Position Paper: Code Lists

Proposal 09, 28 May 2002

Document identifier:

p-maler-codelists-09 (Word)

Location:

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

Authors:

Eve Maler, Sun Microsystems <eve.maler@sun.com>
Fabrice Desré, France Télécom <fabrice.desre@francetelecom.com>

Abstract:

This position paper outlines several options for handling code lists in the UBL library and customizations of that library, and recommends one option for NDR SC consideration. That option was approved by the SC in its 15 May 2002 teleconference.

Status:

This is V09 of the code lists position paper intended for consideration by the OASIS UBL Naming and Design Rules subcommittee and other interested parties. It is complete, and no new revisions are planned. The recommendations made here and additional specific implementation recommendations have been incorporated into the *Universal Business Language Naming and Design Rules Specification*, and that document should now serve as the normative source for code list handling.

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

Table of Contents

1 Guidance to the UBL Modeling Process

Where possible, UBL should identify external code lists rather than design its own internal code lists. 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.

The UBL documentation must identify, for each UBL construct containing a code, the one or more code lists that must be *minimally* supported when the construct is used. Our recommendations for how to represent code lists in UBL schema modules have the effect of encapsulating this information in schema form as well.

2 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

3 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.

3.1 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.

3.1.1 Instance

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

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

3.1.2 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"/>
```

3.1.3 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>
```

3.1.4 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

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.
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**	

3.2 QName in Content Method

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

3.2.1 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.

3.2.2 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>

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

3.2.3 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.

3.2.4 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

Requirement	Score	Rank
		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 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**	

3.3 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.

3.3.1 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>
```

3.3.2 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>
```

3.3.3 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.

3.3.4 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.

Requirement	Score	Rank
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.
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**	

3.4 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.

3.4.1 Instance

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

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

3.4.2 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>

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

3.4.3 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.

3.4.4 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
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.

Requirement	Score	Rank
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**	

3.5 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.

3.5.1 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.

3.5.2 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>

<xsd: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>
</xsd: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.

3.5.3 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.

3.5.4 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.

Requirement	Score	Rank
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.
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**	

3.6 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.

3.6.1 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>
```

3.6.2 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.

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

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

<xsd: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>
</xsd: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.

3.6.3 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.

3.6.4 Assessment

This is a strong contender in every area.

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.

Requirement	Score	Rank
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**	

518

519 # 4 Analysis and Recommendation

520 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.

521 We recommend the multiple namespaced types method, with the addition of strong documented
522 expectations on the external organizations that define schema modules for code lists in order to
523 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.

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.