UBL NDR Position Papers

First Public Release
16 March 2002

This version:

http://oasis-open.org/committees/ubl/200203/ndrsc/review/draft-ndr-20020316.pdf

Previous version:

none

Intended audience:

EDI experts, business experts, and XML experts interested in the development of an international standard for basic XML business schemas.

Status:

This is the first public-review distribution of position papers produced by the OASIS UBL Naming and Design Rules Subcommittee. You can find additional NDR SC materials at the NDR portal:

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

These papers are approaching completion, but they are in different stages of development. We have been working with the people in the Library Content SC, particularly those generating the schemas, to test out the recommendations in these papers. More testing will follow, and we would be grateful for additional review.

Please send any comments to ubl-comment@lists.oasis-open.org.

Position Paper: Definition of Elements, Attributes, and Types

Authors: Mark Crawford (mcrawford@lmi.org), Arofan Gregory (arofan.gregory@commerceone.com), Eve Maler (eve.maler@sun.com)

Date: 16 March 2002

Filename: draft-arofan-tagspec-03.doc

Status: Unreviewed by the Naming and Design Rules SC in its V03 form but deemed ready for external review regardless.

1 Definition of Elements, Attributes, and Types

In W3C XML Schema (known as XSD), 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.

1.1 Relation of XML Constructs to ISO 11179 and ebXML Core Components

These rules refer to the following concepts taken from ISO 11179 and used subsequently in the ebXML Core Components work: *(TBD: need formal references)*

- Object Class
- Property Term
- Qualifier
- Representation Term (RT)
- Core Component Type (CCT)

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…

In UBL, types are all named and therefore “top-level”, whereas most 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.

Rules are given below on documenting XML constructs to indicate the unambiguous relationship of each construct to its corresponding Core Component-based semantic representation.

1.2 UBL Documentation

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.

Each dictionary entry defines 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.

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
- *(TBD: possibly others, including summaries of code lists, UBL-specific core component types, and UBL-specific representation terms; for RTs, we're supposed to start with the official CC list and liaise with UN/CEFACT in proposing new ones that we need to add for our own purposes)*

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

*(Note: Throughout this paper, the rules for using the **xsd:documentation** element's **source** attribute are incorrect; it is supposed to be a URI, not a keyword. This will be corrected in the next version.)*

1.2.1 Naming Rules for Dictionary Full Names

The fully qualified path for an element or attribute is constructed as follows:

(TBD)

1.2.2 Contents of Dictionary Entries

(TBD)

1.2.3 Contents of Other UBL Documentation

(TBD)

1.3 XML Constructs in UBL

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).
- **Attributes:**
    - **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.
- **Types**: Complex or simple XSD types. Note that UBL has no anonymous types; all types are assigned a name in their definition. In the UBL structural definitions, all complex type definitions should be grouped together, and all simple types similarly grouped together, for ease of reference.

The following sections define the naming and usage rules of these constructs.

1.3.1 General Naming Rules for XML Constructs

Following are the naming rules that apply to all names of XML constructs in UBL:

1. Names MUST use Oxford English.
2. *(TBD: Tentative; needs more Library Content SC input)* Names of XML constructs MUST NOT use non-alphabetic delimiters.
3. Names MUST NOT use acronyms, abbreviations, or other word truncations, with the following exceptions:
    - The Representation Term **Identifier** MUST be represented in XML names as **ID**.

- *(More TBD)*

4. Names MUST NOT contain non-letter characters unless required by language rules. *(More TBD)*
5. Names MUST be in singular form unless the concept itself is plural (example: **Goods**).
6. 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
   - *(More TBD; should these be enumerated, or can a more general rule be stated or referred to?)*

1.3.2 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. *(Note: This rule is proposed, but has not yet been decided as a recommendation of the Naming and Design Rules SC.)*

Example: **Order**, **AdvanceShipNotice**.

1.3.3 Naming and Definition of Lower-Level Elements

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 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. *(TBD: Our future work on role models may end up modifying these rules. E.g., right now we assume implicitly that the type bound to the element is used somehow in the property term name, but this need not be the case.)*

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

Example: *(TBD).*

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.
- *(More TBD)*

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.

Mixed-content elements are considered to be leaf elements with a Representation Term of **Prose**. *(Note: This rule is proposed, but has not yet been decided as a recommendation of the Naming and Design Rules SC.)*

Empty elements are named as follows:

(TBD)

Example: *(TBD).*

(TBD: Rules governing elements of the same name and their respective types.)

The following extended example shows a complex type that locally declares two lower-level elements:

```
<xsd:complexType name="…">
  <xsd:sequence>
    <xsd:element name="Name" type="NameType"
      minOccurs="1" maxOccurs="1">
      <xsd:annotation>
        <xsd:documentation source="Use">
The name information for an entity.
        <xsd:documentation>
    </xsd:element>
    <xsd:element name="Address" type="AddressType"
      minOccurs="0" maxOccurs="1">
      <xsd:annotation>
        <xsd:documentation source="Use">
The address information for an entity.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Following is another extended example of the documentation fields for the locally declared elements within their parent type:

```
<xsd:complexType name="…">
```

```
  <xsd:sequence>
    <xsd:element name=”PartyID” type=”IdentifierType”
      minOccurs=”1” maxOccurs=”1”>
      <xsd:annotation>
        <xsd:documentation source=”Use”>
A standard identification of an entity doing business as assigned
By a standards agency.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name=”MDFBusiness” type=”xsd:Boolean”
      minOccurs=”1” maxOccurs=”1”>
      <xsd:annotation>
       <xsd:documentation source=”Use”>
An indicator of whether the party is a minority, disadvantaged,
or female owned business.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    …
  </xsd:sequence>
</xsd:complexType>
```

1.3.4 Naming and Definition of Attributes

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.

Example: **amountCurrencyIDCode**. *(TBD: Is this a good example?)*

(TBD: Do global attributes have any differences in naming or declaration from regular local attributes?)

1.3.5 Naming and Definition of Types

Complex XSD types in UBL declare (usually) a set of local elements and (possibly) some attributes. These types correspond to Object Classes, with the local elements and the attributes corresponding to Properties of that Object Class. *(TBD: There may be a few exceptions for complex types that serve merely as convenient XML “containers” and do not correspond in a semantically significant way to Object Classes. We have not identified any of these yet.)*

All types MUST have names (that is, they are not anonymous) and MUST appear as top-level constructs in UBL schema modules (that is, they are not embedded within element or attribute declarations). The type name is the Object Class name, with “Type” appended and with a Qualifier optionally prepended:

[Qualifier] + ObjectClass + “Type”

Example: **CodeNameType**.

(TBD: How should the naming of simple types, and complex types that contain ***simpleContent****, differ from regular complex types? For example, are Representation Terms used?)*

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:

[TBD: We need to specify which sets of values are used for Contexts (reference to the official UBL list), and we also need to present the controlled lists of Representation Terms. Finally, we need to reference an official version of the Core Components Library, if possible, so that the UIDs can be resolved.]

- **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.
- **Property Term**: The Property Term of the type. *(TBD: Won't this always be NA?)*
- **Representation Term**: The representation term of the type.
- **Core Component Type**: The CCT per the UBL list.
- **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".

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=”Property Term” xml:lang=”en”>NA
    </xsd:documentation>
    <xsd:documentation source=”Representation Term”
      xml:lang=”en”>Details
    </xsd:documentation>
    <xsd:documentation source=”Core Component Type”
      xml:lang=”en”>NA
    </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>
```

Position Paper: Code Lists

Author: Eve Maler (eve.maler@sun.com)

Date: 27 February 2002

Filename: draft-maler-codelists-04.doc

1 Code Lists

As defined in the Core Components specification, V1.8, a **code** is:

> "A character string (letters, figures or symbols) that for brevity and/or language independence may be used to represent or replace a definitive value or text of an attribute. Codes usually are maintained in code lists per attribute type (e.g. colour)."

It has the core component type **Code. Type**; however, this type assignment does not require it to be handled in any particular way in syntax bindings, such as in XSD by an enumeration of strings.

A **code list**, for our purposes, is a closed set of codes (possibly with a provision for indicating custom codes) that is defined and maintained by an organization along with documentation of the meaning of each code.

An **external code list**, for our purposes, is a code list that is maintained by an organization other than the UBL SC and incorporated into UBL by reference. An **internal code list**, for our purposes, is a code list that is defined in the body of the UBL set of specifications. Thus, a code list that is considered internal from the perspective of ANSI X12 might be considered external from the perspective of UBL.

On 13 February 2002, the NDR SC agreed to the following proposal:

"We should use external code lists as much as possible, and in those cases leave validation and subsetting up to the application (except perhaps for pattern matching). We should create our own validatable code lists sparingly. This is a short-term solution. In the long term, we would have the option to use validatable forms of the external code lists provided by external organizations."

This position paper proposes a specific formulation of this solution that is designed to be suitable for use in the NDR document.

> **Note:** All naming and markup design in examples in this paper is ad hoc and does not necessarily adhere to the NDR rules developed to date.

1.1 Design Principles

The definition and management of code lists in UBL adheres to the following design principles:

- **Semantic clarity**

  It must be possible to interpret the meaning of any non-custom code (and also, ideally, any custom code as well) accurately and consistently. Thus, it must be possible to uniquely identify the relevant code list for each UBL markup construct that contains a code, and as a corollary, it must be possible to distinguish between different versions of the "same" code list in case of backwards-incompatible changes. We should encourage documentation of custom codes to the extent possible.

- **Management of code list maintenance costs**

It is expensive to maintain internal versions of code lists that already exist externally. Also, it is expensive to develop new code lists. UBL should try to leverage existing work where possible.

- **Validation**

  It should be possible to validate that a legitimate code from a code list is being used, but some or all of this validation may happen at run time, using application-specific means.

- **Subsetting**

  It should be possible to restrict the legitimate codes available.

- **Extension**

  It should be possible to add to the universe of possible codes that can be used in a UBL construct, but the new codes should be given semantic clarity.

1.2 Criteria for Choosing and Defining Code Lists

Where possible, external code lists should be used in preference to internal code lists in the design of UBL. Potential reasons for designing an internal code list include the need to combine multiple existing external code lists, or the lack of any suitable external code list. The lack of "easy-to-read" or "easy-to-understand" codes in an otherwise suitable code list is not sufficient reason to define an internal code list.

1.3 Documenting UBL Use of External Code Lists

UBL must document the following items related to code lists:

- For a specific version of each internal and external code list used by UBL: A URI reference (in the style of XML namespace names) that UBL will use to refer to that list

  Since most external standards bodies have not defined such a URI reference for the code lists under their purview, in these cases UBL must define its own URI references to stand for these lists.

- The requirements that UBL extensions must follow in documenting code lists of their own invention

- For each UBL element or attribute containing a code: An indication (by mention of the corresponding URI references) of the one or more code lists that must be minimally supported when the construct is used, and, if necessary, the specific version of the code list associated with this version of UBL

  If an external code list is updated without a corresponding update to UBL and new codes have been added to the list, these new codes may legitimately be used in UBL documents (with an expectation that document recipients may not be configured to handle them). However, existing codes are to be interpreted strictly as in the version of the code list identified in the UBL documentation. If any codes change in a backwards-incompatible fashion, it is an error to interpret a code in the sense defined by the new version until UBL itself is updated.

1.4 Code List Namespaces

Issue: Do we need to recommend a basic style of URI reference for external code lists? Example URIs are used below, but they are not normative. Who invents these? Who maintains the list? Where does the list appear in the documentation?

1.5 Code List Schema Framework

The mechanism for handling all appearances of codes in UBL markup is the same, whether the code is internal or external. The code is an XML qualified name, or "QName", consisting of a namespace prefix and a local part separated by a colon. Following is an example of a QName, where "baskin" is the namespace prefix and "Chocolate" is the local part:

```
baskin:Chocolate
```

QNames are defined by the built-in XSD simple type called `QName`. The schema definition of UBL must make reference to a UBL type based on `QName` wherever a code is allowed to appear, rather than enumerating a closed set of value options. For example:

```
<xsd:simpleType name="UBLCodeType">
  <xsd:restriction base="xsd:QName"/>
</xsd:simpleType>
…
<xsd:element name="IceCream">
  <xsd:attribute
    name="IceCreamFlavorCode" type="UBLCodeType" use="required"/>
</xsd:element>
```

The intent is for the namespace prefix in the QName to be mapped, through the use of the `xmlns` attribute as part of the normal XML Namespace mechanism, to a URI reference that stands for the code list from which the code comes. The local part identifies the actual code in the list that is desired. Following is an example of a mapping of the "baskin" prefix to Version 1.0 of a Baskins-Robbins ice cream flavor namespace, assuming that UBL has had to define its own URI reference for this namespace:

```
<IceCream
  xmlns:baskin="http://www.oasis-open.org/committees/ubl/codelists/BR31-V1.0"
  IceCreamFlavorCode="baskin:Chocolate"/>
```

As noted in Section 1.3, the documentation for the `IceCreamFlavorCode` attribute must indicate the minimum code lists that are expected to be used in this attribute. However, the attribute is allowed to contain codes from additional code lists, as long as they are in the form of a QName.

Applications that produce and consume UBL documents are responsible for validating and interpreting the codes contained in the documents.

1.6 Creating and Using Code List Extensions and Subsets

If it is desired to supply a code that is not in any of the code lists identified as being minimally supported for a particular field, but the desired code is in a code list that is already defined with a namespace, the creator of the UBL document need only supply the corresponding QName. For example:

```
<IceCream
  xmlns:un="http://www.oasis-open.org/committees/ubl/codelists/UN-icecream"
  IceCreamFlavorCode="un:ChocolateChocolateChip"/>
```

If it is desired to supply a code that is neither in the minimally supported code lists for the field nor in any other code lists already defined, an extension designer must create a new external code list in a new namespace. For example:

```
<IceCream
  xmlns:my=http://www.example.com/codes/icecream/V1.3"
  IceCreamFlavorCode="my:DragonflyRipple"/>
```

There is no need for this usage to be associated with XSD code. It is not necessary to use the context methodology to indicate where such custom code lists are expected to be used.

> **Issue**: Should we recommend/require the use of the context methodology for doing this? Even if not, there is an issue of how it would accommodate such a thing even on a volunteer basis.

As noted in Section 1.3, it is intended that the extension namespace (code list) be documented sufficiently by the extension designer to provide semantic clarity when the codes from this list are used.

If it is desired to define an explicit subset of an existing code list, rather than building an implicit understanding of subsets into applications, a subset designer may create a new external code list in a new namespace that contains the desired subset. In this case, it is critical that the documentation of the namespace (code list) include a mapping back to the codes on which it is based.

1.7 Code List Validation Futures

The QName solution is considered short-term. In the future, if any of the organizations that maintain UBL-referenced code lists choose to offer a schema-based representation of the code lists that can be incorporated into UBL for greater validation, UBL may consider incorporating them.

However, for maximum flexibility with maximum semantic clarity in the long term, the ideal solution might be for QNames to be able to be validated according to, respectively, the namespace *URI* and the local part, not the namespace *prefix* and the local part. The reason for this is that the prefix is merely an indirection mechanism to get to the URI reference, and is insignificant – and potentially variable – all by itself.

Thus, until such time as this type of validation becomes an option in XSD validation, schema modules that are non-QName-based (for example, enumerated lists of non-prefixed codes or codes with hard-wired namespace prefixes) may not be very helpful to any version of UBL that uses the QName solution. And unfortunately, schema modules that *are* QName-based offer just as little “early validation” as the solution proposed here.

Elements versus Attributes

18. March 2002
Gunther Stuhec
Verteiler: UBL-Group

1 Introduction

A common cause of confusion, or at least uncertainty, in the design of a schemas is the choice between specifying parts of the document as elements or attributes. Elements and Attributes are both containers for information. Many times the choice between an Element and an Attribute seems very arbitrary, almost matter of style.

There is some information that could go either way. For example, *Country* could be an Attribute or an Element. Neither way is right or wrong, it is just a choice. While the choice may indeed be arbitrary in some cases, the 'typical' roles of Elements and Attributes and the different types of content models and constraints of these two containers will be explained in this document very shortly.

2 Characteristics

The fundamental difference between Elements and Attributes in XML 1.0 is to be define the limits of what the two containers can be used for. It means that elements can contain child elements as well as content and attributes can only hold content only**.** The distinction between attribute and content element then becomes the distinction between an attribute and a containment relationship with another object.

The following table shows the elementary differences of Elements and Attributes:

Elements	Attributes
Can have child Elements nested within them	Can't have nested Elements or Attributes; can contain only strings, or lists of strings
Typically used for structured data items but can be and are used for simple data items as well	Typically used for "atomic" items of data
Elements must appear in the order specified in the schema, but may appear several times.	Each Attribute of a particular Element can only be specified once, but more than one Attribute inside of one Element can be specified in any order
Elements usually represent the natural, core content, which would generally appear in every	Attributes represent data of secondary importance; often metadata?

printout/display?	
(Sub-)Elements usually represent parts of an Element	Attributes usually represent properties of an Element

2.1 Elements

Elements are logical units of information in a schema. They represent information objects. Elements either contain information (text), or have a structure of subelements. Therefore elements are good for representing structurally significant information.

Elements are more extensible than attributes in an evolving standard because elements can contain other elements or substructures directly while attributes cannot. If a concept is defined as an attribute initially, and then needs to be expanded to hold fine-grained information, it must be changed to an element to be modeled correctly.

Elements can have attributes attached to them as metadata, while attributes cannot. Elements are repeatable within the same container structure, but attributes can only appear once in the attribute list of an element. In addition, if order of occurrence is significant, elements are the only option because attributes do not have order.

2.2 Attributes

Attributes are atomic, referentially transparant characteristics of an object that have no identity of their own. Generally this corresponds to primitive data types (e.g., Strings, Date, etc.). Taking a more logical view, an attribute names some characteristic of an object that models part of its internal state, and is not considered an object in its own right. That is, no other objects have relationships to an attribute of an object, but rather to the object itself.

Attributes can be divided into the following types:

- The type of attribute that relating to element identification (ID and IDREF type attributes, and those attributes of type CDATA that have application-specific identification rules, such as the name attribute of the A element in HTML)
- Those containing tokens that identify one or more contexts in which the element applies, or which identify one or more options to be used during processing of the element (entity names, notation names, name tokens or values from a predefined set of tokens)
- Those tokens that carry data to be used as part of the application (typically CDATA type attributes).
- Attributes can also be describing the characteristics of information: a property of an information object. For example, notation attributes clearly define the coding of the data within the element, and so clearly control the processing of the contents. Similarly entity attributes clearly identify external, unparsed, entities that will need to be processed according to the rules applicable to the notation defined in the entity declaration.

The general characteristics of attributes are:

- Attribute values can have no substructure

- Attributes are unordered, so there is no standard way to specify that one attribute's value should precede the other's (there is no guarantee that an API will give you the attributes in the same order that you specified them)
- Attributes can only contain multiple values if they are tokens (*e.g.*, NMTOKEN) or references to other elements (*e.g.*, IDREFS)
- Attributes can only describe structures by using for example “xsi:type” and they can link to them (IDREF or ENTITY) but they cannot contain subelements directly in markup.

3 Advantages and Disadvantages of Attributes

It is much more easier to describe any general rule for using attributes esspecially, if the advantages and disadvantages are putted into the opposite before.

The advantages of using Attributes are:

- In XML 1.0, and in the XML Schema, only attributes may have default values assigned to them by the schema.
- Attributes can have names that indicate the role the value plays in the element. Element contents have content names, but there can be by Attributes only to say what role the content plays in any particular element that contains it.
- Attributes have (minimal) data types.
- Attributes take up less space as there is no need for an end tag. Using attributes for data points results in a drastically smaller document representing the same information.
- Attributes are easier to access in DOM.
- Attributes can be built in are unordered.
- Attributes can be used for data points disambiguates structure and information. Code is much cleaner when using attributes for data points – attributes always contain data points, and elements always contain structure.
- When extracting information from an XML document to store to an RDBMS, or vice-versa, using attributes for data points forms a very clean mapping between the systems - attributes always correspond to columns, while elements always correspond to tables. This makes code to import and export data between RDBMS systems and XML documents easy to write and very flexible.
- Attributes can be constrained against a predefined list of enumerated values.
- Attributes can have default values.
- Attributes are concise and easier to parse than elements.

The disadvantages of using Attributes are:

- Attributes aren't as convenient for long text, large values, or binary entities.
- Attributes can't contain other elements. Therefore, there can't contain nested info.
- Attribute values are harder to search for in search engines
- Attribute values often don't appear on the screen in editing tools (you have to open a special dialog or popup to see them)
- Attribute values can be slightly more awkward to access in processing APIs

- Attributes are ambiguity and not expandable for future changes. Each attribute is either there or not. There is no way to indicate that if you provide this one, you can't provide that one, or if these two are present, then you can't have that one, or if this one is present, then that one has to also be present, and so on
- Whitespace can't be ignored in an attribute value.
- Attributes can only contain multiple values by using tokens.
- Attributes can describe structures in a difficult form by using “xsi:type” only. There is no way to describe a srtucture by using like child elements.
- Attributes are more difficult to manipulate by program code

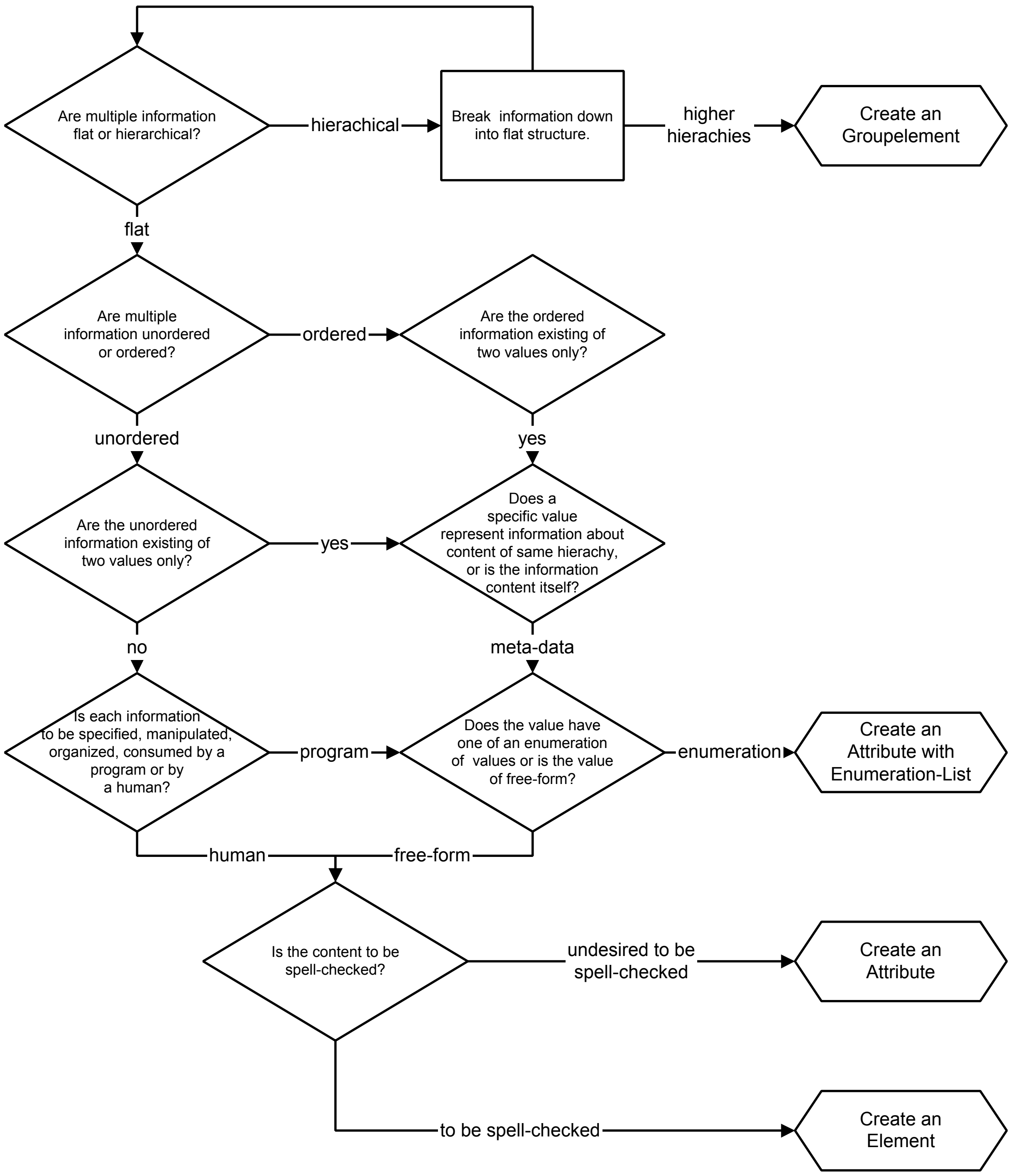
4 Guidelines

Attributes can actually be used to display of the information what would otherwise be displayed withing the child elements. How can be done a decision when a piece of information is an child-element or an attribute? Tim Bray has written to this proplem: "...when the property has a simple value like a string, we put that in the content of the element; when the property's value is another object, we put a pointer to it in an attribute value and leave the element decribing the property empty."

That solution is one way but a efficient choice for definition depents not on values only. It must be done additionally a consideration of the limitations and special properties of Elements and Attributes which are depending on the disadvantages and advantages of each too.The following considertions may be helpful for using of Elements or Attributes:

- The definition of an Element is advantangeous if the document property relate to the structure of the document.
- An Element should be used to represent a piece of information that can be considered an indpendent object.
- An Element should be used when the information is related via a parent/child relationship to another piece of information. In this case, the element is also a subelement of the element to which it is related.
- An Attribute should be used to represent any information "left over" after defining the objects that have relationships to other objects (and should thus be elements and subelements).
- An Attribute should be used to represent any information that describes other information, such as a status or id.
- An Elemente must be used, if an item needs to occur multiple times, because attributes can have only occur once in an element.
- An Attribut is very useful, if it necessary to limit values to a predefined list, since it is possible to specify a valid list of values for an element.
- Attributes are a better choice, to minimize the file size of target documents.

The following diagram illustrates a way to find out how want to be an Element or an Attributes necessary to be define it. This definition process depends by considering the limitation and special propertiers which are in the following diagram included.


Are multiple information flat or hierarchical?
hierachical
Break information down into flat structure.
higher hierachies
Create an Groupelement
flat
Are multiple information unordered or ordered?
ordered
Are the ordered information existing of two values only?
unordered
yes
Are the unordered information existing of two values only?
yes
Does a specific value represent information about content of same hierachy, or is the information content itself?
no
meta-data
Is each information to be specified, manipulated, organized, consumed by a program or by a human?
program
Does the value have one of an enumeration of values or is the value of free-form?
enumeration
Create an Attribute with Enumeration-List
human
free-form
Is the content to be spell-checked?
undesired to be spell-checked
Create an Attribute
to be spell-checked
Create an Element

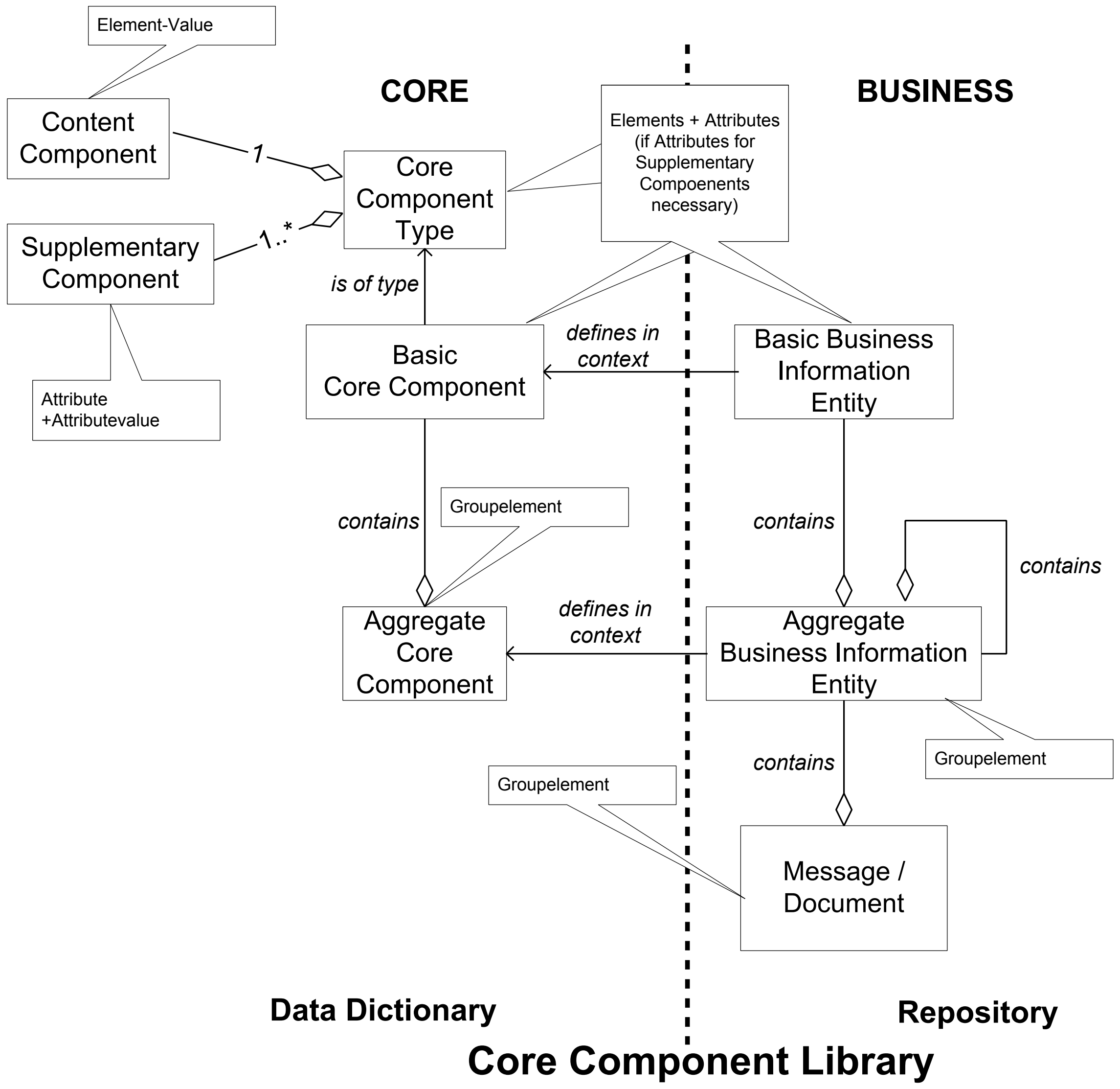
5 Recommendation

In the Core Components Technical Specification a Core Component Type will be used for the creation of Core Components. It consists of one Content Component for the value and one ore more Suplementary Components for giving an essential extra definition to the Core Component itself. The Core Component Type will be used for creation of Basic Core Component (BBC) or Aggregate Core Components (ACC) respectively, which are necessary for building of Basic Business Information Entities (BBIEs) or Aggregate Business Information Entities.

Since this BBIEs are a derivation of BCCs and must have a human-readable business semantic definition, the BCCs itself has to be defined as Elements. The content of each Component Content are to be spell-checked in the most of situations. Therefore the Component Content will be represented as an Element-Value.

The Supplementary Components will be represented as Attributes. Since, as the most of the information of each Supplementary Components are restricting attributes, will be used by programs and represented can be represented in a unordered form. Furthermore, the Supplementary Components could be including information as enumerations.

All Aggregate Components (ACCs and ABIEs) are nodes in an hierachical order and nodes inside of hierachies will be defined as Groupelements. The following figure describes the relationship between the Core Components and the Business Information Entities and type of representation in XML-syntax of each component.

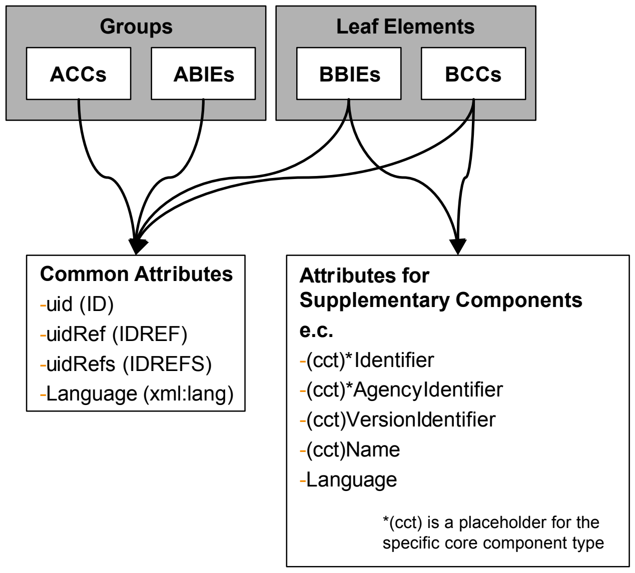


6 Proposal

I would like to do the following proposal for using attributes:

1. Attributes must be used for representation of the Global Unique Identifier by using ID within all components like BCC, ACC, BBIE and ABIE.
2. Attributes should be used for defining the relationship between components. The relationships in XML can also be represented with ID-IDREF(S) attributes. Using these attributes, an element may refer to one or more other elements (by including the value of those elements' ID fields in the pointing element's own IDREF or IDREFS field). While this may seem to be directly analogous to a relational database's key mechanisms, there's one important difference: Most parsers treat these pointers as unidirectional. In other words, given an IDREF or IDREFS field, it's possible to quickly find the element or elements with the associated ID or IDs, but not the other way around. As you'll see when I discuss modeling solutions, this turns out to be a real impediment to design.
3. An attribute should be be used, if the tagname of each CC or BIE will be represented in another language as the Oxford English language. The language Oxford English will be used as default.
4. Attributes should be used only inside of Core Component Types which are used for defining of the Basic Core Components (Leaf Elements).
5. Attributes should be used only for defining of supplementary components only. The supplementary components are fixed defined inside of the ebXML Core Component Specification and must not expanded normally.
6. The content component should be defined only as an element content of each leaf element.
7. An attribute should be not necessary, if it exists a default value for the specific supplementary component.

As the following diagram shows, it will be two types of attributes are necessary:



The summary of the properties and the advantages of the proposed way is:

- For each Basic Core Component (BCC) is only one leaf-element necessary. We don't need a element group, which includes a bunch of child elements for the content components as well as for the supplementary components.
- This definition is well structured and easy to read / easy to understand by a user.
- On the other hand the context-dependent BIEs can be easily used in the OO-design and in the implementing coding.
- The information about supplementary components contained in the attribute value only. This attribute values can be omitted in the instances, if the default value is defined.

6.1 Empty Elements

All of the following type of empty elements are not necessary for building Basic Core Components (BCCs) are Basic Business Information Entities (BBIEs) respectively:

```
<ElementName/>

<ElementName></ElementName>

<ElementName attributeName=”Value”/>

<ElementName attributeName=”Value”></ElementName>
```

Every BBIE derived from a BCC includes a content which is expressed by the element value of a leaf element. Otherwise, it is a content not needed, the specific BBIE must not to be expressed.

6.2 Common Attributes

For the definition of the common attributes (ID, IDREF, IDREFs and language) which will be used within every Core Component and Business Information Entity respectively, there is a attributegroup (the choosen name of that attribute group is “UidAttributeGroup” yet) defined. This attribute group includes the following attributes:

- uid – The attribute “uid” identify each CC or BIE uniquely by expressing the GUID (Global Unique Identifier). The “uid” based on the built-in datatype “ID”. The ID must be represented in every CC and BIE.ID represents the ID attribute type from [XML 1.0 (Second Edition)].
- uidRef – The attribute “uidRef” use a single IDREF relationship to point the relating element back to the element it needs to reference. IDREF represents the IDREF attribute type from [XML 1.0 (Second Edition)].
- uidRefs – The attribute “uidRefs” based on the built-in datatype IDREFS. IDREFS can habve serveral targets (IDs). IDREFS represents the IDREFS attribute type from [XML 1.0 (Second Edition)].
- xs:language – The Attribute “language” may be inserted in documents to specify the language used for the tagnames for BIEs and CCs. The attribute represents natural

language identifiers as defined by [RFC 1766]. It will be used, if the tagname of each CC or BIE will be not in the Oxford English language. The language Oxford English will be used as default.

namespace	CoreComponentTypes.xsd						
used by	complexTypes	**AmountType CodeType DateTimeType IdentifierType MeasureType NumericType QuantityType TextType**					
attributes	Name	Type	Use	Default	Fixed	Annotation	
	uid	xs:ID	required				
	uidRef	xs:IDREF	optional				
	uidRefs	xs:IDREFS	optional				
	language	language	optional	en			

source	`<attributeGroup name="UidAttributeGroup">`
	`<attribute name="uid" type="xs:ID" use="required"/>`
	`<attribute name="uidRef" type="xs:IDREF" use="optional"/>`
	`<attribute name="uidRefs" type="xs:IDREFS" use="optional"/>`
	`<attribute name="lang" type="language" use="optional" default="en"/>`
	`</attributeGroup>`						

6.3 Attributes for Supplementary Components

Attributes are useful for supplementary components especially. This will show the first example:

```
<complexType name="AmountType" id="000105">
  <annotation>
    <documentation source="CCTS V1.7" xml:lang="en">
      A number of monetary units specified in a currency where the unit of currency is explicit or implied.
    </documentation>
  </annotation>
  <simpleContent>
    <extension base="cct:AmountContentType">
      <attribute name="amountCurrencyIdentificationCode" type="cct:AmountCurrencyIdentificationCodeType"/>
      <attributeGroup ref="cct:UidAttributeGroup"/>
    </extension>
  </simpleContent>
</complexType>
```

The first example represents the core component type "AmountType". The AmountType is derived by the content component "AmountContentType". Therefore it is possible, that the value of AmountType will represent in the derived Core Component directly and not by an additional child element. And the "AmountType" includes on supplementary component "AmountCurrencyIdentificationCode". This supplementary component is derived by "AmountCurrencyIdentificationCodeType" and is represented as an attribute. The XML instance of this "AmountType" is represented as is follows:

```
<Amount amountCurrencyIdentificationCode="EUR" uid="ID000000" uidRef="ID000000" uidRefs="ID000000 ID000001"
language="en">3.14</Amount>
```

The next example shows the "AmountType" without any attribute:

```
 <xs:complexType name="AmountType_0p2">
  <xs:sequence>
   <xs:element name="AmountContent" type="cct:AmountContentType"/>
   <xs:element name="AmountCurrencyIdentificationCode" type="cct:AmountCurrencyIdentificationCodeType"/>
  </xs:sequence>
 </xs:complexType>
```

It will be more complicated for the parser as well as the user, if the content will be represented in an additional childelement inside the complexType "AmountType". Therefore it is necessary to create two childelements inside of "AmountType". The XML instance will be then shown as the following example:

```
<Amount_0p2>
 <AmountContent>33.34</AmountContent>
 <AmountCurrencyIdentificationCode>EUR</AmountCurrencyIdentificationCode>
</Amount_0p2>
```

It will be much more data necessary for building an instance of "AmountType". This will be influenced the parsing of big documents especially. As well as that example is not better readable as the first example. The new version of DOM as well as the SAX parser parsing all attributes in a very fast and elegant way, faster as a lot of additional childelements.

The following example shows the creation of date-time elements in two different ways:

The first example shows the definition of the dateTime format with a built-in simpleType:

```
   <element name="DateTime_0p3" type="dateTime"/>
```

It is although possible to create the Date Time format in a special format, based on ISO 8601:

```
<complexType name="DateTimeType_0p1">
 <simpleContent>
  <extension base="cct:DateTimeContentType">
   <attribute name="dateTimeFormat" type="cct:DateTimeFormatType"/>
  </extension>
 </simpleContent>
</complexType>
```

The attribute “dateTimeFormat” gives the information about the representation of the special format:

```
 <DateTime_0p1 dateTimeFormat="YY-MM-DD">02-02-05</DateTime_0p1>
```

This XML instance represented the content in the same element. Therefore, there is no changing of the representation. That will be not so, if the description of the format will be done by an additional child element:

```
<DateTime_0p2>
 <DateTimeContent>02-02-05</DateTimeContent>
 <DateTimeFormat>YY-MM-DD</DateTimeFormat>
</DateTime_0p2>
```

There are two child elements necessary. One for the content and the other for the format description. That makes much more data and is not so easy understandable as the example before.

There are might be a problem by using more than one supplementary components for one Core Component Type. For example "codeType". Since as the values of each supplementary components do represent some processable data or codes respectively.

There are some examples:

A.)

```
 <Code_0p1 codeListIdentifier="1B" codeListAgencyIdentifier="28" codeListVersionIdentifier="1" codeName="Special Code"
languageCode="en">ABCX</Code_0p1>
```

In the first example (A) are all supplementary components represented as attributes. The problem is that will happen no direct relationship between codeName and languageCode. This must be necessary, because the languageCode is related to the codeName.

B.)

```
<Code_0p2 CodeListAgencyIdentifier="1B" CodeListIdentifier="28" CodeListVersionIdentfier="1">
  <CodeContent>ABCX</CodeContent>
  <CodeName languageCode="en">Special Code</CodeName>
 </Code_0p2>
```

In the second example (B) are the supplementary components shared in attributes and child elements. Supplementary components would like represented as attributes, if the data could be processably or coded information respectively. Supplementary components which represents user readable information represented as child elements. The content component is represented as child element, too. One expection have the attribute "languageCode" due to related to the readable name of the code it will be placed inside of the child element "CodeName".

C.)

```
<Code_0p3>
  <CodeContent>ABCX</CodeContent>
  <CodeListAgencyIdentifier>1B</CodeListAgencyIdentifier>
  <CodeListIdentifier>28</CodeListIdentifier>
  <CodeListVersionIdentifier>1</CodeListVersionIdentifier>
  <CodeName>Special Code</CodeName>
  <LanguageCode>us</LanguageCode>
 </Code_0p3>
```

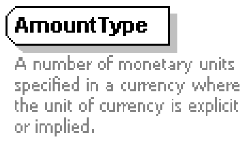
The last example the CodeType without any attributes. There are much more data and there is no relationship between CodeName and LanguageCode, too.

The attributes doesn't make the readability much more complicated. It help us, to build relationship in a very short and easy matter. The XML instances are much shorter and there are not so much hierachies for representing that data. That helps that the parsing of that structure is much more faster. And is helpful to map elements in an internal workflow or database respectively.

6.4 Attributes within the Core Component Types

The following subchapters shows the different core component types with the use of attributes as examples.

6.4.1 complexType AmountType

diagram	 AmountType A number of monetary units specified in a currency where the unit of currency is explicit or implied.
namespace	CoreComponentTypes.xsd
type	extension of **cct:AmountContentType**
facets	totalDigits 10 fractionDigits 2

	Name	Type	Use	Default	Fixed	Annotation
attributes	amountCurrencyIdentificationCode	cct:AmountCurrencyIdentificationCodeType				
	uid	xs:ID	required			
	uidRef	xs:IDREF	optional			
	uidRefs	xs:IDREFS	optional			
	language	language	optional	en		

annotation	documentation	A number of monetary units specified in a currency where the unit of currency is explicit or implied.

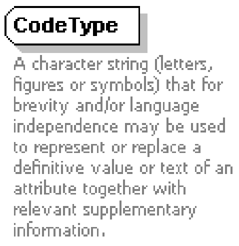
source

```
<complexType name="AmountType" id="000105">
  <annotation>
    <documentation source="CCTS V1.7" xml:lang="en">A number of monetary units specified in a currency where the unit of currency is explicit or implied.</documentation>
  </annotation>
  <simpleContent>
    <extension base="cct:AmountContentType">
      <attribute name="amountCurrencyIdentificationCode" type="cct:AmountCurrencyIdentificationCodeType"/>
      <attributeGroup ref="cct:UidAttributeGroup"/>
    </extension>
  </simpleContent>
</complexType>
```

example

```
<Amount amountCurrencyIdentificationCode="EUR" uid="ID000000" uidRef="ID000000" uidRefs="ID000000 ID000001" language="en">3.14</Amount>
```

6.4.2 complexType CodeType

diagram	 CodeType A character string (letters, figures or symbols) that for brevity and/or language independence may be used to represent or replace a definitive value or text of an attribute together with relevant supplementary information.
namespace	CoreComponentTypes.xsd
type	extension of **cct:CodeContentType**

attributes

Name	Type	Use	Default	Fixed	Annotation
codeListIdentifier	cct:CodeListIdentifierType				
codeListAgencyIdentifier	cct:CodeListAgencyIdentifierType				
codeListVersionIdentifier	cct:CodeListVersionIdentifierType				
codeName	cct:CodeNameType				
languageCode	cct:LanguageCodeType				
uid	xs:ID	required			
uidRef	xs:IDREF	optional			
uidRefs	xs:IDREFS	optional			
language	language	optional	en		

annotation	documentation	A character string (letters, figures or symbols) that for brevity and/or language independence may be used to represent or replace a definitive value or text of an attribute together with relevant supplementary information.

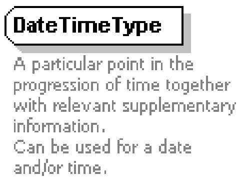
source

```
<complexType name="CodeType" id="000089">
  <annotation>
    <documentation source="CCTS V1.7" xml:lang="en">A character string (letters, figures or symbols) that for brevity and/or language independence may be used to represent or replace a definitive value or text of an attribute together with relevant supplementary information.</documentation>
  </annotation>
  <simpleContent>
    <extension base="cct:CodeContentType">
      <attribute name="codeListIdentifier" type="cct:CodeListIdentifierType"/>
      <attribute name="codeListAgencyIdentifier" type="cct:CodeListAgencyIdentifierType"/>
      <attribute name="codeListVersionIdentifier" type="cct:CodeListVersionIdentifierType"/>
      <attribute name="codeName" type="cct:CodeNameType"/>
      <attribute name="languageCode" type="cct:LanguageCodeType"/>
      <attributeGroup ref="cct:UidAttributeGroup"/>
    </extension>
  </simpleContent>
</complexType>
```

example

```
<Code codeListIdentifier="CODEID01" codeListAgencyIdentifier="CodeAgency" codeListVersionIdentifier="V01"
codeName="CodeName" languageCode="en-us" uid="ID000001" uidRef="ID000001" uidRefs="ID000000 ID000001"
language="en">COD</Code>
```

6.4.3 complexType DateTimeType

diagram	 DateTimeType A particular point in the progression of time together with relevant supplementary information. Can be used for a date and/or time.
namespace	CoreComponentTypes.xsd
type	extension of **cct:DateTimeContentType**

attributes

Name	Type	Use	Default	Fixed	Annotation
dateTimeFormatText	cct:DateTimeFormatTextType				
uid	xs:ID	required			
uidRef	xs:IDREF	optional			
uidRefs	xs:IDREFS	optional			
language	language	optional	en		

annotation	documentation	A particular point in the progression of time together with relevant supplementary information. Can be used for a date and/or time.

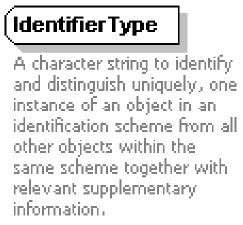
source

```
<complexType name="DateTimeType" id="000066">
  <annotation>
    <documentation source="CCTS V1.7" xml:lang="en">A particular point in the progression of time together with relevant supplementary information.
Can be used for a date and/or time.
</documentation>
  </annotation>
  <simpleContent>
    <extension base="cct:DateTimeContentType">
      <attribute name="dateTimeFormatText" type="cct:DateTimeFormatTextType"/>
      <attributeGroup ref="cct:UidAttributeGroup"/>
    </extension>
  </simpleContent>
</complexType>
```

example

```
<DateTime dateTimeFormatText="YYYY-MM-DD" uid="ID000002" uidRef="ID000002" uidRefs="ID000001 ID000002" language="en">2002-03-05</DateTime>
```

6.4.4 complexType IdentifierType

diagram	 IdentifierType A character string to identify and distinguish uniquely, one instance of an object in an identification scheme from all other objects within the same scheme together with relevant supplementary information.
namespace	CoreComponentTypes.xsd
type	extension of **cct:IdentifierContentType**

attributes

Name	Type	Use	Default	Fixed	Annotation
identificationSchemeName	cct:IdentificationSchemeNameType				
iIdentificationSchemeAgencyName	cct:IdentificationSchemeAgencyNameType				
languageCode	cct:LanguageCo				

	Name	Type	Use	Default	Fixed	Annotation
		deType				
	uid	xs:ID	required			
	uidRef	xs:IDREF	optional			
	uidRefs	xs:IDREFS	optional			
	language	language	optional	en		

annotation	documentation A character string to identify and distinguish uniquely, one instance of an object in an identification scheme from all other objects within the same scheme together with relevant supplementary information.

source

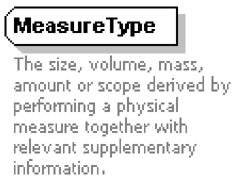
```
<complexType name="IdentifierType" id="000101">
  <annotation>
    <documentation source="CCTS V1.7" xml:lang="en">A character string to identify and distinguish uniquely, one instance of an object in an identification scheme from all other objects within the same scheme together with relevant supplementary information. </documentation>
  </annotation>
  <simpleContent>
    <extension base="cct:IdentifierContentType">
      <attribute name="identificationSchemeName" type="cct:IdentificationSchemeNameType"/>
      <attribute name="iIdentificationSchemeAgencyName" type="cct:IdentificationSchemeAgencyNameType"/>
      <attribute name="languageCode" type="cct:LanguageCodeType"/>
      <attributeGroup ref="cct:UidAttributeGroup"/>
    </extension>
  </simpleContent>
</complexType>
```

example

```
<Identifier identificationSchemeName="IDNAME01" iIdentificationSchemeAgencyName="IdAgency" languageCode="en-us" uid="ID000003" uidRef="ID000003" uidRefs="ID000002 ID000003" language="en">ID01022-XX</Identifier>
```

6.4.5 complexType MeasureType

diagram


MeasureType
The size, volume, mass, amount or scope derived by performing a physical measure together with relevant supplementary information.

namespace	CoreComponentTypes.xsd
type	extension of **cct:MeasureContentType**

attributes

Name	Type	Use	Default	Fixed	Annotation
measureUnitCode	cct:MeasureUnitCodeType				
uid	xs:ID	required			
uidRef	xs:IDREF	optional			
uidRefs	xs:IDREFS	optional			
language	language	optional	en		

annotation	documentation The size, volume, mass, amount or scope derived by performing a physical measure together with relevant supplementary information.

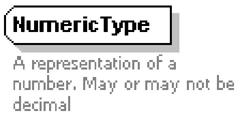
source

```
<complexType name="MeasureType" id="000152">
  <annotation>
    <documentation source="CCTS V1.7" xml:lang="en">The size, volume, mass, amount or scope derived by performing a physical measure together with relevant supplementary information.</documentation>
  </annotation>
  <simpleContent>
    <extension base="cct:MeasureContentType">
      <attribute name="measureUnitCode" type="cct:MeasureUnitCodeType"/>
      <attributeGroup ref="cct:UidAttributeGroup"/>
    </extension>
  </simpleContent>
</complexType>
```

example

```
<Measure measureUnitCode="KGM" uid="ID000004" uidRef="ID000004" uidRefs="ID000003 ID000004" language="en">3.14</Measure>
```

6.4.6 complexType NumericType

diagram	 NumericType A representation of a number. May or may not be decimal
namespace	CoreComponentTypes.xsd
type	extension of **cct:NumericContentType**

attributes

Name	Type	Use	Default	Fixed	Annotation
numericFormatTextType	cct:NumericFormatTextType				
uid	xs:ID	required			
uidRef	xs:IDREF	optional			
uidRefs	xs:IDREFS	optional			
language	language	optional	en		

annotation	documentation	A representation of a number. May or may not be decimal

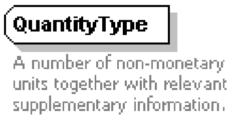
source

```
<complexType name="NumericType" id="000182">
  <annotation>
    <documentation source="CCTS V1.7" xml:lang="en">A representation of a number. May or may not be
decimal</documentation>
  </annotation>
  <simpleContent>
    <extension base="cct:NumericContentType">
      <attribute name="numericFormatTextType" type="cct:NumericFormatTextType"/>
      <attributeGroup ref="cct:UidAttributeGroup"/>
    </extension>
  </simpleContent>
</complexType>
```

example

```
<Numeric numericFormatTextType="nnnnnn" uid="ID000005" uidRef="ID000005" uidRefs="ID000004 ID000005"
language="en">123324</Numeric>
```

6.4.7 complexType QuantityType

diagram	 QuantityType A number of non-monetary units together with relevant supplementary information.
namespace	CoreComponentTypes.xsd
type	extension of **cct:QuantityContentType**

attributes

Name	Type	Use	Default	Fixed	Annotation
quantityUnitCode	cct:QuantityUnitCodeListAgencyIdentifierType				
quantityUnitCodeListIdentifier	cct:QuantityUnitCodeListIdentifierType				
quantityUnitCodeListAgencyIdentifier	cct:QuantityUnitCodeListAgencyIdentifierType				
uid	xs:ID	required			
uidRef	xs:IDREF	optional			
uidRefs	xs:IDREFS	optional			
language	language	optional	en		

annotation	documentation	A number of non-monetary units together with relevant supplementary information.

source

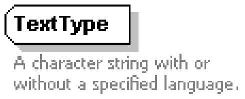
```
<complexType name="QuantityType" id="000108">
  <annotation>
    <documentation source="CCTS V1.7" xml:lang="en">A number of non-monetary units together with relevant
supplementary information.</documentation>
  </annotation>
  <simpleContent>
    <extension base="cct:QuantityContentType">
      <attribute name="quantityUnitCode" type="cct:QuantityUnitCodeListAgencyIdentifierType"/>
      <attribute name="quantityUnitCodeListIdentifier" type="cct:QuantityUnitCodeListIdentifierType"/>
      <attribute name="quantityUnitCodeListAgencyIdentifer" type="cct:QuantityUnitCodeListAgencyIdentifierType"/>
      <attributeGroup ref="cct:UidAttributeGroup"/>
    </extension>
  </simpleContent>
</complexType>
```

example

```
<Quantity quantityUnitCode="token" quantityUnitCodeListIdentifier="token"
quantityUnitCodeListAgencyIdentifer="token" uid="ID000006" uidRef="ID000006" uidRefs="ID000005 ID000006"
language="en">10</Quantity>
```

6.4.8 complexType TextType

diagram



namespace: CoreComponentTypes.xsd

type: extension of **cct:TextContentType**

attributes

Name	Type	Use	Default	Fixed	Annotation
languageCode	cct:LanguageCodeType				
uid	xs:ID	required			
uidRef	xs:IDREF	optional			
uidRefs	xs:IDREFS	optional			
language	language	optional	en		
language	xs:language	optional			

annotation

documentation	A character string with or without a specified language.

source

```
<complexType name="TextType" id="000090">
  <annotation>
    <documentation source="CCTS V1.7" xml:lang="en">A character string with or without a specified
language.</documentation>
  </annotation>
  <simpleContent>
    <extension base="cct:TextContentType">
      <attribute name="languageCode" type="cct:LanguageCodeType"/>
      <attributeGroup ref="cct:UidAttributeGroup"/>
      <attribute name="language" type="xs:language" use="optional"/>
    </extension>
  </simpleContent>
</complexType>
```

example

```
<Text languageCode="en-us" uid="ID000007" uidRef="ID000007" uidRefs="ID000006 ID000007" language="en"
lang="en-us">Hello World</Text>
```

Position Paper: Modularity, Namespaces and Versioning

Author: Bill Burcham (bill_burcham@stercomm.com)

Date: 3/15/02

Filename: draft-burcham-modnamver-03.doc

1 Summary

There are many possible mappings of XML schema constructs to namespaces and to operating system files. This paper explores some of those alternatives and sets forth some rules governing that mapping in UBL.

2 Problem Description

Namespaces are a syntactic convenience supporting the association of a "context" with either a lexical scope (default namespace), or a shorthand identifier (namespace qualifier). This context, applied either implicitly (in a lexical scope) or explicitly (via qualified names) supports compression of what would otherwise be long identifiers. In the absence of namespaces, identifier names are all long.

It is common for an instance document to carry namespace declarations, so that it might be validated. Processing logic (such as a stylesheet) typically carries namespace declarations pertaining to the language in which it is specified in (XSLT) as well as the namespaces on which it *operates*. The latter must match namespaces in the instance document under translation in order for useful work to be carried out.

In practice, namespaces are often given names denoting a hierarchy. XML processing tools may or may not use this hierarchy information. This sort of hierarchical naming though can be useful for the human reader.

As with other significant software artifacts, schemas can become large. In addition to the logical taming of complexity that namespaces provide, we might like to also divide the physical realization of that schema into multiple operating system files.

Schemas change over time. UBL will be no exception. What sort of version information (if any) will a schema carry? How shall that information be carried so as to conveniently support the needs of users operating on document instances with XML processing tools.

This position paper will address these three topics related to namespaces:

1. **Namespace Structure**: What shall be the mapping between namespaces and XML Schema constructs (e.g. type definitions)?
2. **Module Structure**: What shall be the mapping between namespaces and XML Schema constructs and operating system files?
3. **Versioning**: What support for versioning of schema shall be provided?

In subsequent sections, we'll examine each topic in turn, presenting first the options, then a recommendation.

3 Assumptions

Much of this discussion will be based on the expected complexity of the UBL vocabulary. We structure systems into components in order to manage complexity.

3.1 Problem Size

How big will UBL be? How interconnected?

One source for complexity estimation is xCBL. TBD: how many type definitions, element declarations, "instance roots" in xCBL?

Another source for estimation is X12 that according to [NDR-MSG-88] has:

> a bit over 1,000 data elements (…) a smaller number of segments, and
> 300 or so transaction sets

Also from [NDR-MSG-88] we have EDIFACT:

- There are just under 650 data elements which are
- used in approx 200 composite structures (sort of equivalent to low level Aggregate Core Components (ACCs)).
- These elements and composites are reused within just over 150 segment structures (sort of equivalent to higher level ACCs).
- Combinations of all the above make up just under 200 messages (doc types).

So an estimate of 1000 types and 250 message types seems reasonable for UBL.

3.2 Optimal Component Size

We don't want to define 1000 types all in one XML namespace, nor would we want to define them all in one file. Such an approach would lack structure necessary for understanding both by maintainer and users. Additionally, performance would be far from optimal for instance documents that needed only a subset of the UBL types.

For these reasons we presume that we need to structure and divide UBL into a hierarchy of components. We will strive to balance coupling and cohesion between the components in order to:

- Manage the complexity of each component while not creating too many components[1]
- Provide for useful subsetting of components

We envision that many useful instance documents (messages) will be possible that require only a fraction of the overall UBL schema. In those cases it should be possible to avoid processing of the unneeded parts.

4 Options: XML Namespace Identification

This section presents some options for the form that UBL namespace names might take.

---

[1] The "seven plus or minus two" rule [SEVEN-TWO] is a good, general rule of thumb. It's especially useful when you don't have any other rule. It says that if you want people to be able to keep a set of concepts in mind, then you are limited to about seven concepts. Implications for XML for example might be: a type would define no more than seven (or so) elements, a namespace would define no more than about seven types, etc.

4.1 Option 1: Namespace Name = Namespace Location

There is certainly precedent for this approach. See for example the ebXML Message Service schema `http://www.oasis-open.org/committees/ebxml-msg/schema/msg-header-2_0.xsd`.

4.2 Option 2: Namespace Name is OASIS URN namespace

This option exemplifies the current best practice within OASIS. See RFC 3121 [OASIS-URN-NS] for details. See Namespaces in XML for background [NAMESPACE].

Under this option, the namespace names for UBL namespaces would have the following form while the schemas are at draft status:

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

When they move to specification status the form will change to:

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

Where the form of {document-id} is TBD but should match the schema module name (see section 6, Recommendation: Schema Location).

5 Recommendation: Namespace Identification

We pick Option 2: *Namespace Name is OASIS URN namespace.*

This recommendation probably needs more justification.

Will document-id include versioning information or will versioning be handled outside this identifier? See section 12, Recommendations: Versioning.

6 Recommendation: Schema Location

A question related to Namespace identification is schemaLocation. Schema location includes 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`

TBD does this recommendation need more justification?

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

7 Options: Namespace Structure

In this section we'll explore some mappings between XML Schema structures and namespaces.

7.1 Option 1: One Big Namespace

We could have one big namespace for UBL. On the plus side, it would be fairly easy to remember. The downside is that we would forfeit the opportunity to use hierarchical namespaces to communicate the structure of the vocabulary.

7.2 Option 2: One Namespace Per Type

This approach represents the other end of the spectrum. If you've got a namespace per type then why not just use the type name. The namespace fails to be shorthand for anything. It fails to be memorable, or to group related types together.

7.3 Option 3: Core Plus "Functional" Namespaces

This option represents a space between 7.1 and 7.2. There would be one namespace for "core" types and there would be namespaces for each of the TBD functional areas e.g. Order, Invoice.

Purpose	Namespace name
UN Core Component Types	`urn:oasis:names:tc:ubl:cc:CoreComponentTypes`
UBL Core	`urn:oasis:names:tc:ubl:Core`
Order Domain	`urn:oasis:names:tc:ubl:Order`
Invoice Domain	`urn:oasis:names:tc:ubl:Invoice`
TBD	TBD

This represents a top-level decomposition of the vocabulary into multiple vertical (functional) slices and a single (horizontal) slice – the so-called core namespace.

The downside of this approach is that with seven or so functional namespaces, they are going to get awfully "crowded" (on the order of one hundred types per namespace).

7.4 Option 4: Core Plus "Functional" Namespaces Plus Internal Structure as Needed

A refinement on 7.3 this option frees each of the functional and core namespaces to have their own hierarchy as necessary in order to further manage complexity.

8 Recommendation: Namespace Structure

	Pro	Con
Option 1: one big namespace	Easy to remember namespace	When anything in UBL changes, all processing code must be changed (at a minimum to use new namespace name)
Option 2: namespace per type	Total compartmentalization	Why use namespaces at all? With this option the namespace

		ceases to provide useful contextualization.
Option 3: core plus "functional" namespaces	Allows parts of UBL to change independently. When a functional area changes, processing code depending on core needn't change.	Doesn't allow for intermediate structure. What if the functional namespaces may require further subdivision?
Option 4: core plus "functional" namespaces plus internal structure as needed	(same as Option 3)	By allowing intermediate namespaces, they will certainly flourish. Design rules must be developed to avoid regressing toward Option 2 over time.

Option 3 is recommended. We reserve the right to revisit this decision when we are further along in the process of defining types. If we find that we need more structure, we can move to option 4.

8.1 Into What Namespace Do Extensions Go

Extensions (by users) go into user-defined namespaces outside of UBL.

9 Options: Module Structure

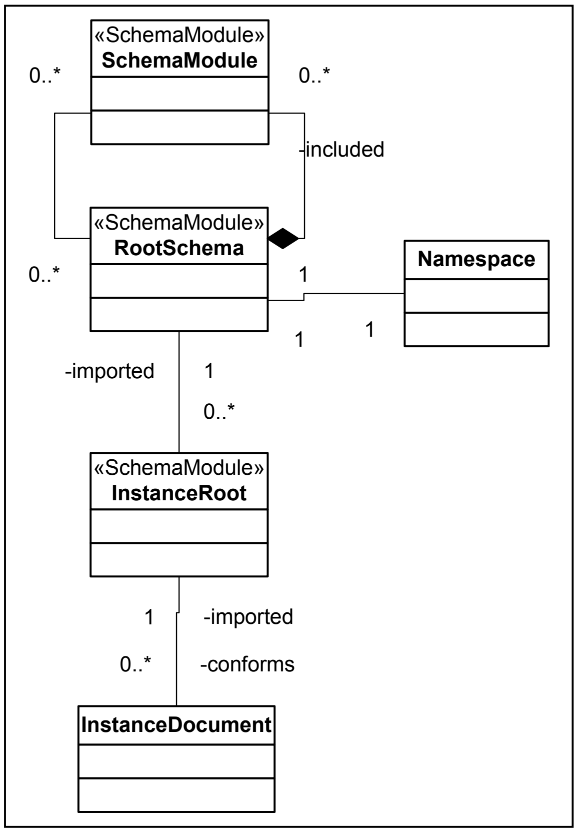
TBD: what are some other options?

10 Recommendation: Module Structure

The UBL vocabulary consists of a set of instance roots and root schemas. The instance roots comprise a ready-to-use set of business document types. The instance roots import type definitions from root schemas.

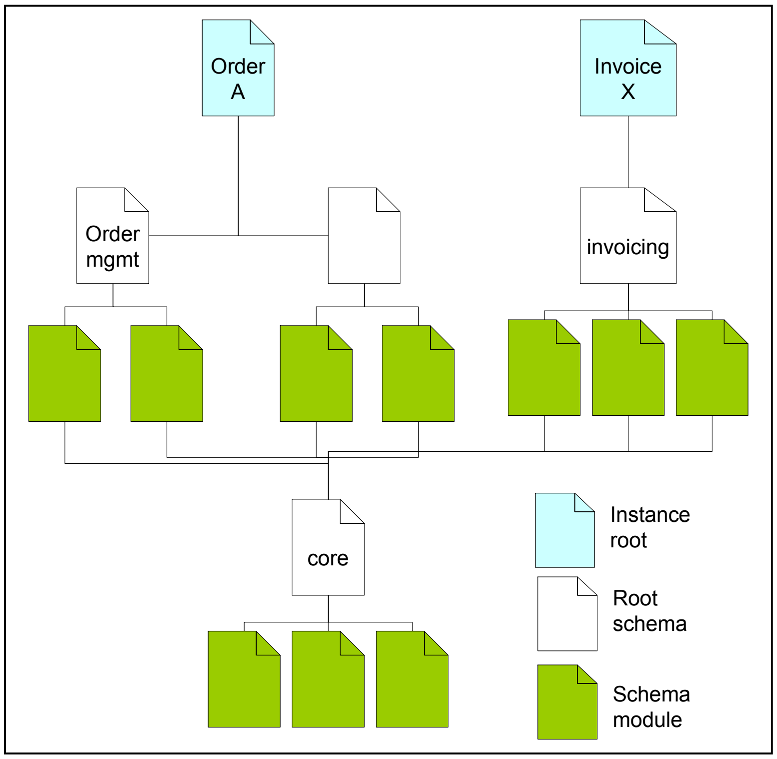
Each root schema defines a BIE. If a root schema is large, it may be broken up into multiple schema modules. The schema modules are imported in a root schema.

Here is a depiction of the component structure:



10.1 Recursive Composition

A schema module, or by extension, a root schema, may depend upon other root schemas for its definition.



10.2 Instance Root Types

If preferring type definitions over global element definitions is good, why not take it to the extreme [NDR-MSG-70]. **The type of the root element in an instance root is a global type** (not an anonymous type).

10.3 Number of Instance Roots

In some cases, various actions in the protocol (create vs. delete) will have totally different document structure requirements. But in some cases (create vs. update), the content might be identical. However, we still think we should design in favor of more document types rather than less, e.g. one for each transmission (a la RosettaNet). It avoids confusion on the part of developers to have a separate document type for each thing. We might then decide to optimize some of them by merging them together.

11 Options: Versioning

[XFRNT-VER] does a great job of laying out the problem and solution space for schema versioning as it is traditionally practiced. The options presented in that document are not really disjoint rather they are building blocks. If you look at the recommendations in that document, you will see that the options are used in concert.

11.1 Option XF-1: Change the (internal) schema "version" attribute

11.2 Option XF-2: Create a "schemaVersion" attribute on the root element

11.2.1 Usage A: Conformance enforced by validator

11.2.2 Usage B: Conformance enforced by an extra processing pass

11.3 Option XF-3: Change the schema's target namespace

11.4 Option XF-4: Change the name/location of the schema

11.5 Option 5: Schema Version as Context Classifier

In [NDR-MSG-13] the point was made that schema version might just be another context classifier.

12 Recommendations: Versioning

Each namespace should have a version. Other things shouldn't (e.g. schema modules shouldn't).

Each of core and functional areas will have a version. How shall we communicate compatibility/incompatibility?

One approach is to follow a convention whereby a schema's version identifier consists of two parts: a major number and a minor number. The major number changes when a backward-incompatible change is made:

- changing a default value (legal issue)
- adding a new required element
- removing a required or optional element

The minor number changes when a backward-compatible change is made:

- adding an optional element

How do we communicate version compatibility between core and functional areas:

- between core and f/a's
- between f/a's

TBD: Include input from SAML versioning paper (to be released 1-10-02)

The following table summarizes the tradeoffs between the options.

	Pro	Con
Option XF-1: Change the (internal) schema "version" attribute		Not enforced by validator
Option XF-2-A: Create a "schemaVersion" attribute on the root element -- Conformance enforced by validator		Conformance requires exact match on version string
Option XF-2-B: Create a "schemaVersion" attribute on the root element -- Conformance enforced by an extra processing pass		Extra processing step.
Option XF-3: Change the schema's target namespace		With this approach, instance documents will not validate until they are changed to designate the new targetNamepsace. However, one does not want to force all instance documents to change, even if the change to the schema is really minor and would not impact an instance. +Include problems.
Option XF-4: Change the name/location of the schema		Ugh!
Option 5: Schema Version as Context Classifier	Leverages the context machinery	Requires the context machinery

13 Definitions

Backward compatibility – TBD.

BIE – Business Information Entity. A description of a business concept. Represented as an XML schema by a *root schema.*

extension a.k.a. customization – specification of new BIE's with well-defined, enforced relationships to old BIE's. Relationship types include: restriction, extension. In some cases processing logic will need to treat the base and the extension as the same, in other cases it will need to distinguish between them.

Forward compatibility – TBD

instance root, a.k.a. 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.

Namespace – a name that scopes a related group of XML type definitions.

processing logic – software logic that operates on BIE instances to achieve some business function

root schema – A *schema module* that directly, or via inclusion of other schema modules, defines all types for a particular namespace. This is the XML Schema representation of a BIE. (Compare that definition, with the one we came up with last week in Menlo Park: *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 document – as defined by the XSD specification – per that specification, a schema document defines types into exactly one namespace, the target namespace.

schema module – A *schema document*. A schema module need not define all types in a particular namespace. Contrast with *root schema*. (Compare that definition, with last week's: *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.*)

versioning – reification of revisions to BIE's in order to support coexistence in a system, of two or more revisions of a BIE.

14 References

NAMESPACE	*Namespaces in XML*	http://www.w3.org/TR/REC-xml-names/
NDR-MSG-13	*schema version as context classifier*, Burcham, Bill; Maler, Eve; a post to the UBL-NDR mailing list.	http://lists.oasis-open.org/archives/ubl-ndrsc/200111/msg00013.html
NDR-MSG-70	*Fwd: Straw Man on Namespaces, Schema Module Architecture,etc.*, Rawlins, Mike; a post to the UBL-NDR mailing list.	http://lists.oasis-open.org/archives/ubl-ndrsc/200111/msg00070.html
NDR-MSG-88	*Fwd: Straw Man on Namespaces,Schema Module Architecture, etc.*, Probert, Sue; Maler, Eve.; a post to the UBL-NDR mailing list.	http://lists.oasis-open.org/archives/ubl-ndrsc/200111/msg00088.html
OASIS-URN-NS	IETF RFC 3121 *A URN Namespace for OASIS*	http://www.faqs.org/rfcs/rfc3121.html
SCHEMA-PRIM	*XML Schema Part 0: Primer*	http://www.w3.org/TR/xmlschema-0/

SEVEN-TWO	*The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information*, George A. Miller, Psychological Review, 63, 81-97.	http://psychclassics.yorku.ca/Miller/
XFRNT-VER	*XML Schema Versioning,* MITRE Corporation and xml-dev list group members.	http://www.xfront.com/Versioning.pdf
XML-URI-LIST	*XML-URI List* at w3.org	http://lists.w3.org/Archives/Public/xml-uri/