

# Position Paper: Modeling Roles in UBL

**Author:** Bill Burcham (bill\_burcham@stercomm.com)

**Date:** 3/11/02

**Filename:** draft-burcham-rolemodel-04.doc

1	Summary .....	2
2	Problem Description .....	2
3	Option 0: Role-Based Naming of Elements.....	4
4	Option 1: if tag names match then types must match .....	5
4.1	Option 1a: Hungarian Code .....	5
5	Option 2: if types match then tag names must match .....	7
6	Option 3: 1-1 Correspondence Between Types and Tag Names .....	7
7	Recommendation .....	7
7.1	Enter: Properties and Roles.....	8
8	Appendix: Sample UBL Modeling Artifact Schema .....	12
9	Appendix: Schema eg0 .....	13
10	Appendix: Schema eg1 .....	14
11	Appendix: Schema eg1a .....	15
12	References.....	15

# 1 Summary

There was much discussion NDRSC during the recent face-to-face meetings regarding possible rules regarding the cardinality between Tag Names and Types. Various options were explored and candidate rules were put to straw poll. The exercise did not result in firm resolution. At least one candidate rule is still on the docket for further discussion.

This paper presents a fairly complete exploration of the options. The exploration results in the conclusion that none of the candidate rules are viable.

That result (the absence of rule providing guidance in this area) has been termed “anarchy” by some NDR SC members. A new concept – that of *role*, is introduced to address the issue. Rules relating to the identification and definition of roles in UBL are presented.

## 2 Problem Description

The problem, as it originally arose in the NDR SC seemed to center on the cardinality between tag names and types. Given two elements:

- If the elements have the same tag name, do they have the same type?
- If the elements have the same type, do they have the same tag name?

In this section we consider various combinations of tag name/type name uniqueness. In thinking about it some more there are four top-level cases:

- If tag names match then...
- If tag names don't match then...
- If types match then...
- If types don't match then...

Then for each there are three sub-cases, e.g.

- a. If tag names match then type names *must* match
- b. If tag names match then type names *must not* match
- c. If tag names match then we can draw no conclusion regarding types (don't care)

Combining these cases (cross product) we arrive at 12 possibilities. Again, remember that for this discussion we are considering two elements here. The cases below capture possible inferences we can make given that the element's tag names match (or don't) – cases 1-6, or given that the element's types match (or don't) – cases 7-12.

case	Thing-1	Match	Thing-2	Must Match	Must Not Match	Don't care
1	Tag name	✓	Type	✓		
2	Tag name	✓	Type		✓	
3	Tag name	✓	Type			✓
4	Tag name		Type	✓		
5	Tag name		Type		✓	
6	Tag name		Type			✓
7	Type	✓	Tag name	✓		
8	Type	✓	Tag name		✓	
9	Type	✓	Tag name			✓
10	Type		Tag name	✓		
11	Type		Tag name		✓	
12	Type		Tag name			✓

The way to read this table is, e.g. case 1: if tag names (of the two elements under consideration) match then their types must match, or more formally:

$$tagname_a = tagname_b \rightarrow type_a = type_b$$

Cases 1-6 draw conclusions about type names from statements about tag names. Cases 7-12 do the reverse.

Cases 3, 6, 9, 12 correspond to the absence of any design rule – the default case should we decide to make no rule. Those cases encompass the “may match” and “may not match” cases as well. Those cases are **grayed** to show that will not be considered further.

Case 2: “if tag names match then type must not match” can be eliminated intuitively.

Cases 4, 8, 10 can be eliminated similarly. Those have been **grayed** as well.

This leaves for candidate rules, cases: 1, 5, 7, 11.

If we express cases 5 and 7 as propositions, however, we see they are identical:

$$\text{Case 5 is: } tagname_a \neq tagname_b \rightarrow type_a \neq type_b$$

In propositional form that's:  $NOT(tagname_a \neq tagname_b) OR (type_a \neq type_b)$

$$\text{Simplifying: } (tagname_a = tagname_b) OR (type_a \neq type_b)$$

$$\text{Case 7 is: } type_a = type_b \rightarrow tagname_a = tagname_b$$

In propositional form:  $NOT(type_a = type_b) OR (tagname_a = tagname_b)$

$$\text{Simplifying and rearranging: } (tagname_a = tagname_b) OR (type_a \neq type_b)$$

Similarly for cases 1 and 11:

Case 1 is:  $tagname_a = tagname_b \rightarrow type_a = type_b$

In propositional form that's:  $NOT(tagname_a = tagname_b)OR(type_a = type_b)$

Simplifying and rearranging:  $(type_a = type_b)OR(tagname_a \neq tagname_b)$

Case 11 is:  $type_a \neq type_b \rightarrow tagname_a \neq tagname_b$

In propositional form:  $NOT(type_a \neq type_b)OR(tagname_a \neq tagname_b)$

Simplifying:  $(type_a = type_b)OR(tagname_a \neq tagname_b)$

Similar derivations show that cases 2 and 8 are identical, as are cases 4 and 10. All the duplicate cases have been grayed-out in the table to show that they can be ignored.

So the only options requiring consideration are cases 1 and 5 and possibly combinations of those. These options are explored in subsequent sections.

### 3 Option 0: Role-Based Naming of Elements

An element's tag name is simply the **PropertyTerm** for that property. There is no requirement that elements sharing a tag name must also have the same content type. There is no requirement that a tag name convey anything about the content type.

This option represents the absence of any rule dictating a relationship between type and tag name. This option does not proscribe a correspondence where such a correspondence makes sense.

This option also gives guidance on tag names, to wit that they are precisely `PropertyTerm`.

Here is a sample Order document containing element names conforming to this option (the schema for this example can be found in Appendix: Schema eg0 on page 13):

```
<?xml version="1.0" encoding="UTF-8"?>
<eg0:OrderDocument xmlns:eg0="eg0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="eg0
eg0.xsd">
  <Header>
    <Buyer>
      <Name>Fred</Name>
    </Buyer>
    <Seller>
      <Name>Ted</Name>
    </Seller>
  </Header>
</eg0:OrderDocument>
```

- + Short tag names conserve space in instance documents
- + Less redundancy in tag names compared to other options. This eliminates confusing tag names like "HeaderOrderHeader"
- + The names are by definition, meaningful in the context where they occur since that's exactly what a `PropertyTerm` is meant to do. They are not required to be meaningful

outside that scope and as a result carry none of the baggage necessary for that purpose.

## 4 Option 1: if tag names match then types must match

(from Case 1) For each type there is a set of reserved tag names, usable only for elements of that type. It is ok for two elements of the same type to have different tag names so long as both names came from the list for that type.

Here is a sample Order document containing element names conforming to this option (the schema for this example can be found in Appendix: Schema eg1 on page 14):

```
<?xml version="1.0" encoding="UTF-8"?>
<eg1:OrderDocument xmlns:eg1="eg1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="eg1
eg1.xsd">
  <OrderHeader>
    <Buyer>
      <Name>Fred</Name>
    </Buyer>
    <Seller>
      <Name>Ted</Name>
    </Seller>
  </OrderHeader>
</eg1:OrderDocument>
```

Notice that the element containing the header structure is named “OrderHeader” (as opposed to simply “header”) since invoices will also have headers, but of a different type. To distinguish the two occurrences this option dictates that the tag names must differ.

- + When a (local) tag name is encountered it would be possible (using the aforementioned lists of type-to-tag-name associations) to infer the type name from the (local) tag name. If those lists were mostly short (length 1) then this might even be possible from memory.
- If the lists are short (length 1) then we have essentially devolved into a 1-1 correspondence between type and tag names. (1-1 correspondence is considered in section 6 below).
- Requires LC SC to record with each type, the list of (local) tag names used for that type, and to reconcile candidate tag names against those lists – changing candidate names when clashes occur.

This last drawback adds to the overhead of the LC SC activity. We might avoid that overhead at the expense of some redundancy in the UBL instance documents. This gives rise to a sub-option...

### 4.1 Option 1a: Hungarian Code

An important drawback of Option 1 is:

- Requires LC SC to record with each type, the list of (local) tag names used for that type, and to reconcile candidate tag names against those lists – changing candidate names when clashes occur.

One way to eliminate this procedural overhead would be to *include the type name as part of the tag name*. In C circles it has been called *Hungarian Code*<sup>1</sup>.

This option builds on **option 1** this way:

(from Case 1) For each type there is a set of reserved tag names, usable only for elements of that type. It is ok for two elements of the same type to have different tag names so long as both names came from the list for that type. The way this uniqueness is guaranteed is by including the type name as part of the tag name.

So we might structure tag names like:

[Qualifier.]PropertyTerm[.RepresentationTerm | ObjectClass]

Where ObjectClass is the type of this property=element.

There is precedent for this naming convention. RepresentationTerm is very much like an ObjectClass where the ObjectClass comes from a distinguished list, namely the list of RepresentationTerms. Option 1a would extend that naming to “non-leaf” elements.

Option 1a trades the procedural drawback of Option 1 for redundancy in UBL instance documents. It also improves on the readers ability to know the type name without resorting to other documentation.

Here is a sample Order document containing element names conforming to this option (the schema for this example can be found in Appendix: Schema eg1a on page 15):

```
<?xml version="1.0" encoding="UTF-8"?>
<eg1a:OrderDocument xmlns:eg1a="eg1a" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="eg1a
eg1a.xsd">
  <HeaderOrderHeader>
    <BuyerParty>
      <Name>Fred</Name>
    </BuyerParty>
    <SellerParty>
      <Name>Ted</Name>
    </SellerParty>
  </HeaderOrderHeader>
</eg1a:OrderDocument>
```

An invoice document would similarly carry a HeaderInvoiceHeader.

- + When a (local) tag name is encountered the type name is immediately apparent.
- Tag names are long. This makes UBL instance documents larger.

---

<sup>1</sup> See [http://shamit.virtualave.net/charles\\_simonyi.htm](http://shamit.virtualave.net/charles_simonyi.htm) for a discussion with Charles Simonyi of Microsoft – creator of this naming style. In the original Hungarian Code the type name was actually encoded for brevity. While it is certainly reasonable to consider two kinds of Hungarian Code: abbreviated and non-abbreviated, in this paper we consider only the latter. Abbreviation runs afoul of other design rules already adopted by the SC.

- Since an element’s tag name carries its type name, that tag name must change whenever the element’s type changes. An important outcome is that processing code (e.g. stylesheets) will have to change as well.
- As usual with structured names there is the possibility of redundancy (e.g. HeaderOrderHeader )

## 5 Option 2: if types match then tag names must match

(from Case 5<sup>2</sup>) For each type there is a single tag name. Every element of that type uses that tag name. However, two types may share a tag name.

This option precludes the creation/use of tag names tailored for their role in a particular type. Instead, given the type of the element, you’d be stuck with a particular tag name.

- If a type contained two (local) elements of the same type, you’d have to either break this rule (and give one element a different tag name), or use *position* to distinguish the meaning of the two elements (in the context of the type).
- This option allows for elements of two different types to share tag names, so it is not possible to infer the type from the tag name when reading an instance document.

## 6 Option 3: 1-1 Correspondence Between Types and Tag Names

We can also express these situations simply in terms of their cardinality. For instance in Case 1 the cardinality is: *type* (1-0..\*) *tag name*, and for Case 5: *type* (1..\*-0..1) *tag name*.

In order to arrive at a 1-1 cardinality we would have to take these two rules together.

(from Case 1 and 5 taken together): each tag name corresponds to one Type and all elements of a particular Type share the same tag name.

- This option is tantamount to global tag names. That option has already been rejected by the SC.

## 7 Recommendation

During the face-to-face there was some discussion of high level design drivers. Three important ones that kept coming up were:

- Readability of an instance document
- Ease of instance construction
- Ease of instance processing

---

<sup>2</sup> In the NDRSC we were discussing case 7 (disguised as it’s twin – case 5). When it is worded as in 5 it may be harder to understand. This may have led to confusion.

When considered against those three drivers, none of the candidate rules (options 1-3 above) has significant value. Therefore we recommend **Option 0**.

The illusory benefits of options 1 or 2 taken in isolation devolve quickly into option 3 (1-1 correspondence of type name to tag name). The latter has already been rejected by the NDR SC.

Perhaps the story doesn't end there though. There is may be a need to capture recurring patterns of structure *use*. The problem with global element names is that in our zeal to capture usage patterns we enforce *everywhere* the overhead of formulating a globally unique and meaningful name.

We already capture recurring *structures/semantics* with XML types (corresponding to CC ABIE's). Is there a way to capture (and identify) recurring *usage* patterns while not imposing the use of a globally unique name for every single element in the schema?

## **7.1 Enter: Properties and Roles**

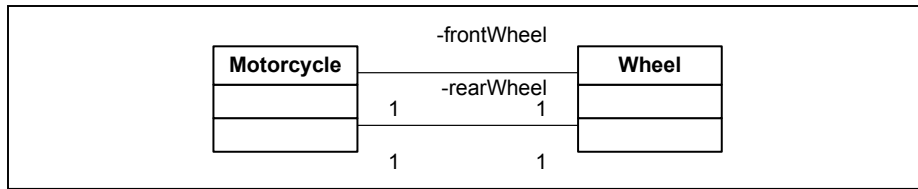
*Roles*<sup>3</sup> are an essential part of many modeling languages, such as UML and Entity Relationship or ER. Unfortunately roles and associations seem to be absent in the UBL model, and the UN Core Components [CC-UN] and ISO 11179 [NAMING-ISO] models upon which it is based.

---

<sup>3</sup> There is a difference between a role and an association. Generally, a role is one side of an association. A role is a one-way mapping. An association is usually 2-way, but may in general be n-way. The “arity” of the association corresponds to the number of roles in that association. Also an association in most modeling methods may also carry its own data and is usually given a “first class” identifier whereas roles generally are simply named (and described) concepts.



A simple example will illustrate the role concept. The following picture depicts a motorcycle with two wheels, front and rear.



It is common practice when *realizing* a model including roles (in a modeling language such as UML or ER), in particular implementation language (such as XSD, Java or SQL) to use the role name in the implementation. This mapping is very natural, for instance:

- In Java, role names become names of fields (of reference types) (see section 23.4.2 on page 300 of [UML-APPLY])
- In SQL, role names become names of foreign key fields
- In XML role names become element names (see section *Mapping UML Compositions* on page 107 of [XML-UML])

So mapping the UML model into XML might yield a scheme like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:complexType name="Motorcycle">
    <xs:all>
      <xs:element name="FrontWheel" type="Wheel"/>
      <xs:element name="RearWheel" type="Wheel"/>
    </xs:all>
  </xs:complexType>
  <xs:simpleType name="Wheel">
    <xs:restriction/>
  </xs:simpleType>
</xs:schema>

```

Notice how type Motorcycle has e.g. an element called FrontWheel of type Wheel. Notice also, how the role name “frontWheel” from the model was used to arrive at the element name “FrontWheel” in the schema.

Much of the contention over element naming in UBL stems from the imprecise treatment of “properties” in the UN Core Components Technical Specification [CC-UN]. While that specification *does* talk extensively about “property terms” – which are part of a “dictionary entry name” for a “data element” (a la [NAMING-ISO]), we are left to *infer* the existence and makeup of a “first class” property concept.

The term “property” is used often in that specification, but it is never formally defined<sup>4</sup>. Further, it never appears in any of the conceptual diagrams. Little wonder therefore, that the concept of *role* as discussed here would be entirely absent, tied up as it is with the concept of property.

---

<sup>4</sup> Additionally, the term “child field” is used in some of the examples in that specification. That term is used synonymously to “property”, and is also left undefined.

So here we are (in particular the LC SC) trying to build analysis artifacts. We are trying to give “property terms” to things. What things are we trying to give them to? Well CC doesn’t tell us! Let us propose:

**P0: The UBL model must include the concept of *property*.** Property is the model element named by a *property term* in the same way as a *BIE* or a *CC* is the model element named by an object class (name).

Further, once we identify and describe these properties, what shall we call them? Could a set of rules around role definition satisfy our need to capture recurring component usage patterns (and name them)? Perhaps the central tenet would be:

**P1: Role-based element/property naming:** *every* element’s tag name should reflect the role played by that element’s content *relative to* the XSD type in which that element is declared.

In this way, roles are divorced from types. Then we might make rules like this:

**P2:** A catalog of roles will be maintained. Each role will be uniquely named and described.

For instance, we might have roles: *Header*, *Summary* and *Detail* in such a catalog. When these came up in NDR SC it was amazing to me how polarized we were. One faction believed that since Order and Invoice both have these components that they should be called the same thing in both situations. The other thought that would be confusing since an OrderHeader is different from an InvoiceHeader. Both factions felt that their approach would be less confusing.

This catalog need *not* require an entry for *every* element/property/child field. Such a requirement would cause devolution into an element catalog, which is not what we’re after. It would also dilute the strength of the more powerful entries such as Header, Summary, and Detail.

**P3:** Candidacy for this catalog could be left to a matter of taste, or we could come up with a metric that e.g. only roles occurring or expected to occur more than once are candidates. It will boil down to a combination of experience and taste.

Then where these roles occur in the analysis model, we could use the role name to induce the tag name:

**P4:** When naming an element/property consider its role. Reconcile against the role catalog.

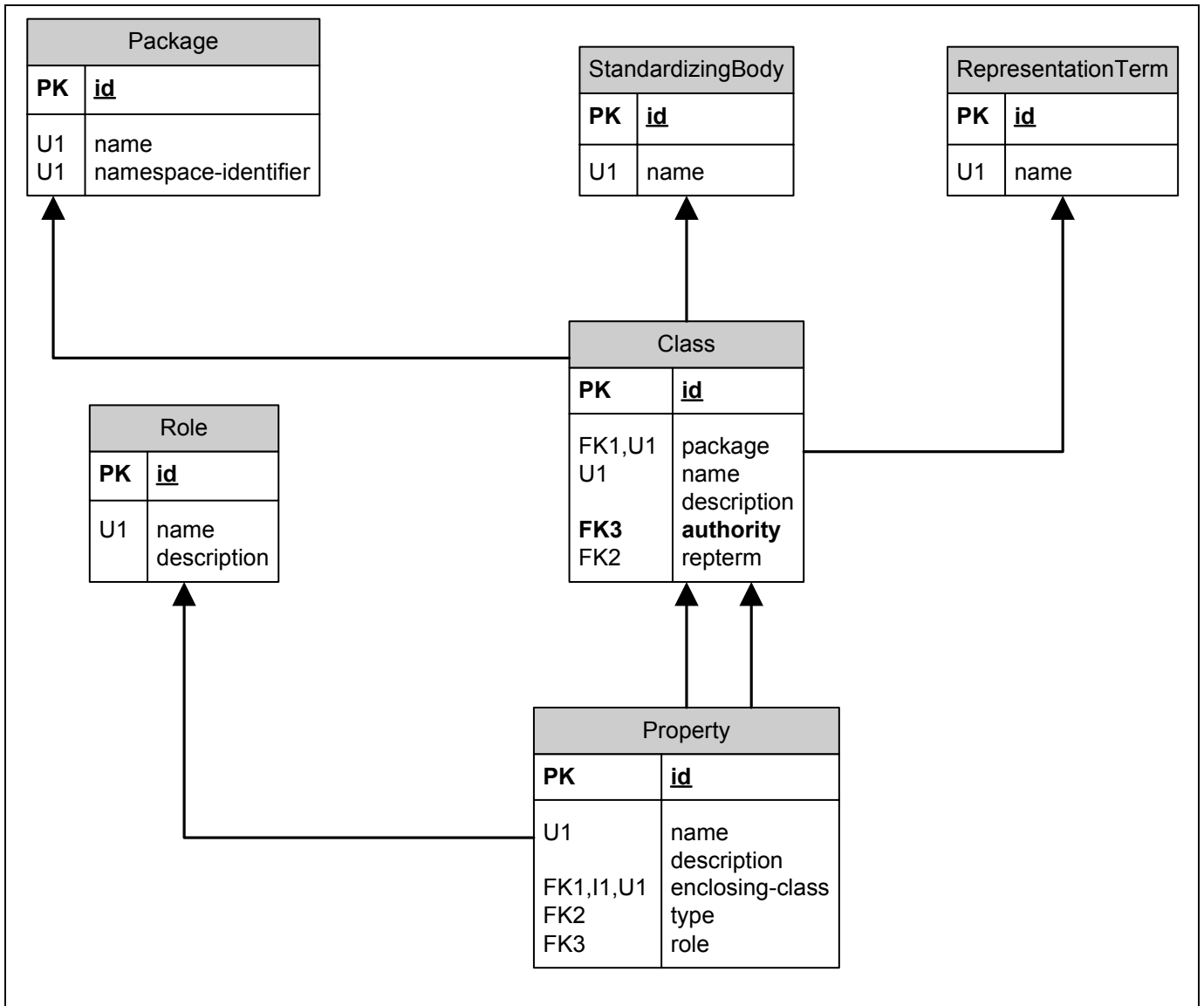
The roles would be linked to the element catalog. Where appropriate, a (local) element definition would refer to the (cataloged) role represented by that element.

**P5:** The element catalog would associate an element with its role definition (if any).



## 8 Appendix: Sample UBL Modeling Artifact Schema

Here is a non-normative peek at an actual catalog structure supporting the recommendations:



## 9 Appendix: Schema eg0

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="eg0" xmlns:xsi="http://www.w3.org/2000/XMLSchema-instance" xmlns:eg0="eg0"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="eg0" elementFormDefault="unqualified"
attributeFormDefault="unqualified">
  <xs:complexType name="OrderHeader">
    <xs:all>
      <xs:element name="Buyer" type="Party"/>
      <xs:element name="Seller" type="Party"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="InvoiceHeader">
    <xs:all>
      <xs:element name="Buyer" type="Party"/>
      <xs:element name="Seller" type="Party"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="Party">
    <xs:all>
      <xs:element name="Name" type="xs:string"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="Order">
    <xs:sequence>
      <xs:element name="Header" type="OrderHeader"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Invoice">
    <xs:all>
      <xs:element name="Header" type="InvoiceHeader"/>
    </xs:all>
  </xs:complexType>
  <xs:element name="OrderDocument" type="Order"/>
</xs:schema>
```

## 10 Appendix: Schema eg1

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="eg1" xmlns:xsi="http://www.w3.org/2000/XMLSchema-instance" xmlns:eg1="eg1"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="eg1" elementFormDefault="unqualified"
attributeFormDefault="unqualified">
  <xs:complexType name="OrderHeader">
    <xs:all>
      <xs:element name="Buyer" type="Party"/>
      <xs:element name="Seller" type="Party"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="InvoiceHeader">
    <xs:all>
      <xs:element name="Buyer" type="Party"/>
      <xs:element name="Seller" type="Party"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="Party">
    <xs:all>
      <xs:element name="Name" type="xs:string"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="Order">
    <xs:sequence>
      <xs:element name="OrderHeader" type="OrderHeader"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Invoice">
    <xs:all>
      <xs:element name="InvoiceHeader" type="InvoiceHeader"/>
    </xs:all>
  </xs:complexType>
  <xs:element name="OrderDocument" type="Order"/>
</xs:schema>
```

# 11 Appendix: Schema eg1a

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="eg1a" xmlns="eg1a" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:eg1a="eg1a" xmlns:xsi="http://www.w3.org/2000/XMLSchema-instance" elementFormDefault="unqualified"
attributeFormDefault="unqualified">
  <xs:complexType name="OrderHeader">
    <xs:all>
      <xs:element name="BuyerParty" type="Party"/>
      <xs:element name="SellerParty" type="Party"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="InvoiceHeader">
    <xs:all>
      <xs:element name="BuyerParty" type="Party"/>
      <xs:element name="SellerParty" type="Party"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="Party">
    <xs:all>
      <xs:element name="Name" type="xs:string"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="Order">
    <xs:sequence>
      <xs:element name="HeaderOrderHeader" type="OrderHeader"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Invoice">
    <xs:all>
      <xs:element name="HeaderInvoiceHeader" type="InvoiceHeader"/>
    </xs:all>
  </xs:complexType>
  <xs:element name="OrderDocument" type="Order"/>
</xs:schema>

```

# 12 References

CC-UN	<i>UN/CEFACT Draft Core Components Specification, Part 1</i> , 15 January, 2002, version 1.75	
NAMING-ISO	<i>ISO/IEC 11179</i> , Final committee draft, Parts 1-6.	
UML-APPLY	<i>Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design</i>	
XML-UML	<i>Modeling XML Applications with UML: Practical e-Business Applications</i> , David Carlson, 2001, Addison-Wesley.	

