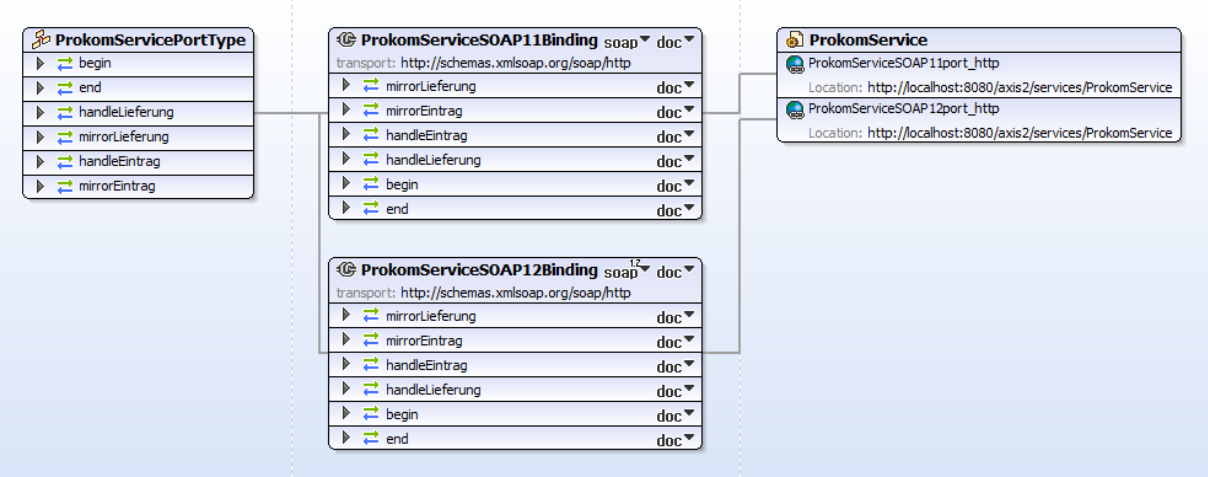
NeRTHUS XML Object Framework



Entwicklerdokumentation

Webservice, XML-Databinding, Marshalling und Unmarshalling

NeRTHUS Informationssysteme

Inhalt

Aktuelle Version: DRAFT 0.3	Datum: 29.04.2011

1 Einleitung

XML/EDIFACT, cXML, GAS-XML, OCIT, ACI ACRIS u.v.a.m. – standardisiertes XML hat sich für den normierten Geschäftsnachrichtenaustausch durchgesetzt. In einer Serviceorientierten Architektur (SOA) wird überwiegend mit Webservices kommuniziert, das hauptsächlich angewandte Protokoll ist SOAP (bzw. REST), die Daten werden als XML modelliert und mit XML-Schema definiert, Webservice werden mit WSDL eindeutig beschrieben. Prozesse und Abläufe werden als fachliche Workflows in BPMN (oder EPK) abgebildet, verschiedene Webservices im technischen Workflow über BPEL bzw. BPMN orchestriert.

Grundlage dieser Technologie ist die objektorientierte Verarbeitung von XML, d.h. der Übergang von XML-Daten in Objekte und umgekehrt, mit dem Ziel, die Entwicklungsarbeit insoweit zu automatisieren, dass man sich auf die Lösung der Aufgabenstellung konzentrieren kann, d.h. die Fachlogik implementieren, mit Datenobjekten arbeiten, ohne sich dabei mit XML oder dessen Modellierung (JDOM, AXIOM, …) auseinander setzen zu müssen.

Muss man mit seinem System mehrere unterschiedliche XML-Standards unterstützen, so fällt einem schnell auf, dass es sehr unterschiedliche Ausprägungen und Modellierungsprinzipien gibt, wofür unterschiedliche XML-Databinding-Frameworks (JAXB, CASTOR, XMLBeans) mehr oder weniger geeignet sind. In der einen Modellierung werden fast nur XML-Elemente verwendet, andere bevorzugen Attribute, dann werden Elemente als Container für Sequenzen konsequent genutzt usw..
Zudem entstehen bei der Generierung der Klassenbibliotheken aus den diversen Frameworks eine Vielzahl von Klassen, die nah am XML orientiert sind, weniger an den Fachobjekten.

Bei GAS-XML gibt es z.B. ein Vererbungsprinzip von Attributen, d.h. die Werte gelten in allen Unterelementen, wo ein gleichnamiges Attribut definiert, aber nicht explizit angegeben ist (Währungen, Zeitstempel, phys. Größen usw.). Durch dieses Vererbungsprinzip ist auch ein Wert für das „unendliche“ offene Ende eines Zeitraums bedingt (end = ““, Leerstring).

Das hier beschriebene NeRTHUS XML Object Framework (NeXOF) hat einen Vorläufer in dem GAS-XML Object Framework der Steria Mummert Consultung AG (www.gas-xml.de), das fast in der gesamten Gaswirtschaft im Umfeld von GAS-X verwendet wird und gleichzeitig mit GAS-XML als Kommunikationsstandard entwickelt wurde.
Das Thema „XML-Kommunikation“ drückt sich aber durch eine Vielzahl von Standards aus. So war es an der Zeit, die gewonnenen Erfahrungen mit GAS-XML zu verallgemeinern, mit moderner Programmiertechnologie (ab Java 5) zu verbinden und auf andere Standards anzuwenden. Damit wurde eine Neuentwicklung notwendig, in die alle Erfahrungen strukturiert einfließen und die Effizienz gesteigert werden konnte.

Das Ergebnis: ein neues XML Object Framework, offen und lizenzfrei, getestet und bewährt durch viele bekannte Einsatzszenarien und jahrelangen Erfahrungen. Der Mehrwert besteht in der klaren OO-Modellierung von XML-Informationseinheiten als Java-Klassen, dem AXIS2-Databinding, der Flexibilität, der Beherrschung der Zeit (UTC), der einfachen Verwendung von Aufzählungstypen, dem aus GAS-XML übernommenen, möglichen, Vererbungsprinzip, der Standardorientiertheit in den Basisklassen (ISO 4217, 3166-1, 639, 1000, 31, 8601), u.v.a.m..

2 Bestandteile

Das NeXOF basiert im Wesentlichen auf AXIOM (StAX) und Java 5/6, wobei in der Anwendung keinerlei Berührung mit den Basis-API's, z.B. AXIOM, notwendig ist.
Folgende Bibliotheken gehören zu dem Umfang:

- nexof.jar – NeRTHUS XML Object Framework (NeXOF)
- activaion.jar
- axiom-api.jar
- axiom-dom.jar
- axiom-impl.jar
- commons-logging.jar
- jaxen.jar
- log4j.jar
- stax-api.jar
- wstx-asl.jar
- xml-apis.jar

Darüber hinaus existiert ein Databinding zu AXIS2 (nexof-db-axis2.jar), um das NeXOF in der Webservice-Implementierung anwenden zu können.

Als Beispiele dienen die voll funktionsfähigen und sich im Einsatz befindlichen Implementierungen zu:

- OCIT® (XML Verkehrsinformationen – Parking)
- GAS-XML (Auszug betriebsw. Daten – Buchungen, Kapitel Klassenbibliothek)
- cXML (Commerce XML)
- PROKOM (XML Eintragsdaten DeTeMedien)

die als Vorlage zum Verständnis und in der Anwendung zur Anpassung dienen sollen.

Die Quellen des NeXOF werden nur zur Verfügung gestellt:

- in Kooperation, d.h. wenn Optimierungen und Weiterentwicklungen sicher geteilt werden
- im Rahmen von kommerziellen Vereinbarungen.

3 Grundlagen

3.1 Annotations

Mit Java 5 wurde der Sprachumfang um Annotations (Runtime) erweitert, die sofort umfangreiche Anwendung, z.B. bei Hibernate (z.B. JPA) und Spring 3.0 fanden. Damit wurde es möglich, die Zuordnung (Mapping) von Java-Klassen und Instanzvariablen zu Datenobjekte und Attributen (XML, Datenbank) einfach und übersichtlich zu gestalten.
Für das NeXOF sind folgende Annotations (com.nerthus.annotation) definiert:

- XMLEntity(tag, type)
- XMLAttribute(name, type, format, defaultValue, inheritable, unusedBy)
- XMLElement(tag, container, type, format, nullable)
- XMLContentValue(type, format)
- XMLObject(tag, container, nullable)
- XMLObjectList(tag, container, nullable)
- XMLNode(tag, container, nullable)

3.1.1 XMLEntity

Die Annotation XMLEntity kennzeichnet Klassen, die direkt einer XML-Informationseinheit zugeordnet sind. Typischer Weise sind XML-Informationseinheiten solche XML-Fragmente, die aus Sicht der Objektorientierung als Objekte verstanden werden können und die als Geschäftsobjekte Bestandteil eines Datenaustausches sind, z.B. <ocit-c:Parking>.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface XMLEntity {
        String tag();
        String type() default "element"; // [element, document]
}
```

Das Attribut „tag“ kennzeichnet den Namen des XML-Elements, dem die Klasse zugeordnet ist.

Attribut	Beschreibung
tag	Name Element der zugeordneten XML-Informationseinheit
type	Typ [element (default), document] XML-Element oder XML-Dokument

Ein besonders Element ist das Root-Element eines XML-Dokumentes, deren Klasse durch XMLEntity mit dem Attribut type = “document“ angegeben wird.

```
/**
* Businessclass for PROKOM Document <prokom:lieferung>
*/
@XMLEntity(tag = "lieferung", type = "document")
public class Lieferung extends NEEnterpriseObject implements NEXMLInterface {
```

Damit wird die Interpretation und Ausgabe des XML-Prolog und der Namespaces gesteuert.

```
/**
* BusinessObject for <ocit-c:Parking>
*/
@XMLEntity(tag = "Parking")
public class Parking extends NEEnterpriseObject implements NEXMLInterface {
```

I.d.R. gibt es eine 1:1-Beziehung von Element und Klasse, selbst wenn die XML-Informationseinheit im Schema als ComplexType definiert ist, wird der Typ nur unter einem einzigsten Elementnamen (Tag) verwendet. Für den ganz seltenen Fall (<prokom:eintrag_pflegen>/<prokom: eintrag_anlegen>, <gas-ml:partner>/<gas-xml:businesspartner>) kann man a) die Klassen dublizieren oder b) die Instanzvariable NEEnterpriseObject.tag benutzen.

Klassen mit der Annotation XMLEntity als Datentyp, die in anderen Klassen verwendet werden, werden mit der Annotation XMLObject gekennzeichet.

3.1.2 XMLAttribute

Die Annotation XMLAttribute kennzeichnet eine Instanzvariable für ein XML-Attribut im XML-Element, das der Klasse als XML-Informationseinheit zugeordnet ist.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface XMLAttribute {
        String name();
        String defaultValue() default "";
        String type() default "xs:string";
        String format() default "";
        boolean inheritable() default false;
        String unusedBy() default "";
}
```

Attribut	Beschreibung
name	Name des XML-Attributes
defaultValue	Default-Wert des XMLAttributes (wenn Attribut optional)
type	Schema Datentyp für Typemapping (long, xs:dateTime)
format	Format-Pattern Zeitangaben, Werte (DD.MM.YYYY, ###.##0)
inheritable	Attribut-Wert wird geerbt (aus übergeordneten Elementen)
unusedBy	Attribut wird nicht ausgegeben, wenn angegebenes != null

Die Verwendung ist relativ einfach, es gilt jedoch ein paar nahe liegende Regeln für die Verwendung der Datentypen zu beachten.

XSD-Datentyp	use	Java-Datentyp
xs:int, xs:short	optional	Integer
xs:int, xs:short	required	int
xs:double, xs:float	optional	Double
xs:double, xs:float	required	double
xs:dateTime	optional/required	NECalendarDate
...	...	...

Einerseits werden einige Datentypen im Java nicht benutzt, z.B. Float und Short.
Ist ein Atrribut optional, d.h. es muss nicht angegeben werden, so ist ein nullable Datentyp nahe liegend (java.lang.Boolean statt boolean). Bei Datum und Zeitstempel wird auf NECalendarDate gemappt. Ist ein Attribut vom Typ xs:dateTime required, so wird in der Java-Klasse die Instanzvariable bei der Deklaration instanziiert.

Beispiel <prokom:eintragsversion> Schema:

```
<xsd:attribute name="gueltig_von" type="pk:t_date" use="required"/>
```

Eintragsversion.java:

```
@XMLAttribute(name = "gueltig_von", format = "YYYY-MM-DD")
protected NECalendarDate gueltig_von = new NECalendarDate();
```

In seltenen Fällen kann es zu einer unterschiedlichen Ausgabe (XOR) von Attributen kommen, z.B. bei der Preisliste <gas-xml:priceset> werden Preise als Liste indiziert zu den Attributen */priceset/@values-per-period und */priceset/@period ohne Zeitangaben mit */price/@i (**$begin_{Einzelwert} = begin_{Zeitreihe} + (i * period / values\text{-}per\text{-}period)$**), oder aber ohne diese Attribute mit Zeitangaben */price/@begin und */price/@end ausgegeben.
Um das zu ermöglichen wird unusedBy verwendet:

```
@XMLEntity(tag = "price")
public class Price extends NEEnterpriseObject implements NEXMLInterface {

        @XMLAttribute(name = "i")
        protected Integer index = null;

        ...

        @XMLAttribute(name = "begin", inheritable = true, unusedBy = "index")
        protected NECalendarDate begin = null;

        @XMLAttribute(name = "end", inheritable = true, unusedBy = "index")
        protected NECalendarDate end = null;
```

3.1.3 XMLElement

Die Annotation XMLElement kennzeichnet eine Instanzvariable für ein XML-Unterelement (Datenelement) des XML-Elements, das der Klasse als XML-Informationseinheit zugeordnet ist, der Wert ist der Textcontent des Datenelements. Das Datenelement und das mögliche Container-Element selbst hat keine Attribute (für den Fall Klasse implementieren und als XMLObject deklarieren)!

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface XMLElement {
        String tag();
        String container() default "";
        String type() default "xs:string";
        String format() default "";
        boolean nullable() default true;
}
```

Attribut	Beschreibung
tag	Name des XML-Unterelementes (Datenelement)
container	Datenelement als Unterelement eines Container-Elementes
type	Schema Datentyp für Typemapping (long, xs:dateTime)
format	Format-Pattern Zeitangaben, Werte (DD.MM.YYYY, ###.##0)
nullable	Container-Element ist optional (minOccurs = “0“)

Die Verwendung gleicht dem XMLAttribut.

XSD-Datentyp	use	Java-Datentyp
xs:int, xs:short	optional	Integer
xs:int, xs:short	required	int
xs:double, xs:float	optional	Double
xs:double, xs:float	required	double
xs:dateTime	optional/required	NECalendarDate
...	...	...

Beispiel <gas-xml:person> Schema:

```
<xs:complexType name="personDef">
	<xs:annotation>
		<xs:documentation>individual person data</xs:documentation>
	</xs:annotation>
	<xs:sequence>
		...
		<xs:element name="lastname" type="xs:string" minOccurs="0"/>
		<xs:element name="gender" minOccurs="0">
			<xs:simpleType>
				<xs:restriction base="xs:string">
					<xs:enumeration value="f"/>
					<xs:enumeration value="m"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:element>
```

Person.java:

```
@XMLElement(tag = "lastname")
protected String lastname = null;

@XMLElement(tag = "gender")
protected EnumGender gender = null;
```

Container sind für Elemente eher selten, weil exakt ein Datenelement unter einem Containerelement in der Modellierung sehr eigen ist. Auf eine Annotation XMLElementList wurde verzichtet, in dem Fall ist XMLObject zu nutzen.

3.1.4 XMLContentValue

Die Annotation XMLContentValue kennzeichnet eine Instanzvariable für den Textcontent des XML-Elements, das der Klasse als XML-Informationseinheit zugeordnet ist.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface XMLContentValue {
	String type() default "string";
	String format() default "";
}
```

Attribut	Beschreibung
type	Schema Datentyp für Typemapping (long, xs:dateTime)
format	Format-Pattern Zeitangaben, Werte (DD.MM.YYYY, ###.##0)

Eine Annotation XMLContentValue kann nur exakt einmal in einer Klasse logischer Weise erfolgen!

Beispiel <ocit-c:DriveIn> Schema:

```
<xsd:element name="DriveIn" minOccurs="0">
	<xsd:complexType>
		<xsd:simpleContent>
			<xsd:extension base="xsd:int">
				<xsd:attribute name="interval" type="xsd:unsignedInt"
					use="optional" default="15"/>
			</xsd:extension>
		</xsd:simpleContent>
	</xsd:complexType>
</xsd:element>
```

DriveIn.java:

```
@XMLAttribute(name = "interval", defaultValue = "15")
protected int interval = 15;

@XMLContentValue()
protected Integer value = null;
```

3.1.5 XMLObject

Die Annotation XMLObject kennzeichnet eine Instanzvariable, die als Klasse für eine andere XML-Informationseinheit implementiert ist.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface XMLObject {
        String tag() default "";
        String container() default "";
        boolean nullable() default true;
}
```

Attribut	Beschreibung
tag	XML-Element-Name des Objektes (**keine Angabe = xs:any**)
container	Objekt als Unterelement eines Container-Elementes
nullable	Container-Element ist optional (minOccurs = “0“)

Als Beispiel dient die Verwendung von <ocit-c:DriveIn> in <ocit-c:Parking><Data><Value> ParkingValue.java:

```
/**
* BusinessObject for <ocit-c:Parking><Data><Value>
*/
@XMLEntity(tag = "Value")
public class ParkingValue extends NEEnterpriseObject implements NEXMLInterface {

        @XMLElement(tag = "Occupacy")
        protected Integer occupacy = null;

        ...

        @XMLObject(tag = "DriveIn")
        protected DriveIn driveIn = null;

        @XMLObject(tag = "DriveOut")
        protected DriveOut driveOut = null;

        @XMLElement(tag = "Type")
        protected EnumParkingType type = null;
```

Eine Besonderheit besteht darin, wenn in der Annotation das Attribut „tag“ nicht angegeben wird. Das entspricht dem XML-Schema für xs:any, d.h. an der Stelle kann ein beliebiges Objekt/Element im Definitionsumfang (##targetNamespace) folgen, der Datentyp ist dann NEXMLInterface:

```
@XMLObject()
protected NEXMLInterface object = null;
```

3.1.6 XMLObjectList

Die Annotation XMLObjectList kennzeichnet eine Liste/Array eines Objekts, das als Klasse für eine andere XML-Informationseinheit implementiert ist.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface XMLObjectList {
        String tag() default "";
        String container() default "";
        boolean nullable() default true;
}
```

Attribut	Beschreibung
tag	XML-Element-Name der in der Liste enthaltenen Objekte (**keine Angabe = xs:any**)
container	Liste als Unterelement eines Container-Elementes
nullable	Container-Element ist optional (minOccurs = “0“)

Die Liste wird als Java-Liste typisiert implementiert:

```
/* <prokom:eintragsbestand><eintrag> */
@XMLObjectList(tag = "eintrag", container = "eintragsbestand")
protected Vector<Eintrag> eintragsbestand = new Vector<Eintrag>();
```

Eine Besonderheit besteht darin, wenn in der Annotation das Attribut „tag“ nicht angegeben wird. Das entspricht dem XML-Schema für xs:any, d.h. an der Stelle können beliebige Objekte/Elemente im Definitionsumfang (##targetNamespace) folgen, der Datentyp der Typisierung ist dann NEXMLInterface:

```
@XMLObjectList()
protected Vector<NEXMLInterface> anyList = new Vector<NEXMLInterface>();
```

3.1.7 XMLNode

Die Annotation XMLNode stellt eine Besonderheit im NeXOF dar. Der Hintergrund besteht darin, dass einige XML-Spezifikationen, wie z.B. GAS-XML (<gas-xml:extension>, Informationseinheiten generell definieren, aber deren Erweiterung durch beliebiges, in anderen Spezifikationen definiertes XML, vorsehen. Das entspricht dem XML-Schema <xs:any namespace="##other"> und stellt eine sehr effektive Möglichkeit dar, nicht für jede Ausnahme das verallgemeinerte Datenmodell anzupassen.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface XMLNode {
        String tag() default "";
        String container() default "";
        boolean nullable() default true;
}
```

Attribut	Beschreibung
tag	XML-Element-Name (i.d.R. nicht angebbar)
container	in einem Container-Element
nullable	Container-Element ist optional (minOccurs = "0")

Bisher gibt es nur eine Verwendung von XMLNode, in <gas-xml:extension>:

```
@XMLEntity(tag = "extension")
public class Extension extends NEEnterpriseObject implements NEXMLInterface {

        // namespaced extensions ##other
        @XMLNode()
        protected NEXMLNode extension;
```

Der Datentyp ist NEXMLNode, ein Interface, das in Ableitung von

```
org.apache.axiom.om.impl.llom.OMElementImpl;
```

implementiert ist. Damit wird ab dort das XML mit der AXIOM-API verarbeitet, weil dafür keine Modellierung existiert.

3.2 Datentypen

Für Werte (Attribute, Content Elemente) gilt folgendes allgemeines Typmapping:

XSD-Datentyp	use/minOccurs	Java-Datentyp
xs:dateTime	optional/required	NECalendarDate
enumeration	optional/required	enum implements NEEnumType
xs:string	optional/required	String
xs:int, xs:short	optional/0	Integer
xs:int, xs:short	required/1	int
xs:long	optional/0	Long
xs:long	required/1	long
xs:double, xs:float	optional/0	Double
xs:double, xs:float	required/1	double
xs:boolean	optional/0	Boolean
xs:boolean	required/1	boolean
xs:int, xs:double, xs:float, xs:boolean, ...	optional/0 mit default=wert `<xsd:attribute name="anzahl" type="xs:int" use="optional" default="15"/>`	einfacher Datentyp (int, double, boolean, ..) Defautwert-Zuweisung Deklaration Klasse, z.B.: `protected int anzahl = 15;`
...	...	...

Ist z.B. ein Atrribut optional, d.h. es muss nicht angegeben werden, so wird ein komplexer Datentyp (Integer statt int, Double statt double), der null sein kann, verwendet. Eine Ausnahme stellt die Angabe eines Defaults dar, dabei wird im Java der Wert einem einfachen Datentyp (int, double, boolean, …) deklarativ zugewiesen.

Alle Datum- und Zeitstempel-Angaben werden auf NECalendarDate gemappt.

3.2.1 Aufzählungstypen – NEEnumType

Im XML-Schema wird sehr häufig der Wertebereich durch Aufzählung festgelegt. Dafür werden im NeXOF eigene Typ-Klassen implementiert, die von Java-Typ enum sind. Damit sie im Rahmen des Frameworks funktionieren, implementieren sie ein Interface NEEnumType.

Beispiel <ocit-c:direction> Schema:

```
<xsd:element name="direction" minOccurs="0">
	<xsd:annotation>
		<xsd:documentation>direction</xsd:documentation>
	</xsd:annotation>
	<xsd:simpleType>
		<xsd:restriction base="xsd:string">
			<xsd:enumeration value="oneSided"/>
			<xsd:enumeration value="doubleSided"/>
		</xsd:restriction>
	</xsd:simpleType>
</xsd:element>
```

EnumDirection.java

```
public enum EnumDirection implements NEEnumType {

	ONESIDED("oneSided"),
	DOUBLESIDED("doubleSided");

	private String value;

	EnumDirection(String value){
		this.value = value;
	}

	public static EnumDirection getValue(String value){
		if (value == null) { return null; }
		for (EnumDirection pt : EnumDirection.values()) {
			if(pt.value.equals(value)){
				return pt;
			}
		}
		return null;
	}

	public String getValue() {
		return value;
	}
}
```

Innerhalb des NeXOF-NEEnumType wird ausgenutzt, dass XML Text ist, d.h. das alle Werte als String interpretierbar sind. Damit wird dieser Typ auch zum Wertmapping benutzt, d.h. man kann unterschiedliche 1:1-Ausprägungen damit beherrschen.

In dem folgenden Beispiel wird lediglich der Integer-Wert zugeordnet.

Beispiel //prokom:eintrag_pflegen/@liefergrund Schema:

```
<xsd:attribute name="liefergrund">
	<xsd:simpleType>
		<xsd:restriction base="pk:t_nummer_1">
			<xsd:enumeration value="1"/>
			<xsd:enumeration value="2"/>
			<xsd:enumeration value="3"/>
			<xsd:enumeration value="4"/>
		</xsd:restriction>
	</xsd:simpleType>
</xsd:attribute>
```

EnumLiefergrund.java

```
public enum EnumLiefergrund implements NEEnumType {

	EINS(1, "1"),
	ZWEI(2, "2"),
	DREI(3, "3"),
	VIER(4, "4");

	private String value;
	private Integer intValue;

	EnumLiefergrund(Integer value, String str){
		this.intValue = value;
		this.value = str;
	}
	public static EnumLiefergrund getValue(String value){
		if (value == null) { return null; }
		for (EnumLiefergrund pt : EnumLiefergrund.values()) {
			if(pt.value.equals(value)){
				return pt;
			}
		}
		return null;
	}
	public static EnumLiefergrund getValue(Integer value){
		if (value == null) { return null; }
		for (EnumLiefergrund pt : EnumLiefergrund.values()) {
			if(pt.intValue.equals(value)){
				return pt;
			}
		}
		return null;
	}
	public String getValue() {
		return value;
	}
	public int getValueAsInteger() {
		return intValue;
	}
}
```

3.2.2 Datum, Zeitstempel – NECalendarDate

Das NECalendarDate ist eine Komfortklasse und dient dem Umgang mit Zeitangaben (ISO 8601, UTC). Im NeXOF wird damit der Datentyp xs:dateTime gemappt.

(siehe: http://www.w3.org/TR/NOTE-datetime)

```
   Year:
      YYYY (eg 2011)

   Year and month:
      YYYY-MM (eg 2011-07)

   Complete date:
      YYYY-MM-DD (eg 2011-07-16)

   Complete date plus hours and minutes:
      YYYY-MM-DDThh:mmTZD (eg 2011-07-16T19:20+01:00)

   Complete date plus hours, minutes and seconds:
      YYYY-MM-DDThh:mm:ssTZD (eg 2011-07-16T19:20:30+01:00)

where:
     YYYY = four-digit year
     MM   = two-digit month (01=January, etc.)
     DD   = two-digit day of month (01 through 31)
     hh   = two digits of hour (00 through 23) (am/pm NOT allowed)
     mm   = two digits of minute (00 through 59)
     ss   = two digits of second (00 through 59)
     TZD  = time zone designator (Z or +hh:mm or -hh:mm)
```

UTC-Zeitangaben sind immer in Weltzeit, d.h. bezogen auf Greenwich Mean Time (GMT), ein Beispiel gleichwertiger Angaben für einen Zeitpunkt:

2011-12-24T03:00-02:00	2011-12-24T06:00+01:00	2011-12-24T08:00+03:00

Die Abbildung in lokaler, gesetzlicher Zeit ist im XML nicht zwingend. Für den internationalen Geschäftsnachrichtenverkehr, d.h. über Zeitzonen-Grenzen hinweg, sind es also normalisierte Zeitangaben. NECalendarDate verfügt über eine Methode localize(TimeZone tz), womit die Zeitangabe für Darstellungsgaufgaben (Zeitzonen-Offset, Tag, Stunde) umgewandelt werden kann.

Mit einer Formatangabe (XMLAttribute, XMLContentValue) können auch andere Zeitformate erzeugt bzw. gelesen werden.

4 Anwendung NeXOF

4.1 Implementierung ausgewählter XML-Standards

Es hat sich bewährt, für jeden benötigten XML-Standard ein eigene Komponente (JAR), basierend auf dem NeXOF, zu implementieren. Diese besteht aus der Klassenbibliothek für die den XML-Infomationseinheiten entsprechenden Klassen und einem DataMarshaller, der Mappingoptionen für ausgewählte XML-Informationseinheiten, sogenannte Top-Level-Objekte, enthält. D.h. der DataMarshaller verwaltet alle XML-Element-Bezeichnungen (Tags), mit denen kommuniziert wird, damit nicht die ganze Klassenbibliothek nach der Annotation XMLEntity durchsucht werden muss. Weiterhin werden 2 Parser implementiert, einer für SAX-, der andere für StAX-Parsing.

Folgende Eclipse-Projekte dienen als Beispiele:

- nexof_finance (Subset GAS-XML)
- nexof_ocit (Subset OCIT-C)
- nexof_prokom (Eintragsdaten DeTeMedien)
- nexof_cxml (Commerce XML)

Für den Einsatz innerhalb der AXIS2-Webservices ist nur die Klassenbibliothek notwendig, weil das Unmarshalling/Marshalling von der AXIS-Engine übernommen wird und der zugehörige DataMarshaller Bestandteil des NeXOF-AXIS2-Databinding ist und über eine Datei konfiguriert wird.

4.1.1 Klassenbibliothek

Die Klassenbibliothek entsteht unmittelbar aus dem XML-Schema, in einfachster Weise mittels Stylesheet. Dabei entsteht i.d.R. eine Klasse für ein XML-Dokument, mehrere Klassen für jeweils eine XML-Informationseinheit (XML-Element, Complex Type) und Klassen für Aufzählungstypen (NEEnumType, Enumeration), je nach Struktur des Schemas.

Ich persönlich benutze äußerst ungern Code-Generatoren, weil ich keinen Überblick habe, was ich im Detail bekomme, und bevorzuge die manuelle Implementierung. Das ist auch relativ schnell und einfach machbar.

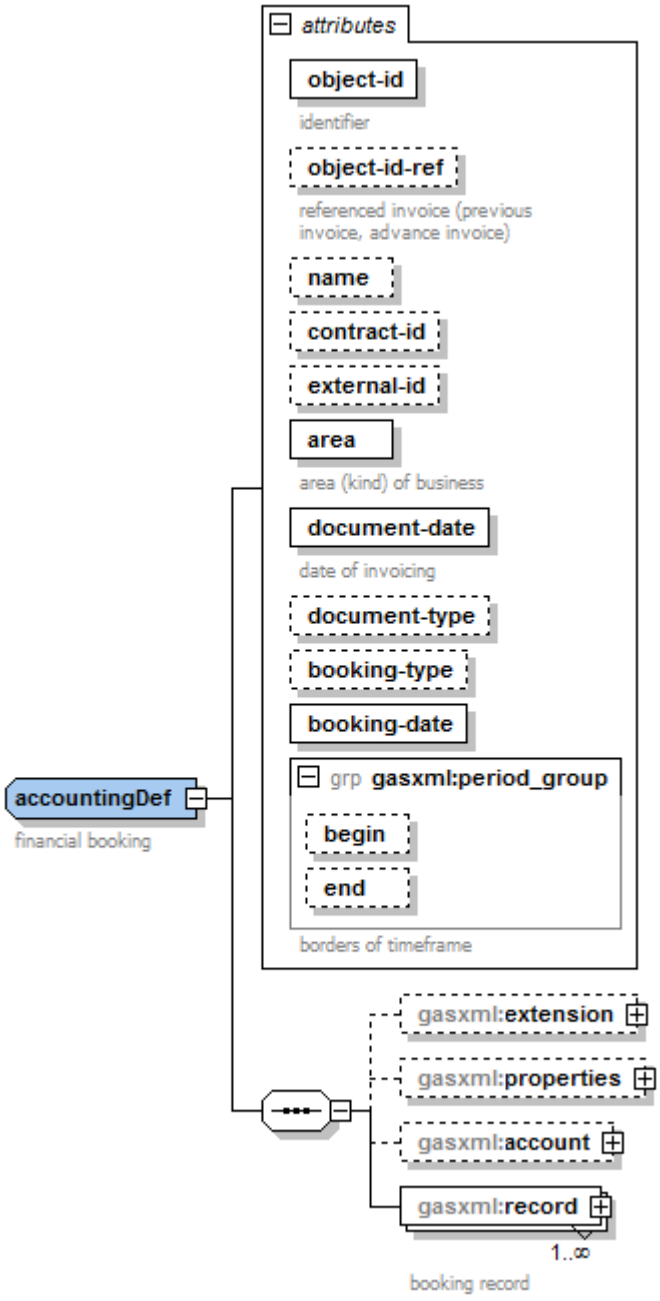
Als Beispiel dient <gas-xml:accountig> (nexof_finance), Buchungssätze aus GAS-XML, weil diese Aufgabenstellung branchenunabhängig ist und ein typischer praxisrelevanter Anwendungsfall in einer SOA, z.B. Richtung SAP FiBu über SAP PI mit Buchungsbestätigung, darstellt.

Buchungen sind eine Folge von Rechnungen (Eingang wie Ausgang) und mussten parallel zur <gas-xml:invoice> modelliert werden, da der Informationsgehalt doch sehr unterschiedlich ist bzw. sein kann.

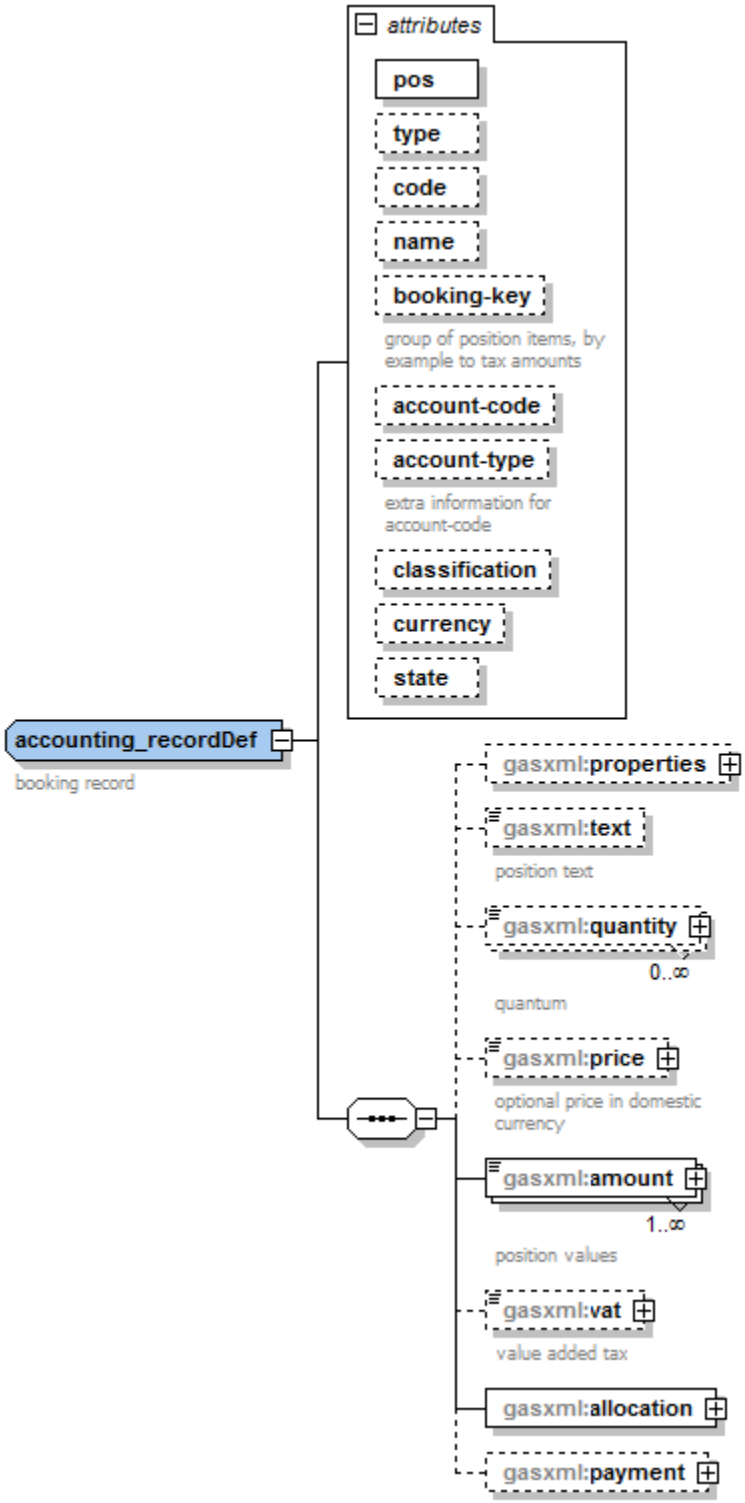
Die Informationseinheiten <gas-xml:accounting> bzw. <gas-xml:invoice> spielen darüber hinaus noch in vielen andern Schnittstellen, z.b. zu SAP BW/BO, u.U. eine Rolle.

Informationen zu GAS-XML sind über die Webseite www.gas-xml.de erhältlich. Es können Handbücher (Abrechnungsdaten, Messdaten) angefordert und das Schema (www.gas-xml.de/*3.2*/gas-xml.xsd) herunter geladen werden.

Beispiel Buchung <gas-xml:accounting>



mit den einzelnen Buchungssätzen <gas-xml:record>



Dieses Beispiel ist zugegebenermaßen sehr komplex, d.h. man könnte die Implementierung auch anhand eines einfacheren Beispiels erläutern. Aber einfache Beispiele sind meistens konstruiert und Praxisnähe dient auch dem Verständnis.

Ein dazugehöriges XML könnte folgendes Aussehen haben:

```
<accounting object-id="00002576" object-id-ref="4711003" contract-id="4711" document-type="invoice"
                document-date="2011-08-01T00:00:00+01:00" booking-type="standard"
                name="Warenlieferung" area="3" booking-date="2011-08-02T00:00:00+01:00">
        <account number="0471108015" iban="DE21500100600471108015" owner="Baumarkt Pudelig Potsdam" >
                <bank name="Postbank Frankfurt am Main" code="50010060" swift-code="PBNKDEFF"/>
        </account>
        <record pos="1" booking-key="A8" account-code="8316" classification="170" currency="EUR">
                <text>Steckdose 16A</text>
                <quantity unit="piece" type="document">3.0</quantity>
                <price currency="EUR" market="Potsdam" unit="piece">12.00</price>
                <amount type="document" currency="EUR" currency-type="contractual">36.00</amount>
                <vat code="MwSt16" type="vat" unit="%">16.0</vat>
                <allocation account-code="3334044" cost-centre="KST0816" cost-object="KPL0816/1"/>
                <payment target="2011-09-01T00:00:00+01:00" method="transfer"/>
        </record>
        ...
</accounting>
```

Daraus ergibt sich, das folgende Klassen zu implementieren sind:

- Accounting (Buchung)
- Account (Bankverbindung) mit Bank
- Record (Buchungssatz)
- Quantity (Menge)
- Price (Preis)
- Amount (Betrag)
- Vat (Steuer)
- Allocation (Kontierung)
- Payment (Zahlung)

Dazu werden noch einige Aufzählungstypen benötigt:

- Currency (vorhanden: Standard NeXOF, EnumCurrency, ISO 4217)
- Booking-type
- …

Im Grunde gibt es hier nur ein Top-Level-Objekt, nämlich Accounting. Alle anderen Objekte existieren nur als Bestandteil (z.B. Price in PriceSet, InvoicePosition und eben AccountingRecord), d.h. sie werden mit @XMLObject bzw @XMLObjectList gekennzeichnet verwendet.

*/accounting/record/text ist nur ein XML-Element mit Textcontent, also ohne Attribute und Unterelementen, und wird deshalb im AccountingRecord als @XMLElement gekennzeichnet sein.

Günstiger Weise fängt man mit der Implementierung bei den Unterklassen an, da diese in höheren verwendet werden, z.B. die Kontierung <gas-xml:allocation>.

```
/**
 * BusinessObject for <gas-xml:allocation>
 */
@XMLEntity(tag = "allocation")
public class Allocation extends NEEnterpriseObject implements NEXMLInterface {

	@XMLAttribute(name = "type")
	protected String type;

	@XMLAttribute(name = "account-code")
	protected String account_code;

	@XMLAttribute(name = "cost-centre")
	protected String cost_centre;

	@XMLAttribute(name = "cost-object")
	protected String cost_object;

	@XMLAttribute(name = "division")
	protected String division;

	public Allocation(
				NEXMLDataMarshaller dm,
				NEXMLNode node,
				NEXMLContext context) {
		super();
		this.initWith(dm, node, context);
	}
```

Wir sehen, die Informationseinheit hat nur Attribute, der Konstruktor in der Form DataMarshaller, XMLNode, Context) ist Pflicht. Auf die Darstellung der Getter- und Setter-Methoden wird, der Einfachheit wegen, verzichtet.

Der Buchungssatz hat ein Attribut mit der Bezeichnung type (record/@type) als Enum:

```
public enum EnumPositionType implements NEEnumType {

	UNDEFINED("debit"),
	CREDIT("credit"),
	DEBIT("debit"),
	TAX("tax"),
	FEE("fee"),
	SUBTOTAL("sub total"),
	TAXTOTAL("tax total"),
	NETTOTAL("net total"),
	GROSSTOTAL("gross total");

	private String value;

	EnumPositionType(String value){
		this.value = value;
	}
```

```
        public static EnumPositionType getValue(String value) {

                for (EnumPositionType pt : EnumPositionType.values()) {
                        if((pt.value != null) && pt.value.equals(value)){
                                return pt;
                        }
                }
                return null;
        }

        public String getValue() {
                return value;
        }
}
```

Damit ist es dann möglich, eine komplexere Klasse zu implementieren, den Buchungssatz:

```
/**
 * BusinessObject for <gas-xml:accounting><record>
 */
@XMLEntity(tag = "record")
public class AccountingRecord extends NEEnterpriseObject implements NEXMLInterface {

        @XMLAttribute(name = "pos")
        protected int pos;

        @XMLAttribute(name = "type")
        protected EnumPositionType type;

        @XMLAttribute(name = "code")
        protected String code;

        @XMLAttribute(name = "booking-key")
        protected String booking_key;

        @XMLAttribute(name = "account-code")
        protected String account_code;
        ...
        @XMLElement(tag = "text")
        protected String text = null;

        @XMLObjectList(tag = "quantity")
        protected Vector<Quantity> quantities = new Vector<Quantity>();

        @XMLObject(tag = "price")
        protected Price price = null;

        @XMLObjectList(tag = "amount")
        protected Vector<Amount> amounts = new Vector<Amount>();

        ...

        public AccountingRecord(NEXMLDataMarshaller dm,
                                        NEXMLNode node, NEXMLContext context) {
                super();
                this.initWith(dm, node, context);
        }
```

4.1.2 DataMarshaller

Zu jedem Modul, jeder mit NeXOF implementierten Klassenbibliothek für einen XML-Standard, existiert ein eigener DataMarshaller in Ableitung des NEXMLDataMarshaller.

```
public class DataMarshaller extends NEXMLDataMarshaller {

        public static final String CL = "GAS-XML 3.2";

        /** Creates a new instance of DataMarshaller */
        public DataMarshaller(NEXMLContext context) {
                this();
                this.context = context;
        }

/       ** Creates a new instance of DataMarshaller */
        public DataMarshaller() {
                super();

                businessClasses.put("accounting",
                                        "com.nerthus.gasxml.object.Accounting");
                businessClasses.put("invoice",
                                        "com.nerthus.gasxml.object.Invoice");
                businessClasses.put("pricelist",
                                        "com.nerthus.gasxml.object.Pricelist");

        }
}
```

Diese Implementierung dient der festen Zuordnung von Element-Namen (Tag) und Klasse, damit die Klassenbibliothek beim Unmarshalling nicht vollstandig nach @XMLEntity-Einträgen durchsucht werden muss. Diese Zuordnung ist innerhalb einer Klassenbibliothek fest definiert.

4.1.3 NEXMLObjectHandler, NEXMLObjectParser

Im NeXOF ist noch ein sehr praktischer Mechanismus hinterlegt, der es erlaubt, in beliebigen XML-Dokumenten, ab einem definierten Element, bekannte Informationseinheiten zum Unmarshalling zu parsen.

Der NEXMLObjectParser schickt eine Nachricht (unmarshalled(object, context)) mit dem Objekt an ein anderes Objekt, das das NEXMLObjectHandler-Interface implementiert hat.

Ein Beispiel aus der PROKOM-Implementierung:

```
// this == ObjectHandler, nur Einträge unter <lieferung> interessieren
// der DataMarshaller kennt nur Einträge
objectParser = new NEXMLObjectParser(new DataMarshaller(), this, "lieferung");
objectParser.parseFile(absolutefilename);
```

und die Methode des ObjectHandlers, in dem Falle „this“:

```
public void unmarshalled(NEXMLInterface object, NEXMLContext context) {
        // könnte ja doch später in Weiterentwicklung was anderes kommen
        if (object instanceof Eintrag) {
                Eintrag eintrag = (Eintrag) object;
                // was damit machen
                ….
        }
        return;
}
```

4.2 AXIS2 Webservice

Was nutzt das beste XML-Object-Framework, wenn es bei Webservice-Implementierungen nicht einsetzbar ist und dann dort mit einfachen XML (AXIOM, JDOM) gearbeitet werden muss oder dort andere Frameworks (XMLBeans, Castor, …) genutzt werden müssen?

AXIS2 ist sicherlich eines der stärksten Webservice-Frameworks auf dem Markt, das diverse WS-*-Standards unterstützt und sich hervorragend professionelle, d.h. stabile und effiziente, Webservice entwickeln lassen.

Für die Implementierung von Webservice unter AXIS2 sei auf die verfügbare Literatur verwiesen, insbesondere auf das schon berühmte Buch „Java Web Services mit Apache Axis2“ von Thilo Frotscher, Marc Teufel und Dapeng Wang, das neben dem Studium der Axis2-Quellen sehr geholfen hat.

4.2.1 NeXOF AXIS2 Data Binding

Für das Data Binding steht die Datei nexof-axis2-databinding.jar zur Verfügung, die auf das NeXOF (nexof.jar) aufsetzt und mit AXIS2 (*/WEB-INF/lib) verteilt werden muss. Ein bestimmter Webservice nutzt das Data Binding, wenn in seiner service.xml die MessageReceiver des NeXOF-Axis2-Databinding-Frameworks (com.nerthus.axis2.rpc.receivers) genutzt werden:

- NERPCMessageReceiver
- NERPCInOnlyMessageReceiver
- NERPCInOutAsyncMessageReceiver

service.xml:

```
<messageReceivers>
        <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
                class="com.nerthus.axis2.rpc.receivers.NERPCInOnlyMessageReceiver"/>
         <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
                class="com.nerthus.axis2.rpc.receivers.NERPCMessageReceiver"/>
</messageReceivers>
```

Im NeXOF-Axis2-Databinding-Framework ist nicht bestimmt, mit welcher konkreten Klassenbibliothek gearbeitet wird, d.h. welcher DataMarshaller (aus OCIT, aus GAS-XML) für das Unmarshalling benötigt wird.

Aus diesem Grund ist ein spezieller DataMarshaller Teil des Databindings, der seine Informationen aus einer Konfigurationsdatei erhält, wo also die Zuordnung von Elementen zu Klassen, im Gegensatz der DataMarshaller einer bestimmten Klassenbibliothek, nicht fest definiert ist. Diese Flexibilität birgt viele Vorteile, insbesondere, wenn sogenannte Speziallösungen (Sonderlocken) erforderlich werden, die genau ein Kunde in einem ganz speziellen Fall braucht. Dann kann man nämlich auch besondere Klassen nutzen, die diese Aufgabenstellung, abweichend vom Standard, repräsentieren und man damit Regel von der Ausnahme trennen kann.

Die Konfigurationsdatei für den DataMarshaller des NeXOF-Axis2-Databinding-Frameworks heißt nexof_databinding.xml und wird in */WEB-INF/ von AXIS2 abgespeichert.

Beispiel für OCIT-, PROKOM- und GAS-XML-Webservices:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
	<!-- OCIT-C -->
	<entry key="Parking">com.nerthus.ocit.object.Parking</entry>
	<!-- PROKOM -->
	<entry key="lieferung">com.nerthus.prokom.object.Lieferung</entry>
	<entry key="eintrag">com.nerthus.prokom.object.Eintrag</entry>
	<entry key="eintrag_anlegen">com.nerthus.prokom.object.EintragAnlegen</entry>
	<entry key="eintrag_pflegen">com.nerthus.prokom.object.EintragPflegen</entry>
	<entry key="eintrag_loeschen">com.nerthus.prokom.object.EintragLoeschen</entry>
	<entry key="geodaten">com.nerthus.prokom.object.Geodaten</entry>
	<entry key="eintragsversion">com.nerthus.prokom.object.Eintragsversion</entry>
	<entry key="hierarchie">com.nerthus.prokom.object.Hierarchie</entry>
	<!-- GAS-XML Finance -->
	<entry key="pricelist">com.nerthus.gasxml.finance.object.Pricelist</entry>
	<entry key="priceset">com.nerthus.gasxml.finance.object.Priceset</entry>
	<entry key="price">com.nerthus.gasxml.finance.object.Price</entry>
	<entry key="invoice">com.nerthus.gasxml.finance.object.Invoice</entry>
	<entry key="accounting">com.nerthus.gasxml.finance.object.Accounting</entry>
	<entry key="partner">com.nerthus.gasxml.finance.object.Partner</entry>
	<entry key="businesspartner">com.nerthus.gasxml.finance.object.Partner</entry>
</properties>
```

4.3 Beispiel PROKOM WS (AXIS2)

Die DeTeMedien liefert ab 2011 Eintragsdaten (Telefon-, Branchenbücher) im sogenannten PROKOM-Format, ein XML-Format, das mit einem XML-Schema (XSD) definiert ist. U.U. kann davon ausgegangen werden, dass Eintragsdaten über DeTeMedien hinaus zukünftig in diesem Format ausgetauscht werden.

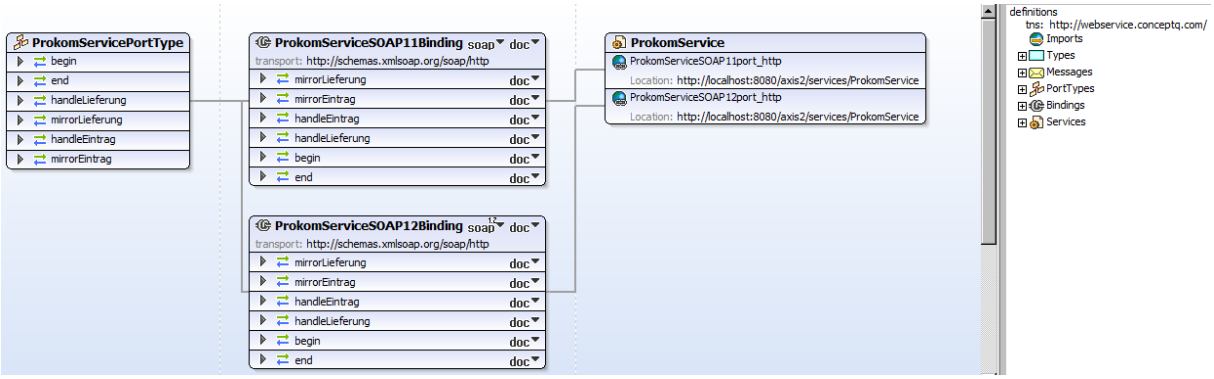
NeXOF und das AXIS2-Databinding werden konkret an einem einsatzfähigen Beispiel (www.nerthus.de/download) demonstriert, das einen komplett eingerichteten Tomcat + AXIS2 + Prokom-WS und die Eclipse-Projekte zu der Klassenbibliothek und dem Webservice enthält.

4.3.1 Webservice

In dem Eclipse-Project ws_prokom ist die Web Service-Klasse implementiert, die mittels service.xml im AXIS2 konfiguriert wird. Alle Bestandteile befinden sich bereits compiliert in dem AXIS2-Server (siehe Literatur und apache-tomcat-6.0.29\webapps\prokom-ws\WEB-INF\services\prokom-ws).

Wesentlich ist die WSDL, die eineindeutig den WebService beschreibt.

(…/WEB-INF/ProkomWebservice.wsdl)



Das Databinding, also das Mappen der XML-Informationseinheit auf die zugehörige Klasse, wird bei den Webservices durch eine Ressourcen-Datei konfiguriert, da das Webservice-Framework unabhängig von dem jeweils implementierten Objectframework (PROKOM, OCIT-C, GAS-XML, XML/EDIFACT) ist, d.h. der DataMarshaller des Prokom-Objectframeworks serverseitig keine Verwendung findet. Die Datei befindet sich in …\webapps\prokom-ws\WEB-INF\ und heißt nexof_databinding.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
   <entry key="lieferung">com.nerthus.prokom.object.Lieferung</entry>
   <entry key="eintrag">com.nerthus.prokom.object.Eintrag</entry>
   <entry key="eintrag_anlegen">com.nerthus.prokom.object.EintragAnlegen</entry>
   <entry key="eintrag_pflegen">com.nerthus.prokom.object.EintragPflegen</entry>
   <entry key="eintrag_loeschen">com.nerthus.prokom.object.EintragLoeschen</entry>
</properties>
```

In der Klasse com.nerthus.service.prokom.ProkomService werden die Webservice-Operationen als Methoden implementiert, z.B.

```
public Properties handleLieferung(Lieferung lieferung, Properties props)
                                                                    throws AxisFault {
        // HIER: unternehmensspezifische Verarbeitung
        return result;
}
```

In denen ist die Verarbeitung der erhaltenen Daten unternehmensspezifisch zu implementieren. Im Beispiel wird eine Ursprungsdatei in Lieferungen zu je 10 Einträgen gesplittet und an den Web Service mit der Operation handleLieferung() gesendet. Hier erfolgt dann die Verarbeitung z.B. mit einem Spring-Hibernate-Backend Richtung Datenbank. Die Beispielimplementierung zeigt, wie die Daten „angefasst“ werden und wie ein mögliches Result unter Verwendung von Nachrichten <nexml:messages><nexml:msg> aussehen könnte.

```
<properties name="result">
   <property name="service" value="ProkomService" />
   <property name="operation" value="handleLieferung" />
   <property name="datenbestand" value="2010-10-05T00:00:00+02:00" type="xs:datetime" />
   <property name="bestand" value="10" type="xs:integer" />
   <property name="anlegen" value="0" type="xs:integer" />
   <property name="loeschen" value="0" type="xs:integer" />
   <property name="pflegen" value="0" type="xs:integer" />
   <property name="messages">
      <messages>
         <msg object="*/eintrag[@prokom_id='203041060']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203041160']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203043260']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203046060']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203046660']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203051360']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203052060']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203053860']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203055260']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203056760']" level="I">Bestand OK</msg>
      </messages>
   </property>
   <property name="state" value="true" type="xs:boolean" />
</properties>
```

Die Verarbeitungszeit eines einzelnen Eintrags, aus einer Datei geparst, an den PROKOM-WS gesendet und von diesem in der Verarbeitung bestätigt, d.h. Response gesendet und ausgewertet, beträgt hier auf dem Testsystem ca. 1,2 ms.

```
================================================================
= WebServiceClient: Ende Import!                               =
= Bearbeitungszeit:    354698 ms                               =
= Anzahl Objekte:      303514                                  =
================================================================
```

Unter realen Bedingungen ist noch die Verarbeitungszeit des Backends und ggf. Netzwerk verursachte Zeit hinzu zu rechnen. Aber die Zeit, die das Framework als Grundlast verursacht, ist sicher zu vernachlässigen.

4.3.2 Webservice-Client

Um einen Webservice zu testen oder aber gelieferte Dateien genau so verarbeiten zu können, als würden die Daten über den Webservice geliefert, steht eine Klasse zu Verfügung:

```
com.nerthus.ws.client.batch.JobWebServiceClient
```

Diese dient für den Aufruf aus der Kommandozeile oder aber aus einem Scheduler als Batch-Job. Damit werden Dateien aus dem Dateisystem geparst (oder aus einer Mail-Inbox) und die Daten an den Web Service geschickt und ist konfigurierbar.
Die Konfiguration für die Jobs ist eine XML-Datei (imex/conf), für PROKOM beispielhaft:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<imex session="true" timeout="60000">
        <!-- config of import -->
        <import name="Eintragsdaten" desc="Import/Update Prokom">
                <file format="NULL" source-dir="C:\prokom\imex\examples"
                        filename-mask="testdaten.xml" source-mask="*"/>
                <tasks>
                        <task  endpoint="http://localhost:8080/prokom-ws/services/ProkomService"
                                object="lieferung" operation="handleLieferung" return="properties" split="10"/>
                </tasks>
                <post-processing old-dir="/tmp/old" err-dir="/tmp/err"/>
        </import>
</imex>
```

wo angegeben wird, wo die Dateien zu finden sind (source-dir) oder welcher Web Service (endpoint und operation) aufgerufen wird. Die vorhandene import_prokom.xml ist beispielhaft und dient dem Test der vorhandenen Implementierung. Die Struktur ist mit ein wenig Dokumentation dem XML-Schema imex/imex_conf.xsd entnehmbar.

```
#!/bin/ksh
#-------------------------------------------------------------
#  Aufruf WS-Client
#-------------------------------------------------------------
# --- Standard fuer UNIX Java ---
NECLASSPATH="$JAVA_HOME/lib/classes.zip"

# --- alle *.jar aus lib ---
for i in $AXIS2_HOME/WEB-INF/lib/*.jar
do
   NECLASSPATH=$NECLASSPATH:$i
done
NECLASSPATH=$NECLASSPATH:$AXIS2_HOME/WEB-INF/classes

# --- start ws-client ---
java -Daxis2.repo=$AXIS2_HOME/WEB-INF -Daxis2.xml=$AXIS2_HOME/WEB-INF/conf/axis2.xml
-Dset.JAVA_HOME=$JAVA_HOME -classpath $NECLASSPATH -Xms32m -Xmx1024m
com.nerthus.ws.client.batch.JobWebServiceClient $1 $2 $3 $4
```

5 Abschließende Bemerkung

Eine Entwicklerdokumentation kann immer nur mehr oder weniger vollständig sein. Ein grundlegendes Übel liegt auch immer darin, dass der ursprüngliche, primäre Entwickler die Dokumentation schreibt und eigentlich überhaupt keine Vorstellung hat, was der Anwender seiner Kreation an Informationen braucht. Er schätzt Sachen als trivial ein, die dann den Anwendern Kopfschmerzen bereiten oder betrachtet etwas als sehr wichtig, das die Anwender ohnehin schon ahnten und wussten.

Ein paar Stunden beratender Beistand bewirken manchmal mehr als jede Dokumentation und das sei als Unterstützung ausdrücklich angeboten. Das ist auch ein bisschen der Zweck hinter der ganzen Offenheit und Kostenfreiheit.

Trotzdem soll sich eine solche Dokumentation auch weiterentwickeln und verbessern, d.h. den Bedürfnissen der Anwender immer besser gerecht werden. Deshalb sind auch zum Thema Dokumentation alle Hinweise und Anregungen willkommen, auch wenn damit ein beratender Beistand vollkommen überflüssig werden sollte.

Die NeRTHUS Informationssysteme GmbH existiert seit 2002 nicht mehr, d.h. die wirtschaftliche Tätigkeit wurde beendet und NeRTHUS dient nur noch als persönliches Label.

Für vertragliche Vereinbarungen, Projekte mit Gewährleistung usw., existiert eine Kooperation mit der CCDM GmbH (www.ccdm.de) in Potsdam, ein Institut der Fachhochschule Brandenburg/Havel und der Projektträger der eCOMM Brandenburg (www.ecomm-brandenburg.de). Diese bundesweite Initiative „Netzwerk Elektronischer Geschäftsverkehr“ (eCOMM) passt fachlich hervorragend zu dem mit dem NeXOF technologisch getriebenen Thema des standardisierten und prozessorientierten Nachrichtenaustausches. Normierte Geschäftsnachrichten bedingen normierte Geschäftsobjekte, die Bestandteil der Nachrichten sind. Es bestehen viele standardisiert Ansätze, von EDIFACT bis ebXML.

XML ist die Technologie, SOA und BPM sind die Werkzeuge.

Potsdam, 29. April 2011