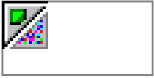


# Guidelines For The Customization of UBL v1.0 Schemas

## Working Draft 1.0-beta2, 04/29/04

Document identifier:

wd-cmsc-cmguidelines-1.0-beta2

Editor:

Eduardo Gutentag, *Sun Microsystems, Inc.* <eduardo.gutentag@sun.com>

Authors:

Matthew Gertner <*matthew@acepoint.cz*>
Eduardo Gutentag, *Sun Microsystems, Inc.* <eduardo.gutentag@sun.com>
Arofan Gregory, *Aeon LLC* <agregory@aeon-llc.com>

Contributors:

Eve Maler, *Sun Microsystems, Inc.*
Dan Vindt, *Accord*
Bill Burcham, *Sterling Commerce*

Abstract:

This document presents guidelines for a compatible customization of UBL schemas, and how to proceed when that is impossible.

Status:

This is a draft document and is likely to change on a regular basis.

If you are on the <ubl@lists.oasis-open.org> list for committee members, send comments there. If you are not on that list, subscribe to the <ubl-comment@lists.oasis-open.org> list and send comments there. To subscribe, send an email message to <ubl-comment-request@lists.oasis-open.org> with the word "subscribe" as the body of the message.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the UBL TC web page (http://www.oasis-open.org/committees/ubl/).

Copyright © 2003, 2004 OASIS Open, Inc. All Rights Reserved.

31 # Table of Contents

32 1. Introduction
33 1.1. Goals of this document
34 2. Background
35 2.1. The UBL Schema
36 2.2. Customization of UBL Schemas
37 2.3. Customization of customization
38 3. Compatible UBL Customization
39 3.1. Use of XSD Derivation
40 3.2. Some observations on extensions and restrictions
41 3.3. Documenting the Customization
42 3.4. Use of namespaces
43 4. Non-Compatible UBL Customization
44 4.1. Use of Ur-Types
45 4.2. Building New Types Using Core Components
46 5. Customization of Codelists
47 6. Use of the UBL Type Library in Customization
48 6.1. The Structure of the UBL Type Library
49 6.2. Importing UBL Schema Modules
50 6.3. Selecting Modules to Import
51 6.4. Creating New Document Types with the UBL Type Library
52 7. Future Directions

53 ## Appendixes

54 A. Notices
55 B. Intellectual Property Rights
56 References

57

58 # 1. Introduction

59 ### Note

60 It is highly recommended that readers of the current document first consult the CCTS paper
61 [**Reference**] before proceeding, in order to understand some of the thinking behind the concepts
62 expressed below.

63 With the release of version 1.0-beta of the UBL library it is expected that subsequent changes to it will be few
64 and far between; it contains important document types informed by the broad experience of members of the
65 UBL Technical Committee, which includes both business and XML experts.

66 However, one of the most important lesson learned from previous standards is that no business library is
67 sufficient for all purposes. Requirements differ significantly amongst companies, industries, countries, etc., and
68 a customization mechanism is therefore needed in many cases before the document types can be used in real-
69 world applications. A primary motivation for moving from the relatively inflexible EDI formats to a more
70 robust XML approach is the existence of formal mechanisms for performing this customization while retaining
71 maximum interoperability and validation.

72 It is an UBL expectation that:

73 1. Customization will indeed happen,

74 2. It will be done by national and industry groups and smaller user communities,

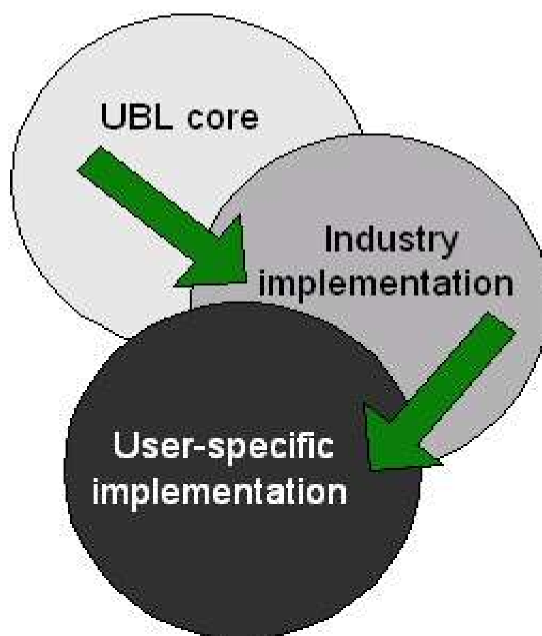
75 3. These changes will be driven by real world needs, and

76 4. These needs will be expressed as context drivers.

77 EDI dealt with the customization issue through a subsetting mechanism that took took a standard (the
78 UN/EDIFACT standard, the AINSI X12 standard, etc.) [**References**] and subsetted it through industry
79 Implementation Guides (IG), which were then subsetted into trading partners IGs, which were then subsetted
80 into departamental IGs. UBL proposes dealing with this through schema derivation.

81 Thus UBL starts as generic as possible, with a set of schemas that supply all that's likely to be needed in the
82 80/20 or core case, which is UBL's primary target. Then it allows both subsetting and extension according to
83 the needs of the user communities, industries, nations, etc., according to what is permitted in the derivation
84 mechanism it has chosen, namely W3C XML Schema.

85 **Figure 1.**




86 These customizations are based on the eight context drivers identified by ebXML (see below ). Any given
87 schema component always occupies a location in this eight-space, even if not a single one has been identified
88 (that is, if a given context driver has not been narrowed, it means that it is true for all its possible contextual
89 values). For instance, UBL has an Address type that may have to be modified if the Geopolitical region in
90 which it will be used is Thailand. But as long as this narrowing down of the Geopolitical context has not been
91 done, the Address type applies to all possible values of if, thus occupying the "any" position in this particular
92 axis of the eight-space.

93 In order for interoperability and validation to be achieved, care must be taken to adhere to strict guidelines when
94 customizing UBL schemas. Although the UBL TC intends to produce a customization mechanism that can be
95 applied as an automatic process in the future, this phase (known as Phase II, and predicted in the UBL TC's
96 charter) has not been reached. Instead, Phase I, the current phase, offers the guidelines included in this
97 document.

98 In what follows in this document, "Customization" always means "context motivated customization", or
99 "contextualization".

100 ## 1.1. Goals of this document

101 This document aims to describe the procedure for customizing UBL schemas, with three distinct goals.

102 1. The first goal is to ensure that UBL users can extend UBL schemas in a manner that:
103     - allows for their particular needs,
104     - can be exchanged with trading partners whose requirements for data content are different but related,
105       and
106     - is UBL compatible.

107 2. The second goal is to provide some canonical escape mechanisms for those whose needs extend beyond what
108    the compatibility guidelines can offer. Although the product of these escape mechanisms cannot claim UBL
109    compatibility, at least it can offer a clear description of its relashionship to UBL, a claim that cannot be made
110    by other *ad hoc* methods.

111 3. The third goal is to gather use case data for the future UBL context extension methodology, the automatic
112    mechanism for creating customized UBL schemas, scheduled for Phase II. To achieve this goal users are
113    strongly encouraged to provide feedback.

114 # 2. Background

115 The major output of the UBL TC is encapsulated in a series of UBL Schemas [**Reference**]. It is assumed that in
116 many cases users will need to customize these schemas for their own use. In accordance with ebXML
117 [**Reference** to CCTS] the UBL TC expects this customization to be carried out only in response to contextual
118 needs (**see** [xxx]) and by the application of any one of the eight identified context drivers and their possible
119 values.

120 It must be noted that the UBL schemas themselves are the result of a theoretical customization:

121 Behind every UBL Schema, a hypothetical schema exists in which all elements are optional and all types are
122 abstract. This is what we call the "Ur-schema". As mandated in the XSD specification, abstract types cannot be
123 used as written; they can only be used as a starting point for deriving new, concrete types. Ur-types are
124 modelled as abstract types since they are designed for derivation. Whether the UBL TC actually produces and
125 publishes a copy of these Ur-schemas is irrelevant, since it is possible for any one to reconstruct
126 deterministically the appropriate Ur-schema from any of the schemas produced by the UBL TC.

127 ## 2.1. The UBL Schema

128 The first set of derivations from the abstract Ur-types is the UBL Schema Library itself, which is assumed to be
129 usable in 80% of business cases. These derivations contain additional restrictions to reduce ambiguity and
130 provide a minimum set of requirements to enable interoperable trading of data by the application of one context,
131 Business Process. The UBL schema may then be used by specific industry organizations to create their own
132 customized schemas. When the UBL Schema is used, conformance with UBL may be claimed. When a Schema
133 that has been customized through the UBL sanctioned derivation processs is used, conformance with UBL may
134 also be claimed.

135 ## 2.2. Customization of UBL Schemas

136 It is assumed that in many cases specific businesses will use customized UBL schemas. These customized
137 schemas contain derivations of the UBL types, created through additional restrictions and/or extensions to fit
138 more precisely the requirements of a given class of UBL users. The customized UBL Schemas may then be
139 used by specific organizations within an industry to create their own customized schemas.

140 ## 2.3. Customization of customization

141 Due to the extensiblilty of W3C Schema, this process can be applied over and over to refine a set of schemas
142 more and more precisely, depending on the needs of specific data flows.

143 In other words, there is no theoretical limit to how many times a Schema can be derived, leading to the possible
144 equivalent of infinite recursion. In order to avoid this, the Rule of Once-per-Context has been developed, as
145 presented later, in "Context Chains "

146 # 3. Compatible UBL Customization

147 Central to the customization approach used by UBL is the notion of schema derivation. This is based on object-
148 oriented principles, the most important of which are inheritance and polymorphism. The meaning of the latter
149 can be gleaned from its linguistic origin: poly, meaning "many", and morph, meaning "shape". By adhering to
150 these principles, document instances with different "shapes" (that is, that conform to different but related
151 schemas,) can be used interchangeably.

152 The UBL Naming and Design Rules Subcommittee (NDRSC) has decided to use XSD, the standard XML
153 schema language produced by the World Wide Web Consortium (W3C), to model document formats. One of
154 the most significant advances of XSD over previous XML document description languages, such as DTDs, is
155 that it has built-in mechanisms for handling inheritance and polymorphism, which we will refer to as "XSD
156 derivation". It therefore fits well with the real-world requirements for business data interchange and our goal of
157 interoperability and validation.

158 There are two important types of modification that XSD derivation does not allow. The first can be summarized
159 as the deletion of required components (that is, the reduction of a component's cardinality from x..y to 0..y). The
160 second is the *ad hoc* location of an addition to the content model through extension. There may be some cases
161 where the user needs a different location for the addition, but XSD extension only allows addition at the end of
162 a sequence.

163 Thus, there are three different scenarios covering the derivation of new types from existing ones:

164 - **Compatible UBL Customization**

165 - An existing UBL type can be modified to fit the requirements of the customization through XSD
166 derivation. These modifications can include extension (adding new information to an existing
167 type), and/or refinement (restricting the set of information allowed to a subset of what is
168 permitted by the existing type).

169 - **Non-compatible UBL Customization**

170 - An existing UBL type could be modified to fit the requirements of the customization, but the
171 changes needed go beyond those allowed by XSD derivation.

172 ○ No existing UBL type is found that can be used as the basis for the new type. Nevertheless, the
173 base library of core components that underlies UBL can be used to build up the new type so as to
174 ensure that interoperability is at least possible at the core component level.

175 These Guidelines will deal with each of the above scenarios, but we will first and foremost concentrate on the
176 first, as it is the only one that can produce UBL-compatible schemas.

## 177 3.1. Use of XSD Derivation

178 XSD derivation allows for type extension and restriction. These are the only means by which one can customize
179 UBL schemas and claim UBL compatibility. Any other possible means, even if allowed by XSD itself, is not
180 allowed by UBL. For instance, although XSD does permit the redefinition of a type to be something other than
181 what it originally is, UBL has decided to reject this approach, because by default `<xsd:redefine>` does not
182 leave any traces of having been used (such as a new namespace, for instance) and because of the danger of
183 circular redefinitions.

184 The examples in the following sections will be based on the following complex type (and note that in all cases
185 the `<xsd:annotation>` elements have been removed in order to achieve maximum legibility):

```
186 <xsd:complexType name="PartyType">
187     <xsd:sequence>
188       <xsd:element ref="PartyIdentification"
189        minOccurs="0" maxOccurs="unbounded">
190       </xsd:element>
191       <xsd:element ref="PartyName"
192        minOccurs="0" maxOccurs="1">
193       </xsd:element>
194       <xsd:element ref="Address"
195        minOccurs="0" maxOccurs="1">
196       </xsd:element>
197       <xsd:element ref="PartyTaxScheme"
198        minOccurs="0" maxOccurs="unbounded">
199       </xsd:element>
200       <xsd:element ref="Contact"
201        minOccurs="0" maxOccurs="1">
202       </xsd:element>
203       <xsd:element ref="Language"
204        minOccurs="0" maxOccurs="1">
205       </xsd:element>
206     </xsd:sequence>
207   </xsd:complexType>
```

### 208 3.1.1. Extensions

209 XSD extension is used when additional information must be added to an existing UBL type. For example, a
210 company might use a special identification code in relation to certain parties. This code should be included in
211 addition to the standard information used in a Party description (PartyName, Address, etc.) This can be achieved
212 by creating a new type that references the existing type and adds the new information:

```
213         <xsd:complexType name="MyPartyType">
214             <xsd:extension base="cat:PartyType">
215                 <xsd:element ref="MyPartyID" minOccurs="1" maxOccurs="1"/>
216                </xsd:element>
217             </xsd:extension>
218          </xsd:complexType>
```

219 Some observations:

220 • Notice that derivation can be applied only to types and not to elements that use those types. This is not a
221 problem: UBL uses explicit type definitions for all elements, in fact disallowing XSD use of anonymous
222 types that define a content model directly inside an element declaration.

223 • This derived type, `MyPartyType`, can be used anywhere the original `PartyType` is allowed. The
224 instance document should use the xsi:type attribute to indicate that a derived type is being used. This
225 does not enforce the use of the new type inside a given element, however, so an `Order` instance could
226 still be created using the standard UBL `PartyType`. If the user wishes to require the use of the derived
227 type, blocking the possibility of using the original type in an instance, a new derived type must be
228 created from the `Order` type using refinement and specifying that the `MyPartyType` must used.

229 • UBL defines global elements for all types, and these elements, rather than the types themselves, are used
230 in aggregate element declarations. The same procedure can be used for derived types, so a global
231 `MyParty` element should be created based on the `MyPartyType`.

232 • All derived types should be created in a separate namespace (which might be tied to the user
233 organization) and reference the UBL namespaces as appropriate. [Appropriate **reference** to UBL's
234 namespace usage, and below]

235 **3.1.2. Restrictions**

236 XSD restriction is used when information in an existing UBL type must be constrained or taken away. For
237 instance, the UBL `PartyType` permits the inclusion of any number of Party identifiers or none. If a specific
238 organization wishes to allow exactly one identifier, this is achieved as follows (note that the annotation fields
239 are removed from the type definition to make the example more readable):

```
240 <xsd:complexType name="MyPartyType">
241     <xsd:restriction base="cat:PartyType">
242      <xsd:sequence>
243       <xsd:element ref="PartyIdentification"
244        minOccurs="1" maxOccurs="1">
245       </xsd:element>
246       <xsd:element ref="PartyName"
247        minOccurs="0" maxOccurs="1">
248       </xsd:element>
249       <xsd:element ref="Address"
250        minOccurs="0" maxOccurs="1">
251       </xsd:element>
252       <xsd:element ref="PartyTaxScheme"
253        minOccurs="0" maxOccurs="unbounded">
254       </xsd:element>
255       <xsd:element ref="Contact"
256        minOccurs="0" maxOccurs="1">
257       </xsd:element>
258       <xsd:element ref="Language"
259        minOccurs="0" maxOccurs="1">
260       </xsd:element>
261     </xsd:sequence>
262    </xsd:restriction>
263   </xsd:complexType>
```

264 Note that the entire content model of the base type, with the appropriate changes, must be repeated when
265 performing restriction.

266 A very important characteristic of XSD restriction is that it can only work within the limits substitutability, that
267 is, the resulting type must still be valid in terms of the original type; in other words, it must be a true subset of
268 the original such that a document that validates against the original can also validate against the changed one.

269 Thus:

- 270 you can reduce the number of repetitions of an element (that is, change its cardinality from 1..100 to 271 1..50, for instance)
- 272 you can eliminate an optional element (that is, change its cardinality from 0..3 to 0..0)
- 273 you cannot eliminate a required element or make it optional (that is, change its cardinality from 1..3 to 274 0..3)

## 275 3.2. Some observations on extensions and restrictions

- 276 Extensions and restrictions can be applied in any order to the same Type; it is recommended, though, 277 that they be applied close to each other to improve understanding of the resulting schema.
- 278 Notice that derivation can be applied only to types and not to elements that use those types. This is not a 279 problem: UBL uses explicit type definitions for all elements, in fact disallowing XSD use of anonymous 280 types that define a content model directly inside an element declaration.
- 281 This derived type, `MyPartyType`, can be used anywhere the original `PartyType` is allowed. The 282 instance document should use the xsi:type attribute to indicate that a derived type is being used. This 283 does not enforce the use of the new type inside a given element, however, so an `Order` instance could 284 still be created using the standard UBL `PartyType`. If the user wishes to require the use of the derived 285 type, blocking the possibility of using the original type in an instance, a new derived type must be 286 created from the `Order` type using refinement and specifying that the `MyPartyType` must used.
- 287 UBL defines global elements for all types, and these elements, rather than the types themselves, are used 288 in aggregate element declarations. The same procedure can be used for derived types, so a global 289 `MyParty` element should be created based on the `MyPartyType`.
- 290 All derived types should be created in a separate namespace (which might be tied to the user 291 organization) and reference the UBL namespaces as appropriate. [Appropriate **reference** to UBL's 292 namespace usage, and below]

## 293 3.3. Documenting the Customization

294 Every time a derivation is performed on a UBL- or UBL-derived-Schema, the context driver and the driver 295 value used must be documented. If this is not done, then *by definition* the derived Schema is not UBL- 296 compliant.

297 Context is expressed using a set of name/value pairs (context driver, driver value), where the names are one of a 298 limited set of context drivers established by the UBL TC on the basis of CCTS (**Reference**):

- 299 Business process
- 300 Official constraint
- 301 Product classification
- 302 Business process role
- 303 Industry classification

304 • Supporting role

305 • Geopolitical

306 • System constraint

307 There is no pre-set list of values for each driver. Users are free at this point to use whatever codification they
308 choose, but they should be consistent; therefore while not obliged to do so, communities of users are strongly
309 encouraged to always use the same values for the same context (that is, those who use "U.S.A" to indicate a
310 country in the North American Continent, should not intermix it with "US" or "U.S." or "USA"). And if a
311 particular standardized codification is used, it should also be identified in the documentation. (Some standard
312 sets of values are provided in the CCTS specification.)

313 There is no predetermined order in which context drivers are applied.

314 More than one context driver might be applied to various types within the same set of schema extensions.
315 Therefore, documentation at the root level, although desirable, is not enough. Context should be included within
316 a `<Context>` child of the element `<Contextualization>` (in the UBL namespace) inside the
317 documentation for each customized type, with the name of the context driver expressed as in the list above, but
318 using the provided elements within that element. For example, if a type is to be used in the French apparel
319 industry (shoes), the Context documentation would appear as follows:

```
320 <xsd:annotation>
321     <xsd:documentation>
322        <ubl:Contextualization>
323          <ubl:Context>
324             <ubl:Geopolitical>France</ubl:Geopolitical>
325             <ubl:IndustryClassification>Apparel</ubl:IndustryClassification>
326             <ubl:ProductClassification>Shoes</ubl:ProductClassification>
327           </Context>
328         </ubl:Contextualization>
329     </xsd:documentation>
330 <xsd:annotation>
```

331 The `<Context>` element can be repeated, once of each incremental change.

332 If a customization is made that does not fit into any of the existing context drivers, it should be described in
333 prose inside the <Context> element:

```
334 <xsd:annotation>
335     <xsd:documentation>
336       <ubl:Contextualization>
337         <ubl:Context>Used for jobs performed on weekends to specify
338                      additional data required by the trade union</ubl:Context>
339       </ubl:Contextualization>
340     </xsd:documentation>
341 <xsd:annotation>
```

342 **Note**

343 Any issues with the set of context drivers currently defined or the taxonomies to be used for
344 specifying values should be communicated to the UBL Context Driver Subcommittee.

345 For each of the context drivers (`Geopolitical`, `IndustryClassification`, etc.) the following
346 characteristics should also be specified (a later version will provide the requisite attributes for doing so):

347 • listID (List Identifier) - string: The identification of a list of codes. Can be used to identify the URL of a
348 source that defines the set of currently approved permitted values.

349 • listAgencyID (List Agency Identifier) - string: An agency that maintains one or more code lists.
350 Defaults to the UN/EDIFACT data element 3055 code list.

351 • listAgencyName (List Agency Name) - string: The name of the agency that maintains the code list.

352 • listName ( List Name) - string: The name of a list of codes.

353 • listVersionID (List Version Identifier) - string: The Version of the code list. Identifies the Version of the
354 UN/EDIFACT data element 3055 code list.

355 • languageID (Language Identifier) - string: The identifier of the language used in the corresponding text
356 string (ISO 639: 1998)

357 • listURI (List URI) - string: The Uniform Resource Identifier that identifies where the code list is
358 located.

359 • listSchemeURI (List Scheme URI) - string: The Uniform Resource Identifier that identifies where the
360 code list scheme is located.

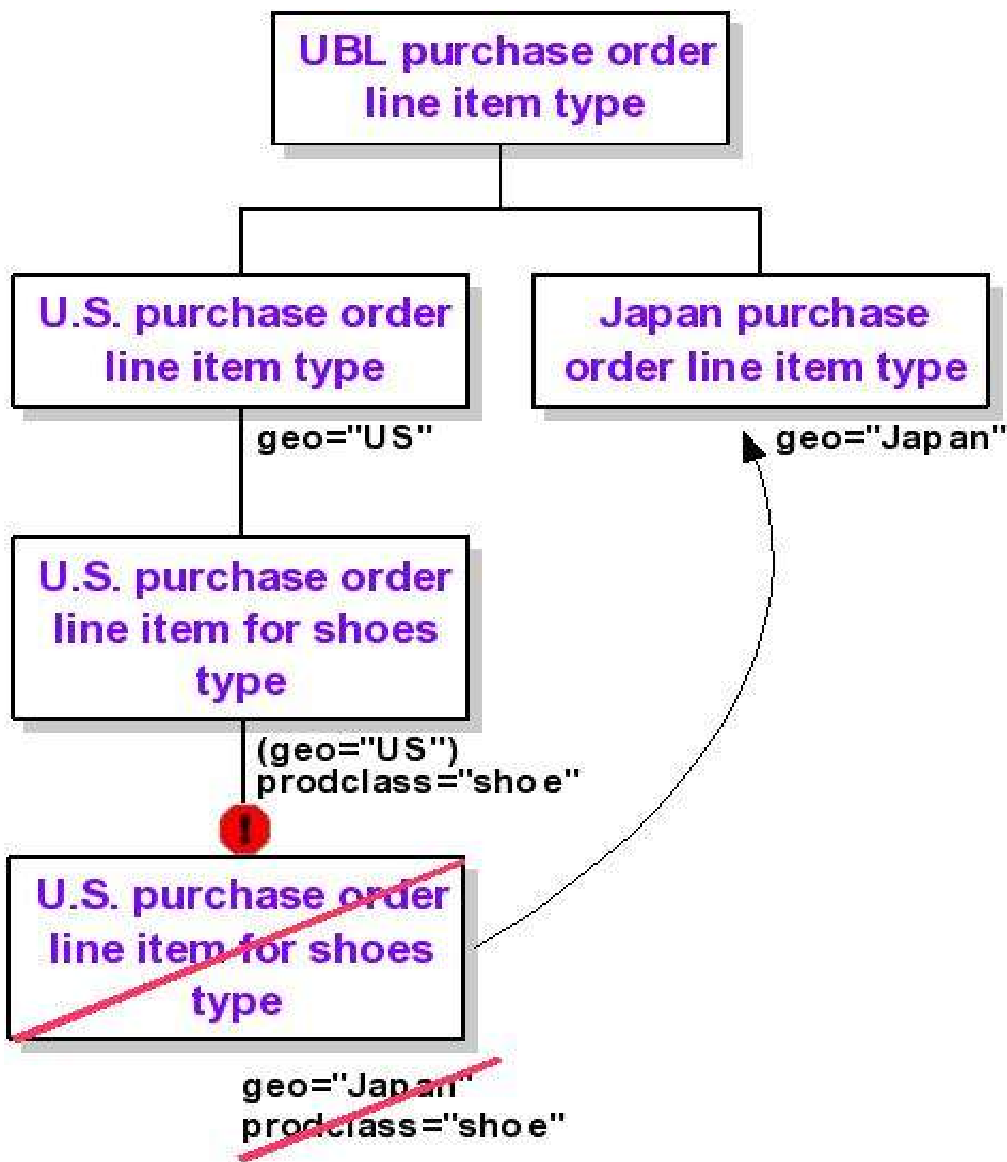
361 • Coded Value: A value or set of values taken from the indicated code list or classification scheme.

362 • Text Value: A textual description of the set of values.

363 **3.3.1. Context chains**

364 As mentioned in "Customization of Customization", there is a risk that derivations may form extremely long
365 and unmanageable chains. In order to avoid this problem, the Rule of Once-per-Context was formulated: no
366 context can be applied, at a given hierarchical level of that context, more than once in a chain of derivations. Or,
367 in other words, any given context driver can be specialized, but not reset. Thus, if the Geopolitical context
368 driver with a value of "USA" has been applied to a type, it is possible to apply it again with a value that is a
369 subset, or that occupies a hierarchically lower level than that of the original value, like California or New York,
370 but it cannot be applied with a value equal or higher in the hierarchy, like Japan. In order to use that latter value,
371 one must go up the ladder of the customization chain and derive the type from the same location as that from
372 which the original was derived.

373 **Figure 2.**




374

### 3.4. Use of namespaces

Every customized Schema or Schema module must have a namespace name different from the original UBL one. This may end up having an upward-moving ripple effect (a schema that includes a schema module that now has a different namespace name must change its own namespace name, for instance). However, it should be noted that all that has to change is the local part of the namespace name, not the prefix, so that XPaths in existing XSLT stylesheets, for instance, would not have to be changed except inasmuch as a particular element or type has changed.

Although there is not constraint as to what namespace name should be used for extensions, or what method should be used for constructing it, it is recommended that the method be, where appropriate, the same as the method specified in [**Reference** to NDR document, section on namespace construction]

## 4. Non-Compatible UBL Customization

There are two important types of customization that XSD derivation does not allow. The first can be summarized as the deletion of required components (that is, the reduction of a component's cardinality from x..y to 0..y). The second is the *ad hoc* location of an addition to a content model. There may be some cases where the user needs a different location for the addition than the one allowed by XSD extension, which is at the end of a sequence.

391 Because XSD derivation does not allow these types of customization, any attempts at enabling them (which in
392 some cases simply mean rewriting the schema with the desired changes as a different schema in a different,
393 non-UBL namespace) must by necessity produce results that are not UBL compatible. However, in order to
394 allow users to customize their schemas in a UBL-friendly manner, the notion of an Ur-schema was invented: for
395 each UBL Schema, an theoretical Ur-schema exists in which all elements are optional and all types are abstract.
396 The use of abstract types is necessary because an Ur-type can never be used as is; a derived type must be
397 created, as per the definition of abstract types in the XSD specification.

## 398 4.1. Use of Ur-Types

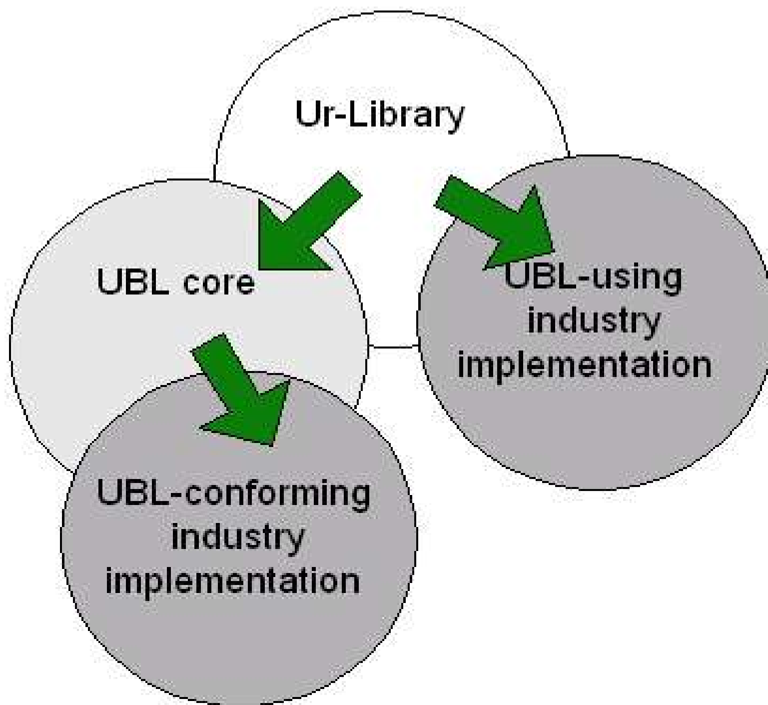
399 XSD derivation is sufficient for most cases, but as mentioned above, in some instances it may be necessary to
400 perform changes to the UBL types that are not handled by standard mechanisms. In this case, the UBL Ur-types
401 should be used. Remember, an Ur-type exists for each UBL standard type and differs only in that all elements in
402 the content model are optional, including elements that are required in the standard type. By using the Ur-type,
403 the user can therefore make modifications, such as eliminating a required field, that would not be possible using
404 XSD derivation on the standard type.

405 For instance, suppose an organization would like to use the UBL `PartyType,` but does not want to use the
406 required ID element. In this case, normal XSD refinement is used, but on the Ur-type rather than the standard
407 type:

```
408 <xsd:complexType name="MyPartyType">
409     <xsd:restriction base="ur:PartyType">
410      <xsd:sequence>
411       <xsd:element ref="PartyIdentification"
412        minOccurs="0" maxOccurs="0">
413       </xsd:element>
414       <xsd:element ref="PartyName"
415        minOccurs="0" maxOccurs="1">
416       </xsd:element>
417       <xsd:element ref="Address"
418        minOccurs="0" maxOccurs="1">
419       </xsd:element>
420       <xsd:element ref="PartyTaxScheme"
421        minOccurs="0" maxOccurs="unbounded">
422       </xsd:element>
423       <xsd:element ref="Contact"
424        minOccurs="0" maxOccurs="1">
425       </xsd:element>
426       <xsd:element ref="Language"
427        minOccurs="0" maxOccurs="1">
428       </xsd:element>
429     </xsd:sequence>
430    </xsd:restriction>
431   </xsd:complexType>
```

432 The new type is no longer compatible with the UBL `PartyType`, so standard processing engines that know
433 about XSD derivation will not recognize the type relationship. However, some level of interoperability is still
434 preserved, since both UBL `PartyType` and `MyPartyType` are derived from the `PartyType` Ur-type. If
435 this additional flexibility is required, a processor can be implemented to use the Ur-type rather than the UBL
436 type. It will then be able to process both the UBL type and the custom type, since they have a common ancestor
437 in the Ur-type (at the expense, of course, of an added level of complexity in the implementation of the
438 processor).

439 **Figure 3.**




440

441 Once again: changes to the Ur-type do not enforce changes in the enclosing type, so the UBL `OrderType` has
442 to be changed as well if the user organization wants to ensure that only the new `MyPartyType` is used. In
443 fact, the new `OrderType` will not be compatible with the UBL `OrderType`, since `MyPartyType` is no
444 longer derived from UBL's PartyType. However, the new `OrderType` can be derived from the `OrderType`
445 Ur-type to achieve maximum interoperability.

446 It is possible that at some point one ends up with a schema that contains customizations that were made in a
447 compatible manner as well as customizations that were made in a non-compatible manner. If that is the case,
448 then the schema must be considered non-compatible.

## 449 4.2. Building New Types Using Core Components

450 Sometimes no type can be found in the UBL library or Ur-type library that can be used as the basis for a new
451 type. In this case, maximum interoperability (though not compatibility) can be achieved by building up the new
452 type using types from the core component library that underlies UBL. (See below)

453 For example, suppose a user organization needs to include a specialized product description inside business
454 documents. This description includes a unique ID, a name and the storage capacity of the product expressed as
455 an amount. The type definition would then appear as follows:

```
456 <xsd:complexType name="ProductDescriptionType">
457         <xsd:sequence>
458                 <xsd:element name="ID" type="cct:IdentifierType"/>
459                 <xsd:element name="Name" type="cct:NameType"/>
460                 <xsd:element name="Capacity" type="cct:AmountType"/>
461         </xsd:sequence>
462 </xsd:complexType>
```

### 463 Note

464 The above example should belong to a clearly non-UBL namespace.

465 It goes without saying that all new names defined when creating custom types from scratch should also conform
466 to the UBL Naming and Design Rules [**Reference**].

# 5. Customization of Codelists

The guidelines presented in this document do not include the customization of Codelists. This topic is not addressed here. It is expected that it will be addressed during the 1.1 timeframe.

# 6. Use of the UBL Type Library in Customization

UBL provides a large selection of types which can be extended and refined as described in the preceding sections. However, the internal structure of the UBL type library needs to be understood and respected by those doing customizations. UBL is based on the concept of compatible reuse where possible, and there are cases where it would be possible to extend different types within the library to achieve the same end. This section discusses the specifics of how namespaces should be imported into a customizer's namespace, and the preference of types for specific extension or restriction. What follows applies equally to UBL-compatible and UBL-non-compatible extensions.

## 6.1. The Structure of the UBL Type Library

The UBL type library is exhaustively modelled and documented as part of the standard; what is provided here is a brief overview from the perspective of the customizer.

Within the UBL type library is an implicit hierarchy, structured according to the rules provided by the UBL NDR. When customizing UBL document types, the top level of the hierarchy is represented by a specific business document. The business document schema instances are found inside the control schema modules, which consist of a global element declaration and a complex type declaraion (referenced by the global element declaration) for the document type. Also within these control schema modules are imports of the other UBL namespaces used (termed "external schema modules"), and possibly includes of schema instances specific to that module (termed "internal schema modules"). The control schema modules import the *Common Aggregate Components (CAC)* and *Common Basic Components (CBC)* namespaces, which include global element and type declarations for all of the reusable constructs within UBL. These namespace packages in turn import the *Specialized Datatype* and *Unspecialized Datatype* namespaces, which include declarations for the constructs which describe the basic business uses for data-containing elements. These namespaces in turn import the CCT namespace, which provides the primitives from which the UBL library is built.[**Reference the picture in NDR**]

This hierarchy represents the model on which the UBL library is based, and provides a type-intensive environment for the customizer. The basic structure is one of semantic qualification: as you move from the modeling primitives (CCTs) and go up the hierarchy toward the business documents, the semantics at each level become more and more completely qualified. This fact provides the fundamental guidance for using these types in customizations, as discussed more fully below.

## 6.2. Importing UBL Schema Modules

UBL schema modules are included for use in a customization through the importing of their namespaces. Before extending or refining a type, you must import the namespace in which that type is found directly into the customizing namespace. While inclusion may be used to express internal packaging of multiple schema instances within a customizer's namespace, the include mechanism should never be used to reference the UBL type library.

The UBL NDR provides a mechanism whereby each schema module made up of more than a single schema instance has a "control" schema instance, which performs all of the imports for that namespace. Customizers should follow this same pattern, since their customizations may well be further customized along the lines described above. In the same vein, when a UBL document type is imported, it should be the control schema module for that document type which is imported, bringing in all of the doctype-specific constructs, whether in

the control schema instance for that namespace or one of the "internal" schema instances.

## 6.3. Selecting Modules to Import

In many cases, the customizer will have no choice about importing or not importing a specific module: if the customizer needs to extend the document-type-level complex type, there is only a single choice: the control schema for the document type must be imported. Not all cases are so clear, however. When creating lower-level elements, by extending the types found in the *CAC* and *CBC* namespaces (for example), it is possible to either extend a provided type, or to build up a new one from the types available within the *Specialized Datatypes* and *Unspecialized Datatypes* namespace packages.

UBL compatible customization always involves reuse at the highest possible level within the hierarchy described here. Thus, it is always best to reuse an existing type from a higher-level construct than to build up a new type from a lower-level one. Whenever faced with a choice about how to proceed with a customization, you should always determine if there is a customizable type within the *CAC* or *CBC* before going to the Datatype namespace packages. This rule further applies to the use of the datatype namespaces: never go directly to the CCT namespace to create a type if something is available for extension or refinement within the datatype namespaces. By the same token, it is always preferable to extend a complex datatype than to create something with reference to an XSD primitive datatype, or a custom simple type.

It is important to bear in mind that the structure of the UBL library is based around the ideas of semantic qualification and reuse. You should never introduce semantic redundancy into a customized document based on UBL. You should always further qualify existing semantics if at all possible.

## 6.4. Creating New Document Types with the UBL Type Library

UBL provides many useful document types for customization, but for some business processes, the needed document types will not be present. When creating a new document type, it is recommended that they be structured as similarly as possible to existing documents, in accordance with the rules in the UBL NDR. The basic structure can easily be seen in an examination of the existing document types. What is not so obvious is the approach to the use of types. The design here is to primarily use the types provided in the *CAC* and *CBC*, and only then going to the Datatypes namespace packages. This is the same approach described for modifying UBL document types in the preceding section.

# 7. Future Directions

It is planned that in Phase II of the development of this Context Methodology, a context extension method will be designed to enable automatic customization of UBL types based on context, as outlined in the charter of the UBL TC. This methodology will work through a formal specification of the reasons for customizing the type, i.e. the context driver and its value. By expressing the context formally and specifying rules for customizing types based on this context, most of the changes that need to be made to UBL in order for it to fit in a given usage environment can be generated by an engine rather than performed manually. In addition, significant new flexibility may be gained, since rules from two complementary contexts could perhaps be applied simultaneously, yielding types appropriate for, say, the automobile industry and the French geopolitical entity, with the appropriate documentation and context chain produced at the same time.

UBL has not yet progressed to this stage of development. For now, one of the main goals of the UBL Context Methodology Subcommittee is to gather as many use cases as possible to determined what types of customizations are performed in the real world, and on what basis. Another important goal is to ensure that types derived at this point from UBL's version 1 can be still used later on, intermixed with types derived automatically in the future.

# A. Notices

Copyright © The Organization for the Advancement of Structured Information Standards [OASIS] 2003, 2004. All Rights Reserved.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS's procedures with respect to rights in OASIS specifications can be found at the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification, can be obtained from the OASIS Executive Director.

OASIS invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to implement this specification. Please address the information to the OASIS Executive Director.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to OASIS, except as needed for the purpose of developing OASIS specifications, in which case the procedures for copyrights defined in the OASIS Intellectual Property Rights document must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS has been notified of intellectual property rights claimed in regard to some or all of the contents of this specification. For more information consult the online list of claimed rights.

# B. Intellectual Property Rights

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the UBL TC web page.

# References

## Normative

[RFC 2119] S. Bradner. *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels*. IETF (Internet Engineering Task Force). 1997.