ESPRITSYSTEMS

Neutral Data Format (NDF)

Beschreibung des „*Neutral Data Formats*“
zur Speicherung und Langzeitarchivierung
beliebiger Informationen

NDF Format-Version 1.6

2014-03
Rainer Büsch

Inhaltsverzeichnis

1 Einführung....3

2 Schreiben und Lesen von NDF....3

3 Hauptstrukturen....4

3.1 Namensvergabe....4

3.2 Properties....4

3.2.1 Property Format....4

3.2.2 Header-Properties....5

3.2.3 Wertetyp-Konvertierung....5

3.2.4 Document-Header....5

3.2.5 Document-Encoding....5

3.3 Kommentare....6

3.3.1 Line-Comment....6

3.3.2 Block-Comment....6

3.4 OBJECT-Struktur....6

3.5 LIST-Struktur....7

4.1 TABLE-Struktur....7

4.2 ARRAY-Struktur....8

4.3 TEXT-Struktur....8

5 Zusammengesetzte Strukturen....9

5.1 Eingebettete Strukturen....9

5.2 Hierarchische Strukturen....9

5.3 Properties mit Struktur-Werten....10

6 Erweiterbarkeit....11

7 Vergleich mit XML....11

7.1 Datenmasse....12

7.2 Konsistenzprüfung....12

7.3 Root-Element....12

7.4 Performance....12

8 Fazit....13

1 Einführung

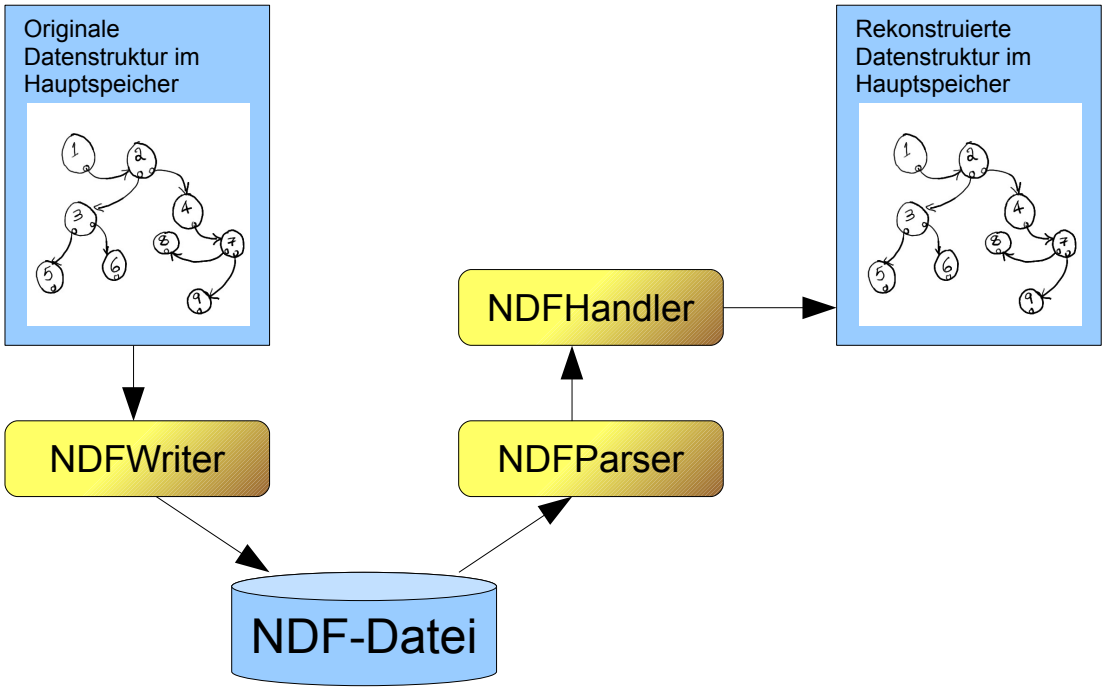
Das ***N**eutral **D**ata **F**ormat* (NDF) ist ein Datenformat zur Speicherung und Langzeitarchivierung von beliebigen Informationen in einem lesbaren ASCII-Format. Die Daten werden in einer hierarchischen Struktur gespeichert, die für Mensch und Computer leicht lesbar ist. Anders als bei XML ist NDF ein sehr kompaktes Format mit sehr wenig Overhead an Meta-Information. Die Strukturen des Formats werden in diesem Dokument beschrieben.

Das programmiertechnische Generieren und Parsen von NDF-Dateien ist in folgendem Dokument beschrieben: http://esprit-systems.de/downloads/esprit/docu/EspritNdfProgrammingDE.pdf

2 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 empfangenen Informationen auszuwerten und ggf. die betreffenden Objekt-Strukturen im Hauptspeicher wieder aufzubauen.



Bei der Enwicklung des *NdfWriters* und -*Parsers* wurde höchsten 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.

3 Hauptstrukturen

Es werden derzeit folgende Strukturen unterstützt: ***OBJECT, TABLE, ARRAY*** und ***TEXT***. Jede der Strukturen hat einen ***Namen*** und eine eigene *Syntax* zur Speicherung der Daten. Darüber hinaus gibt es jeweils mehrere Arten von ***Properties*** und ***Comments***. Im folgenden werden alle Strukturen an einfachen Beispielen gezeigt.

3.1 Namensvergabe

Der Name einer Strukur muss folgende Kriterien erfüllen:

- Er muss ein einfaches Wort sein, ohne Leerzeichen.
- Er darf keine sog. *Break-Character* enthalten. Dies sind ASCII-Zeichen, die zur Identifizierung von Wortgrenzen bzw. Strukturen dienen. Im Einzelnen sind dies: Leerzeichen, Tabulatorzeichen und folgende Zeichen: **{}[]<>\^,;"=@$#**.
  Explizit erlaubt sind hingegen folgende Zeichen: **:._-+**.

➔ Empfohlen werden Namen in *CamelCaseNotation*, wobei Wörter zusammen geschrieben sind und Wortanfänge jeweils mit einem großen Buchstaben beginnen. Namen von Strukturen bzw. Objekten sollten mit einem großen Buchstaben beginnen, Namen von Properties hingegen mit einem Kleinen.

3.2 Properties

Properties sind granuläre Informationen im Format *key=value*. Sie sind nicht typisiert, das heißt alle sind vom Typ ***String***. Die echte Typisierung bzw. Konvertierung in Zahlen etc. muss beim Lesen und Interpretieren vom Anwender erfolgen.

3.2.1 Property Format

Das folgende Beispiel zeigt den Anfang eines NDF-Dokuments:

```
# This is a NDF file written by the NdfWriter.
# Below you see the mandatory document header properties.
<
ndfDocVersion = 1.6                     # the document format version of the writer
ndfDocEncoding = "UTF-8"                # the encoding used in this file
ndfDocCreated = "2012-09-25 19:03:21"   # creation date of the document
>
# Header properties may occur anywhere in the document on top level
<
author="Rainer Buesch"
checkedBy=["Elma" "Tim" "Josch"]
isCommitted=true
>
...
```

Die mit '#' beginnenden Kommentarzeilen werden ignoriert und einfach überlesen.

Mit '<' wird ein Property-Block eröffnet. Danach können beliebig viele Properties im besagten Format *key=value* definiert werden. Der Block muss mit '>' abgeschlossen werden. Gültige Property-Keys sind einfache alphanumerische Strings ohne Sonderzeichen.

Für Property-Values gibt es mehrere Möglichkeiten: Die Einfachste im obigen Beispiel ist:

```
ndfDocVersion = 1.6
```

Wenn der Wert Leerzeichen oder Sonderzeichen beinhaltet, dann muss er gequotet sein wie folgt:

```
format = "Price for {0} is about <{1}> $."
```

Ein boolescher Wert kann nur *true* oder *false* (oder *T* bzw. *F* als Kurzform):

```
isCommitted=true  isPayed=F
```

Ein Property-Wert kann auch eine Werteliste sein, wie in diesem Beispiel:

```
checkedBy=["Josch" "Tim" "Elma"]
```

Ein leerer Wert muss wie folgt angegeben werden:

```
comment=null  # explicit null value
author=""     # empty String value
checkedBy=[]  # empty list value
```

Aufeinanderfolgende Properties müssen durch mindestens ein Leerzeichen getrennt sein. Zeilenumbrüche spielen im NDF Format grundsätzlich keine Rolle und werden wie ein Leerzeichen gewertet. Das *key* und *value* trennende Gleichheitszeichen muss nicht durch Leerzeichen abgetrennt sein. Es können also auch mehrere Properties in etwas kompakterer Weise in die gleiche Zeile geschrieben werden.

```
<ndfDocVersion=1.6 ndfDocCreated="2012-09-25 19:03:21" isConfirmed=true>
```

3.2.2 Header-Properties

Properties, die auf Top-Ebene im Dokument definiert sind, sind sog. *Header-Properties*. Wann immer ein Block von Header-Properties gefunden wird, werden diese vom Parser in einer Map aufgesammelt und an den *NdfHandler* übergeben.

Die Möglichkeit, Header-Properties mehrfach zu definieren ist z.B. sehr nützlich, um ein Dokument in Sektionen einzuteilen, wie in diesem Beispiel:

```
<section=1 description="Input model">
…
<section=2 description="Processing">
…
<section=3 description="Result data">
```

3.2.3 Wertetyp-Konvertierung

Da Properties aus einer NDF-Datei stammen, sind sie per se alle vom Typ String. Ein eingelesener Satz von Properties wird in einer *NdfProperties*-Map an den *NdfHandler* übergeben. Letztere bietet Methoden wie *getAsInt(key), getAsDouble(key), getAsTimestamp(key), getAsList(key)* etc. um die Werte in den gewünschten Java-Datentyp zu konvertieren.

3.2.4 Document-Header

Der erste Property-Block in einem NDF-Dokument ist der sog. *Document-Header.* Er beinhaltet die beiden im obigen Beispiel gezeigten Properties *ndfDocVersion, ndfDocEncoding* und *ndfDocCreated.* Diese müssen am Anfang eines jeden NDF-Dokuments erscheinen, da der *NdfParser* zwingend die Versions-Information benötigt - vorher kann er keine Strukturen einlesen. Der Dokument-Header kann beliebig viele zusätzliche, kundenspezifische Properties beinhalten.

3.2.5 Document-Encoding

Die Header-Property *ndfDocEncoding* gibt an, in welchem Encoding das Dokument geschrieben wurde und gelesen wird. Wird kein Encoding angegeben, so wird das Standard-Encoding der betreffenden Plattform verwendet. Um NDF-Dokumente plattformunabhängig zu halten wird das *UTF-8* Encoding empfohlen, welches einen guten Kompromiss aus Dateigröße und Performance darstellt.

3.3 Kommentare

Kommentare werden vom Parser grundsätzlich überlesen und nicht ausgewertet, also auch nicht an den Handler übergeben. Sie dienen lediglich der Kommentierung der geschriebenen Daten.

3.3.1 Line-Comment

Line-Comments werden mit dem Zeilen-Kommentarzeichen '**#**' eingeleitet und gelten bis zum Zeilenende. Das Kommentarzeichen muss dabei nicht am Zeilenanfang stehen. Daher ist es möglich, auch einzelne Properties zu kommentieren, wie im obigen Beispiel gezeigt. Innerhalb von Strukturen werden Kommentare automatisch vom *NdfWriter* eingerückt.

3.3.2 Block-Comment

Block-Comments werden mit dem Block-Kommentarzeichen '**$**' eingeleitet und gelten auch über mehrere Zeilen hinweg, bis ein weiteres '**$**' Zeichen gefunden wird. Innerhalb von Strukturen werden Block-Kommentare automatisch vom *NdfWriter* eingerückt – und zwar alle Zeilen des Kommentarblocks.

Der *NDFWriter* schreibt *Block-Comments* zur besseren Übersicht mit einer einleitenden Kopf- und einer abschließenden Fuss-Separatorzeile, wie das folgende Beispiel zeigt. Beide können - falls gewünscht - einen eingebetteten Titel haben. Länge und Zeichen des Separators sind frei wählbar.

```
      $---Start of block comment--------------
      This is a multi-line block-comment.
      Note that the text is automatically
      indented according to the current
      indentation level during write.
      ----------------------------------------$
```

3.4 OBJECT-Struktur

Eine Strukturdefinition wird am beginnenden '@' Zeichen erkannt – hier *@OBJECT*. Die *OBJECT*-Struktur speichert die Zustandsinformation eines beliebigen Objektes. Es hat einen Namen und besteht im allgemeinen aus einer Property-Liste und kann beliebig viele weitere Strukturen (werden unten beschrieben) beinhalten. Die allgemeine Syntax ist:

```
@OBJECT Name { <property-list> <other-structure> ... }
```

Hier ein einfaches Beispiel:

```
@OBJECT Customer {
      <
      street = "Märchenweg 10"
      zipcode = 89766
      city = "Münchhausen"
      lastContactTs="2012-09-13 12:25:10"
      projects=["APP-SUITE" "NET-SUITE" "DYNAMIT"]
      >
      ...
}
```

Die *OBJECT* Struktur erhält einen Namen (hier *Customer*), der typischerweise einem Java-Datentyp entspricht. Der optionale Property-Block <...> definiert Eigenschaften des Objekts, die i.d.R. den Instanzvariablen einer Klasse entsprechen. Der restliche Datenblock bis zur schließenden Klammer '}' enthält ggf. weitere Objekt-Strukturen in beliebig tiefer Schachtelung. Das im folgenden Beispiel gezeigte Objekt *Sample* beinhaltet weitere Objekte vom Typ *Position*:

```
@OBJECT Sample {
      < positionCount=2 >
      @OBJECT Position {
```

```
            <x=100 y=20>
      }
      @OBJECT Position {
            <x=100 y=20>
      }
      @TEXT Comment {
            [^|]
            This text obtains to the object.
            The TEXT structure is explained below.
            |
      }
}
```

Eine OBJECT Struktur ist die einzige Struktur, die aus anderen Strukturen quasi zusammengebaut werden kann.

3.5 LIST-Struktur

Eine LIST-Struktur dient der Speicherung einer Liste von Werten. Die allgemeine Syntax ist:

```
 4 @LIST Name { <property-list> value value value … }
```

Die Struktur besitzt einen Namen und enthält eine optionale Liste von Properties, gefolgt von einer beliebig langen Liste von Datenwerten. Für Letztere gilt die gleiche Syntax wie für Property-Werte. Auch Listen-Werte werden hier unterstützt. Das folgende Beispiel zeigt eine einfache Liste der im Esprit-Framework zur Verfügung stehenden Log-Level:

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

4.1 TABLE-Struktur

Eine TABLE-Struktur dient der Speicherung relationaler Daten, ähnlich wie in einer relationalen Datenbank. Die allgemeine Syntax ist:

```
@TABLE Name { <property-list> [header-list] row-values; …}
```

Die Struktur besitzt einen Namen und enthält eine optionale Liste von Properties, gefolgt von einer Liste der Spaltennamen in eckigen Klammern '[…]'. Daraufhin folgen beliebig viele Datenzeilen, die mit ';' abgeschlossen sind. Die Anzahl der Datenwerte muss stets mit der Anzahl der Spaltennamen übereinstimmen. Für die Datenwerte gilt die gleiche Syntax wie für Property-Werte. Auch Listen-Werte werden hier unterstützt. Das folgende Beispiel zeigt eine einfache Tabelle:

```
@TABLE Adresses {
      < primaryKey = [Country Zipcode] >

      [Country Zipcode Street       City]
       de      70654   "Waldweg 5"  "Isselshausen";
       de      70655   "Feldweg 1"  "Hasenheim";
}
```

Die Anzahl der Datenzeilen ist – ähnlich wie bei einer relationalen Datenbank – nicht vorgegeben, sondern erst nach vollständigem Lesen aller Tabellendaten bekannt.

4.2 ARRAY-Struktur

Eine ARRAY-Struktur dient der Speicherung von Arrays. Anders als bei einer Tabelle hat ein Array eine von vornherein festgelegte Größe und alle Werte sind vom gleichen Datentyp. Die Struktur definiert einen Namen und kann ebenfalls eine optionale Property-Liste haben. In eckigen Klammern '[…]' folgt dann ein Zahlenwert für die Größe, bzw. eine Liste von Größen bei multi-dimensionalen Arrays. Dahinter steht die Liste der Datenwerte, die mit einem ';' als Dimensionsabschluss terminiert sein muss. Für die Datenwerte gilt die gleiche Syntax, wie für Property-Werte. Lediglich Wertelisten werden hier nicht unterstützt. Die allgemeine Syntax ist:

```
@ARRAY Name { <property-list> [dimension-list] value, value ...; value, value,...; }
```

Das folgende Beispiel zeigt ein drei-dimensionales Array mit unterschiedlichen Dimensionsgrößen und Datentypen:

```
@ARRAY 3Dim {
      [10 24 3]

      0 1 2 3 4
      5 6 7 8 9;

      0.00   1.00   2.00   3.00   4.00
      5.00   6.00   7.00   8.00   9.00
      10.00  11.00  12.00  13.00  14.00
      15.00  16.00  17.00  18.00  19.00
      20.00  21.00  22.00  23.00;

      "good morning" "good evening" "good night";
}
```

In diesem Falle besteht die erste Dimension aus 10 *int*-Werten, gefolgt von 24 *double*-Werten, wiederum gefolgt von 3 *String*-Werten. Die Anzahl der pro Zeile geschriebenen Werte kann im *NdfWriter* eingestellt werden, ebenso wie das Dezimalformat von Fließkommazahlen. Jede Dimension ist mit einem ';' Zeichen abgeschlossen.

4.3 TEXT-Struktur

Eine TEXT-Struktur dient der Speicherung von gewöhnlichem Text. Die allgemeine Syntax ist:

```
@Text Name { <property-list> [LF EOT] text... <LF> text ... <EOT> }
```

Dabei steht LF für ein Sonderzeichen, welches einen Zeilenumbruch (**L**ine**F**eed) repräsentiert, EOT steht für ein Sonderzeichen, welches das das Ende des Textes markiert (**E**nd**O**f**T**ext).

➔ Beide Zeichen LF und EOT *müssen* angegeben werden und dürfen im zu schreibenden Text nicht vorkommen. Aus technischen Gründen müssen sie Sonderzeichen des ASCII-Codes unterhalb von 127 sein.

Das folgende Beispiel zeigt eine einfache TEXT Struktur:

```
@TEXT Alphabet {
      [^~]
      the quick brown fox
      jumps over the lazy
      dog
      ~
}
```

Hier wurde LF=^ und EOT=~ gesetzt und markieren damit Zeilenumbrüche bzw. das Textende. Es ist irrelevant, ob die Textzeilen eingerückt oder zusätzliche Leerzeichen oder Zeilenumbrüche enthalten, vielmehr ist er als Fließtext anzusehen. Das nächste Beispiel zeigt, wie ein Text gezielt in Abschnitte eingeteilt werden kann:

```
 @TEXT MultipleParagraphs {
```

```
      < font="Courier" style="BOLD" size=12 >
      [^|]
      This is the first paragraph
      consisting of two lines.
      ^
      This is the second paragraph
      consisting of another two lines.
      ^
      This is the last paragraph.
      |
}
```

Hier wurde das LF Zeichen verwendet, um den Text in Absätze zu gliedern. LF und EOT sind aber selbst nicht Bestandteil des Textes. Das Beispiel zeigt darüberhinaus, dass eine TEXT-Struktur - so wie andere Strukturen auch - mit beliebigen Eigenschaften versehen werden kann.

5 Zusammengesetzte Strukturen

Alle oben beschriebenen Strukturen (*Properties, OBJECT, TABLE, ARRAY*) können in einem NDF-Dokument auf Top-Ebene nacheinander in beliebiger Reihenfolge auftauchen. Eine *OBJECT*-Struktur ist die Einzige, die ihrerseits auch andere Strukturen beinhalten kann.

5.1 Eingebettete Strukturen

Das folgende Beispiel zeigt das Einbetten von Strukturen in übergeordneten Strukturen. Objekte können beliebig viele andere Strukturen von beliebigem Typ beinhalten. Hier wird eine einfache Liste von Adressen gespeichert:

```
@OBJECT AddressBook {
      <owner=Harry>
      @OBJECT Address {<
            id=1
            street="Löffelweg 5"
            zipcode=72186
            city="Hasenheim"
      >}
      @OBJECT Address {<
            id=2
            street="Stachelweg 10"
            zipcode=71376
            city="Igelshausen"
      >}
}
```

Die gleiche Information hätte man etwas kompakter auch so abspeichern können:

```
@OBJECT AddressBook {
      <owner=Harry>
      @TABLE Addresses {
            [id street           zipcode city]
             1  "Löffelweg 5"    72186   "Hasenheim";
             2  "Stachelweg 10"  71376   "Igelshausen";
      }
}
```

Analog können natürlich auch ARRAY-Strukturen innerhalb einer OBJECT-Struktur gespeichert werden.

5.2 Hierarchische Strukturen

OBJECT-Strukturen können beliebig tief ineinander geschachtelt werden. Damit sind alle Arten von

Baum-Strukturen abbildbar, wie im folgenden Beispiel gezeigt. Objekte, die keine weiteren *OBJECT*-Strukturen enthalten bilden dabei die Blattknoten des Baumes.

```
@OBJECT Bilder {
      @ OBJECT 2010 {
            @OBJECT Januar { … }
            @OBJECT Februar { … }
            …
      }
      @OBJECT 2011 {
            @OBJECT Januar { … }
            …
      }
}
```

5.3 Properties mit Struktur-Werten

Besondere Möglichkeiten ergeben sich daraus, dass Property-Werte auch Strukturen sein können! Die folgende Property definiert eine X/Y-Position, die von einem *NdfHandler* beispielsweise auf den Java-Datentyp *Point* abgebildet werden könnte:

```
<
position = @OBJECT Position {
               <x=100 y=25>
          }
>
```

Ebenso können Property-Werte auch *ARRAY* oder *TABLE*-Strukturen sein. Das folgende Objekt *Contact* beinhaltet Properties mit verschiedenen Struktur-Werten:

```
@OBJECT Contact {
      <
      name="Rainer Büsch"
      email=rainer.buesch@esprit-systems.de
      # The following property has a value list
      hobbies = ["Saxophon" "Lindy Hop" "Häusle bauen"]
      # The following property has an object value
      pc = @OBJECT Pc {
            < os=Windows7 diskSpace=500GB cpus=8 >
      }

      # The following property has a table value
      memberships = @TABLE Clubs {
            [club website]
            "Saxophon Ensemble" nuart.org;
            "Java User Group"   jugs.de;
      }
      # The following property has an array value
      jobDates = @ARRAY JobDates {
            <comment="geplant bis ende des Jahres">
            [4]
            # Note that dates must be written in the java neutral format
            2012-10-15  2012-10-29  2012-11-12  2012-11-26;
      }
      >
      @TEXT Comment {
            [^|]
            Writing and parsing NDF is a lot of fun!!
            |
      }
}
```

Es ist für einen *NdfHandler* sehr einfach, aus dieser Information ein Java-Objekt namens *Contact* aufzubauen, welches Instanzvariablen vom Typ *Pc*, *List<Club>* und *Date[]* besitzt. Eine entsprechende Java-Klasse, die diese Information aufnimmt, könnte z.B. folgendermaßen aussehen:

```
public class Contact {
     private String name = "Rainer Buesch";
     private String email = "rainer.buesch@esprit-systems.de";
     private String[] hobbies = { "Saxophon", "Lindy Hop", "Häusle bauen" };

     private Pc pc = new Pc("Windows7", 500, 8);
     private List<Club> memberships= new ArrayList(); // needs to be filled
     private Date[] jobDates = new Date[4];           // needs to be filled
     private String comment = "Writing and parsing NDF is a lot of fun!!";
     …
}
```

6 Erweiterbarkeit

NDF ist im Prinzip frei erweiterbar. Zu den Hauptstrukturen *OBJECT, TABLE ARRAY* und *TEXT* könnten relativ leicht weitere hinzu erfunden werden. Denkbar wäre beispielsweise eine Struktur namens *LINK* zum Speichern von Referenzen auf externe Dateien. Die Syntax könnte etwa folgendermaßen aussehen:

```
@LINK name { <property-list> [target-url] description-text }
```

Das folgende hypothetische Beispiel zeigt einen in NDF eingebetteten Link auf eine im Internet verfügbare Datei:

```
@LINK EspritLogo {
     <
     mimeType = image/gif
     dataSize = 5096
     >
     [ http://esprit-systems.de/downloads/esprit/EspritLogo.gif ]
     This is an image file showing the Esprit logo. Note that the
     Server needs to be available in order to download the image data.
}
```

7 Vergleich mit XML

Zum Vergleich von NDF und XML wurde im folgenden Beispiel die gleiche Information einmal mit NDF und dann mit XML geschrieben.

Hier die Information im NDF Format:

```
# This is a NDF file written by the NdfWriter
< ndfDocVersion="1.6" ndfDocCreated="2012-09-28 22:31:02" >

@OBJECT Database {
     < databaseName="traindb" >

     @TABLE course {
          < primaryKey=[course_id] >

          [course_id course_name                  days  price]
          "J1"       "Java Fundamentals"            5  3250.0;
          "J2"       "Java Application Programming" 5  3900.0;
          "J3"       "Java Performance Tuning"      5  4200.0;
     }
}
```

Hier die Information im XML Format:

```
<!-- This is a XML file written by the XmlWriter -->
<?xml version="1.0" encoding="UTF-8" created="2007-11-18 13:35:22" ?>
<!DOCTYPE database SYSTEM "database.dtd">
<database name="traindb">
  <table name="course">
    <dbobject name="course">
      <field name="course_id" primaryKey="true">J1</field>
      <field name="course_name">Java Fundamentals</field>
      <field name="days">5</field>
      <field name="price">3250.0</field>
    </dbobject>
    <dbobject name="course">
      <field name="course_id" primaryKey="true">J2</field>
      <field name="course_name">Java Application Programming</field>
      <field name="days">5</field>
      <field name="price">3900.0</field>
    </dbobject>
    <dbobject name="course">
      <field name="course_id" primaryKey="true">J3</field>
      <field name="course_name">Java Performance Tuning</field>
      <field name="days">5</field>
      <field name="price">4200.0</field>
    </dbobject>
  </table>
</database>
```

7.1 Datenmasse

Man sieht deutlich, dass das NDF Format wesentlich kompakter und übersichtlicher ist. Allein die Tatsache, dass in XML ein Tag-Name sich im schließenden Tag wiederholt, ist eine im Prinzip unnötige Redundanz. Da XML keine Blockdatenstrukturen unterstützt, muss für jeden Datensatz die gesamte Meta-Information wiederholt geschrieben werden. Dies bläht die zu schreibende Datenmasse erheblich auf und verschlechtert das Verhältnis von Information zu Meta-Information.

7.2 Konsistenzprüfung

In der XML Datei findet man am Anfang eine Referenz auf eine sog. DTD-Datei (Document Type Definition). Sie beinhaltet eine Beschreibung der Keywort-Struktur, anhand derer beim Lesen der Information eine Konsistenzprüfung durchgeführt werden kann.

Diesbezüglich ist die Philosophie bei NDF eine andere: Der *NdfWriter* lässt es erst gar nicht zu, dass inkonsistente Daten geschrieben werden. Daher kann die Konsistenzprüfung beim Lesen im Prinzip entfallen.

7.3 Root-Element

Bei NDF können beliebig viele Strukturen aneinander gereiht werden. Es gibt nicht – wie bei XML – ein Root-Element, in das alles eingebettet werden muss. Daher ist das NDF-Format auch geeignet zum Schreiben von ständig wachsenden Daten, wie z.B. strukturierten Logging Dateien, die leicht auswertbar sein sollen.

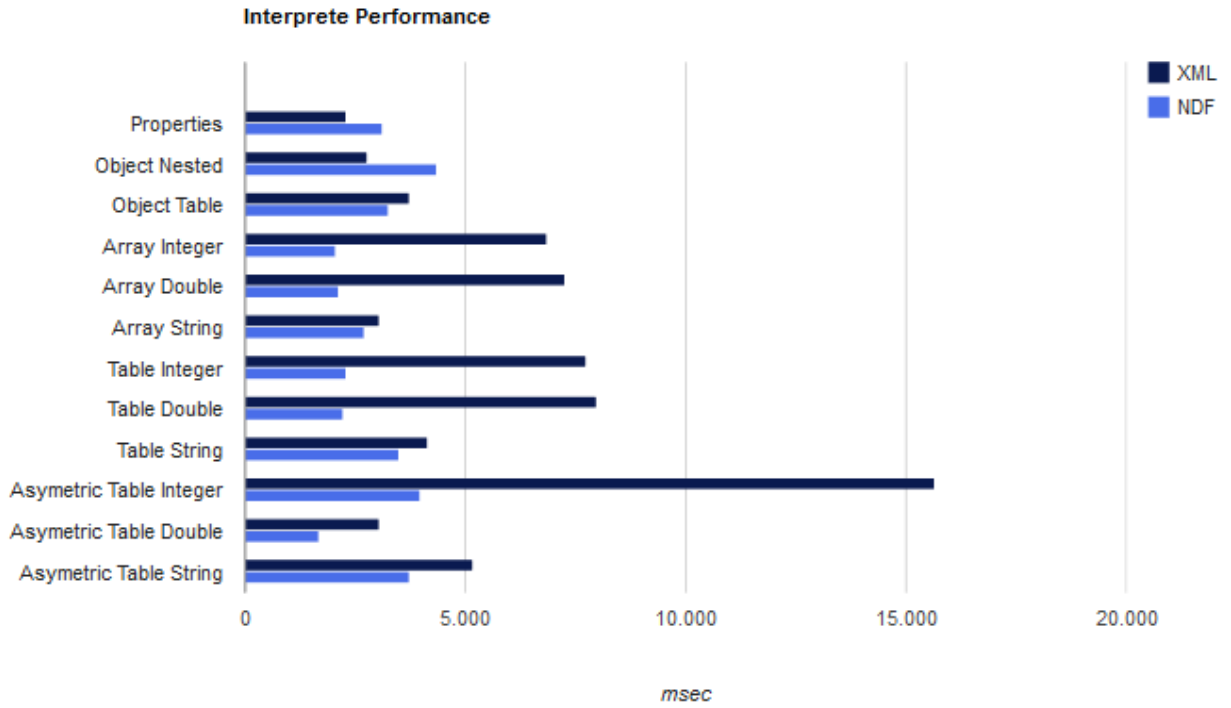
7.4 Performance

Bei der Enwicklung des *NdfWriters* und -*Parsers* wurde höchsten 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. Beide nutzen die Java NIO-Klassen für das IO-Processing und sind somit hoch performant.

Das folgende Bild zeigt einen XML/NDF Performance-Vergleichstest, bei dem jeweils eine 100MB Datei mit verschiedenen Datentypen eingelesen und ausgewertet wurde[1].



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

8 Fazit

NDF ist ein neutrales Format, in dem alle üblichen Datenstrukturen in übersichtlicher und gut lesbarer Form gespeichert werden können. Im Gegensatz zu XML ist es sehr kompakt und benötigt nur wenig Overhead an Meta Informationen. Da es neben *OBJECT*- auch *TABLE*- und *ARRAY*-Strukturen unterstützt, ist es insbesondere zur Speicherung von Massendaten geeignet. Alle NDF-Strukturen können darüber hinaus mit beliebiger Meta-Information in Form von *Properties* versehen werden, so dass es möglich sein sollte, alle in der Informatik typischen Informationsstrukturen in NDF vollständig abzubilden und mit hervorragender Performance zu schreiben und zu lesen.

1 Ergebnisse eines Studien-Projekts bei der DHBW-Stuttgart (Duale Hochschule Baden Würtemberg) im Januar 2014.