Universal Business Language (UBL) Code List Representation

Working Draft 2004-02-08

Document identifier:

WD-UBLCLSC-CODELIST-20040206.DOC

Location:

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

Editor:

Marty Burns for National Institute of Standards, burnsmarty@aol.com

Contributor:

Anthony Coates abcoates@londonmarketsystems.com
Mavis Cournane mavis.cournane@cognitran.com
Anne Hendry anne.hendry@sun.com
G. Ken Holman gkholman@CraneSoftwrights.com
Sue Probert sue.probert@dial.pipex.com
Alan Stitzer alan.stitzer@marsh.com

Abstract:

This specification provides rules for developing and using reusable code lists. This specification has been developed for the UBL Library and derivations thereof, but it may also be used by other technologies and XML vocabularies as a mechanism for sharing code lists and for expressing 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 Code List Subcommittee **[CLSC]**. Your comments are invited. Members of this subcommittee should send comments on this specification to the ubl-clsc@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/).

Change History

Revision	Editor	Description
2004-01-13	Marty Burns	First complete version converted from NDR revision 05
2004-01-14	Marty Burns	Minor edit of chapter heading 3 & 4
2004-01-20	Marty Burns	Incorporated descriptions from AS and KH
2004-02-06	Marty Burns	Cleaned up requirements and other sections – removed some redundant content from merge of contributions. Explicitly identified Data Model and Metadata models separately from XML representations of the same.

Table of Contents

1 Introduction 6

1.1 Scope and Audience 6

1.2 Terminology and Notation 6

2 Requirements for Code Lists 8

2.1 Overview 8

2.2 Use and management of Code Lists 8

2.2.1 [R1] First-order business information entities 8

2.2.2 [R2] Second-order business information entities 8

2.2.3 [R3] Data and Metadata model separate from Schema representation 8

2.2.4 [R4] XML and XML Schema representation 9

2.2.5 [R5] Machine readable data model 9

2.2.6 [R6] Conformance test for code lists 9

2.3 Types of code lists 9

2.3.1 [R7] UBL maintained Code List 9

2.3.2 [R8] Identify and use external standardized code lists 9

2.3.3 [R9] Private use code list 9

2.4 Technical requirements of Code Lists 9

2.4.1 [R10] Semantic clarity 10

2.4.2 [R11] Interoperability 10

2.4.3 [R12] External maintenance 10

2.4.4 [R13] Validatability 10

2.4.5 [R14] Context rules friendliness 10

2.4.6 [R15] Upgradability 10

2.4.7 [R16] Readability 10

2.4.8 [R17] Code lists must be unambiguously identified 10

2.4.9 [R18] Ability to prevent extension or modification 11

2.5 Design Requirements of Code List Data Model 11

2.5.1 [R19] A list of the values (codes) for a code list 11

2.5.2 [R20] Multiple lists of equivalents values (codes) for a code list (e.g. integers & mnemonics) 11

2.5.3 [R21] Unique identifiers for a code list 11

2.5.4 [R22] Unique identifiers for individual values of a code list 11

2.5.5 [R23] Names for a code list 11

2.5.6 [R24] Documentation for a code list 11

2.5.7 [R25] Documentation for individual values of a code list 12

2.5.8 [R26] The ability to import, extend, and/or restrict other code lists 12

2.5.9 [R27] Support for describing code lists that cannot be enumerated 12
2.5.10 [R28] Support for references to equivalent code lists 12
2.5.11 [R29] Support for individual values to be mapped to equivalent values in other code lists 12
2.5.12 [R30] Support for users to attach their own metadata to a code list 12
2.5.13 [R31] Support for users to attached their own metadata to individual values of a code list 12
2.5.14 [R32] Support for describing the past and future time-variance of the values 12
2.5.15 [R33] Identifier for UN/CEFACT DE 3055. 13
3 Data and Metadata Model for Code Lists 14
3.1 Data Model Definition 14
3.2 Supplementary Components (Metadata) Model Definition 14
3.3 Examples of Use 15
4 XML Schema representation of Code Lists 17
4.1 Data Model Mapping 17
4.2 Supplementary Components Mapping 17
4.3 Namespace URN 18
4.4 Namespace Prefix 18
4.5 Schema Location 19
4.6 Code List Schema Usage 19
4.7 Instance 20
4.8 Associating UBL Elements with Code List Types 20
4.9 Deriving New Code Lists from Old Ones 21
4.9.1 Extending code lists 21
4.9.2 Restricting code lists 22
5 Conformance to UBL Code Lists 23
6 References 24
Appendix A. Rationale for the Selection of the Code List Mechanism (Historical Non-Normative) 25
Contenders 25
A.1 Enumerated List Method 25
A.2 QName in Content Method 27
A.3 Instance Extension Method 29
A.4 Single Type Method 30
A.5 Mltiple UBL Types Method 33
A.6 Multiple Namespaced Types Method 35
A.7 Analysis and Recommendation 37
Appendix B. - ebXML Registry ClassificationScheme 39
B.1 What is ebXML Registry ClassificationScheme 39
B.2 Using ebRIM ClassificationScheme To Represent UBL Code Lists 39
B.3 Mapping Between UBL Code Lists and ebRIM ClassificationScheme 40

B.3 References 41
Appendix C. List of Rules for Codes *42*
Appendix D. Notices *43*

1 Introduction

Trading partners utilizing the Universal Business Language (UBL) must agree on restricted sets of coded values, termed "code lists", from which values populate particular UBL data fields. Code lists are accessed using many technologies, including databases, programs and XML. Code lists are expressed in UBL for XML using W3C XML Schema for authoring guidance and processing validation purposes.

It is important to note that XML schema languages are not purely abstract data models. They provide only a particular representation of the data. In addition, there are many roughly equivalent design choices (e.g. elements versus attributes). The underlying logical model is obscured, and can be difficult to extract. Therefore, XML schema languages are principally useful as a way of specifying rules to an XML validation engine. Database schemas and programming language class models provide similarly independent representations of logical data models.

A good logical data model format should allow the information about code lists to be expressed in a format that is as simple and unambiguous as possible. To maximize the abstraction on one hand, and the utility of the code list representations on the other, this document first derives an abstract data model of a code list, and then, an XMLSchema representation of that data model.

The document begins with a section expositing the requirements adopted by the committee in order to make certain that design follows requirements. These requirements were used to steer the design choices elected in the balance of the document.

This specification was developed by the OASIS UBL Code List Subcommittee **[CLSC]** to provide rules for developing and using reusable code lists expressed using W3C XML Schema **[XSD]** syntax.

The contents combine requirements and solutions previously developed by UBL's Library, Naming, and Design Rules subcommittee, the work of the National Institute of Standards "eBusiness Standards Convergence Forum" **[eBSC]**, and position papers by Anthony Coates and Gunther Stuhec.

The data model attempts to be sufficiently general to be employable with other technologies in other scenarios that are outside the scope of this committee's work. This specification is organized as follows:

- Section 2 provides requirements for code lists;
- Section 3 provides a data and metadata model of code lists;
- Section 4 is an XMLSchema representation of the model;
- Section 5 is the recommendations for code producers and the compliance rules.

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 Requirements for Code Lists

"There can be no solution without a requirement!"

This section summarizes the requirements to be addressed by this paper.

2.1 Overview

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 vocabularies as a mechanism for sharing code lists. If enough code-list-maintaining agencies adhere to these rules, we anticipate that a more open marketplace in code lists will emerge for all vocabularies.

The goal is to provide a representation for code lists that are extensible, restrictable, traceable, and cognizant of the need for code lists to be maintained by various organizations who are authorities on their content.

2.2 Use and management of Code Lists

This section describes requirements for the use and management of code lists.

2.2.1 [R1] First-order business information entities

As first-order business information entities (BIEs). 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]. For example, in XML a country code might appear as:

```
<Country>UK</Country>
```

2.2.2 [R2] Second-order business information entities

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. For example, in XML a currency code might appear as an attribute:

```
<Currency code="EUR">2456,000</Country>
```

2.2.3 [R3] Data and Metadata model separate from Schema representation

Since all uses of code lists will not be exclusively within the XML domain – ie. Databases, etc…, it is desirable to separate the description of the data model from its XML representative form. This will facilitate use for other purposes of the semantically identical information.

The current UBL code list documents speak of other XML specifications re-using UBL's code list Schemas. While this may occur, there are already many specifications whose use of XML is sufficiently different from UBL's that re-use of UBL Schemas (or Schema fragments) is not an option. That does not mean that those other specifications cannot be interoperable with UBL at the level of code lists.

Code list operability comes about when different specifications or applications use the same enumerated values (or aliases thereof) to represent the same things/concepts/etc. Sharing XML schemas (or fragments) is one way of achieving this, but it is not a necessary method for achieving this goal.

Broader interoperability can be achieved instead by defining a format which models code lists independently of any validation or choice mechanisms that they may be used with. Such a data model should be able to be processed to produce the required XML

Schemas, and should also be able to be processed to produce other artifacts, e.g. Java type-safe enumeration classes, database Schemas, code snippets for HTML forms or XForms, etc.

2.2.4 [R4] XML and XML Schema representation

The principal anticipated use of the code list model will be in XML forms – XML for usage, and XMLSchema for validation of instance documents. This paper should realize a proper XML / XMLSchema representation for the code list model.

2.2.5 [R5] Machine readable data model

A data model is an abstraction and it must be converted to explicit representation for use. The principal such use anticipated by this effort is that of XML data exchange. A machine readable representation of the data model makes the lossless transfer of all meaning to the representation of choice easier since it can be automated.

It is therefore desirable that the data model be expressed in a machine readable form.

2.2.6 [R6] Conformance test for code lists

[1/7/04 GKH] During today's coordination meeting it was suggested that CLSC address in our report criteria for the measurement of conformance ... how will someone who instantiates a code list for use in UBL measure that what they've done will conform in the UBL environment?

I'm not sure I know how myself, but it is an issue we need to either address or justify that we won't be addressing it.

2.3 Types of code lists

2.3.1 [R7] UBL maintained Code List

UBL will make use of code lists that describe information content specific to UBL.

In some cases the UBL Library may extend an existing code list to meet specific business requirements. In others cases the UBL Library may have to create and maintain a code list where a suitable code list does not exist in the public domain. Both of these type of code lists would be considered UBL-internal code lists.

2.3.2 [R8] Identify and use external standardized code lists

Because the majority of code lists are owned and maintained by external agencies, UBL will make maximum use of such external code lists where they exist. The UBL Library SHOULD identify and use external standardized code lists rather than develop its own UBL-native code lists.

2.3.3 [R9] Private use code list

This model must support the construction of private code lists where an existing external code list needs to be extended, or where no suitable external code list exists.

2.4 Technical requirements of 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.

2.4.1 [R10] 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.

2.4.2 [R11] 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.

2.4.3 [R12] 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.

2.4.4 [R13] 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.

2.4.5 [R14] 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>
```

2.4.6 [R15] 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".

2.4.7 [R16] 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.

2.4.8 [R17] Code lists must be unambiguously identified

(1) - any two uses of the same URI represent the use of the very same code list definition

(2) - no two differing code list definitions shall be represented by the same URI

The business issue is that when two trading partners identify the use of a code list, there must not be any ambiguity. Should either partner create a code list or change an existing code list, the identification of the resulting code list must be distinct from that of its origin.

[**ISSUE:** Note: for implementation considerations, Gunther has suggested the approach of namespace URI fields for code list supplemental identification values in draft-stuhec-codeListNamespaces-0p2.doc ... the "ripple effect" of this ensures that when non-UBL code lists are in use, non-UBL namespace URI strings must be used (because the UBL-standard W3C Schema fragments must be changed to utilize the non-standard code list URI strings). This guarantees the unambiguous identification of the entire schema and two UBL partners who are using the same namespace URI for a UBL schema are guaranteed to be talking about the identical element and attribute structures and code list definitions.

In contrast, original proposed UBL approaches to storing code list supplemental identification values in defaulted attributes can "hide" changes in such a way that two uses of the same namespace URI string would not represent the identical *complete* schema definition. This ambiguity could produce interoperability problems.]

2.4.9 [R18] Ability to prevent extension or modification

Certain code lists should not be extensible. For example, the list of colors, RED ORANGE YELLOW GREEN BLUE INDIGO VIOLET. I should be possible to indicate that such a code list is not extensible so the users can be assured of this constancy in its usage.

2.5 Design Requirements of Code List Data Model

What follows is a list of some of the features that a code list data model should provide.

2.5.1 [R19] A list of the values (codes) for a code list

The code list must contain at least two (2) valid values to be considered a code list and not a constant.

2.5.2 [R20] Multiple lists of equivalents values (codes) for a code list (e.g. integers & mnemonics)

Individual code values must be able to be represented in multiple ways to account for individual business requirements.

2.5.3 [R21] Unique identifiers for a code list

The code list must contain a unique identifier to be able to reference the entire code list as an item.

2.5.4 [R22] Unique identifiers for individual values of a code list

Each code within the code list must contain a unique identifier to be able to reference that particular code without knowing the code value or decode value for that code.

2.5.5 [R23] Names for a code list

Each code list must have a unique name that adequately describes the content of the list.

2.5.6 [R24] Documentation for a code list

Each code list must contain documentation which describes, in detail, the business usage for this code list.

2.5.7 [R25] Documentation for individual values of a code list

Each code value on the code list must not only contain valid values and decode values, but must also contain a long description which describes, in detail, the business meaning and usage for this code value.

2.5.8 [R26] The ability to import, extend, and/or restrict other code lists

Each code list must provide the ability to extend, restrict or import additional values for this list.

2.5.9 [R27] Support for describing code lists that cannot be enumerated

Either because of size, volatility, or proprietary restrictions (e.g. a WSDL description of a Web service that can validate which of a set of codes are members of a particular code list)

2.5.10 [R28] Support for references to equivalent code lists

Each code list must be able to refer to other code lists that may or may not be used in place of it. These references are not necessarily exactly the same, but may be equivalent based on business usage.

2.5.11 [R29] Support for individual values to be mapped to equivalent values in other code lists

Each code list value must be able to refer to other code list values that may or may not be used in place of it. These references are not necessarily exactly the same, but may be equivalent based on business usage.

2.5.12 [R30] Support for users to attach their own metadata to a code list

Each code list must have the flexibility to have additional descriptive information added by an individual user to account for unique business requirements.

2.5.13 [R31] Support for users to attached their own metadata to individual values of a code list

Each code value must have the flexibility to have additional descriptive information added by an individual user to account for unique business requirements.

2.5.14 [R32] Support for describing the past and future time-variance of the values

An effective date and expiration date should be established so that the code list can be scoped in time. See, for example, "Patterns for things that change with time", http://martinfowler.com/ap2/timeNarrative.html

2.5.15 [R33] Identifier for UN/CEFACT DE 3055.

Many code lists have been defined by UN/CEFACT. The code list model requires a representation of an identifier for this standard UNTDED 3055 [UNTDED 3055%%%% add reference]. This identifier uniquely identifies UN/EDIFACT standard code lists.

3 Data and Metadata Model for Code Lists

This section provides rules for developing and using reusable code lists. 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 vocabulary wishing to share code lists. See section 4.0 Conformance.

> **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.

Since the UBL Library is based on the ebXML Core Components Version1.9, 11 December 2002; see **[3166-XSD]** UN/ECE XSD code list module for ISO 3166-1, **[CCTS1.9]**), the supplementary components identified for the *Code. Type* core component type are used to identify a code as being from a particular list.

3.1 Data Model Definition

The data model of a code list is presented below.

CCT	UBL Name	Object Class	Property Term	Represen-tation Term	Primitive Type	Card.	Remarks
	Code.Content	Code	Content	Text	String	1..1	Required
	Code.Description	Code Description	Description	Text	String	0..n	Optional
	Code.Value	Code Value	Value	Numeric	Number	1..1	Optional

3.2 Supplementary Components (Metadata) Model Definition

The following model contains the supplementary components description of a code list.

CCT	UBL Name	Object Class	Property Term	Represen-tation Term	Primitive Type	Card.	Remarks
	name	Code	Name	Text	String	0..1	Optional
	listID	Code List	Identification	Identifier	String	0..1	Optional
	listName	Code List	Name	Text	String	0..1	Optional
	listVersionID	Code List	Version	Identifier	String	0..1	Optional
	listAgencyID	Code List Agency	Identification	Identifier	String	0..1	Optional
	listAgencyName	Code List Agency	Name	Text	String	0..1	Optional

	listAgencySchemeID	Code List Agency	Scheme	Identifier	String	0..1	Optional
	listAgencySchemeAgencyID	Code List Agency	SchemeAgency	Identifier	String	0..1	Optional

340 # 3.3 Examples of Use

341 The data type "Code" is used for all elements that should enable coded value representation in the communication between
342 partners or systems, in place of texts, methods, or characteristics. The list of codes should be relatively stable and should not be
343 subject to frequent alterations (for example, CountryCode, LanguageCode, ...). Codelists must have versions.

344 If the agency that manages the code list is not explicitly named and is specified using a role, then this takes place in a tag name.

345 The following types of code can be represented:

346 a.) Standardized codes whose code lists are managed by an agency from the code list DE 3055.

Code	Standard
listID	Code list for standard code
listVersionID	Code list version
listAgencyID	Agency from DE 3055 (excluding roles)
listAgencySchemeID	-
listAgencySchemeAgencyID	-

347 b.) Proprietary codes whose code lists are managed by an agency that is identified by using a standard.

Code	Proprietary
listID	Code list for the propriety code
listVer	Version of the code list
listAgencyID	Standardized ID for the agency (normally the company that manages the code list)
listAgencySchemeID	ID schema for the schemeAgencyId
listAgencySchemeAgencyID	Agency DE 3055 that manages the standardized ID 'listAgencyId'

348 c.) Proprietary codes whose code lists are managed by an agency that is identified without the use of a standard.

Code	Proprietary
listID	Code list for the proprietary code
listVer	Code list version
listAgencyID	Standardized ID for the agency (normally the company that manages the code list)
listAgencySchemeID	ID schema for the schemeAgencyId
listAgencySchemeAgencyID	'ZZZ' (mutually defined from DE 3055)

349 d.) Proprietary codes whose code lists are managed by an agency that is specified by using a role or that is not specified at all.

The role is specified as a prefix in the tag name. listID and listVersionID can optionally be used as attributes if there is more than one code list. If there is only one code list, no attributes are required.

Code	Proprietary
listID	ID schema for the proprietary identifier
listVer	ID schema version
listAgencyID	-
listAgencySchemeID	-
ListAgencySchemeAgencyID	-

4 XML Schema representation of Code Lists

This section describes how the data model is mapped to XMLSChema [needs reference???].

Note that the code list is derived in two pieces – a simpleType that contains the actual content of the code list, and, a complexType with simple content that attaches the optional supplementary components to the enumeration.

1) Define an abstract element for inclusion in extensible schemas (note: this is "placebo")
2) Define a simpleType to hold the enumerated values
3) Define a complexType to add the supplementary components
4) Define an element that substitutes for the abstract type to enable usage in unextended schemas
5) Define a comprehensive URN to hold supplementary components that can qualify uniqueness of usage

4.1 Data Model Mapping

The following table summarizes the component mapping of the data model

UBL Name	XMLSchema Mapping
Code.Content	1. Abstract element `<xs:element name="{code.name}A" type="xsd:token" abstract="true"/>` 2. Simple type to hold code list values and optional annotations `<xs:simpleType name="{code.name}Type"> <xs:restriction base="xsd:token"> <xs:enumeration value="{code.content}"/> <xs:enumeration value="{code.content}"/> <xs:enumeration value="{code.content}"/> . . . </xs:restriction> </xsd:simpleType>` 3. Complex type to associate supplementary values with code list values that substitutes for the abstract type. `<xs:element name="{code.name}" type="{code.name}Type" substitutionGroup="{code.name}TypeA">` 4. Element to substitute for abstract element in non-exended schemas `<xs:element name="LocaleCode" type="LocaleCodeType" substitutionGroup="LocaleCodeTypeA"/>`
Code.Description	xsd:annotation/ xsd:documentation/
Code.Value	xsd:annotation/ xsd:documentation/

4.2 Supplementary Components Mapping

The following table shows all supplementary components of the code type. It shows additionally the current representation by using attributes and the recommended representation by using namespaces and annotations.

UBL Name	XMLSchema Mapping	Optional
	URN mapping	complex type attribute mapping
Code.name	xsd:annotation/ xsd:documentation/	

	cc:codename	
Code.listID	namespace (URN) 1. position Mandatory	`<xs:attribute name="listID" type="xs:token"/>`
Code.listName	namespace (URN) 2. position Optional	`<xs:attribute name="listName" type="xs:token"/>`
Code.listVersionID	namespace (URN) 3. position Mandatory	`<xs:attribute name="listVersionID" type="xs:string"/>`
Code.listAgencyID	namespace (URN) 4. position optional	`<xs:attribute name="listAgencyID" type="xs:token"/>`
Code.listAgencyName	namespace (URN) 5. position optional	`<xs:attribute name="listAgencyName" type="xs:token"/>`
Code.listAgencySchemeID	namespace (URN) 6. position optional	`<xs:attribute name="listID" type="xs:token"/>`
Code.listAgencySchemeAgencyID	namespace (URN) 7. position optional	`<xs:attribute name="listAgencySchemeID" type="xs:token"/>`

4.3 Namespace URN

The following construct represents the construct for the URN of a code list, according OASIS URN:

```
urn:oasis:tc:ubl:codeList:<Code List. Identification. Identifier>:<Code List. Name.
Text>:<Code List. Version. Identifier>:<Code List. Agency Identifier>:<Code List.
Agency Name. Text>:<Code List. Agency Scheme. Identifier>:<Code List. Agency Scheme
Agency. Identifier>
```

The first four parameters are fixed by Uniform Resource Name (URN) [see RFC 2141] and OASIS URN [see RFC 3121]:

- urn --> leading token of URNs
- oasis --> registered namespace ID “oasis”
- tc --> Technical Committee Work Products
- ubl --> From Technical Committee UBL (Universal Business Language)

The parameter “codeList” identifies the schema type “code list”.

The following parameters from <Code List. Identifier> to <Code List. Agency Scheme Agency. Identifier> represents the specific code list supplementary components of the CCT codeType.

Example:

```
urn:oasis:tc:ubl:codeList:ISO639:Language%20Code:3:ISO:International%20Standardizati
on%20Organization::
```

4.4 Namespace Prefix

Namespace prefix could be freely defined. However, it is helpful for better understanding, to identity the code lists by a convention of namespace prefixes.

The prefix provides the namespace prefix part of the qualified name of each code list. It is recommended that this prefixe should contain the information of the supplementary component <Code List. Identification Identifier> and if it is necessary for separation, the information of the supplementary component <Code List. Version. Identifier> separated by a dash "-". All letters should be lower case.

Example:

```
iso639
iso639-3 (with version)
```

4.5 Schema Location

A question for code lists related to namespace identification is also the schemaLocation. The schema location includes the complete URI, which is used to identify code list schemas.

Every code list must normally be provided by the specific responsible agency. Therefore the following URI should be used for these codelists:

```
http://www.<Code List. Agency Name. Text>.org/ubl/codeLists/<Code List.
Identification. Identifier>_<Code List. Version. Identifier>.xsd
```

The name "ubl" specifies that the specific code list be based on the UBL convention. Under "codeLists" will be listed all specific code lists of this responsible agency.

Example:

```
http://www.iso.org/ubl/codeLists/iso639_3.xsd
```

If some responsible agencies cannot provide their own code lists by a URI, it is possible that these code lists could be provided by OASIS. In the fashion of other OASIS specifications, UBL specific code lists of other responsible agencies will be located under the UBL committee directory:

```
http://www.oasis-open.org/committees/ubl/codeLists/<Code List. Agency Name.
Text>/<Code List. Identification. Identifier>_<Code List. Version. Identifier>.xsd
```

Example:

```
http://www.oasis-open.org/committees/ubl/codeLists/ISO/iso639_3.xsd
```

4.6 Code List Schema Usage

For every code list, there exists a specific code list schema. This code list schema must have a targetNamespace with the UBL specific code list namespace and have a prefix with the code list identifier itself.

The element in the code list schema can be used for the representation as a global declared element in the document schemas. The name of the element is the UBL tag name of the specific BIE for a code.

The simpleType represents the possible codes and the characteristics of the code content. The name of the simpleType must be always ended with "..Content". Within the simpleType is a restriction of the XSD built-in data type "xsd:token". This restriction includes the specific facets "length", "minLength", "maxLength" and "pattern" for regular expressions to describe the specific characteristics of each code list.

Each code will be represented by the faucet "enumeration" after the characteristics. The value of each enumeration represents the specific code value and the annotation includes the further definition of each code, like "Code. Name", "Language. Identifier" and the description.

The schema definitions to support this might look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
```

```
       targetNamespace="urn:oasis:ubl:codeList:ISO4217:Currency%20Code:3:5:ISO::"
       xmlns:iso4217="urn:oasis:ubl:codeList:ISO4217:Currency%20Code:3:5:ISO::"
       xmlns:xs="http://www.w3.org/2001/XMLSchema"
       elementFormDefault="qualified" attributeFormDefault="unqualified">

<xs:element name="LocaleCodeTypeA" type="xs:token"
        abstract="true">
  <xs:annotation>
    <xs:documentation>
       An abstract place holder for a code list element
    </xs:documentation>
  </xs:annotation>
</xs:element>

<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"
                    substitutionGroup="LocaleCodeTypeA">
    <xs:annotation>
      <xs:documentation>
             A substitution for the abstract element based
             on aStdEnum
      </xs:documentation>
    </xs:annotation>
</xs:element>

<xs:element name="LocaleCode" ref="LocaleCodeTypeA"/>
</xs:schema>
```

4.7 Instance

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

```
<LocaleCode>US</LocaleCode>
```

4.8 Associating UBL Elements with Code List Types

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 global `CountryIdentificationCode` element is assumed to require a code from the hypothetical ISO 3166 code list defined in Section 3.1. Thus, it needs to reference the `iso3166:ISO3166Code` global element.

Every first-order code appearing in the UBL Library must be double-wrapped.

[**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
      ref="ubl:CountryIdentificationCode"/>
  </xs:sequence>
</xs:complexType>

<xs:element name=”CountryIdentificationCode”>
    ...
    <xs:element ref=”iso3166:ISO3166Code”/>
</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 element definition for `CountryIdentificationCode` would change as follows (assuming the Codes “R” Us module were properly imported):

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

<xs:element name=”CountryIdentificationCode”>
  ...
  <xs:choice>
    <xs:element ref=”iso3166:ISO3166Code”/>
    <xs:element ref=”codesrus:CodeRUsCode”/>
  </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.

4.9 Deriving New Code Lists from Old Ones

In order to promote maximum reusability and ease code lists maintenance, code list designers are expected to build new code lists from existing lists. They could for example combine several code lists or restrict an existing code list.

These new code lists must be usable in UBL elements the same manner the “basic” code lists are used.

4.9.1 Extending code lists

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

```
<xs:schema targetNamespace="cust"
  xmlns:std="std"
  xmlns="cust"
  xmlns:cust="custom"
  xmlns:xs=http://www.w3.org/2001/XMLSchema
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

<xs:import namespace="std"
```

```
  schemaLocation="D:\_PROJECT\NIST\XMLSchema\test0513\std.xsd"/>

<xs:element name="LocaleCode" substitutionGroup="std:LocaleCodeA">
  <xs:annotation>
    <xs:documentation>A substitute for the abstract LocaleCodeA
      that extends the enumeration
    </xs:documentation>
  </xs:annotation>
  <xs:simpleType>
    <xs:union memberTypes="std:aStdEnum">
      <xs:simpleType>
        <xs:restriction base="xs:token">
         <xs:enumeration value="IL"/>
         <xs:enumeration value="GR"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
</xs:element>
</xs:schema>
```

4.9.2 Restricting code lists

The base schema shown above could be restricted to support a subset of codes as follows:

```
<xs:import namespace="std"
  schemaLocation="D:\_PROJECT\NIST\XMLSchema\test0513\std.xsd"/>
<xs:element name="LocaleCode" substitutionGroup="std:LocaleCodeA">
  <xs:annotation>
    <xs:documentation>
      A substitute for the abstract LocaleCodeA that restricts
        the enumeration
    </xs:documentation>
  </xs:annotation>
  <xs:simpleType>
    <xs:restriction base="xs:token">
    <xs:enumeration value="DE"/>
    <xs:enumeration value="US"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Let's consider we want to union the code"R"Us code list and the ISO3166 code list to create a compound list.

5 Conformance to UBL Code Lists

This section is for Producers of Code Lists outside of UBL. These lists could be owned by a number of different type of organizations. The conformance

We probably need a Conformance section in this document so that code list producers (who, in general, won't be UBL itself) will know how/when to claim conformance to the requirements (MUST) and recommendations (SHOULD/MAY) in this specification. This spec is not for the UBL TC, but for code list producers (which may occasionally include UBL itself).

6 References

[3166-XSD] UN/ECE XSD code list module for ISO 3166-1, **[CCTS1.9]** *UN/CEFACT Draft Core Components Specification*, Part 1, 11 December, 2002, Version 1.9.

[CLSC] OASIS UBL Code List Subcommittee. Portal: http://www.oasis-open.org/committees/sc_home.php?wg_abbrev=ubl-clsc . Email archive: http://lists.oasis-open.org/archives/ubl-clsc/.

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

[eBSC] "eBusiness Standards Convergence Forum", http://www.nist.gov/ebsc.

[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/.

[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. http://www.unece.org/etrades/unedocs/repository/codelist.htm.

Appendix A. Rationale for the Selection of the Code List Mechanism (Historical 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.

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.

A.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.

A.1.1 Instance

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

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

A.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"/>
```

A.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>
```

A.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.
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.

Requirement	Score	Rank
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	

A.2 QName in Content Method

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

A.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.

A.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>
```

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

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

694 A.2.3 Derivation Opportunities

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

698 A.2.4 Assessment

699 The idea of using XML namespaces to identify code lists is potentially useful, but because this method uses namespaces in a hard-
700 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 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.

Requirement	Score	Rank
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	

A.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.

A.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>
```

A.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>
```

A.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.

A.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.
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	

A.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.

A.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>
```

A.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>
```

A.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.

A.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.
Upgradability	2	High A document creator could merely change the `CodeListVersionIdentifier` value and supply a code available only in the new version.

Requirement	Score	Rank
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	

A.5 MItiple 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.

A.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.

A.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.

A.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.

A.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.

Requirement	Score	Rank
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	

A.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.

A.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>
```

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

<xs:complexType name=”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.

A.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.

A.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	

953 # A.7 Analysis and Recommendation

954 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.

Method	Score	Comments
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.

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

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

Appendix B. - ebXML Registry ClassificationScheme

This section provides the proposed text for inclusion in the UBL specification to add a non-normative recommendation to use ebXML Registry ClassificationScheme XML Schema as a schema for representing UBL Code lists. The author is committed to working with the UBL TC on this proposal as deemed necessary by that body.

B.1 What is ebXML Registry ClassificationScheme

The OASIS ebXML Registry standard defines an abstract information model for representing structured taxonomies. It also defines a normative binding of this model to XML Schema which may be used to define structured taxonomies in a standard XML format.

In this model a taxonomy is represented by a class named ClassificationScheme while taxonomy values are represented by a class named ClassificationNode. Any taxonomy, its taxonomy values and the hierarchical structure of its taxonomy values may be defined using an instance of a ClassificationScheme and a set of ClassificationNode instances arranged in a hierarchical structure. Figure 1 shows the information model for ClassificationScheme in UML format.

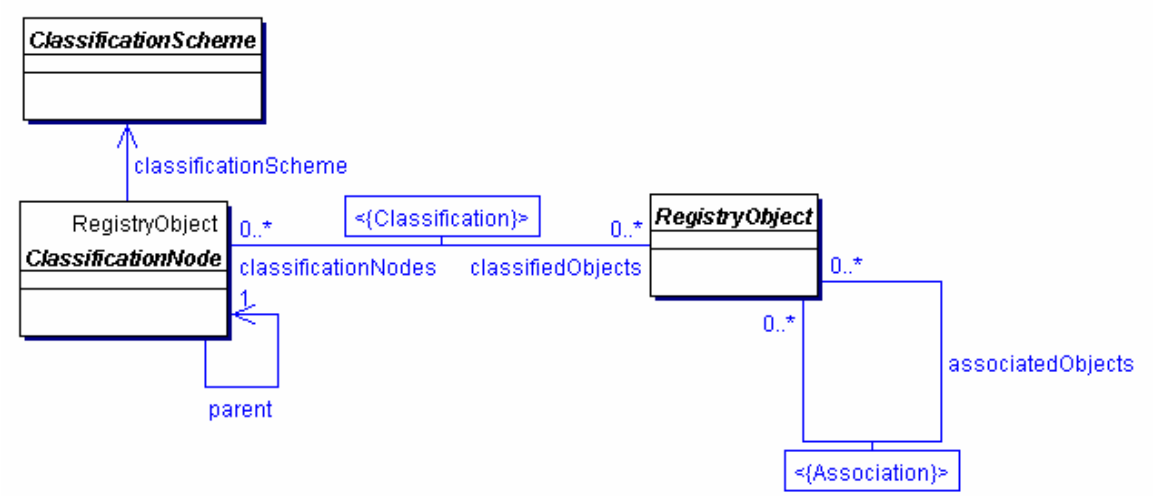

Figure 1: Information Model Classification *View*

In addition to the information model classes defined above, ebRIM also defines a class called Slot which is used to add dynamic attributes to any object (including ClassificationScheme and ClassificationNode). Slots provide for attribute extensibility within ebRIM.

B.2 Using ebRIM ClassificationScheme To Represent UBL Code Lists

The ebRIM ClassificationScheme information model and its normative binding to an XML Schema representation is recommended for representing UBL code lists for the following reasons:

- Provide an open, standards-based XML schema that can be used to represent UBL code lists.
- Supports the " UBL Code List Rules" defined by [wp-ubl-codelist].
- Is extensible to accommodate additional requirements in the future.
- Allows any UBL code lists to be based upon and validated by a single common XML schema.

- Enable the definition of hierarchical UBL code lists.
- Make it easier to use ebXML Registry to store UBL content.

B.3 Mapping Between UBL Code Lists and ebRIM ClassificationScheme

A normative binding to XML schema [ebRIM Schema] has been defined for the abstract ebRIM ClassificationScheme information model shown in Figure 1. This section describes how the ebRIM ClassificationScheme schema may be used to represent UBL code lists.

At the highest level, a UBL code lists maps to an ebRIM ClassificationScheme while the values within the code list map to an ebRIM ClassificationNode. The following example illustrates a very simple code list for representing Gender:

```
<ClassificationScheme id="urn:uuid:d1462ca5-a643-46e9-b3da-eda1403d9d3a"
isInternal="true" nodeType="UniqueCode"  userVersion="1.0">
  <Name><LocalizedString lang="en-US" charset="UTF-8" value="Gender"/></Name>
  <Description><LocalizedString lang="en-US" charset="UTF-8" value="A gender code
list"/></Description>

  <Slot name="xmlNameSpace">
    <ValueList><Value>urn:nameSpaceURN</Value></ValueList>
  </Slot>

  <Slot name="responsibleOrganization">
    <ValueList><Value>urn:orgURN</Value></ValueList>
  </Slot>

  <ClassificationNode id="urn:uuid:4c764c0d-6248-4017-b58e-e0b1667fa2e5"
code="Male">
    <Name><LocalizedString lang="en-US" charset="UTF-8" value="Male"/></Name>
    <Description><LocalizedString lang="en-US" charset="UTF-8" value="Code for
Male"/></Description>
  </ClassificationNode>

  <ClassificationNode id="urn:uuid:078f0d7b-5f3a-4aa6-8b59-af6b91da4185"
code="Female">
    <Name><LocalizedString lang="en-US" charset="UTF-8" value="Female"/></Name>
    <Description><LocalizedString lang="en-US" charset="UTF-8" value="Code for
Female"/></Description>
  </ClassificationNode>

</ClassificationScheme>
```

[wp-ubl-codelist] defines that a UBL code list representations MAY include the following attributes. This section defines the mapping to ebRIM:

Code Attribute Name	Mapping in ebRIM
Name	Name element of ClassificationNode
listID	Slot with same name

listName	Slot with same name
listVersionID	userVersion attribute of ClassificationScheme
listAgencyID	Slot with same name
listAgencyName	Slot with same name
listAgency-SchemeID	Slot with same name
listAgency-SchemeAgencyID	Slot with same name
xml:lang	Lang attribute of LocalizedString in Name and Description
xlink:href	Slot with same name
xlink:role	Slot with same name
xlink:type	Slot with same name

Using the simple mapping provided above, any UBL code lists may be represented within ebRIM Classification XML Schema and be adherent to [wp-ubl-codelist].

B.3 References

[ebRIM] ebXML Registry Information Model version 2.1
http://www.oasis-open.org/committees/regrep/documents/2.1/specs/ebRIM.pdf

[ebRIM Schema] ebXML Registry Information Model Schema
http://www.oasis-open.org/committees/regrep/documents/2.1/schema/rim.xsd

(Note version 2.5 will soon be TC approved. Note sure which you want to reference. Version 2.1 is OASIS approved 2.5 has just been TC approved this week and will be available on web site in next 3 weeks).

Appendix C. List of Rules for Codes

[R 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.

[R 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.

[R 3] In the code list type, attributes must be defined at least for the code list identification identifier (`listID`), code list agency identifier (`listAgencyID`), and code list version identifier (`listVersionID`). Defining attributes for the code name (`name`) 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.

[R 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.

[R 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.

[R 6] A global element in the agency's namespace may optionally be defined and associated with the code list type.

Be aware 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).

[R 7] Every first-order code appearing in the UBL Library must be double-wrapped.

Appendix D. 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.