Universal Business Language (UBL) Code List Rules

Working Draft 01, 25 August 2002

Document identifier:

wd-ublndrsc-codelist-01

Location:

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

Editor:

Eve Maler, Sun Microsystems <eve.maler@sun.com>

Contributor:

Fabrice Desré, France Telecom

Abstract:

This specification provides rules for developing and using reusable code lists. This specification was originally developed for the UBL Library and derivations thereof, but it may also be used by other XML vocabularies as a mechanism for sharing code lists in W3C XML Schema form.

Status:

This is a draft document. It may change at any time.

This document was developed by the OASIS UBL Naming and Design Rules subcommittee **[NDRSC]**. Your comments are invited. Members of this subcommittee should send comments on this specification to the ubl-ndrsc@lists.oasis-open.org list. Others should subscribe to and send comments to the ubl-comment@lists.oasis-open.org list. 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 was developed by the OASIS UBL Naming and Design Rules subcommittee **[NDRSC]** to provide rules for developing and using reusable code lists in W3C XML Schema **[XSD]** form. It is organized as follows:

- Section 2 offers guidance to the OASIS UBL Technical Committee in incorporating code lists into the UBL Library.
- Section 3 provides rules on how to define and use reuable code list schema modules.
- Section 4 is non-normative. It provides the analysis that led to the recommendation of the XSD datatype mechanism for creating reusable code lists.

1.1 Scope and Audience

The rules in this specification are designed to encourage the creation and maintenance of code list modules by their proper owners as much as possible. It was originally developed for the UBL Library and derivations thereof, but it is largely not specific to UBL needs; it may also be used with other XML vocabularies as a mechanism for sharing code lists in XSD form. If enough code-list-maintaining agencies adhere to these rules, we anticipate that a more open marketplace in XML-encoded code lists will emerge for all XML vocabularies.

This specification assumes that the reader is familiar with the UBL Library and with the ebXML Core Components concepts and ISO 11179 concepts that underlie it.

1.2 Terminology and Notation

The text in this specification is normative for UBL Library use unless otherwise indicated. The key words *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* in this specification are to be interpreted as described in **[RFC2119]**.

Terms defined in the text are in **bold**. Refer to the UBL Naming and Design Rules **[NDR]** for additional definitions of terms.

Core Component names from ebXML are in *italic*.

```
Example code listings appear like this.
```

Note: Non-normative notes and explanations appear like this.

Conventional XML namespace prefixes are used throughout this specification to stand for their respective namespaces as follows, whether or not a namespace declaration is present in the example:

- The prefix `xs:` stands for the W3C XML Schema namespace **[XSD]**.
- The prefix `xhtml:` stands for the XHTML namespace.
- The prefix `iso3166:` stands for a namespace assigned by a fictitious code list module for the ISO 3166-1 country code list.

2 Guidance to the UBL Modeling Process

Where possible, the UBL Library should identify and use external standardized code lists rather than develop its own UBL-native code lists. Designing an internal code list is justified in cases where, for example, an existing external code list needs to be extended, or where no suitable external code list exists. The lack of “easy-to-read” or “easy-to-understand” codes in an otherwise suitable code list is not sufficient reason to define a UBL-native code list.

Where the UBL Library does create its own native code lists, the lists should be **globally scoped** (designed for reuse and sharing, using named types and namespaced schema modules) rather than **locally scoped** (not designed for others to use and therefore hidden from their use). Globally scoped code lists are much preferable because the additional work is negligible and the benefits of reuse are great.

For each UBL construct containing a code, the UBL documentation must identify the zero or more code lists that must be minimally supported when the construct is used. The rules in this specification for how to represent code lists in UBL schema modules have the effect of encapsulating this minimal-support information in schema form as well. It is assumed that whole code lists, and not subsets of those code lists, will be identified; however, users of the UBL Library may identify any subset they wish from an identified code list for their own trading community conformance requirements.

3 Defining and Using Code Lists

This section provides rules for developing and using reusable code lists in XSD form. These rules were developed for the UBL Library and derivations thereof, but they may also be used by other code-list-maintaining agencies as guidelines for any XML vocabulary wishing to share code lists.

> **Note:** The OASIS UBL Naming and Design Rules subcommittee is willing to help any organization that wishes to apply these rules but does not have the requisite XSD expertise.

3.1 Overview

This section introduces important terminology and concepts.

UBL uses codes in two ways:

- As **first-order** business information entities (BIEs) in their own right. For example, one property of an address might be a code indicating the country. This information appears in an element, according to the Naming and Design Rules specification **[NDR]**.
- As **second-order** information that qualifies some other BIE. For example, any information of the Amount core component type must have a supplementary component (metadata) indicating the currency code. This information appears in an attribute.

Every first-order code appearing in the UBL Library must be double-wrapped. The **inner code element** is dedicated to holding codes only from a single list. For example, the `ISO3166CountryCode` element below is designed to hold codes only from the ISO 3166-1 list of two-letter country codes; here it happens to contain the code for Belgium. The inner code element is wrapped in an **outer code element**, in this case a `CountryIdentificationCode` element representing a BIE for the country portion of an address.

```
<Address>
  ...

  <!-- outer code element -->
  <CountryIdentificationCode>

    <!-- inner code element -->
    <ISO3166CountryCode>BE</ISO3166CountryCode>
  </CountryIdentificationCode>

</Address>
```

The inner element is associated with two XSD datatypes that uniquely define the ISO 3166-1 code list in a way that allows for efficient reuse:

- A simple type (**code content type**) represents the string of characters supplying the code inside the element’s start- and end-tags. It provides constraints that ensure, to one degree or another, that the code supplied is a legitimate member of the list.
- A complex type (**code list type**) represents the code list as a whole. It provides attributes that hold metadata about the code list.

The code content type is connected to the code type using the XSD “simple content” mechanism, which allows the element to have both string content and attributes:

```
<xs:simpleType name=”...code content type name...”>
  ...
</xs:simpleType>

```

```
<xs:complexType name=”...code type name...”>
  ...
  <xs:simpleContent>
    <xs:extension base="...code content type name reference...">
      <xs:attribute name="...">
        ...
      </xs:attribute>
      ...
  </xs:simpleContent>
</xs:complexType>
```

These two types should be defined in an XSD schema module dedicated to this purpose (a **code list module**) and must have documentation embedded in them that identifies their adherence to the rules in this specification. The code list module must have a proper target namespace for reference by XML vocabularies that wish to use it.

> **Note:** The XSD form prescribed by this specification is not intended to preclude additional definitions of the same code list in other forms, such as other schema languages or different XSD representations. The UBL Library requires an XSD form because the library is itself in XSD.

Code-list-maintaining agencies are encouraged to create their own code list modules; these modules are considered **external** as far as UBL is concerned.The UBL Library, where it has occasion to define its own code lists, must create its own **native** code list modules. In some cases, an external agency that owns a code list in which UBL has an interest might choose (for the moment or forever) not to create a code list module for it. In these cases, UBL must define a code list module on behalf of the agency. It is expected that these **orphan** code list modules will not have the same validating power, nor be maintained with as much alacrity, as other code list modules with proper owners.

To use a code list module, the UBL Library must associate the relevant type with a native element. For example:

```
<xs:element
  name=”ISO3166CountryCode”
  type=”...code type name reference...”>
  ...
</xs:element>
```

3.2 XML Representations for ebXML-Based Codes

Since the UBL Library is based on the ebXML Core Components (currently at V1.8; see **[CCTS1.8]**), the supplementary components identified for the *Code. Type* core component type are used to identify a code as being from a particular list. According to the UBL Naming and Design Rules **[NDR]**, the content component is represented as an XML element and the supplementary components are represented as XML attributes. [**ISSUE:** Note that the current V1.85 work on CCTS may require changes to this specification.]

Following are the components associated with *Code.Type* and the required representation in the code list module and XML instance.

197

Component Name	**Component Definition**	**XML Form**	**Name**
Code. Content	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	Simple content of an element.	Not dictated by this specification. Where the element is in the UBL Library, the Naming and Design Rules specification **[NDR]** provides rules.
Code List. Identifier	The name of a list of codes	Attribute. Required to be supplied as either a "live" value or a default value.	`ID`
Code List. Agency. Identifier	An agency that maintains one or more code lists	Attribute. Required to be supplied as either a "live" value or a default value. [**ISSUE:** Usually the agency ID is itself a code. Does third-order metadata need to be provided indicating the code list?]	`agencyID`
Code List. Version. Identifier	The version of the code list	Attribute. Required to be supplied as either a "live" value or a default value.	`versionID`
Code. Name	The textual equivalent of the code content	Attribute. Optional to define and supply.	`codeName`
Language. Code	The identifier of the language used in the corresponding text string (in ISO 639 form)	Attribute. Optional to supply if the attribute containing the Code. Name component above is not defined or supplied. Its value is interpreted as being in ISO 639 form. [**Issue:** Need to document the appropriate code list ID, agency ID, and code list version ID values for the choice of ISO 639 here.]	`languageCode`

3.3 Template and Rules for Code List Modules

Following is a template to follow in creating a code list module. This hypothetical ISO 3166-1 code list for country codes is used merely as an example. A text version of this template is available **[CLTemplate]**.

> **Note:** The UN/ECE organization has made available an XSD representation of the ISO 3166-1 code list **[3166-XSD]**. While that XSD representation serves a purpose that is somewhat different from that targeted in this specification, it is useful to use as a reference while studying this template.

[ISSUE: The embedded documentation shown in this template is not yet approved. The theory is that the supplementary components describing the code list should be on the code content type, as well as the code type, so that the code content type can be safely used for second-order code attributes as well.]

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns=”http://www.w3.org/2001/XMLSchema”
  xmlns:xs=”http://www.w3.org/2001/XMLSchema”
  xmlns:xhtml=”...http://www.w3.org/1999/xhtml...”>
  targetNamespace="...namespace for ISO 3166 code list module..."
  xmlns:iso3166="...namespace for ISO 3166 code list module...">
  <xs:annotation>
    <xs:documentation>
This code list module template corresponds to draft 01 of the
OASIS UBL NDR code list rules document (wd-ublndrsc-codelist-01).
See that document for information on how to use this template:
http://www.oasis-open.org/committees/ubl/ndrsc/archive/.
    </xs:documentation>
  </xs:annotation>
  <xs:simpleType name=”iso3166:CodeContentType”>
    <xs:annotation>
      <xs:documentation>
        <xhtml:div class=”Core_Component_Type”>
          <xhtml:p>Code. Type</xhtml:p>
        </xhtml:div>
      </xs:documentation>
      <xs:documentation>
        <xhtml:div class=”Code_List._Identifier”>
          <xhtml:p>ISO 3166</xhtml:p>
        </xhtml:div>
      </xs:documentation>
      <xs:documentation>
        <xhtml:div class=”Code_List._Agency._Identifier”>
          <xhtml:p>6</xhtml:p>
        </xhtml:div>
      </xs:documentation>
      <xs:documentation>
        <xhtml:div class=”Code_List._Version._Identifier”>
          <xhtml:p>0.2</xhtml:p>
        </xhtml:div>
      </xs:documentation>
    </xs:annotation>
    <xs:extension base=”xs:token”>
      <xs:enumeration value=”AF”/>
      <xs:enumeration value=”AL”/>
      <xs:enumeration value=”DZ”/>
      . . .
    </xs:extension>
  </xs:simpleType>

  <xs:complexType name=”iso3166:CodeType”>
```

```
    <xs:annotation>
      <xs:documentation>
        <xhtml:div class=”Core_Component_Type”>
          <xhtml:p>Code. Type</xhtml:p>
        </xhtml:div>
      </xs:documentation>
      <xs:documentation>
        <xhtml:div class=”Code_List._Identifier”>
          <xhtml:p>ISO 3166</xhtml:p>
        </xhtml:div>
      </xs:documentation>
      <xs:documentation>
        <xhtml:div class=”Code_List._Agency._Identifier”>
          <xhtml:p>6</xhtml:p>
        </xhtml:div>
      </xs:documentation>
      <xs:documentation>
        <xhtml:div class=”Code_List._Version._Identifier”>
          <xhtml:p>0.2</xhtml:p>
        </xhtml:div>
      </xs:documentation>
    </xs:annotation>
    <simpleContent>
      <xs:extension base="iso3166:CodeContentType">
        <xs:attribute name="ID"
          type="xs:token" fixed=”ISO 3166”/>
        <xs:attribute name="agencyID"
          type="xs:token" fixed=”6”/>
        <xs:attribute name="versionID"
          type="string" fixed=”0.2”/>
    </simpleContent>
  </xs:complexType>
</xs:schema>
```

Following are the rules for defining a code list module:

1. All newly defined types must be named; they must not be anonymous.

   Note: Only locally scoped code lists should use anonymous types, to prevent the types from being associated with multiple elements or with elements in other namespaces.

2. A properly named target namespace must be assigned to the code list schema module. It is recommended that the types be defined in their own dedicated schema module, so that the namespace unambiguously refers to a single code list.

3. In the code list type, attributes must be defined at least for the code list identifier (`ID`), code list agency identifier (`agencyID`), and code list version identifier (`versionID`). Defining attributes for the code name (`codeName`) and its language code (`languageCode`) is optional. The attributes may be associated with any appropriate simple types. The attribute values need not be fixed; a default could be provided, or the value could simply be required to appear in the instance.

4. The XSD definitions should be made as reasonably constraining as possible, defining value defaults or fixed values for supplementary components and circumscribing the valid values of the code content without compromising the maintainability goals of the agency. It might make sense not to use enumeration but rather to use pattern-matching regular expressions or to avoid strict code validation entirely.

5. Embedded documentation must be provided as shown in the template above in order to indicate the appropriate code list metadata. If the code list module serves for multiple versions of the same code list, the documentation block for *Code List. Version. Identifier* is

optional. See the Naming and Design Rules specification **[NDR]** for more information on embedded documentation rules.

6. A global element in the agency's namespace may optionally be defined and associated with the code list type. Note that the UBL Library currently does not plan to use such elements, but it might be helpful for use in other XML vocabularies that import global elements from other namespaces.

    > **Note:** Various features of XSD could be used for purposes not related to this specification, such as attribute groups (to manage the attributes for supplementary components) and the use of non-built-in XSD simple types for the attribute values (for tighter management of constraints on these values).

3.4 Associating UBL Elements with Code List Types

No matter whether type pairs for code lists are defined by UBL or by an external agency, the UBL Library must define its own elements for the provision of the actual codes in an instance. (This is according to the rule regarding local unqualified elements in the Naming and Design Rules **[NDR]** specification.) This definition is done in the following manner.

First, the relevant code list module must be imported into the relevant UBL Library module.

```
<xs:import
  namespace="...namespace for ISO 3166 code list module..."
  schemaLocation="...location of code list module..." />
```

Then, an outer code element representing the code BIE must be set up to hold one or more inner code elements. Here, a `CountryIdentificationCode` element is assumed to require a code from the hypothetical ISO 3166 locale code list defined in Section 3.3. Thus, it needs to contain an `ISO3166LocaleCode` element associated with the `iso3166:LocaleCodeType` type.

[ISSUE: We need some rules around the naming and construction of types such as `CountryIdentificationCodeType`, with the types being generated based on the contents of the "Code Lists/Standards" column of the spreadsheet. These rules should probably go in the NDR document, not here.]

```
<xs:complexType name="Address">
  ...
  <xs:sequence>
    ...other content...
    <xs:element
      name="CountryIdentificationCode"
      type="ubl:CountryIdentificationCodeType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name=”CountryIdentificationCodeType”>
    ...
    <xs:element name=”ISO3166Code” type=”iso3166:CodeType”/>
</xs:complexType>
```

In this case, only one code list is allowed to be used for country codes. However, it is possible for the outer element to allow a choice of one or more inner elements, each containing a code from a different list. For example, if a country code from Codes "R" Us were also allowed, the type definition for `CountryIdentificationCodeType` would change as follows (assuming the Codes "R" Us module were properly imported):

```
<xs:complexType name="Address">
  ...
  <xs:sequence>
    ...other content...
    <xs:element
      name="CountryIdentificationCode"
      type="ubl:CountryIdentificationCodeType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name=”CountryIdentificationCodeType”>
  ...
  <xs:choice>
    <xs:element name=”ISO3166Code” type=”iso3166:CodeType”/>
    <xs:element name=”CodesRUsCode” type=”codesrus:CodeType”/>
  </xs:choice>
</xs:complexType>
```

In this way, minimal support for a selection of code lists can be indicated not just through normative prose but through formal schema constraints as well.

3.5 Deriving New Code Lists from Old Ones

[**ISSUE:** This section is to be supplied. It needs to show the proper way to build new code lists, e.g. by unioning multiple existing code lists and by subsetting existing code lists manually.]

4 Rationale for the Selection of the Code List Mechanism (Non-Normative)

This non-normative section describes the analysis that was undertaken by the OASIS UBL Naming and Design Rules subcommittee to recommend a particular XSD-based solution for the encoding of code lists.

Note that some of the examples in this section may be incorrect or obsolete, without compromising the results of the analysis. If you notice problems, please report them and we will attempt to fix them. Otherwise, please consider this section historical.

4.1 Requirements for a Schema Solution for Code Lists

Following are our major requirements on potential code list schemes for use in the UBL library and customizations of that library. For convenience, a weighted point system is used for scoring the solutions against the requirements.

- Semantic clarity

  The ability to “dereference” the ultimate normative definition of the code being used. The supplementary components for “Code.Type” CCTs are the expected way of providing this clarity, but there are many ways to supply values for these components in XML, and it’s even possible to supply values in some non-XML form that can then be referenced by the XML form.

  Points: Low = 0, Medium = 2, High = 4

- Interoperability

  The sharing of a common understanding of the limited set of codes that are expected to be used. There is a continuum of possibilities here. For example, a schema datatype that allows only a hard-coded enumerated list of code values provides “hard” (but inflexible) interoperability. On the other hand, merely documenting the intended shared values is more flexible but somewhat less interoperable, since there are fewer penalties for private arrangements that go outside the standard boundaries. This requirement is related to, but distinct from, validatability and context rules friendliness.

  Points: Low = 0, Medium = 2, High = 4

- External maintenance

  The ability for non-UBL organizations to create XSD schema modules that define code lists in a way that allows UBL to reuse them without modification on anyone’s part. Some standards bodies are already starting to do this, though we recognize that others may never choose to create such modules.

  Points: Low = 0, Medium = 2, High = 4

- Validatability

  The ability to use XSD to validate that a code appearing in an instance is legitimately a member of the chosen code list. For the purposes of the analysis presented here, “validatability” will not measure the ability for non-XSD applications (for example, based on perl or Schematron) to do validation.

  Points: Low = 0, Medium = 2, High = 4

- Context rules friendliness

The ability to use expected normal mechanisms of the context methodology for allowing codes from additional lists to appear (extension) and for subsetting the legitimate values of existing lists (subsetting), without adding custom features just for code lists. This has lower point values because we expect it to be easy to design custom features for code lists. For example, the following is a mock-up of one approach that could be used:

```
<CodeList fromType="LocaleCodeType" toCode="MyCodeType">
<Add>JP</Add>
<Remove>DE</Remove>
</CodeList>
```

Points: Low = 0, Medium = 1, High = 2

- Upgradability

  The ability to begin using a new version of a code list without the need for upgrading, modifying, or customizing the schema modules being used. This has lower point values because requirements related to interoperability take precedence over a “convenience requirement”.

  Points: Low = 0, Medium = 1, High = 2

- Readability

  A representation in the XML instance that provides code information in a clear, easily readable form. This is a subjective measurement, and it has lower point values because although we want to recognize readability when we find it, we don’t want it to become more important than requirements related to interoperability.

  Points: Low = 0, Medium = 1, High = 2

4.2 Contenders

The methods for handling code lists in schemas are as follows:

- The **enumerated list method**, using the classic method of statically enumerating the valid codes corresponding to a code list in an XSD string-based type internally in UBL
- The **QName in content method**, involving the use of XML Namespaces-based “qualified names” in the *content* of elements, where the namespace URI is associated with the supplementary components
- The **instance extension method**, where a code is provided along with a cross-reference to somewhere in the same instance to the necessary supplementary information
- The **single type method**, involving a single XSD type that sets up attributes for supplying the supplementary components directly on all elements containing codes
- The **multiple UBL types method**, where each element dedicated to containing a code from a particular code list is bound to a unique UBL type, which external organizations must derive from
- The **multiple namespaced types method**, where each element dedicated to containing a code from a particular code list is bound to a unique type that is qualified with a (potentially external) namespace

Throughout, an element `LocaleCode` defined as part of the complex type `LanguageType` is used as an example element in a sample instance, and UBL library schema definitions are demonstrated along with potential opportunities for XSD-style derivation. Each method is assessed to see which requirements it satisfies.

4.2.1 Enumerated List Method

The enumerated list method is the “classic” approach to defining code lists in XML and, before it, SGML. It involves creating a type in UBL that literally lists the allowed codes for each code list.

4.2.1.1 Instance

The enumerated list method results in instance documents with the following structure.

```
<LocaleCode>code</LocaleCode>
```

4.2.1.2 Schema Definitions

The schema definitions to support this might look as follows.

```
<xs:simpleType name="LocaleCodeType">
  <xs:restriction base="xs:token">
    <xs:enumeration value="DE"/>
    <xs:enumeration value="FR"/>
    <xs:enumeration value="US"/>
    . . .
  </xs:restriction>
</xs:simpleType>

<xs:element name="LocaleCode" type="LocaleCodeType"/>
```

4.2.1.3 Derivation Opportunities

Using the XSD feature for creating unions of simple types, it is possible to extend the valid values of such an enumeration. However, it seems that we can't *restrict* the list of valid values. This is because `<xs:enumeration>` is not a type construction mechanism, but a facet.

The base schema shown above could be extended to support new codes as follows:

```
<xs:simpleType name="OtherCodeType">
  <xs:restriction base="xs:token">
    <xs:enumeration value="SP"/>
    <xs:enumeration value="DK"/>
    <xs:enumeration value="JP"/>
    . . .
  </xs:restriction>
</xs:simpleType>

<xs:element name="MyLocalCode">
  <xs:simpleType>
    <xs:union memberTypes="LocaleCodeType OtherCodeType"/>
  </xs:simpleType>
</xs:element>
```

4.2.1.4 Assessment

Spelling out the valid values assures validatability, but defining all the necessary code lists in UBL itself defeats our hope that code lists can be defined and maintained in a decentralized fashion.

Requirement	**Score**	**Rank**
Semantic clarity	*0*	Low The supplementary components of the code list could be provided as schema annotations, but they are not directly accessible as first-class information in the instance or schema.

Requirement	**Score**	**Rank**
Interoperability	4	High The allowed values are defined by a closed list defined in the schema itself.
External maintenance	0	Low We have to modify the type union in the base schema to "import" the new codes.
Validatability	4	High The allowed values are defined by a closed list defined in the schema itself.
Context rules friendliness	0	Low The allowed values are defined in the middle of a simple type, whereas the context methodology so far only knows about elements and attributes.
Upgradability	0	Low A schema extension would be needed to add any new codes defined in a new version.
Readability	2	High The instance is as compact as it can be, with no extraneous information hindering the visibility of the code itself.
Total	**11**	

4.2.2 QName in Content Method

The QName method was proposed in V04 of the code lists paper.

4.2.2.1 Instance

With the QName method, 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 used in the `LocaleCode` element, where “iso3166” is the namespace prefix and “US” is the local part. The “iso3166” prefix is bound to a URI by means of an `xmlns:iso3166` attribute (which could have been on any ancestor element).

```
<LocaleCode
  xmlns:iso3166=”http://www.oasis-
open.org/committees/ubl/ns/iso3166”>
iso3166:US
</LocaleCode>
```

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.

The namespace URI shown here is just an example. However, it is likely that the UBL library itself would have to define a set of common namespace URIs in all cases where the owners of external code lists have not provided a URI that could sensibly be used as a code list namespace name.

4.2.2.2 Schema Definitions

QNames are defined by the built-in XSD simple type called `QName`. The schema definition in UBL should make reference to a UBL type based on `QName` wherever a code is allowed to appear, so that this particular use of QNames in UBL can be isolated and documented. For example:

```
<xs:simpleType name=”CodeType”>
  <xs:restriction base=”QName”/>
</xs:simpleType>

<xs:complexType name="LanguageType" id="UBL000013">
  <xs:sequence>
    <xs:element name="IdentificationCode" . . .></xs:element>
    <xs:element name="Name" . . .></xs:element>
    <xs:element name="LocaleCode"
      type="cct:CodeType" id="UBL000016" minOccurs="0">
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

The documentation for the `LocaleCode` element should indicate the minimum set of code lists that are expected to be used in this attribute. However, the attribute can contain codes from any other 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.

4.2.2.3 Derivation Opportunities

The QName type does have several facets: length, minLength, maxLength, pattern, enumeration, and whiteSpace. However, since namespace prefixes are ideally changeable, depending only on the presence of a correct xmlns namespace declaration, the facets (which are merely lexical in nature) are not a sure bet for controlling values.

4.2.2.4 Assessment

The idea of using XML namespaces to identify code lists is potentially useful, but because this method uses namespaces in a hard-to-process (and somewhat non-standard) manner, both semantic clarity and validatability suffer.

Requirement	**Score**	**Rank**
Semantic clarity	1.5	Low to medium You have to go through a level of indirection, and a complicated one at that (because QNames in content are pseudo-illegitimate and are not supported properly in many XML tools), in order to refer back to the namespace URI. Further, the namespace URI might not resolve to any useful information. However, in cases where the URI is meaningful or sufficient documentation of the code list exists (something we could dictate by fiat), clarity can be achieved.
Interoperability	0	Low The shared understanding of minimally supported code

Requirement	Score	Rank
		lists would have to be conveyed only in prose.
External maintenance	0	Low There is no good way to define a schema module that controls QNames in content.
Validatability	0	Low All validation is pushed off to the application.
Context rules friendliness	0	Low This method is similar to the single type method in this respect. If extensions and subsets are to be managed by means of a context rules document at all, there would need to be a code list-specific mechanism added to reflect this method. If extensions and subsets don't need to be managed by means of context rules because everything happens in the downstream application, there is no need to do anything at all.
Upgradability	2	High You need to have a different URI for each version of a code list, but if you do this, using a new version is easy: You just use a prefix that is bound to the URI for the version you want. However, there is no magic in namespace URIs that allows version information to be recognized as such; the whole URI is just an undifferentiated string.
Readability	1	Medium The representation is very compact because the supplementary component details are deferred to another place (and format) entirely, but the QName format and the need for the `xmlns:` attribute make the information a little obscure.
Total	**4.5**	

4.2.3 Instance Extension Method

In the instance extension method, a code is provided along with a cross-reference to the ID of an element in the same instance that provides the necessary code list supplementary information. One XML instance might contain many code list declarations.

4.2.3.1 Instance

The instance extension method results in instance documents with something like the following structure. The `CodeListDecl` element sets up the supplementary information for a code list, and then an element provides a code (here, `LocaleCode`) also refers to the ID of the relevant declaration.

```
<CodeListDecl ID=”ID-LocaleCode”
  CodeListIdentifier=”ISO3166”
  CodeListAgencyIdentifier=”ISO”
  CodeListVersionIdentifier=”1.0”/>
. . .
```

```
<LocaleCode IDRef=”ID-LocaleCode”>
US
</LocaleCode>
```

4.2.3.2 Schema Definitions

The schema definitions to support this might look as follows.

```
<xs:element name=”CodeListDeclaration” type=”CodeListDeclType”/>
<xs:complexType name=”CodeListDeclType”>
  <xs:attribute name="CodeListIdentifier" type="xs:token"/>
  <xs:attribute name="CodeListAgencyIdentifier" type="xs:token"/>
  <xs:attribute name="CodeListVersionIdentifier" type="xs:token">
</xs:complexType>
. . .
<xs:element name=LocaleCode” type=”LocaleCodeType”/>
<xs:complexType name=”LocaleCodeType”>
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="IDRef" type="xs:IDREF"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

4.2.3.3 Derivation Opportunities

Since code lists are declared in the instance document, there are not many opportunities for schema type derivation. Additional attributes for supplementary components could be added by this means, though this is unlikely to be needed.

4.2.3.4 Assessment

This method allows for great flexibility, but leaves validatability and interoperability nearly out of the picture.

Requirement	**Score**	**Rank**
Semantic clarity	3	Medium to high All of the necessary information is present in the code list declaration, but retrieving it must be done somewhat indirectly.
Interoperability	1	Low to medium Standard XML entities could be provided that define the desired code lists, but there is no a machine-processable way to ensure that they get associated with the right code-usage elements.
External maintenance	2	Medium Using XML entities, external organizations could create and maintain their own code list declarations.
Validatability	0	Low Using XSD, there is no way to validate that the usage of a code matches the valid codes in the referenced code list.

Requirement	Score	Rank
Context rules friendliness	0	Low Since this method resides primarily in the instance and not the schema, the context rules have little opportunity to operate on code list definitions.
Upgradability	2	High It is easy to declare a code list with a higher version directly in the instance.
Readability	1.5	Medium to high The instance looks fairly clean, but the code list choice is a bit opaque.
Total	**9.5**	

4.2.4 Single Type Method

The single type method is currently being used in UBL, as a result of a perl script running over the Library Content SC’s modeling spreadsheet. The script makes use of our decision to use attributes for supplementary components of a CCT and elements for everything else.

4.2.4.1 Instance

The single type method results in instance documents with the following structure.

```
<LocaleCode
  CodeListIdentifier=”ISO3166”
  CodeListAgencyIdentifier=”ISO”
  CodeListVersionIdentifier=”1.0”>
US
</LocaleCode>
```

4.2.4.2 Schema Definitions

The relevant UBL library schema definitions are as follows in V0.64 (leaving out all annotation elements). Notice that `CodeType` is a complex type that sets up a series of attributes (the supplementary components for a code) on an element that has simple content of `CodeContentType` (the code itself). Also note that, although a `CodeName` attribute is defined along with its corresponding type, this is a duplicate component for the code itself, and need not be used in the instance.

```
<xs:simpleType name="CodeContentType" id="000091">
  <xs:restriction base="token"/>
</xs:simpleType>

<xs:simpleType name="CodeListAgencyIdentifierType" id="000093">
  <xs:restriction base="token"/>
</xs:simpleType>

<xs:simpleType name="CodeListIdentifierType" id="000092">
  <xs:restriction base="token"/>
</xs:simpleType>

<xs:simpleType name="CodeListVersionIdentifierType" id="000099">
  <xs:restriction base="token"/>
</xs:simpleType>
```

```
<xs:simpleType name="CodeNameType" id="000100">
  <xs:restriction base="string"/>
</xs:simpleType>

<xs:simpleType name="LanguageCodeType" id="000075">
  <xs:restriction base="language"/>
</xs:simpleType>

<xs:complexType name="CodeType" id="000089">
  <xs:simpleContent>
    <xs:extension base="cct:CodeContentType">
      <xs:attribute name="CodeListIdentifier"
        type="cct:CodeListIdentifierType">
      </xs:attribute>
      <xs:attribute name="CodeListAgencyIdentifier"
        type="cct:CodeListAgencyIdentifierType">
      </xs:attribute>
      <xs:attribute name="CodeListVersionIdentifier"
        type="cct:CodeListVersionIdentifierType">
      </xs:attribute>
      <xs:attribute name="CodeName" type="cct:CodeNameType">
      </xs:attribute>
      <xs:attribute name="LanguageCode"
        type="cct:LanguageCodeType">
      </xs:attribute>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="LanguageType" id="UBL000013">
  <xs:sequence>
    <xs:element name="IdentificationCode" . . .></xs:element>
    <xs:element name="Name" . . .></xs:element>
    <xs:element name="LocaleCode" type="cct:CodeType"
      id="UBL000016"
      minOccurs="0">
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

4.2.4.3 Derivation Opportunities

While it is possible to derive new simple types that restrict other simple types (including built-in types such as `xs:token`, used here for the actual code and other components), it is not possible to use such derived simple types directly in a UBL attribute such as `CodeListVersionIdentifier` without defining a whole new element structure. This is because you need to use the XSD `xsi:type` attribute to “swap in” the derived type for the ancestor, and you can’t put an attribute on an attribute in XML.

4.2.4.4 Assessment

This method is strong on semantic clarity because of the attributes for supplementary components, but it loses interoperability and schema flexibility because it is using a single type for everything.

Requirement	Score	Rank

Requirement	**Score**	**Rank**
Semantic clarity	4	High The various supplementary components for the code are provided directly on the element that holds the code, allowing the code to be uniquely identified and looked up.
Interoperability	0	Low The shared understanding of minimally supported code lists would have to be conveyed only in prose.
External maintenance	0	Low There is no particular XSD formalism provided for encoding the details of a code list; thus, there is no way for external organizations to create a schema module that works smoothly with the UBL library. However, there are no barriers to creating a code list (in some other form) for use in any code-based UBL element.
Validatability	0	Low There is no XSD structure for testing the legitimacy of any particular codes. All validation would have to happen at the application level (where the application uses the attribute values to find some code list in which it can do a lookup of the code provided).
Context rules friendliness	0	Low If extensions and subsets are to be managed by means of a context rules document at all, there would need to be a code list-specific mechanism added to reflect this method. If extensions and subsets don't need to be managed by means of context rules because everything happens in the application, there is no need to do anything at all.
Upgradability	2	High A document creator could merely change the `CodeListVersionIdentifier` value and supply a code available only in the new version.
Readability	1.5	Medium to high The code is accompanied by "live" supplementary components in the instance, which swells the size of instance. However, the latter are only in attributes, and it is nonetheless very clear what information is being provided.
Total	**7.5**	

4.2.5 Multiple UBL Types Method

In this method, each list is associated with a unique element, whose content is a code from that list. The element is bound to a type that is declared in the UBL library; the type ensures that the Code.Type supplementary components are documented.

4.2.5.1 Instance

The multiple UBL types method results in instance documents with the following structure.

```
<LocaleCode>
<ISO3166Code>code</ISO3166Code>
</LocaleCode>
```

The `LocaleCode` element doesn’t contain the code directly; instead, it contains a subelement that is dedicated to codes from a particular list. If codes from multiple lists are allowed here, the element could contain any one of a choice of subelements, each dedicated to a different code list.

4.2.5.2 Schema Definitions

There are many different ways that UBL can define the `ISO3166Code` element, but it probably makes sense to base it on something like the single type method (for the supplementary component attributes) and to use the enumerated type method where practical (for the primary component). Thus, the optimal form of the multiple UBL types method is really a hybrid method.

The schema definition of the types governing the `ISO3166Code` element might look like this:

```
<xs:simpleType name=”ISO3166CodeContentType”>
  <xs:extension base=”token”>
    <xs:enumeration value=”DE”/>
    <xs:enumeration value=”FR”/>
    <xs:enumeration value=”US”/>
    . . .
  </xs:extension>
</xs:simpleType>

<xs:complexType name=”ISO3166CodeType”>
  <simpleContent>
    <xs:extension base=" ISO3166CodeContentType">
      <xs:attribute name="CodeListIdentifier"
        type="cct:CodeListIdentifierType" fixed=”ISO3166”/>
      <xs:attribute name="CodeListAgencyIdentifier"
        type="cct:CodeListAgencyIdentifierType"
        fixed=”ISO”/>
      <xs:attribute name="CodeListVersionIdentifier"
        type="cct:CodeListVersionIdentifierType"
        default=”1.0”/>
      <xs:attribute name="LanguageCode"
        type="cct:LanguageCodeType"
        use=”optional”/>
  </simpleContent>
</xs:complexType>
```

Such a definition does several things:

- It enumerates the possible values of the code itself. An alternative would be just to allow the code to be a string or token, or to specify a regular expression pattern that the code needs to match.
- It provides a default value for the version of the code list being used, with the possiblity that the default could be overridden in an instance of a UBL message to provide a different version (though, since the codes are enumerated statically, if new codes were added to a new version they could not be used with this element as

currently defined). Some alternatives would be to fix the version and to require the instance to set the version value.

- It fixes the values of the code list identifier and code list agency identifier for the code list, such that they could not be changed in an instance of a UBL message. Some alternatives would be to provide changeable defaults and to require that the instance set these values.
- It makes the language code optional to provide in the instance.

4.2.5.3 Derivation Opportunities

Because a whole element is dedicated to the code for each code list, the derivation opportunities are more plentiful. A derived type could be created that does any of the following:

- Adds to the enumerated list of values by means of the XSD union technique
- Adds defaults where there were none before
- Adds fixed values where there were none before

In addition, the element *containing* the dedicated code list subelement can be modified to allow the appearance of additional code list subelements.

4.2.5.4 Assessment

This method is quite strong on most requirements; it falls down only on external maintenance.

Requirement	**Score**	**Rank**
Semantic clarity	4	High The supplementary components are always accessible, either through the instance or (through defaulting or fixing of values) the schema.
Interoperability	4	High Each code-containing construct in UBL can indicate, through schema constraints, exactly what is expected to appear there.
External maintenance	0	Low In order to work with the UBL library, the code lists maintained by external organizations would have to derive from the UBL type, which creates a circular dependency (UBL needs to include an external schema module, but the external module needs to derive from UBL). Alternatively, the UBL library has to do all the work of setting up all the desired code list types.
Validatability	4	High The constraint rules can range from very tight to very loose, and anyone who wants to subset or extend the valid values can express this in XSD terms fairly easily. The limitations are only due to XSD's capabilities.

Requirement	**Score**	**Rank**
Context rules friendliness	2	High Since there is a dedicated element for a code, it can be added or subtracted like a regular element – something that is already assumed to be part of the power of the context rules language.
Upgradability	1.5	Medium to high Depending on how the constraint rules have been set up, it might be required to define a new (possibly derived) type to allow for a new version of a code list. However, in many cases, it will be desirable to design the schema module to avoid the need for this.
Readability	1.5	Medium to high Because there is an element dedicated to the list “source” for the code, the code itself is relatively readable. However, the supplementary components are likely to be hidden away from the instance, which makes their values a bit obscure.
Total	**17**	

4.2.6 Multiple Namespaced Types Method

This method is very similar to the multiple UBL types method, with one important change: The UBL elements that each represent a code from a particular list are bound to types that may have come from an external organization’s schema module.

4.2.6.1 Instance

The namespaced type method results in instance documents with the following structure. This is identical to the multiple UBL types method, because the element dedicated to a single code list is still a UBL-native element.

```
<LocaleCode>
<ISO3166Code>code</ISO3166Code>
</LocaleCode>
```

4.2.6.2 Schema Definitions

The schema definitions to support the content of LocaleCode might look as follows. Here, three code list options are offered for a locale code. The `xmlns:` attributes that provide the namespace declarations for the `iso3166:`, `xxx:`, and `yyy:` prefixes are not shown here. It is assumed that an external organization (presumably ISO) has created a schema module that defines the `iso3166:CodeType` complex type and that this module has been imported into UBL.

```
<xs:complexType name="LanguageType">
  <xs:sequence>
    <xs:element name="IdentificationCode" . . .></xs:element>
    <xs:element name="Name" . . .></xs:element>
    <xs:element name="LocaleCode"
      type="cct:LocaleCodeType" minOccurs="0">
    </xs:element>
  </xs:sequence>
</xs:complexType>

```

```
<xs:complexType name=”LocaleCodeType” id=”. . .”>
  <xs:choice>
    <xs:element name=”ISO3166Code” type=”iso3166:CodeType”/>
    <xs:element name=”XXXCode” type=”xxx:CodeType”/>
    <xs:element name=”YYYCode” type=”yyy:CodeType”/>
  </xs:choice>
</xs:complexType>
```

Just as for the multiple UBL types method, there are many different ways that the `iso3166:CodeType` complex type can be defined, but it probably makes sense to base it on something like the single type method (for the supplementary component attributes) and to use the enumerated type method where practical (for the primary component). Thus, the optimal form of the multiple namespaced types method is really a hybrid method. For example, the definition might look like this:

```
<xs:simpleType name=”iso3166:CodeContentType”>
  <xs:extension base=”token”>
    <xs:enumeration value=”DE”/>
    <xs:enumeration value=”FR”/>
    <xs:enumeration value=”US”/>
    . . .
  </xs:extension>
</xs:simpleType>

<xs:complexType name=”iso3166:CodeType”>
  <simpleContent >
    <xs:extension base="iso3166:CodeContentType">
      <xs:attribute name="CodeListIdentifier"
        type="cct:CodeListIdentifierType"
        fixed=”xxx”/>
      <xs:attribute name="CodeListAgencyIdentifier"
        type=" iso3166:CodeListAgencyIdentifierType"
        fixed=”yyy”/>
      <xs:attribute name="CodeListVersionIdentifier"
        type=" iso3166:CodeListVersionIdentifierType"
        default=”1.0”/>
      <xs:attribute name="LanguageCode"
        type=" iso3166:LanguageCodeType"
        use=”optional”/>
  </simpleContent>
</xs:complexType>
```

Because the UBL library would not have direct control over the quality and semantic clarity of the datatypes defined by external organizations, it would be important to document UBL’s expectations on these external code list datatypes.

4.2.6.3 Derivation Opportunities

Just as for multiple UBL types, because a whole element is dedicated to the code for each code list, the derivation opportunities are more plentiful.

Also, if the external organization failed to meet our expectations about semantic clarity and didn’t add the supplementary component attributes, we could add them ourselves by defining our own complex type whose primary component (the element content) is bound to their type, or by deriving a UBL type from their external type.

4.2.6.4 Assessment

This is a strong contender in every area.

Requirement	Score	Rank

Requirement	Score	Rank
Semantic clarity	4	High The supplementary components are always accessible to the parser, either through the instance or (through defaulting or fixing of values) the schema. This assumes that UBL's high expectations on external types are met, but this is a reasonable assumption.
Interoperability	4	High Each code-containing construct in UBL can indicate, through schema constraints, exactly what is expected to appear there.
External maintenance	4	High External organizations can freely create schema modules that define elements dedicated to their particular code lists, and can even make the constraint rules as flexible or as draconian as they want.
Validatability	4	High The constraint rules can range from very tight to very loose, and anyone who wants to subset or extend the valid values can express this in XSD terms fairly easily. The limitations are only due to XSD's capabilities.
Context rules friendliness	2	High 2 Since there is a dedicated element for a code, it can be added or subtracted like a regular element – something that is already assumed to be part of the power of the context rules language.
Upgradability	1.5	Medium to high Depending on how the constraint rules have been set up, it might be required to define a new (possibly derived) type to allow for a new version of a code list. However, in many cases, the organization maintaining the code list might design the schema module in such a way as to avoid the need for this.
Readability	1.5	Medium to high Because there is an element dedicated to the list "source" for the code, the code itself is relatively readable. However, the supplementary components are likely to be hidden away from the instance, which makes their values a bit obscure.
Total	**21**	

4.3 Analysis and Recommendation

Following is a summary of the scores of the different methods.

Method	**Score**	**Comments**
Enumerated list	**11**	Spelling out the valid values assures validatability, but defining all the necessary code lists in UBL itself defeats our hope that code lists can be defined and maintained in a decentralized fashion.
QName in content	**4.5**	The idea of using XML namespaces to identify code lists is potentially useful, but because this method uses namespaces in a hard-to-process (and somewhat non-standard) manner, both semantic clarity and validatability suffer.
Instance extension	**9.5**	This method allows for great flexibility, but leaves validatability and interoperability nearly out of the picture.
Single type	**7.5**	This method is strong on semantic clarity because of the attributes for supplementary components, but it loses interoperability and schema flexibility because it is using a single type for everything.
Multiple UBL types	**17**	This method is quite strong on most requirements; it falls down only on external maintenance.
Multiple namespaced types	**21**	This is a strong contender in every area.

We recommend the multiple namespaced types method, with the addition of strong documented expectations on the external organizations that define schema modules for code lists in order to ensure maximum semantic clarity and validatability.

Note that is is possible that the UBL library will not have many external schema modules to choose from initially, and some external organizations may choose never to create schema modules for their code lists. Thus, UBL might be in the position of having to create dummy datatypes for some of the code lists it uses. In these cases, at least UBL will achieve most of the benefits, while having to balance the costs of maintenance against these benefits. It may be that UBL can even “kick-start” the interest of some external organizations in producing such a deliverable by supplying a starter schema module.

5 References

[CCTS1.8] *UN/CEFACT Draft Core Components Specification*, Part 1, 8 February, 2002, Version 1.8.

[CLTemplate] OASIS UBL Naming and Design Rules code list module template, http://www.oasis-open.org/committees/ubl/ndrsc/archive/.

[NDR] M. Cournane et al., *Universal Business Language (UBL) Naming and Design Rules*, OASIS, 2002, http://www.oasis-open.org/committees/ubl/ndrsc/archive/wd-ublndrsc-ndrdoc-nn/.

[NDRSC] OASIS UBL Naming and Design Rules subcommittee. Portal: http://www.oasis-open.org/committees/ubl/ndrsc/. Email archive: http://lists.oasis-open.org/archives/ubl-ndrsc/.

[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.

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

[3166-XSD] UN/ECE XSD code list module for ISO 3166-1, http://www.unece.org/etrades/unedocs/repository/codelist.htm.

Appendix A. Notices

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

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

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

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

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