Universal Business Language (UBL) Naming and Design Rules

Working Draft 17, 22 October 2002

Document identifier:

wd-ublndrsc-ndrdoc-17 (Word, PDF)

Location:

http://www.oasis-open.org/committees/ubl/ndrsc/drafts/

Editors:

Bill Burcham, Sterling Commerce <Bill_Burcham@stercomm.com>
Mavis Cournane, Cognitran Ltd <mavis.cournane@cognitran.com> (primary editor)
Mark Crawford, LMI <MCRAWFORD@lmi.org>
Arofan Gregory, CommerceOne <arofan.gregory@commerceone.com>
Eve Maler, Sun Microsystems <eve.maler@sun.com>

Contributors:

Fabrice Desré, France Telecom
Matt Gertner, Schemantix
Jessica Glace, LMI
Phil Griffin, Griffin Consulting
Michael Grimley, US NavyEduardo Gutentag, Sun Microsystems
Sue Probert, CommerceOne
Lisa Seaburg, Aeon Consulting
Gunther Stuhec, SAP
Paul Thorpe, OSS Nokalva

Abstract:

This specification documents the naming and design rules and guidelines for the construction of XML components for the UBL vocabulary.

Status:

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

If you are on the ubl-ndrsc@lists.oasis-open.org list for NDR subcommittee 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 Security Services TC web page (http://www.oasis-open.org/committees/security/).

Table of Contents

1 Introduction

This specification documents the rules and guidelines for the naming and design of XML components for the UBL library. It reflects only rules that have been agreed on by the OASIS UBL Naming and Design Rules Subcommittee (NDR SC). Proposed rules, and rationales for decided rules, appear in the accompanying NDR SC position papers, which are available at http://www.oasis-open.org/committees/ubl/ndrsc/.

The W3C XML Schema form of the UBL library is currently constructed automatically from the metamodel developed by the OASIS UBL Library Content Subcommittee (LC SC). Thus, most of the rules in this document are used to guide the development of the engine that generates the XSD schema modules; this engine is produced by the OASIS UBL Tools and Techniques Subcommittee (TT SC). Some of the rules address XML instance constructs and other practices that must be undertaken by humans, such as developers who are customizing UBL for their own purposes.

1.1 Audiences

There are two primary audiences for this document – the internal TC member/perl script writer, and the UBL customizer.

1.2 Terminology and Notation

The key words *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* in this document are to be interpreted as described in **[RFC2119]**.

The terms “W3C XML Schema” and “XSD” are used throughout this document. They are considered synonymous; both refer to XML Schemas that conform to the W3C Schema Recommendations **[XSD]**. See Section 12 for additional term definitions.

1.3 Guiding Principles

1.3.1 Adherence to general UBL guiding principles

The UBL NDRSC is following the high-level guiding principles for the design of UBL as approved by the UBL TC. These principles are:

- Internet Use - UBL shall be straightforwardly usable over the Internet.
- Interchange and Application Use–UBL is intended for interchange and application use.
- Tool Use and Support - The design of UBL cannot make any assumptions about sophisticated tools for creation, management, storage, or presentation being available. . The lowest common denominator for tools is incredibly low (for example, Notepad), and the variety of tools used is staggering. We do not see this situation changing in the near term.
- Time Constraints–Urgency is a key item in the development of UBL. Many facets of XML are still being debated. UBL will make rapid “informed” decisions that may not agree with the ultimate “right” design decisions subsequently reached elsewhere.
- Legibility - UBL documents should be human-readable and reasonably clear
- Simplicity - The design of UBL must be as simple as possible (but no simpler).
- 80/20 Rule - The design of UBL should provide the 20% of features that accommodate 80% of the needs.

- Component Reuse–The design of UBL document types should share as many common features as possible. The essential nature of e-commerce transactions is to pass along information that gets incorporated again into the next transaction down the line. For example, a purchase order contains information that will be copied into the purchase order response. This forms the basis for our need for a core library of reusable components. In fact, reuse in this context is important not only for the efficient development of software, but also for keeping audit trails.
- Standardization - The number of ways to express the same information in a UBL document is to be kept as close to one as possible.
- Domain Expertise–UBL will leverage expertise in a variety of domains through interaction with appropriate development efforts.
- Customization and Maintenance - The design of UBL must enable customization and maintenance.
- Context Sensitivity - The design of UBL must ensure that context-sensitive document types aren't precluded.
- Prescriptiveness–UBL design will balance prescriptiveness in any one usage scenario with prescriptiveness across the breadth of usage scenarios supported. Having precise, tight content models and datatypes is a good thing (and for this reason, we might want to advocate the creation of more document type "flavors" rather than less; see below). However, in an interchange format, it is often difficult to get the prescriptiveness that would be desired in any one usage scenario.
- Content Orientation - Most UBL document types should be as "content-oriented" (as opposed to merely structural) as possible. Some document types, such as product catalogs, will likely have a place for structural material such as paragraphs, but these will be rare.
- XML Technology–UBL design will avail itself of standard XML processing technology wherever possible (XML itself, XML Schema, XSLT, XPath, and so on). However, UBL will be cautious about basing decisions on "standards" (foundational or vocabulary) that are works in progress.
- Relationship to Other Namespaces–UBL design will be cautious about making dependencies on other namespaces. UBL does not need to reuse existing namespaces wherever possible. For example, XHTML might be useful in catalogs and comments, but it brings its own kind of processing overhead, and if its use is not prescribed carefully it could harm our goals for content orientation as opposed to structural markup.
- Legacy formats - UBL is not responsible for catering to legacy formats; companies (such as ERP vendors) can compete to come up with good solutions to permanent conversion. This is not to say that mappings to and from other XML dialects or non-XML legacy formats wouldn't be very valuable.
- Relationship to xCBL–UBL will not be a strict subset of xCBL, nor will it be explicitly compatible with it in any way.

1.3.2 Design For Extensibility

Many basic e-commerce document types are generally useful, but require minor structural modifications for specific tasks or markets. When a truly common XML structure is to be established for e-commerce, it needs to be easy and inexpensive to modify.

In EDI there has been a gradual increase in the number of published components to accommodate market-specific variations. Several efforts within the EDI community are focused on eliminating this problem; variations are a requirement, and one that is not easy to meet. A related EDI phenomenon is the overloading of the meaning and use of existing elements, which greatly complicates interoperation.

To avoid the high degree of cross-application coordination required to handle structural variations in EDI - and in DTD-based systems - it is necessary to accommodate the required variations in basic data structures without either overloading the meaning and use of existing data elements, or requiring wholesale addition of data elements. This can be accomplished by allowing implementers to specify new element types that inherit the properties of existing elements, and to also specify exactly the structural and data content of the modifications.

Many data structures used in e-commerce are very similar to “standard” data structures, but have some significant semantic difference native to a particular industry or process. This can be expressed by saying that extensions of core elements are driven by context [need ref here]. Context driven extensions should be renamed to distinguish them from their parents, and designed so that only the new elements require new processing.

Similarly, data structures should be designed so that processes can be readily engineered to ignore additions that are not needed.

1.3.3 Code Generation

2 Choice of schema language

The UBL vocabulary is expressed in XSD.

3 Relationship to ebXML Core Components

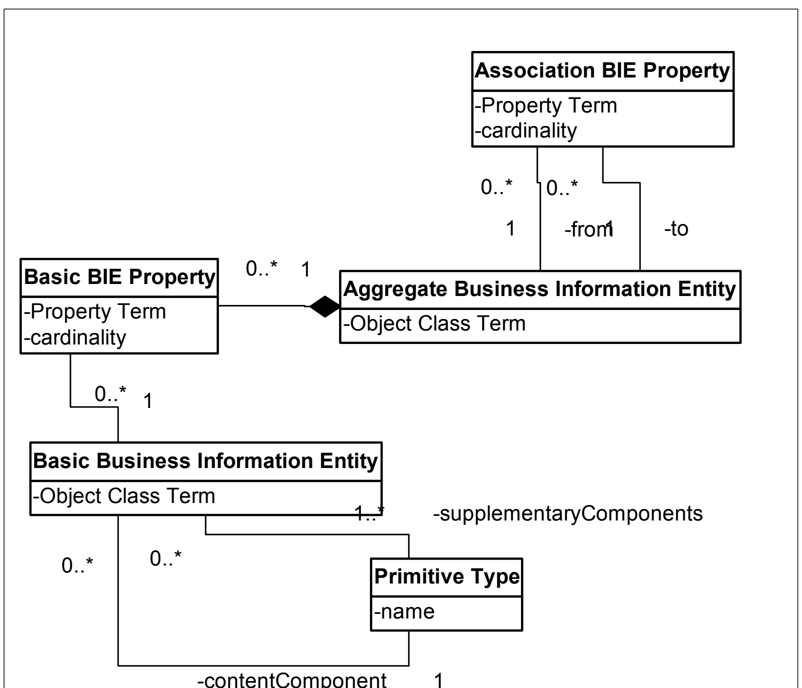
UBL employs the methodology and model described in **[CCTS]**. In the terminology of that specification, the UBL vocabulary consists primarily of *Aggregate Business Information Entities* (ABIE). An ABIE is similar to a Class in object-oriented modeling (e.g. UML). An ABIE is similar to an entity in Entity Relationship modeling.

According to the **CCTS** each ABIE *must* have a unique name (Object Class Term). Each ABIE *must* have one or more BIE Properties. Each BIE Property *must* have a name (Property Term). That name *must* be unique within that ABIE.

There are two kinds of *BIE Property*. A *Basic BIE Property* represents an *intrinsic* property of an ABIE. An *Association BIE Property* represents an *extrinsic* property – in other words an association from one ABIE instance to another ABIE instance. It is the Association BIE Property that expresses the relationship between ABIEs.

In order to actually define the intrinsic structure of an ABIE, a set of *Basic Business Information Entities* is defined. These are the “leaf” types in the system in that they contain *no Association BIE Properties, and no Basic BIE Properties*. A BBIE *must have* a single *Content Component* and one or more *Supplementary Components*. A Content Component is of some *Primitive Type.*

Here’s a picture of the relevant parts of the Core Components metamodel:



(part of) Core Components Metamodel

The preceeding diagram depicts a summary of the Core Components metamodel. Whereas the Core Components metamodel encompasses two broad categories of model element, the Core Component and the Business Information Entity, UBL is concerned with only the latter.

Since UBL is concerning itself only with the development of Business Information Entities, and their realization in XML, the UBL metamodel speaks only in terms of BIE concepts. For instance, while the Core Components metamodel specifies that each BIE is “based on” a particular Core Component – that detail is not considered by UBL. UBL defines no Core Components.

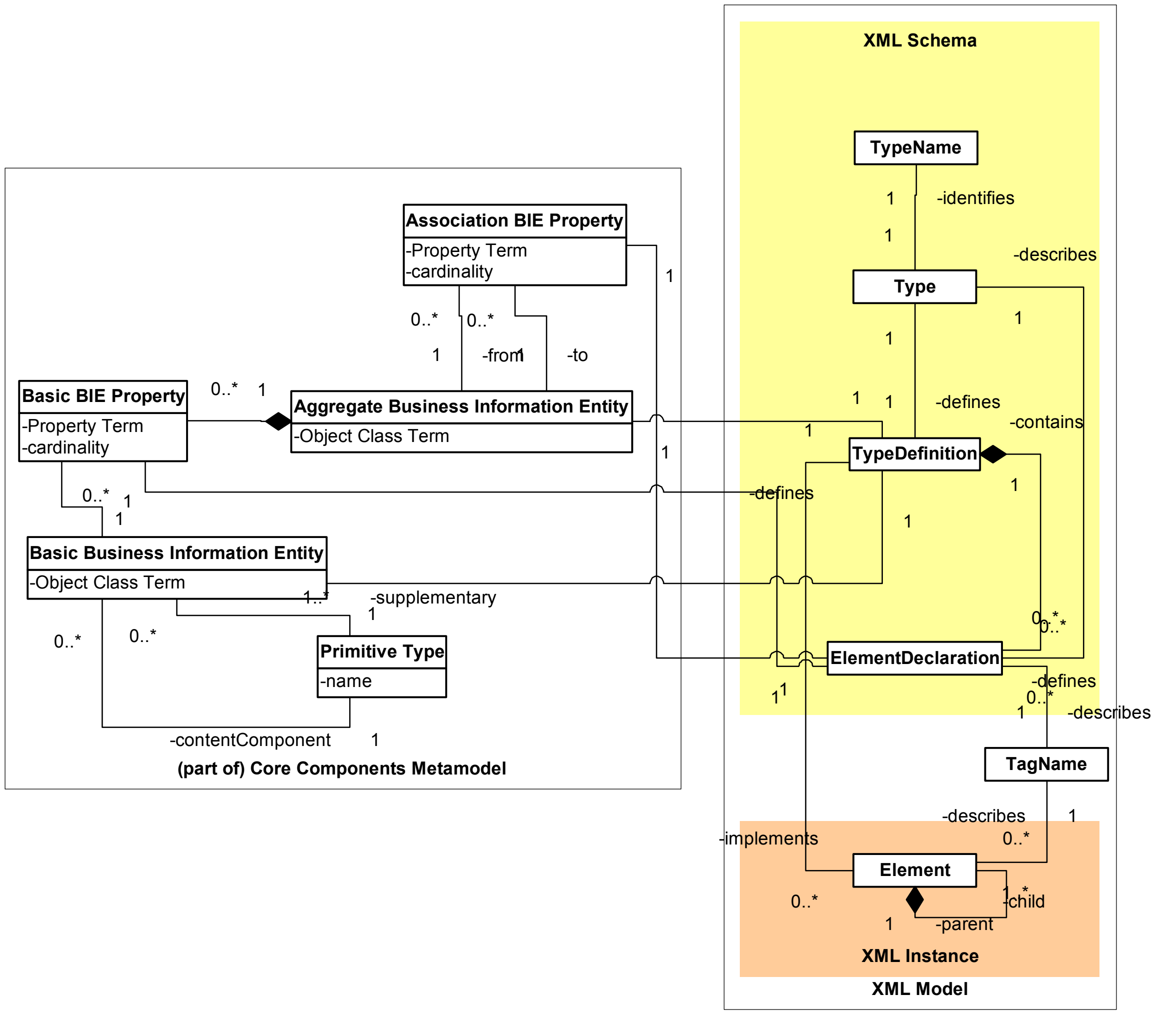
Similarly, the Core Components metamodel describes parallel model elements to capture low-level types such as Identifiers, and Dates etc. In that metamodel, a Core Component Type describes these low-level types for use by Core Components, and (in parallel) a “Data Type” – corresponding to that Core Component Type, describes these low-level types for use by Business Information Entities. UBL is not, therefore concerned with Core Component Types since again, they pertain only to the Core Components model, which UBL is not specifying. UBL defines no Core Components, and UBL defines no Core Component Types.

That being said, you might rightly expect to see Data Type appear in the diagram above, however, since in the Core Components metamodel there is a one-to-one correspondence between a Data Type and a Business Information Entity, UBL has elected to define only the latter. The alternative would be for UBL to define Data Types (e.g. AmountType, CodeType, DateTimeType, etc.) and also to define corresponding BIE’s. To do so would add no value to the work product, so we will model only one. UBL defines no Data Types separate from BIE’s – there is only the BIE’s.

3.1 Rules for Mapping Business Information Entities, Their Properties, and Primitive Types to XML

A primary deliverable of the UBL effort is XML Schemas. These schemas declare a *complex type* for each ABIE, and a complex type for each BBIE. Each Association BIE Property becomes an element definition (within the appropriate complex type). Similarly each Basic BIE Property becomes an element definition within a complex type.

This diagram depicts the relationship between the ABIE model and the XML Schema/XML instance models:



Each ABIE results in a complex type declaration in the XML Schema. The complex type name is derived like this:

<ABIE Object Class Term>"Type"

Here are some examples:

ABIE Object Class Term	Complex Type Name
Address	AddressType
Party	PartyType

Each BBIE results in a complex type declaration in the XML Schema. The name of the complex type is derived like this:

<BBIE Object Class Term>"Type"

Here are some examples:

BBIE Object Class Term	Complex Type Name
Amount	AmountType
DateTime	DateTimeType

Each Basic BIE Property results in an element in the XML Schema. The tag name is derived like this:

<Basic BIE Property Property Term>((<BBIE Object Class Term> != "Text" && <Basic BIE Property Property Term> != <BBIE Object Class Term>) ? (<BBIE Object Class Term> == "Identifier" ? "ID" : <BBIE Object Class Term>)

So the tag name is the name of the Basic BIE Property followed by the name of the pertinent BBIE. If the BBIE is named "Text" or if the name of the Basic BIE Property is the same as the name of the BBIE then it *must* be *elided*. If the BBIE Object Class Term is Identifier then it is translated to "ID" in the tag name.

Here are some examples:

Basic BIE Property Property Term	BBIE Object Class Term	Tag name
Purpose	Code	PurposeCode
Name	Text	Name
Party	Identifier	PartyID

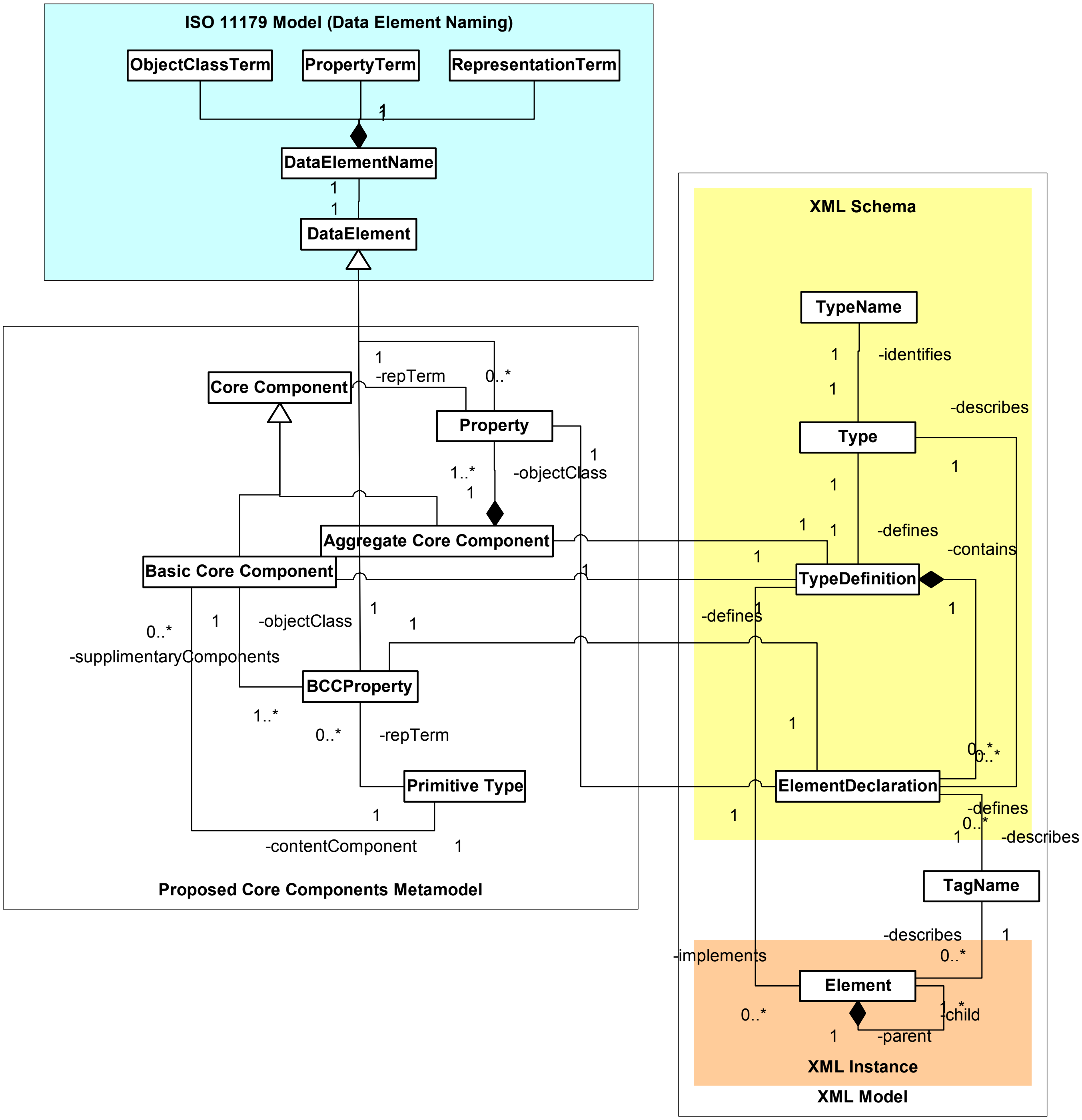
Each Association BIE Property results in an element definition in the XML Schema. The tag name is derived like this:

<Association BIE Property Property Term>((<Association BIE Property Property Term> != < ABIE Object Class Term of ABIE in the "to" role>) ? (<ABIE Object Class Term of ABIE in the "to" role >)

Here are some examples:

Association BIE Property Property Term	ABIE Object Class Term of ABIE in the “to” role	Tag name
Receiving	Contact	ReceivingContact
Address	Address	Address

TODO: we need to add the excruciating details of mapping Basic Business Information Entities, and their associated content component and supplementary components to XSD and XMI.


ISO 11179 Model (Data Element Naming)
ObjectClassTerm
PropertyTerm
RepresentationTerm
DataElementName
DataElement
Proposed Core Components Metamodel
Core Component
Property
Aggregate Core Component
Basic Core Component
BCCProperty
Primitive Type
-repTerm
-objectClass
-supplimentaryComponents
-contentComponent
XML Model
XML Schema
TypeName
Type
TypeDefinition
ElementDeclaration
TagName
-identifies
-describes
-defines
-contains
XML Instance
Element
-implements
-child
-parent

4 XML Constructs

In W3C XML Schema, elements are defined in terms of complex or simple types and attributes are defined in terms of simple types. The rules in this section govern the consistent naming and structuring of these constructs and the manner of unambiguously and thoroughly documenting them.

4.1 UBL Documentation

4.1.1 The UBL Dictionary

The primary component of the UBL documentation is its dictionary. The entries in the dictionary fully define the pieces of information available to be used in UBL business messages. Each dictionary entry has a **full name** that ties the information to its standardized semantics, while the name of the corresponding XML element or attribute is only a shorthand for this full name. The rules for element and attribute naming and dictionary entry naming are different.

[d1] Each dictionary entry name *must* define *one* and only *one* **fully qualified path** (FQP) for an element or attribute.

The fully qualified path anchors the use of that construct to a particular location in a business message. The dictionary definition identifies any semantic dependencies that the FQP has on other elements and attributes within the UBL library that are not otherwise enforced or made explicit in its structural definition. The dictionary serves as a traditional data dictionary, and also serves some of the functions of traditional implementation guides in this way.

4.1.2 Other UBL Documentation

Additional components of the UBL documentation include definitions of:

- XSD complex and simple types in the UBL library, including whether and how that type maps to a core component type
- The top-level elements in UBL that contain whole UBL messages
- Global attributes
- Summaries of Code Lists
- UBL-specific Core Component Types
- UBL-specific representation terms

The UBL documentation should be automatically generated to the extent possible, using embedded documentation fields in the structural definitions.

4.1.3 Embedded documentation

4.2 General Naming Rules for XML Constructs

The following are the naming rules that apply to **all** names of XML constructs in UBL:

Names *must* use Oxford English.

Names *must not* use acronyms, abbreviations, or other word truncations, with the exception of **Identifier**. Other exceptions *may* be identified in the future.

The Representation Term **Identifier** MUST be represented in XML names as `ID`.

Names *must not* contain non-letter characters unless required by language rules.

Names *must* be in singular form unless the concept itself is plural (example: **Goods**).

Names for XML constructs *must* use “camel-case” capitalization, such that each internal word in the name begins with an initial capital followed by lowercase letters (example: `AmountContentType`). As noted below, all XML constructs other than attributes use “upper camel-case”, with the first word initial-capitalized, while attributes use “lower camel-case”, with the first word all in lowercase. Exceptions are as follows:

```
DUNS for Dun & Bradstreet numbers
```

4.3 General Overview of Types

In XSD, elements are declared to have types, and most types (those complex types that are defined to have “complex contents”) are defined as a pattern of subelements and attributes. Thus, XSD has an indirect nesting structure of elements and types (where, for example, Type 1 below is the **parent type** of Element A and where Type 2 is the parent type of Element B and the type **bound to** Element A):

- Type 1
  - Element A
    - Type 2
      - Element B

4.4 Elements and Attributes

4.4.1 Rules for UBL Elements

These rules distinguish the following constructs within the structural definitions of messages and their component parts. Note that some of these distinctions are specific to UBL and are not part of the formal definition of XML or XSD.

- Elements:

Top-level elements: Globally declared root elements, functioning at the level of a whole business message.

Lower-level elements: Locally declared elements that appear inside a business message.

Intermediate elements: Elements not at the top level that are of a complex type, only containing other elements and attributes.

Leaf elements: Elements containing only character data (though they may also have attributes). Note that, because of the XSD mechanisms involved, elements that contain only character data but also have attributes must be declared with complex types, but such elements with no attributes may be declared with simple types or complex types.

Mixed-content elements: Elements that allow both element content and data in their content models, and which may have attributes.

Empty elements: Elements that contain nothing (though they may have attributes).

4.4.1.1 Rules for the Naming and Definition of Top-Level Elements

Each UBL business message has a single root element that is a UBL top-level element. This element *must* be globally declared in a UBL root schema (which *may* contain definitions of additional root elements for other related messages in a functional area; see the Modularity, Namespaces, and Versioning paper) with a reference to a named type definition. Only top-level elements are declared globally.

Top-level elements are named according to the portion of the business process that they initiate. Example: <Order>, <AdvanceShipNotice>.

4.4.1.2 Naming and Definition of Lower-Level Elements

<!—This section has a strong dependency on the local global decision. Additionally, some of the information on naming is now redundant and has been replaced with the information in section 3 on the relationship to CCTS. After the local/global decision is made this section will be re-edited. The purpose of this section will be to elaborate and give detail on the information in Section 3.-->

4.4.1.2.1 General Rules

Lower-level elements (as well as attributes) are considered Properties of the Object Class represented by their parent type.

Lower-level elements *must* be locally declared (Note: This recommendation is now under discussion and may change) as namespace-unqualified elements by reference to a named type, whether complex or simple, and be accompanied by documentation in the form of an `<xsd:annotation>` element with an `<xsd:documentation>` element that has a `source` attribute value of "Use". The documentation specifies the use of the element within its parent type.

There are several kinds of lower-level elements, each with distinct naming rules discussed in the following sections.

<!—since we are using unqualified any customizer has to use qualified to avoid name clashes. It is very unusual to have unqualified elements and this rule is under reconsideration.-->

4.4.1.2.2 Rules for Intermediate Elements

The names of intermediate elements *must* contain the Property Term describing the element and MAY be preceded by an appropriate Qualifier term as necessary to create semantic clarity at that level. The Object Class *may* be used as a qualifier.

```
[Qualifier] + PropertyTerm
```

4.4.1.2.3 Rules for Leaf Elements

Leaf elements are named as follows:

```
[Qualifier] + PropertyTerm + RepresentationTerm
```

The naming of leaf elements follows these exceptions:

- The Representation Term **Text** is always removed.
- Leaf elements with substantially similar Property Terms and Representation Terms *must* remove the Property Term.

Examples: If the Object Class is **Goods**, the Property Term is **DeliveryDate**, and the Representation Term is **Date**, the element name is truncated to `<GoodsDeliveryDate>`; the element name for an identifier of a party `<PartyIdentificationIdentifier>` is truncated to `<PartyIdentifier>` – and then to `<PartyID>` because of the truncation rule.

4.4.1.2.4 Rules for Mixed-Content Elements

Mixed content in business documents is undesirable for a variety of reasons:

White space is difficult to handle and complicates processing.

Mixed content models allow little useful control over cardinality of elements.

For now mixed-content elements should have a Representation Term of **Prose**. This is currently under discussion with the LC SC.

4.4.1.2.5 Rules for Empty Elements

Empty elements are not permitted in UBL. For further details on the discussion details surrounding this recommendation consult the Elements vs Attributes position paper.

4.4.1.2.6 Rules Governing Elements of the Same Name and Their Respective Types

In those cases where it seems beneficial to have two elements that have the same tag name but are bound to different types, as is currently the case with the BIE Order.Header.Details (tag name Header), it is permissible.

4.4.2 Rules for the Naming and Definition of Attributes General Overview

There are two types of attribute:

- **Global attributes**: Attributes that have common semantics on the multiple elements on which they appear. These might be fixed attributes expressing an XML architectural form, attributes for assigning a unique element identifier, or attributes containing natural-language information (such as `xml:lang`).
- **Local attributes**: Attributes that are specific to the element on which they appear. Most attributes are local.

Attributes, like lower-level elements, are Properties of the Object Class represented by their parent type. They are named identically to leaf elements, except that they use lower camel-case rather than upper camel-case e.g. `amountCurrencyIDCode`.

4.4.2.1 Rules for Global Attributes

A global attribute *should* be used only when its semantics are absolutely unchanged no matter what element it's used on, AND it's made available on every single element. This rule applies to both external and UBL-specific global attributes. This allows common attributes that are everywhere but are *not* global, and that need documentation of their meaning in each XML environment in which they're used.

UBL-specific global attributes should be named just like regular attributes and subelements (i.e. as properties of an object class). Hence, by definition, the name of such a property *must* be consistent across all objects.

4.4.2.2 Rules for Local Attributes

All attributes that are not globally declared in UBL are considered to be local attributes.

Rules:

The names of the attributes are not decided yet. So we don't have any naming rules for attributes. The supplementary components have long names and we need to cut these names.

If the name of the representation term and the name of the object class of the supplementary component is the same then remove the object class that repeats the name of the representation term

Concatenate all terms removing all punctuation

If a Uniform Resource Identifier exists within a supplementary component then abbreviate it to URI.

If a representation term contains the word `text` then `text` must be omitted.

4.4.2.3 Rules for the Naming and Definition of Types

4.4.2.3.1 General Rules

In UBL all types *must* be named and therefore they are "top-level". Most UBL elements are declared locally inside complex types and are therefore "lower-level". In terms of ebXML Core Components, UBL complex types are Object Classes, subelements declared within them are Properties of those Object Classes, and the types bound to those subelements are themselves Object Classes which have their own Properties. See below:

```
[Qualifier] + ObjectClass + "Type"
```

Example: `CodeNameType`.

The definition *must* contain a structured set of XSD annotations in an `<xsd:annotation>` element with `<xsd:documentation>` elements that have `source` attribute values indicating the names of the documentation fields below:

- **UBL UID**: The unique identifier assigned to the type in the UBL library.
- **UBL Name**: The complete name (not the tag name) of the type per the UBL library.
- **Object Class**: The Object Class represented by the type.
- **UBL Definition**: Documentation of how the type is to be used, written such that it addresses the type's function as a reusable component.
- **Code Lists/Standards**: A list of potential standard code lists or other relevant standards that could provide definition of possible values not formally expressed in the UBL structural definitions.
- **Core Component UID**: The UID of the Core Component on which the Type is based.
- **Business Process Context**: A valid value describing the Business Process contexts for which this construct has been designed. Default is "In All Contexts".
- **Geopolitical/Region Context**: A valid value describing the Geopolitical/Region contexts for which this construct has been designed. Default is "In All Contexts".
- **Official Constraints Context**: A valid value describing the Official Constraints contexts for which this construct has been designed. Default is "None".
- **Product Context**: A valid value describing the Product contexts for which this construct has been designed. Default is "In All Contexts".
- **Industry Context**: A valid value describing the Industry contexts for which this construct has been designed. Default is "In All Contexts".
- **Role Context**: A valid value describing the Role contexts for which this construct has been designed. Default is "In All Contexts".
- **Supporting Role Context**: A valid value describing the Supporting Role contexts for which this construct has been designed. Default is "In All Contexts".

- **System Capabilities Context**: A valid value describing the Systems Capabilities contexts for which this construct has been designed. Default is “In All Contexts”.

The following is an extended example of the documentation fields for the type:

```
<xsd:complexType name=”PartyType”>
  <xsd:annotation>
    <xsd:documentation source=”UBL UID” xml:lang=”en”>PS1
    </xsd:documentation>
    <xsd:documentation source=”xCBL Name” xml:lang=”en”>Party
    </xsd:documentation>
    <xsd:documentation source=”Object Class” xml:lang=”en”>Party
    </xsd:documentation>
        <xsd:documentation source=”UBL Definition”
      xml:lang=”en”>
    </xsd:documentation>
    <xsd:documentation source=”Code Lists/Standards”
      xml:lang=”en”>NA
    </xsd:documentation>
    <xsd:documentation source=”Core Component UID”
      xml:lang=”en”>[None]
    </xsd:documentation>
    <xsd:documentation source=”Business Process Context”
      xml:lang=”en”>NA
    </xsd:documentation>
    <xsd:documentation source=”Geopolitical/Region Context”
      xml:lang=”en”>NA
    </xsd:documentation>
    <xsd:documentation source=”Official Constraints Context”
      xml:lang=”en”>NA
    </xsd:documentation>
    <xsd:documentation source=”Product Context”
      xml:lang=”en”>NA
    </xsd:documentation>
    <xsd:documentation source=”Industry Context”
      xml:lang=”en”>NA
    </xsd:documentation>
    <xsd:documentation source=”Supporting Role Context”
      xml:lang=”en”>NA
    </xsd:documentation>
    <xsd:documentation source=”System Capabilities Context”
      xml:lang=”en”>NA
    </xsd:documentation>
  </xsd:annotation>
  …
</xsd:complexType>
```

4.5 Containership and element design

5 Modularity, Namespaces, and Versioning

For an overview of current thinking on issues of modularity, namespace and versioning, consult the Modnamver position paper.

5.1 Schema Module Concepts

This section describes the mapping of XML namespaces onto XSD files. A namespace contains type definitions and element declarations. Any file containing type definitions and element declarations is called a SchemaModule.

Every namespace has a special SchemaModule, a RootSchema. Other namespaces dependent upon type definitions or element declaration defined in that namespace import the RootSchema and only the RootSchema.

If a namespace is small enough then it can be completely specified within the RootSchema. For larger namespaces, more SchemaModules may be defined – call these InternalModules. The RootSchema for that namespace then include those InternalModules.

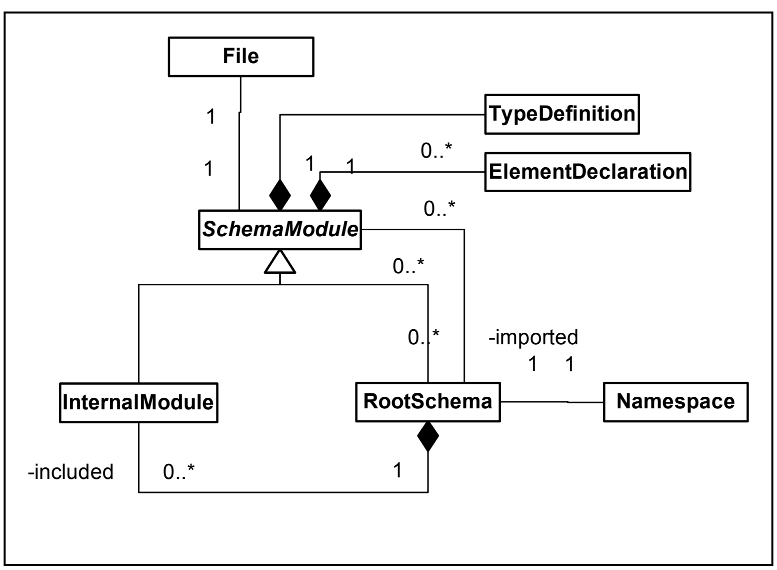
This structure provides encapsulation of namespace implementations.

A namespace "A" dependent upon type definitions or element declaration defined in another namespace "B" *must* import B's RootSchema. "A" *must not* import internal schema modules of "B".

The only place XSD "include" is used is within a RootSchema. When a namespace gets large, its type definitions and element declarations may be split into multiple SchemaModules (called InternalModules) and included by the RootSchema for that namespace.

Thus a namespace as an indivisible grouping of types. A "piece" of a namespace can never be used without all its pieces.

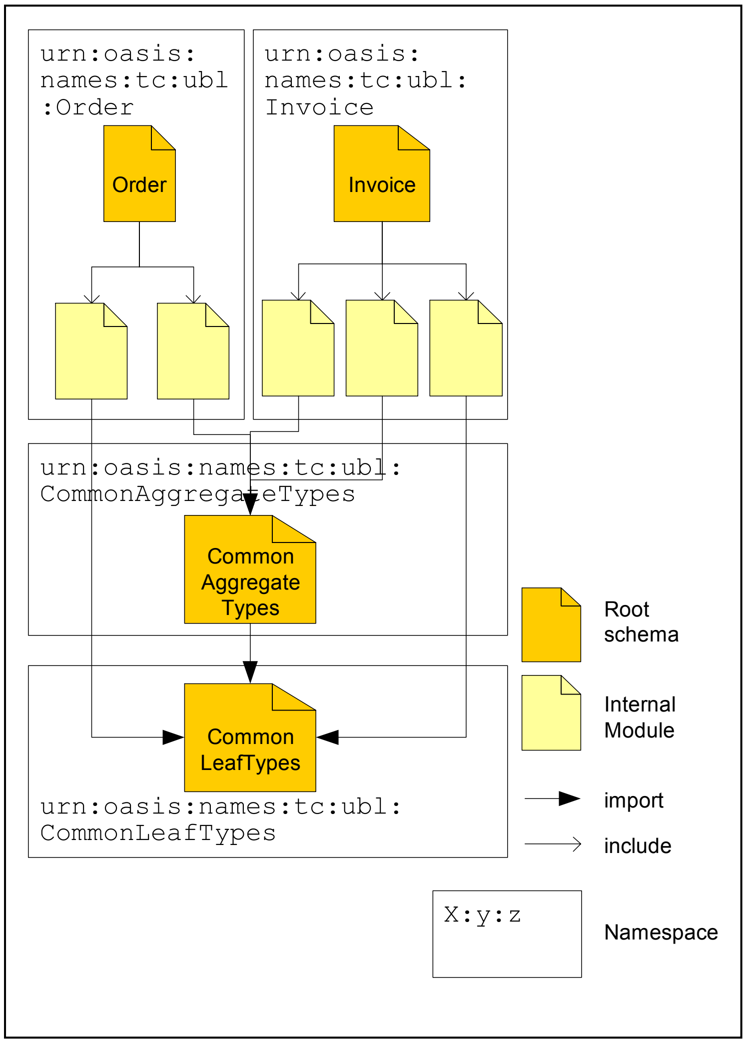
Here is a depiction of the component structure we've described so far. This is a UML Static Structure Diagram. It uses classes and associations to depict the various concepts we've been discussing:



You can see that there are two kinds of schema module: RootSchema and "InternalModule". A RootSchema may have zero or more InternalModules that it includes. Any SchemaModule, be it a RootSchema or an InternalModule may import other RootSchemas.

The diagram shows the 1-1 correspondence between RootSchemas and namespaces. It also shows the 1-1 correspondence between files and SchemaModules. A SchemaModule consists of type definitions and element declarations.

Another way to visualize the structure is by example. The following informal diagram depicts instances of the various classes from the previous diagram.



The preceeding diagram shows how the order and invoice RootSchemas import the "CommonAggregateTypes" and "CommonLeaf Types" RootSchemas. It also shows how e.g. the order RootSchema includes various InternalModules – modules local to that namespace. The clear boxes show how the various SchemaModules are grouped into namespaces.

UBL is structured so that a user can import a piece without getting the whole. It *must* be possible, for instance for a user to import the CommonLeafTypes namespace without causing the CommonAggregateTypes to be imported. It *must* be possible for a user to import the CommonAggregateTypes namespace without causing the Order namespace to be imported. It *must* be possible to import any one of the "vertical" namespaces, e.g. Order without causing another, e.g. Invoice to be imported.

If two namespaces are mutually dependent then clearly, importing one will cause the other to be imported as well. For this reason there *must not* exist circular dependencies between UBL SchemaModules. By extension, there *must not* exist circular dependencies between namespaces. This rule is not limited to *direct* dependencies – transitive dependencies must be taken into account also.

5.2 Rules for Creating Namespaces

Given the conceptual framework of the previous section, important questions remain: how many namespaces are needed? What is the function of each?

This section makes explicit the namespace structure given implicitly in the previous section. The UBL library consists of four namespaces. The Common Leaf Types namespace defines all the Basic Business Information Entities. A Common Aggregate Types namespace defines reusable Aggregate Business Information Entities based on the types defined in the Common Leaf Types namespace.

Two higher-level "domain" namespaces are defined, one for the "ordering" domain and another for the "invoicing" domain. The Order Domain namespace defines message types and ABIEs specific to the ordering domain. Similarly, the Invoice Domain namespace defines message types and ABIEs specific to the invoicing domain.

Purpose	**Namespace name**
Common Leaf Types -- this is where Basic Business Information Entities are defined.	`urn:oasis:names:tc:ubl:CommonLeafTypes[TBD version info]`
Common Aggregate Types – this is where Aggregate BIE's used across various domains are defined.	`urn:oasis:names:tc:ubl:CommonAggregateTypes[TBD version info]`
Order Domain – this is where ordering-related message types and their order-specific ABIE's are defined.	`urn:oasis:names:tc:ubl:Order[TBD version info]`
Invoice Domain – this is where invoicing-related message types and their invoicing-specific ABIE's are defined.	`urn:oasis:names:tc:ubl:Invoice[TBD version info]`

5.3 Rules for Namespace Identification

The namespace names for UBL namespaces *must* have the following structure while the schemas are at draft status:

`urn:oasis:names:tc:ubl:schema{:subtype}?:{document-id}`

When they move to specification status the form *must* change to:

```
urn:oasis:names:specification:ubl:schema{:subtype}?:{document-id}
```

Where the form of {document-id} is TBD but it should match the schema module name (see section).

5.4 Rules for Schema Module Schema Location

Schema location *must* include the complete URI which is used to identify schema modules.

In the fashion of other OASIS specifications, UBL schema modules will be located under the UBL committee directory:

```
http://www.oasis-open.org/committees/ubl/schema/<schema-mod-name>.xsd
```

Where <schema-mod-name> is the name of the schema module file. The form of that name is TBD.

5.5 Rules for Versioning

Each namespace should have a version.

6 Facets

6.1 Introduction

The following rules have been defined for the handling of facets.

6.2 Rules

The content component of a basic core component with attributes *must* be a restriction of a simple type.

For Example:

```
        <xsd:simpleType name="AmountContent">
                <xsd:restriction base="decimal">
                        <xsd:totalDigits value="31"/>
                </xsd:restriction>
        </xsd:simpleType>
```

All basic core components and basic information entities that include content components *must* use user defined types that are based on a simpleType.

Example:

```
        <xsd:simpleType name="AmountContent">
                <xsd:restriction base="decimal">
                        <xsd:totalDigits value="31"/>
                </xsd:restriction>
        </xsd:simpleType>
```

Every basic core component or basic business information entity *must* be created by a ComplexType which refers to the appropriate Simple Type inside of the element <extension>.

Example:

```
        <xsd:complexType name="Amount">
                <xsd:simpleContent>
                        <xsd:extension base="A">
                                <xsd:attribute name="currencyId"
use="required" id="000107">
                                        <xsd:simpleType>
                                                <xsd:restriction
base="token">
                                                        <xsd:length
value="3"/>
                                                </xsd:restriction>
                                        </xsd:simpleType>
                                </xsd:attribute>
                        </xsd:extension>
                </xsd:simpleContent>
        </xsd:complexType>
```

7 Date and Time

7.1 Introduction

Rules for the following aspects of time have been formulated. These aspects of time are:

- specific point of date and/or time
- durations, i.e. measurements of time
- period

7.1.1 Rules for specific points of date/time

For each specific point in time the built in datatype from XML schema (Part 2) *must* be used. These are `xsd:time`, `xsd:date`, `xsd:dateTime`.

7.1.2 Rules for duration

For the expression of the duration the XSD built in datatype xsd:Duration *must* be used. For example

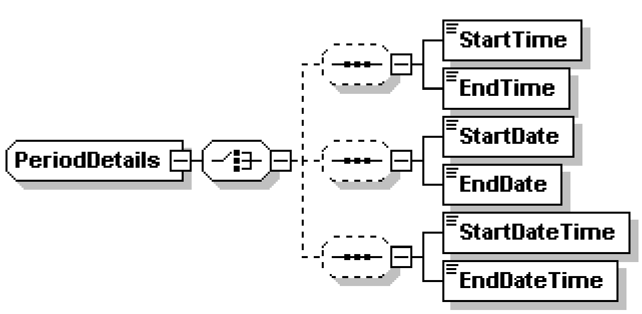
```
<simpleType name="DurationContent"/>
     <complexType name="DurationType">
        <simpleContent>
           <extension base="decimal">
         <attributeGroup ref="cct:commonAttributes"/>
              </extension>
             </simpleContent>
       </complexType>
```

7.1.3 Core Component Types and Representation Terms

There is a one to one correspondence between Core Component Types and Representation Terms. Where additional property terms like `Year`, `YearMonth`, are used then the additional built in datatypes from XML Schema part 2 *must* be used. These additional datatypes are: `xsd:gYear`, `xsd:gYearMonth`, `xsd:gMonth`, `xsd:gMonthDay`, and xsd:gDay.

7.1.4 Period

A period can be expressed use the Aggregate Core Component (ACC) `PeriodDetails`. The ACC is divided into 3 representation types, Date, Time and DateTime. One of these *must* be selected. Each option has a start and end date, start and end time or start `DateTime` and end `DateTime`.



XML-Schema:

```
	<complexType name="PeriodDetails">
		<sequence>
			<choice>
				<element name="StartTime"
type="cct:TimeType"/>
				<element name="StartDate"
type="cct:DateType"/>
				<element name="StartDateTime"
type="cct:DateTimeType"/>
			</choice>
			<choice>
				<element name="EndTime"
type="cct:TimeType"/>
				<element name="EndDate"
type="cct:DateType"/>
				<element name="EndDateTime"
type="cct:DateTimeType"/>
			</choice>
		</sequence>
	</complexType>
```

XML-Instance:

```
	<ValidityPeriod>
		<StartDate>1967-08-13</StartDate>
		<EndDate>1967-08-13</EndDate>
```

This example is stating that the validity period is from the 13th Aug 1967 to 13th August 1967, i.e. that day.

For each representation term the equivalent data type must be used.

8 Rules for Context

For an overview of current thinking on Context Rules, consult the Specialization Architecture position paper from the Context Methodology Subcommittee.

9 Code Lists

See the separate Code List Recommendation paper for details of the NDRSC's recommendations for code lists.

10 UBL Messages

10.1 General Message Rules

The following general rules for messages *must* be applied.

- A UBL message set may be extended where desirable if the business function of the UBL original is retained., but the message exists within its own business context.
- According to the XML Recommendation **[XML]**, the legal characters in XML character data are tab, carriage return, line feed, and the legal characters of Unicode and ISO/IEC 10646, as these standards are updated from time to time. It further notes that "The mechanism for encoding character code points into bit patterns may vary from entity to entity" and requires all XML processors (parsers) to accept the UTF-8 and UTF-16 encodings of 10646. UBL has the same requirements for legal characters in XML instance documents and the same minimal requirements for character encoding support in UBL-aware software. Trading partners *may* agree on other character encodings to use among themselves. It is recommended in all case that encoding declarations be provided in the XML declarations of UBL documents.
- UBL messages *must* express semantics fully in schemas and not rely merely on well-formedness.
- Instances conforming to schemas *should* be readable and understandable, and *should* enable reasonably intuitive interactions.
- In the context of a schema, information that expresses correspondences between data elements in different classification schemes (“mappings”) *may* be regarded as metadata. This information *should* be accessible in the same manner as the rest of the information in the schema.

11 References

[CCTS] *UN/CEFACT Draft Core Components Specification* 30 September, 2002, Version 1.85

[CCFeedback] *Feedback from OASIS UBL TC to Draft Core Components Specification 1.8*, version 5.2, May 4, 2002, http://oasis-open.org/committees/ubl/lcsc/doc/ubl-cctscomments-5p2.pdf.

[GOF] *Design Patterns,* Gamma, et al. ISBN 0201633612

[ISONaming] *ISO/IEC 11179,* Final committee draft, Parts 1-6.

[RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

[UBLChart] UBL TC Charter, http://oasis-open.org/committees/ubl/charter/ubl.htm

[XML] *Extensible Markup Language (XML) 1.0* (Second Edition), W3C Recommendation, October 6, 2000

[XSD] *XML Schema*, W3C Recommendations Parts 0, 1, and 2. 2 May 2001.

12 Technical Terminology

Application-level validation	Adherence to business requirements, such as valid account numbers.
Ad hoc schema processing	Doing partial schema processing, but not with official schema validator software; e.g., reading through schema to get the default values out of it.
Assembly	Using parts of the library of reusable UBL components to create a new kind of business document type.
Context	A particular set of context driver values.
DTD validation	Adherence to an XML 1.0 DTD.
Instance constraint checking	Additional validation checking of an instance, beyond what XSD makes available, that relies only on constraints describable in terms of the instance and not additional business knowledge; e.g., checking co-occurrence constraints across elements and attributes. Such constraints might be able to be described in terms of Schematron.
Generic BIE	A semantic model that has a “zeroed” context. We are assuming that it covers the requirements of 80% of business uses, and therefore is useful in that state.
Instance root/doctype	This is still mushy. The transitive closure of all the declarations imported from whatever namespaces are necessary. A doctype may have several namespaces used within it.
Root Schema	A schema document corresponding to a single namespace, which is likely to pull in (by including or importing) schema modules. Issue: Should a root schema always pull in the “meat” of the definitions for that namespace, regardless of how small it is?
Schema	Never use this term unqualified!
Schema Module	A “schema document” (as defined by the XSD spec) that is intended to be taken in combination with other such schema documents to be used.
Schema Processing	Schema validation checking plus provision of default values and provision of new infoset properties.
Schema Validation	Adherence to an XSD schema.
Well-Formedness Checking	Basic XML 1.0 adherence.

Appendix A. Notices