsen des Dokumentes verschlechtern würde.

Hier wurden Properties zugewiesen. Zusätzlich wurden LF und EOF neu vergeben. Eine Änderung dieser Definition ist nur dann erforderlich, wenn eines der Zeichen tatsächlich im Text benutzt werden muss. Die Ausgabe sieht wie folgt aus:

```
@TEXT Message {
      < format="Header1" >
      [~%]
      All about |TEXT| Structures
      %
}
```

2.7 Schreiben von Objekten

Objekte sind Strukturen, die Properties und ggf. weitere Objekte oder Strukturen beinhalten. Ein einfaches Objekt wird z.B. folgendermaßen erstellt:

```
1  NdfProperties cityProps = new NdfProperties();
2  cityProps.put("cityName", "Stuttgart");
3  cityProps.put("country", "DE");
4  cityProps.put("habitants", 800000);
5
6  openObject("City", cityProps);
7  writeText("Stuttgart is one of the most attractive cities in Germany");
8  closeObject();
```

Die Ausgabe sieht wie folgt aus:

Dateiausgabe: objects.ndf

```
@OBJECT City {
      <
      cityName  = "Stuttgart"
      country   = "DE"
      habitants = 800000
      >
      @TEXT Message {
            [^|]
            Stuttgart is one of the most attractive cities in Germany
            |
      }
}
```

Der obere Property-Block ist ebenso optional, wie die unten enthaltene TEXT-Struktur. Ein Objekt kann also auch völlig leer sein. Die Stärke von Objekt-Strukturen liegt darin, dass sie weitere eingebettete Objekte enthalten können, wie in folgender Erweiterung gezeigt wird:

```
1  … // cityProps as above
2  openObject("City", cityProps);
3
4  NdfProperties geoProps = new NdfProperties();
5  geoProps.put("longitude", 242.5353);
6  geoProps.put("latitude", 179.0764);
7  openObject("GeoLocation", geoProps);
8  closeObject();
9
10 writeText("Stuttgart is one of the most attractive cities in Germany");
11 closeObject();
```

Führt zu folgender Ausgabe:

```
@OBJECT City {
      <
      cityName  = "Stuttgart"
      country   = "DE"
```

```
      habitants = 800000
      >
      @OBJECT GeoLocation {
             < longitude=242.5353 latitude=179.0764 >
      }
      @TEXT Message {
             [^|]
             Stuttgart is one of the most attractive cities in Germany
             |
      }
}
```

Wie man in der Ausgabe sieht, wurde das *GeoLocation*-Objekt innerhalb des *City*-Objekts angelegt. Es können beliebig viele weitere Objekte auf der gleichen Ebene hinzugefügt werden. Diese können ihrerseits weitere Unter-Objekte beinhalten - in beliebiger Schachtelungstiefe. Auch TABLE- und ARRAY-Strukturen können auf diese Weise eingebettet werden wie das folgende Beispiel zeigt:

```
1  setPropertiesCompacted(true);
2  setUseObjectShortCut(true);
3
4  openObject("Route");
5  openPropertyList();
6  writeProperty("id", 1);
7  writeProperty("name", "Way to work");
8  closePropertyList();
9
10 NdfTable table = new NdfTable("GeoLocation");
11 table.addColumns("longitude",  "latitude");
12 table.addRow(245.491, 187.653);
13 table.addRow(245.467, 187.659);
14 table.addRow(245.422, 187.665);
15 writeTable(table);
16
17 closeObject();
```

Die Einstellung in Zeile 1 führt, wie bereits bekannt, zu einer einzeiligen Ausgabe der Properties. Zeile 2 bewirkt, dass Objekte nicht mit *@OBJECT* beginnen, sondern nur noch mit dem Kürzel[2] **@**.

```
@ Route {
      < id=1 name="Way to work" >
      @TABLE GeoLocation {
             [longitude latitude]
              245.491   187.653;
              245.467   187.659;
              245.422   187.665;
      }
}
```

2.8 Schreiben von *NdfStorable* Objekten

Mit Hilfe des Interfaces *NdfStorable* kann das Schreiben von Objekten vereinfacht werden. Es verlangt keine Methoden, sondern dient lediglich der Markierung. Von einem *NdfStorable* Objekt wird erwartet, dass es folgende Annotationen enthält:

- **@NdfType**
  vor dem Klassennamen
- **@NdfField**
  vor jeder Instanzvariablen, die im NDF Format gespeichert werden soll

2 Das Kürzel '@' anstelle von '@OBJECT' ist immer noch ein eindeutiger Bezeichner für Objekt-Strukturen. Die Kurzschreibweise reduziert die Datenmenge bei massenhafter Ausgabe von Objekten.

Beide Annotationen können einen String als Argument tragen. Auf diese Weise kann der zu persistierende Name vorgegeben werden. Ist kein Argument angegeben, so wird der einfache Klassenname (ohne Package), bzw. der Name der Instanzvariablen verwendet. Annotieren wir beispielsweise die *GeoLocation* Klasse wie folgt,

```
1  @NdfType("Location")
2  public class GeoLocation implements NdfStorable {
3      @NdfField() private double longitude;
4      @NdfField() private double latitude;
5      ...
6  }
```

dann können wir sie auf wesentlich einfachere Weise schreiben:

```
1  GeoLocation location = new GeoLocation(2.345, 7.543);

1  openObject(location);
2  closeObject();
```

Aufgrund der Annotationen ist nun bekannt, unter welchen Namen das Objekt und seine Eigenschaften im NDF Format abzubilden sind. Auch das Schreiben in Tabellenform vereinfacht sich nun wie folgt:

```
1  GeoLocation location = new GeoLocation(2.345, 7.543);

1  NdfTable table = new NdfTable(GeoLocation.class);
2  table.addRow(location); // creates the row values from the object's properties
3  writeTable(table);
```

2.9 Schreiben von Kommentaren

Ein NDF-Dokument kann an beliebigen Stellen Kommentare beinhalten, auch innerhalb von *OBJECT*, *TABLE* oder *ARRAY* Strukturen[3]. Sie dienen allein zur Erläuterung der gespeicherten Information und werden beim späteren Parsen **nicht** ausgewertet.

Es gibt zwei Typen von Kommentaren:

- **Zeilenkommentare**
  beginnen mit dem #-Zeichen und gelten bis zum Ende der Zeile.
- **Block-Kommentare**
  beginnen mit dem **$**-Zeichen und gelten bis zum nächsten **$**-Zeichen.

Das folgende Beispiel zeigt beide Typen von Kommentaren.

```
1  writeComment("This is a single line comment");
2  writeLine(); // write an empty line
3  writeBlockComment("This is a \nblock comment\nof three lines.");
```

Führt zu folgender Ausgabe:

```
# This is a single line comment

$----------------------------------------
This is a
block comment
of three lines.
----------------------------------------$
```

3 Innerhalb von TEXT-Strukturen werden Kommentare nicht unterstützt

Block-Kommentare werden automatisch mit einer Kopf- und Fußzeile versehen. Deren Aussehen kann man wie folgt beeinflussen:

```
writeCommentHeader(30);
writeComment("This is a single line comment");
writeCommentHeader(30, '+');
writeLine();

openBlockComment(26, "Block-Header"); // use a titled header...
writeBlockComment("This is a \nblock comment\nof three lines.");
closeBlockComment("Block-Footer"); // ... and a titled footer
```

Sowohl die Länge eines Comment-Headers, als auch das Füllzeichen kann man, wie oben gezeigt, einstellen. Damit sehen die geschriebenen Kommentare so aus:

```
##############################
# This is a single line comment
#++++++++++++++++++++++++++++

$--- Block-Header ---------
This is a
block comment
of three lines.
--- Block-Footer ---------$
```

Die Länge und das Füllzeichen von Comment-Headern kann mit folgenden Methoden global vorgegeben werden:

```
setDefaultCommentHeaderLength(40);     // valid for both block- and line-comment
setFillCharForBlockCommentHeader('*');
setFillCharForLineCommentHeader('=');
```

Beide Arten von Kommentaren können auch innerhalb von Objekt-Strukturen geschrieben werden, wo sie automatisch entsprechend der Schachtelungstiefe der Objekte eingerückt erscheinen.

2.10 Schreiben im Append-Mode

Gelegentlich ist es notwendig, dass in einer vorhandenen NDF-Datei lediglich Daten an Ende hinzugefügt werden müssen. Ein typisches Beispiel wäre eine Logging-Datei im NDF-Format. Zum Anhängen von Daten an eine vorhandene Datei ohne sie zu überschreiben, kann der *NdfWriter* im *Append-Mode* geöffnet werden.

```
    File outFile = new File("hello.ndf");
    new HelloWorldNdfWriter(ctx, outFile, true).execute(); // open file in append mode
```

Man beachte, dass im *Append-Mode* im allgemeinen kein Datei-Header geschrieben wird, es sei denn, die Datei ist zum Zeitpunkt des Öffnens leer. Dann wird sehr wohl der Header geschrieben[4] (ansonsten könnte die Datei nicht mehr geparsed werden).

3 Parsen von NDF Dokumenten

Zum Parsen von NDF-Dokumenten dient die Klasse *NdfParser*. Sie benötigt eine Instanz von *NdfHandler*, der sie per Methodenaufruf die im Dokument gefundenen Informationen mitteilt (Event Driven Parsing). Für eine praktische Anwendung muss eine kundenspezifische Klasse erstellt werden, die von *NdfHandler* ableitet. Diese überschreibt dann die betreffenden Methoden,

4 Das heißt, dass die Methode *openDocument()* nur beim Neuanlegen der Datei aufgerufen wird.

um die mitgeteilte Information auszuwerten und zu verarbeiten.

3.1 Starten des NdfParsers

Als erstes Beispiel entwickeln wir die folgende *DocumentHandler*-Klasse, eine Implementierung von *NdfHandler*. Sie überschreibt die Methoden *startDocument(...)* und *doneDocument(...)* und reagiert somit auf das Öffnen und Schließen des eingelesenen Dokuments.

```
1  public class DocumentHandler extends NdfHandler {
2    @Override
3    public void startDocument(NdfStructure struct, NdfProperties props) throws Exception {
4      logInfo("Started parsing: " +getFileInfo().getFileName());
5      logInfo(props.toDumpString());  // dump full content
6    }
7    @Override
8    public void doneDocument(NdfStructure struct) throws Exception {
9      logInfo("Finished parsing: " +struct.getInfo());
10   }
11 }
```

Die folgende Klasse *ParseTest* startet in ihrer *main(...)* Methode den *NdfParser* mit unserem *DocumentHandler* und liest die im *NdfWriter*-Kapitel geschriebene Eingabedatei *hello.ndf* ein.

```
1  public class ParseTest {
2    public static void main(String[] args) throws Exception {
3      ApplicationContext ctx = AppUtil.getDefaultContext();
4      File inputFile = new File("hello.ndf");
5      NdfHandler handler = new DocumentHandler();
6
7      new NdfParser(ctx, inputFile, handler).execute();
8    }
9  }
```

Wir erhalten folgende Ausgabe auf der Systemkonsole:

```
INF: [esprit] DocumentHandler: Started parsing: hello.ndf
INF: [esprit] DocumentHandler: Content of NdfProperties size=3
  ndfDocVersion: 1.6
 ndfDocEncoding: UTF-8
  ndfDocCreated: 2012-11-22 06:23:17

INF: [esprit] DocumentHandler: Finished parsing: hello.ndf
```

Wie man sieht, wird der Handler durch Aufruf der überschriebenen Methoden über das Öffnen und Schließen der Eingabedatei informiert. Ähnlich geschieht dies mit allen Strukturen, die in einem NDF-Dokument enthalten sein können. Bei *openDocument(...)* werden dem Handler zwei Objekte übergeben:

- *NdfStructure*
  Dieses Objekt ist praktisch ein „Navigator“ durch die im Dokument gefundenen Strukturen. Es kennt die Zeilennummer, die Schachtelungstiefe u.v.m. und wird im letzten Kapitel ausführlich erklärt.
- *NdfProperties*
  Hiermit erhalten wir die zur aktuellen Struktur zugehörigen Properties übergeben. In diesem Falle sind dies die Header-Properties des Dokuments. Der Umgang damit wird im folgenden Kapitel erläutert.

Sehr nützlich bei der Fehlersuche ist die *toDumpString()* Methode, die den gesamten Inhalt des

NdfProperties Objektes auf der Konsole ausgibt[5].

3.2 Parsen von Properties

Als nächstes wollen wir einen *PropertiesHandler* erstellen (abgeleitet von *NdfHandler*), der die im *NdfWriter*-Kapitel geschriebene Datei *properties.ndf* verarbeiten kann:

```
1  public class PropertiesHandler extends NdfHandler {
2    @Override
3    public void foundProperties(NdfStructure struct, NdfProperties props) throws Exception {
4      int intValue             = props.getAsInt("intValue");
5      double doubleValue       = props.getAsDouble("doubleValue");
6      Timestamp timestampValue = props.getAsTimestamp("timestampValue");
7      Date dateValue           = props.getAsDate("dateValue");
8      List<String> listValue   = props.getAsList("listValue");
9      String emptyString       = props.getAsList("emptyString");
10
11     // do something with these values
12   }
13 }
```

Immer, wenn im NDF-Dokument ein Property-Block auf Top-Ebene gefunden wurde, dann wird im *NdfHandler* die überschriebene Methode *foundProperties(..., **NdfProperties**)* aufgerufen und ein entsprechend gefülltes *NdfProperties*-Objekt übergeben. Die Werte können, wie oben gezeigt, einfach abgefragt und auch gleich in den gewünschten Datentyp konvertiert werden.

3.2.1 Document Header-Properties

Der erste Property Block in einem NDF Dokument behinhaltet die sog. Header-Properties. Sie werden beim Aufruf von *startDocument(...)* an den Handler übergeben. Die Namen dieser Properties sind fest vorgegeben (* = unbedingt erforderlich):

- **ndfDocVersion** *
  enthält die Versionsnummer des Dokumentes
- **ndfDocEncoding** *
  enthält das Encoding des Dokumentes
- **ndfDocCreated** *
  enthält den Zeitstempel der Erzeugung des Dokumentes
- **ndfDocAuthor**
  enthält den Autor des Dokumentes
- **ndfDocType**
  enthält den Typ des Dokumentes (ein frei definierbarer String-Wert)

Die Werte dieser Properties können natürlich auf die übliche Weise aus dem *NdfProperties* Objekt erfragt werden. Zur etwas bequemeren Abfrage besitzt der *NdfHandler* aber bereits spezielle Methoden, wie *getDocVersion(), getDocEncoding()* etc.. Sehr praktisch sind zudem folgende Methoden:

- *checkDocumentType("AddressBook");*
  Dieser Aufruf überprüft, ob der Wert der *ndfDocType* Property mit dem angegebenen Wert übereinstimmt und wirft eine entsprechende Exception, falls dies nicht zutrifft.
- *acceptDocumentVersion(String version);*
  Diese Methode wird automatisch im Handler aufgerufen. Durch Überschreiben können Sie

5 Die *toDumpString()* Methode ist im Esprit-Framework in fast allen Klassen enthalten, die Daten enthalten. Sie ist eine hervorragende Debugging-Hilfe, wenn es darum geht, den inneren Zustand einer Klasse festzustellen.

über den Return-Wert (true/false) bestimmen, ob die im Dokument gefundene Version (Wert der *ndfDocVersion* Property) für Sie akzeptabel ist, oder nicht.

3.2.2 Erweiterte Property-Notifizierung

Properties, die nicht auf Top-Ebene definiert sind, gehören immer zu einer bestimmten Struktur und werden derselben im entsprechenden *startXXX(...)* Aufruf übergeben. Mit Hilfe des Schalters *setNotifyAllProperties(true)* erreicht man, dass der Parser sämtliche, auf allen Ebenen gefundenen Properties meldet. So kann man sehr leicht eine Property-Suche durch alle Objekte und Strukturen implementieren.

3.3 Parsen von Listen

Zum Parsen von Listen-Werten sehen wir uns als Beispiel den folgenden *ListHandler* an. Er liest die Werte aus der Datei *list.ndf* ein, die wir im *NdfWriter*-Kapitel geschrieben haben:

```
1  public class ListHandler extends NdfHandler {
2    private List currentList;
3
4    @Override
5    public NotifyListValue startList(NdfStructure struct, NdfProperties props) throws Exception {
6      currentList = new ArrayList();
7      return NotifyListValue.AS_STRING;
8    }
9
10   @Override
11   public void foundListValue(NdfStructure struct, String value) throws Exception {
12     currentList.add(value);
13   }
14
15   @Override
16   public void foundListValue(NdfStructure struct, AsciiWord data) throws Exception {
17     // only called if startList(...) returns NotifyListValue.AS_BUFFERED_WORD
18   }
19
20   @Override
21   public void doneList(NdfStructure struct) throws Exception {
22     logInfo(currentList.toString());
23   }
24 }
```

Das Finden einer LIST-Struktur meldet der Parser durch den Aufruf von *startList(...)*. An dieser Stelle erzeugen wir eine *ArrayList* als Instanzvariable, um die Listenwerte aufzusammeln, die uns durch die folgenden Aufrufe von *foundListValue(...)* mitgeteilt werden.

3.3.1 Notifizierung von Listen-Werten

Wichtig ist der *NotifyListValue* Return-Wert der *startList(...)* Methode. Hier haben wir folgende Alternativen:

- NotifyListValue.**AS_STRING**
  bewirkt den Aufruf von *foundListValue(..., **String**)* für jeden gefundenen Listenwert.
- NotifyListValue.**AS_BUFFERED_WORD**
  bewirkt den Aufruf von *foundListValue(..., **AsciiWord**)* für jeden einzelnen Listenwert. Das übergebene *AsciiWord*[6]-Objekt bietet Methoden wie *asString(), asInt(), asDouble()* etc., um den betreffenden Datenwert in den gewünschten Typ zu konvertieren.

> Die Notifizierung mit **AS_BUFFERED_WORD** ist zwar sehr performant, sie funktioniert allerdings nur für Listen, die ausschließlich Zahlenwerte beinhalten.

6 Das *AsciiWord* ist technisch ein Wrapper über den Einlesepuffer des Parsers, der genau die Stelle herausgreift, wo der betreffende Zellenwert der Tabelle steht. Die Datenkonvertierung in eine Zahl geschieht dabei direkt aus dem Einlesepuffer heraus und ist deshalb äußerst performant.

- NotifyListValue.**SKIP**
  bewirkt, dass keine Notifizierung stattfindet. Wenn man an den Werten explizit nicht interessiert ist, kann man so die Parsing-Geschwindigkeit steigern.

Je nachdem, welchen Return-Wert wir verwenden, brauchen wir also nur die betreffende Methode zu überschreiben – oder im Falle von *SKIP* natürlich keine.

Sind alle Listenwerte eingelesen, wird schließlich *doneList(...)* aufgerufen. Die Bearbeitung der betreffenden LIST-Struktur ist damit abgeschlossen.

3.4 Parsen von Tabellen

Zum Parsen einer Tabelle sehen wir uns als Beispiel den folgenden *TableHandler* an. Er liest die Tabelle *Courses* aus der Datei *tables.ndf* ein, die wir im *NdfWriter*-Kapitel geschrieben haben:

```
1  public class TableHandler extends NdfHandler {
2    private NdfTable currentTable;
3
4    @Override
5    public NotifyTableValue startTable(NdfStructure struct, NdfProperties pr, String[] cols) throws Exception {
6      currentTable = new NdfTable(struct.getName());
7      currentTable.addColumns(cols);
8      return NotifyTableValue.AS_STRING_ARRAY;
9    }
10
11   @Override
12   public void foundTableRow(NdfStructure struct, int rowIndex, String[] rowData) throws Exception {
13     currentTable.addRow(rowData);
14   }
15   @Override
16   public void foundTableRow(NdfStructure struct, int rowIndex, NdfProperties rowProps) throws Exception {
17     // only called if startTable(...) returns NotifyTableValue.AS_PROPERTIES
18   }
19
20   @Override
21   public void foundTableCell(NdfStructure struct, int rowIdx, int colIdx, AsciiWord data) throws Exception {
22     // only called if startTable(...) returns NotifyTableValue.AS_BUFFERED_WORD
23   }
24
25   @Override
26   public void doneTable(NdfStructure struct) throws Exception {
27     logInfo(currentTable.toDumpString());
28   }
29 }
```

Das Finden einer TABLE-Struktur meldet der Parser durch den Aufruf von *startTable(...)*. Dabei übergibt er wie üblich ein *NdfStructure* Objekt, das wir später noch genauer anschauen. Es beinhaltet jedenfalls den Namen der Table-Struktur – in unserem Fall: „*Courses*“. An dieser Stelle legen wir ein *NdfTable*-Objekt als Instanzvariable an, um die im Folgenden einzulesenden Tabellendaten aufzusammeln. Im gleichen Aufruf erfahren wir auch die Namen der Spalten, so dass wir das *NdfTable*-Objekt entsprechend initialisieren können. Sollte die TABLE-Struktur Properties besitzen, können wir diese aus dem übergebenen *NdfProperties*-Objekt auslesen.

3.4.1 Notifizierung von Tabellen-Werten

Wichtig ist der *NotifyTableValue* Return-Wert der *startTable(...)* Methode. Hier haben wir vier Alternativen:

- NotifyTableValue.**AS_STRING_ARRAY**
  bewirkt den Aufruf von *foundTableRow(..., **String[]**)* für jede gefundene Tabellenwertezeile. Wir erhalten die Zeilendaten als *String[]*, welches wir in unserem Beispiel nutzen, um unser *NdfTable* Objekt mit den gefundenen Werten aufzufüllen.
- NotifyTableValue.**AS_PROPERTIES**
  bewirkt den Aufruf von *foundTableRow(..., **NdfProperties**)* für jede gefundene Wertezeile. Das übergebene *NdfProperties* Objekt kann man auf die übliche Art auslesen, wobei die

Spaltennamen als Schlüssel für die Property-Werte dienen. Der Vorteil dieser Methode ist, dass wir die bequemen Datenkonvertierungsmethoden des *NdfProperties* Objekts nutzen können.

- NotifyTableValue.**AS_BUFFERED_WORD**
  bewirkt den Aufruf von *foundTableCell(..., **AsciiWord**)* für jeden einzelnen Zellenwert. Das übergebene *AsciiWord*[7]-Objekt bietet Methoden wie *asString(), asInt(), asDouble()* etc., um den betreffenden Datenwert in den gewünschten Typ zu konvertieren.

  Die Notifizierung mit **AS_BUFFERED_WORD** ist zwar sehr performant, sie funktioniert allerdings nur für Tabellen, die ausschließlich Zahlenwerte beinhalten.

- NotifyTableValue.**SKIP**
  bewirkt, dass keine Notifizierung stattfindet. Wenn man an den Werten einer bestimmten Tabelle explizit nicht interessiert ist, kann man so die Parsing-Geschwindigkeit steigern.

Je nachdem, welchen Return-Wert wir verwenden, brauchen wir also nur die betreffende Methode zu überschreiben – oder im Falle von *SKIP* natürlich keine.

Sind alle Tabellendaten eingelesen, wird schließlich *doneTable(NdfStructure)* aufgerufen. Im obigen Beispiel geben wir an dieser Stelle alle aufgesammelten Daten auf der Konsole aus. Die Bearbeitung der betreffenden TABLE-Struktur ist damit abgeschlossen.

3.5 Parsen von Arrays

Das Parsen von Arrays geschieht ganz analog zum Parsen von Tabellen. Schauen wir uns den folgenden *ArrayHandler* an. Er liest das Array „*Integers*“ aus der Datei *arrays.ndf*, welches wir im *NdfWriter*-Kapitel geschrieben haben:

```
1  public class ArrayHandler extends NdfHandler {
2    private int[] currentArray;
3
4    @Override
5    public NotifyArrayValue startArray(NdfStructure struct, NdfProperties props, int[] dims) throws Exception {
6      int size = dims[0];
7      currentArray = new int[size];
8      return NotifyArrayValue.AS_BUFFERED_WORD;
9    }
10
11   @Override
12   public void foundArrayValue(NdfStructure struct, int dim, int index, String value) throws Exception {
13     // not used here
14   }
15
16   @Override
17   public void foundArrayValue(NdfStructure struct, int dim, int index, AsciiWord data) throws Exception {
18     currentArray[index] = word.asInt();
19   }
20
21   @Override
22   public void doneArray(NdfStructure struct) throws Exception {
23     logInfo(ConvertUtil.toStringList(currentArray));
24   }
25 }
```

Das Finden einer ARRAY-Struktur meldet der Parser durch den Aufruf von *startArray(...)*. Hierbei erfahren wir den Namen des Arrays (aus dem *NdfStructure*-Objekt), sowie die Größe aller Dimensionen. Da unser „*Integers*“ Array nur eine Dimension hat, greifen wir uns den ersten Wert aus *dims[]* und legen eine Instanzvariable *int[] currentArray* mit der entsprechenden Größe an. In diesem Array wollen wir die kommenden Werte aufsammeln.

7 Das *AsciiWord* ist technisch ein Wrapper über den Einlesepuffer des Parsers, der genau die Stelle herausgreift, wo der betreffende Zellenwert der Tabelle steht. Die Datenkonvertierung in eine Zahl geschieht dabei direkt aus dem Einlesepuffer heraus und ist deshalb äußerst performant.

3.5.1 Notifizierung von Array-Werten

Wichtig ist der *NotifyArrayValue* Return-Wert der *startArray(...)* Methode. Hier haben wir drei Alternativen:

- NotifyArrayValue.**AS_STRING**
  bewirkt den Aufruf von *foundArrayValue(..., **String**)* für jeden gefundenen Array-Wert. Wollten wir diesen String-Wert in unserem *currentArray* speichern, dann müssten wir ihn jeweils in einen *int*-Wert konvertieren. Deshalb eignet sich für dieses Beispiel die nächste Methode wesentlich besser.
- NotifyArrayValue.**AS_BUFFERED_WORD**
  bewirkt den Aufruf von *foundArrayValue(..., **AsciiWord**)* für jeden gefundenen Array-Wert. Bei dieser Notifizierung erhalten wir als Übergabeparameter ein *AsciiWord*-Objekt. Dieses bietet hochperformante Methoden zur Zahlenkonvertierung, wie *asInt(), asDouble()* etc. Wir können den erhaltenen Wert, wie im Beispiel gezeigt, also direkt in unserem *currentArray* speichern.

  Die Notifizierung mit **AS_BUFFERED_WORD** ist zwar sehr performant, sie funktioniert allerdings nur für Arrays, die ausschließlich Zahlenwerte beinhalten.
- NotifyArrayValue.**SKIP**
  bewirkt, dass keine Notifizierung stattfindet. Wenn man an den Werten eines bestimmten Arrays explizit nicht interessiert ist, kann man so die Parsing-Geschwindigkeit steigern.

Sind alle Arraydaten eingelesen, wird schließlich *doneArray(NdfStructure)* aufgerufen. Im obigen Beispiel geben wir an dieser Stelle alle aufgesammelten Daten auf der Konsole aus. Die Bearbeitung der betreffenden ARRAY-Struktur ist damit abgeschlossen.

3.6 Parsen von Texten

Analog wie bei den vorher beschrieben Strukturen funktioniert das Parsen von TEXT-Strukturen. Auch hier werden die betreffenden Callback-Methoden im *NdfHander* aufgerufen, wie das folgende Beispiel zeigt:

```
1  @Override
2    public NotifyText startText(NdfStructure struct, NdfProperties props) throws Exception {
3      return NotifyText.AS_STRING;
4    }
5    @Override
6    public void foundText(NdfStructure struct, String text) throws Exception {
7      logInfo(text);
8    }
9    @Override
10   public void foundText(NdfStructure struct, int paragraph List<String> lines) throws Exception{
11
12   }
13   @Override
14   public void doneText(NdfStructure struct) throws Exception {
15   }
```

3.6.1 Notifizierung von gefundenen Texten

Welche der beiden *foundText(...)* Methoden aufgerufen wird hängt davon ab, welcher Return-Wert bei *startText(...)* zurückgegeben wurde. Hier gibt es folgende Möglichkeiten:

- NotifyText.**AS_LINES**
  bewirkt den Aufruf von *foundText(..., **int, List<String>**)*.
  In der übergebenen Liste sind alle Textzeilen einzeln eingetragen. Sie enhalten keine führenden oder nachfolgenden Leerzeichen und auch keine Zeilenumbrüche. Es ist Sache

des Anwenders, wie er die Textzeilen weiterverarbeitet. Im Text vorkommende LF- Symbole bewirken den mehrmaligen Aufruf dieser Methode mit jeweils inkrementiertem *paragrap* Argument.

- NotifyText.**AS_STRING**
  bewirkt den Aufruf von *foundText(..., **String**)*.
  In dem übergebenen String sind sind alle Textzeilen in Fließtext umgewandelt. Er enhält keine führenden oder nachfolgenden Leerzeichen. Im Text vorkommende LF-Symbole finden sich im übergebenen String als '\n' Zeichen wieder.
- NotifyText.**SKIP**
  bewirkt, dass keine Notifizierung stattfindet. Wenn man an dem betreffenden Text explizit nicht interessiert ist, kann man so die Parsing-Geschwindigkeit steigern.

Sind alle Textdaten eingelesen, wird schließlich *doneText(NdfStructure)* aufgerufen. Die Bearbeitung der betreffenden TEXT-Struktur ist damit abgeschlossen.

3.7 Parsen von Objekten

Im Gegensatz zu TABLE-, ARRAY- und TEXT-Strukturen können Objekte weitere Objekte beinhalten und dabei eine beliebige Schachtelungstiefe haben. Im folgenden *CityObjectHandler* Beispiel parsen wir die Datei *objects.ndf*, die wir im *NdfWriter*-Kapitel erzeugt haben.

```
1  public class CityObjectHandler extends NdfHandler {
2    private City currentCity;
3
4    @Override
5    public void startObject(NdfStructure struct, NdfProperties props) throws Exception {
6      if (struct.getName().equals("City")) {
7        String cityName = props.getAsString("cityName");
8        String country = props.getAsString("country");
9        int habitants = props.getAsInt("habitants");
10       currentCity = new City(cityName, country, habitants);
11     }
12     else if (struct.getName().equals("GeoLocation")) {
13       double longitude = props.getAsDouble("longitude");
14       double latitude = props.getAsDouble("latitude");
15       currentCity.setLocation(new GeoLocation(longitude, latitude));
16     }
17   }
18
19   @Override
20   public void foundText(NdfStructure struct, List<String> lines) throws Exception {
21     if (currentCity != null) {
22       currentCity.setComments(lines);
23     }
24   }
25
26   @Override
27   public void doneObject(NdfStructure struct) {
28     if (struct.getName().equals("City")) {
29       logInfo(currentCity.toDumpString());
30       currentCity = null;
31     }
32   }
33 }
```

Das Finden einer OBJECT-Struktur meldet der Parser durch den Aufruf von *startObject(...)*. Hierbei erfahren wir den Namen des Objektes aus *NdfStructure*, sowie alle dessen Eingenschaften aus *NdfProperties*. Ziel des Parsens ist letztendlich, die in der Datei gespeicherten Strukturen in echte Java-Objekte umzuwandeln. Dazu benötigen, wir die Model-Klassen *City* und *GeoLocation*, um die gewünschten Daten aufzunehmen. Dies sind einfache *JavaBeans*, die entsprechende Konstruktoren,

sowie Getter- und Setter-Methoden für die Properties enthalten[8].

Da die Eingabedatei mehrere OBJECT-Strukturen enthält, müssen wir unterscheiden, welche wir bearbeiten wollen. Deshalb befragen wir die *NdfStructure*-Instanz nach dem Namen des Objekts, welches gerade eingelesen wird. Der Parser gibt uns im gleichen Aufruf auch sämtliche Eigenschaften, sodass wir es mit Hilfe dessen Konstruktors instanziieren können [hier: *new City(...)*]. Wir speichern es als Instanzvariable, damit wir die noch kommenden Daten später hinzufügen können.

Wenn in der Eingabe ein *GeoLocation* Objekt erscheint, können wir dieses sofort instantiieren, wie oben gezeigt. Da wir wissen, dass es ein in *City* eingebettetes Objekt ist, übertragen wir es mit *currentCity.setLocation(...)* in unser bereits existierendes *City*-Objekt.

3.7.1 Parsen von *NdfStorable* Objekten

NdfStorable Objekte, die mit den Annotationen @NdfType und @NdfField versehen sind, konnten auf besonders einfache Weise geschrieben werden. Auch das Parsen kann sich erheblich vereinfachen, wenn man die Objekte mit einem Konstruktor versieht, der NdfProperties als Argument akzeptiert, wie im folgenden, erweiterten *GeoLocation*-Beispiel gezeigt:

```
1  @NdfType("Location")
2  public class GeoLocation implements NdfStorable {
3      @NdfField() private double longitude;
4      @NdfField() private double latitude;
5
6      public GeoLocation(NdfProperties p) {
7        longitude = p.getAsDouble("longitude");
8        latitude = p.getAsDouble("latitude");
9      }
10     ...
11 }
```

Dabei ist darauf zu achten, dass die Variablennamen mit den Property-Keys übereinstimmen. Durch diesen Konstruktor vereifacht sich das Einlesen des Objektes zu einem Einzeiler:

```
1   @Override
2   public void startObject(NdfStructure struct, NdfProperties props) throws Exception {
3     ...
4     else if (struct.getName().equals("Location")) {
5       currentCity.setLocation(new GeoLocation(props));
6     }
7   }
```

Man beachte, dass die *GeoLocation* Klasse mit @NdfType("Location") annotiert wurde. Daher wurde sie unter dem Namen "Location" gespeichert und wird in Zeile 4 auch unter diesem Namen erwartet!

3.7.2 Schließen von Objekten

Sind alle Daten eines Objektes eingelesen, wird schließlich *doneObject(NdfStructure)* für das betreffende Objekt aufgerufen. Im obigen Beispiel müssen wir auch hier unterscheiden, welches Objekt geschlossen wurde, um das richtige Objekt zu verarbeiten. Beim Schießen des City-Objekts geben wir dessen Inhalt auf der Konsole aus und können sehen, dass es vollständig aufgebaut wurde.

➔ Es ist guter Programmierstil die temporäre Objekt-Referenz nach der fertigen Bearbeitung auf *null* zu setzen, wie im Beispiel gezeigt. Dadurch wird der belegte Speicher frühzeitig freigegeben.

8 In dem öffentlich zugänglichen Beispielprojekt *EspritWorkbench* finden Sie die Quellcodes dieser Objekte.

3.8 Das NdfStructure Objekt

Das *NdfStructure*-Objekt wird uns bei jedem Aufruf in einer *NdfHandler* Methode übergeben. Bisher haben wir es lediglich benutzt, um den Namen einer Struktur zu erfragen. Es kann aber noch mehr. Es ist sozusagen der **Navigator** durch die geparste Datei. Hier sind seine wichtigsten Methoden:

- *getType()*
  teilt mit, um welchen Struktur-Typ es sich handelt, OBJECT, TABLE, ARRAY, etc.
- *getName()*
  gibt den Namen einer Struktur zurück.
- *getPath()*
  zeigt den vollen Pfadnamen eines Objekts in seiner der gesamten Schachtelungstiefe. Der Pfadname des Objekts ist natürlich eindeutiger, als der einfache Objekt-Name.
- *getParent()*
  gibt eine Referenz auf die übergeordnete Struktur zurück. Die oberste Struktur ist die *Root*-Struktur, die mit *isRoot()* erfragt werden kann.
- *isOpened(), isClosed()*
  teilt mit, ob es sich um das Öffnen oder Schließen einer Struktur handelt.
- *getNestingLevel()*
  teilt mit, in welcher Schachtelungstiefe sich das gerade geparste Objekt befindet.
- *getLineNumber()*
  teilt mit, in welcher Zeile der Eingabedatei sich das gerade geparste Objekt befindet.
- *getInfo()*
  gibt einen String zurück in dem alle wichtigen Informationen enthalten sind. Dies ist sehr nützlich als Debugging-Hilfe.
- *setReference(Object), getReference(), getReferenceByName(String)*
  mit diesen Methoden kann man eine Referenz zwischenspeichern und wieder abfragen. Wie man sie verwendet, wird weiter unten gezeigt.

Im Esprit-Framework existiert ein *DumpingNdfHandler,* der alle Methoden so überschrieben hat, dass *getInfo()* jeweils zum Zeitpunkt des Aufrufs ausgegeben wird. Lassen wir mit diesem Handler die Datei *objects.ndf* parsen, dann sieht das Ergebnis wie folgt aus:

```
###: [esprit] DumpingNdfHandler: (line 5) 0 DOCUMENT PROPERTY /objects.ndf
###: [esprit] DumpingNdfHandler: (line 5) 0 DOCUMENT OPENED /objects.ndf
###: [esprit] DumpingNdfHandler: (line 6) 1 OBJECT OPENED /City
###: [esprit] DumpingNdfHandler: (line 13) 2 OBJECT OPENED /City/GeoLocation
###: [esprit] DumpingNdfHandler: (line 15) 2 OBJECT CLOSED /City/GeoLocation
###: [esprit] DumpingNdfHandler: (line 17) 1 OBJECT TEXT /City
###: [esprit] DumpingNdfHandler: (line 17) 1 OBJECT CLOSED /City
###: [esprit] DumpingNdfHandler: (line 18) 0 DOCUMENT CLOSED /objects.ndf
```

Man sieht sehr deutlich, in welcher Zeile welches Objekt in welcher Schachtelungstiefe gefunden wurde.

3.8.1 Zwischenspeichern von Referenzen

Sehr nützlich ist die Möglichkeit, im *NdfStructure*-Objekt Referenzen speichern zu können. Im folgenden Auszug wurde der *CityObjectHandler* so umgeschrieben, dass das *City* Objekt in der jeweiligen *NdfStructure*-Instanz zwischengespeichert wird.

```
1  public class CityObjectHandler extends NdfHandler {
2    @Override
3    public void startObject(NdfStructure struct, NdfProperties props) throws Exception {
4      if (struct.getName().equals("City")) {
5        String cityName = props.getAsString("cityName");
6        String country = props.getAsString("country");
7        int habitants = props.getAsInt("habitants");
8        struct.setReference(new City(cityName, country, habitants)); // store City on stack
9      }
10     else if (struct.getName().equals("GeoLocation")) {
11       double longitude = props.getAsDouble("longitude");
12       double latitude = props.getAsDouble("latitude");
13
14       City city = struct.getParent().getReference();  // fetch City from stack
15       city.setLocation(new GeoLocation(longitude, latitude));
16     }
17   }
18   ...
```

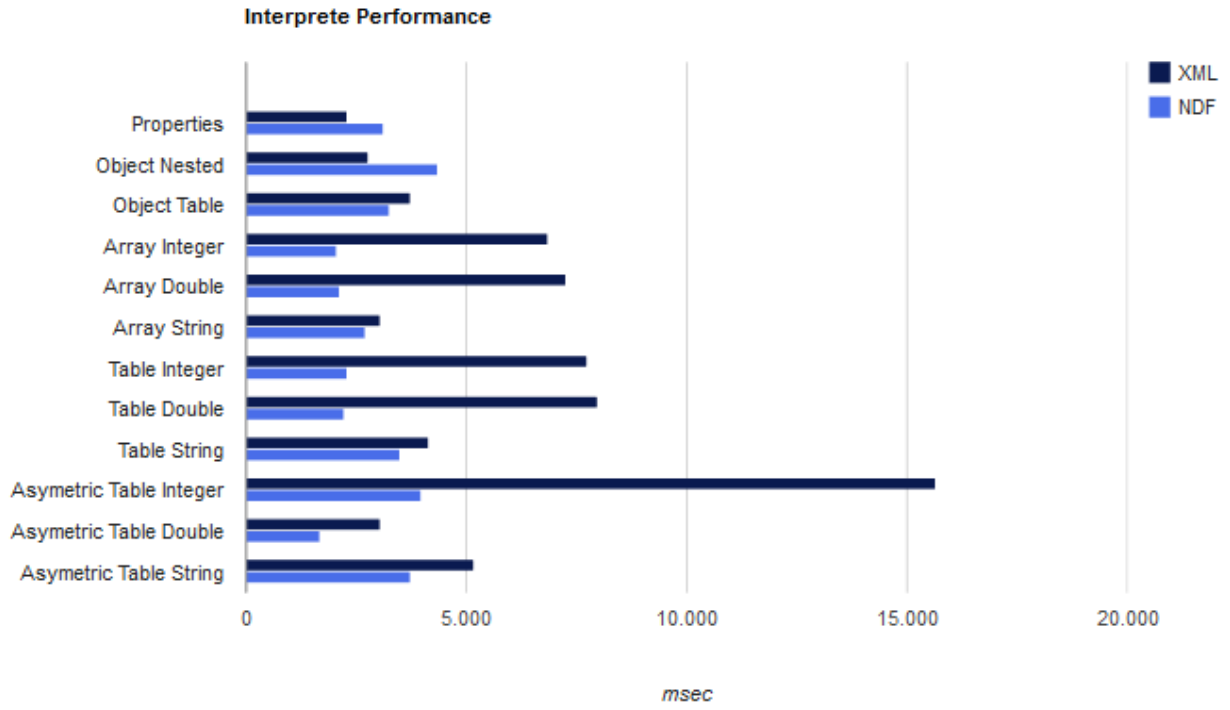
Wie Sie sehen, benötigen wir keine Instanzvariablen mehr! Dies kann die Übersichtlichkeit beim Parsen komplexer Datenstrukturen wesentlich erhöhen. Die Methode *getReference()* gibt die im aktuellen *NdfStructure*-Objekt gespeicherte Referenz zurück. Da das *GeoLocation* Objekt sich innerhalb des City-Objekts befindet gibt der Zugiff *struct.getParent().getReference()* die vorher gespeicherte City-Referenz zurück.

3.9 Performance

Bei der Entwicklung des *NdfWriters* und -*Parsers* wurde größter Wert auf gute Performance gelegt. Das NDF-Datenformat kommt insbesondere zur Speicherung von großen Datenmengen zum Einsatz, wo Performance die entscheidende Rolle spielt.

NdfWriter und -Parser arbeiten daher im *Streaming-Mode*, das heißt, sie sind zustandslos und verbrauchen selbst praktisch keinen Hauptspeicher, egal wie groß die Datenmengen sind, die von ihnen verarbeitet werden.

Das folgende Bild zeigt einen Performance-Vergleichstest zwischen dem NDF- und dem XML-Format, bei dem jeweils eine 100MB Datei mit verschiedenen Datentypen eingelesen und ausgewertet wurde. Der Test wurde im Rahmen eines Studien-Projekts bei der DHBW-Stuttgart (Duale Hochschule Baden Würtemberg) im Januar 2014 durchgeführt.



Das Ergebnis zeigt deutliche Geschwindigkeitsvorteile von NDF, insbesondere bei der Verarbeitung von Massendaten wie Tabellen und Arrays, die aus Zahlen bestehen.