POSTSECONDARY
Electronic Standards Council

PESC Guidelines for XML Architecture and Data Modeling

Version 3.0

April 29, 2005

A publication of the

Postsecondary Electronic Standards Council (PESC)

Executive Summary

Business Problem

Historically, the companies, agencies, and institutions that comprise the education community at large have developed their systems, data models, and data standards independently. This has resulted in a lack of common data definitions and enterprise data standards across the education domain. While the need to conduct business electronically has increased dramatically in recent years, differing data definitions have made mapping and analyzing data between systems difficult, and, consequently, have hindered the building of standard interfaces that ensure data quality and consistency.

Solution

Recognizing the need to standardize its own business processes and data definitions, the Department of Education's Office of Federal Student Aid (FSA) in 2000 embarked upon a comprehensive plan to rationalize its existing systems and processes and base data exchange on XML. In 2002, this initiative was aligned with a broader community initiative when FSA began to work with the Postsecondary Electronic Standards Council (PESC) and the National Council of Higher Education Loan Programs (NCHELP) on the development and design of an XML standard for higher education. Based on the principles originally codified in FSA's XML Framework, this standard seeks to provide all stakeholders maximum benefit from the adoption of XML technology through better, faster, and cheaper information exchange.

Now under the direction of PESC, this community-wide standards initiative seeks to establish enterprise definitions for data that is commonly exchanged by members of the education community as well as the policies, procedures, and standards that will guide further developments. These goals are embodied in two distinct mechanisms maintained by PESC:

- XML Core Component Dictionaries
- Guidelines for XML Architecture and Data Modeling

Together, these components provide a robust technical foundation that ensures quality, consistency, and longevity of data exchange interfaces.

Core Components

Core Components are reusable data structures, modeled in XML that provide standard definitions for key business concepts and entities across the education domain. Having a common set of data definitions and XML models will help the community improve data quality and integration services between disparate systems that must work together. The initial Core Component Dictionaries have been jointly developed by the community under the auspices of PESC, and are now available via the PESC / FSA XML Registry and Repository.

Guidelines for XML Architecture and Data Modeling

Core Components, to be useful for exchanging data, must be assembled into message specifications. Furthermore, message specifications and the Core Components upon which they are based must expand and evolve as business needs change. To guide these processes, this document, the PESC Guidelines for XML Architecture and Data Modeling, and its companion volume, the Standards Forum Policies and Procedures Manual, have been developed. These documents are to be used in conjunction with the XML Core Component Dictionaries to provide the methodology, standards, conceptual framework, and policies needed to assemble, maintain, and expand standard, consistent message specifications.

Table of Contents

Tables and Figures

1 Introduction

1.1 Overview

This document, the Guidelines for XML Architecture and Data Modeling, provides information for developers and implementers of PESC-compliant XML.

Based on PESC's XML Technical Specification for Higher Education, and the XML Technical Reference and Usage Guidelines – developed by the Department of Education's Office of Federal Student Aid (FSA) to guide its usage of XML – it has been revised and expanded by members of the Standards Forum for Education to be a community-wide document. Every effort has been made to build on the experience and work done previously by other standards organizations within and outside of Higher Education: W3C, ebXML, IFX, X12, CommonLine, IMS, IEEE, and ISO, among others.

The reference information provided herein is based on standards and best practices that the PESC membership has developed on community-wide XML Schema development projects. Specifically, these standards and guidelines have been developed through the process of implementing the Common Origination and Disbursement (COD) Common Record and Academic Transcript, as well as defining the draft XML schemas for a number of other initiatives (i.e., Institutional Student Information Record [ISIR] and CommonLine). These guidelines will help developers create XML interfaces which are consistent with PESC's XML standards in order to achieve industry-wide standardization and interoperability.

The development of this specification served to clarify, for the Standards Forum, the most efficient work processes and the ultimate deliverables of the standing and ad hoc work groups that make up the Standards Forum for Education. As these work groups and their needs evolve and expand, so will this document, as it will in conjunction with changes to XML and its related standards.

1.2 Purpose

The purpose of this specification is to guide the work of the Standards Forum, providing a framework for decisions that face the following groups:

- The Standards Forum as an organization, as its structure changes to meet the needs of the higher education community
- The higher education community as it implements XML message data exchanges

This document provides the standards and guidelines the education community can use to ensure consistent and efficient development of XML Schemas for message specifications. It provides numerous XML examples, best practices, and patterns to which developers can refer. Ideally, two developers could be assigned a schema development task, be given a set of requirements, and, using the guidelines in this document, would construct technically similar XML Schema message specifications. Achieving this type of consistency can aid the community greatly in improving the quality of data as it is exchanged across organizations.

1.3 Scope

The scope of the XML standards in this specification is the data that institutions and their partners exchange in support of the business processes within Higher Education, such as student financial aid, admissions, and registration. While schemas may be designed for other purposes, such as for data presentation, this is not the primary focus of the Standards Forum.

Since the business processes within Higher Education require data interchange, PESC schemas are data oriented and may in some cases mirror paper-based documents. Their content models focus more on semantics (or "content") than on presentation or structure (where the content model contains some degree of presentation orientation mixed with semantics). Consequently, the guidelines and standards in this specification have a similar orientation.

The Guidelines for XML Architecture and Data Modeling provides introductory information on XML Data Modeling. Specifically, the document includes information on:

- XML Schema Design Best Practices
- XML Schema Design Patterns
- XML Schema Development Methodology

1.4 Intended Audience

The intended audience of this document is the Standards Forum for Education as well as members of the education community at large wishing to use XML in their data exchanges. It is targeted to advanced XML Schema developers who need to build complex XML Schema message specifications, and assumes that the developer is familiar with XML Schema development.

1.5 Organization of the Document

The Guidelines for XML Architecture and Data Modeling consists of the following sections:

- Section 1: Introduction provides a high level overview, scope, and assumptions of this document.
- Section 2: Overview of XML provides an introduction to XML. This section is not a comprehensive exposition on XML, but rather it highlights topics that readers should be familiar with before reading this document.
- Section 3: XML Schema Design Best Practices provides schema designers with a set of best practices that should be considered for and used in design projects.
- Section 4: PESC XML Schema Structure describes the layered schema structure and conventions that have been adopted by PESC.
- Section 5: Design Patterns for Core Components provides specific XML Schema coding solutions for some of the more common data structures that will be needed for XML message specifications.
- Section 6: XML Schema Development Methodology describes the design / build / test methodology that should be followed when developing PESC-compliant schemas.

- Section 7: XML Schema Object Management provides a discussion on managing XML artifacts, from Core Components, to Sector Libraries, to Message Specifications. It covers the use of versioning and namespace techniques to manage objects.
- Appendix A: Revision History provides a list of changes to this document since its initial publication.

1.6 Assumptions

The Guidelines for XML Architecture and Data Modeling is based on the following assumptions:

- This document is written assuming that the reader has an understanding of XML. It is not intended to be a comprehensive XML tutorial. It only covers topics that can be clarified for the purposes of standardization. It is intended to be a guide for PESC members and the education community in general, to use when implementing XML throughout the community.

- Developers of new XML Schema message specifications will have access to the Core Component definitions stored in the PESC XML Registry and Repository for the Education Community.

2 Overview of XML

2.1 The Basics of XML

eXtensible Markup Language (XML) is a meta-language—a language for describing other languages—that allows for the design of customized vocabularies for different types of documents. In an XML document, data is enclosed in tags that "mark-up" the content, providing it structure and meaning. XML makes it easy for a computer to generate data, read data, and ensure that the data structure is unambiguous.

XML consists of elements that are defined by tags. A start tag precedes the name of an element. An end tag follows it. The following is a sample element for a person's last name.

```
<LastName>Jones</LastName>
```

"<LastName>" is the start tag. "Jones" is the data, or XML content. "</LastName>" is the end tag.

While XML employs the kind of tags used in HTML, XML is not a replacement for HTML. XML uses tags to identify data elements, or what data is, while HTML uses tags to identify data attributes, or how data looks. XML can be used in conjunction with HTML to store data within standard Web pages. It can also be used to store data in files and to pull information from disparate databases.

Simple Elements

Elements can be either complex or simple. A simple element is one that has no child elements. In the following example, the simple elements are in bold type.

Simple Elements

```
<Name>
   <FirstName>Heidi</FirstName>
   <LastName>Smith</LastName>
</Name>
```

Complex Elements

A complex element is one that has child attributes and/or elements. In the following example, the complex element, "Name" is in bold type.

Complex Element

```
<Name>
   <FirstName>Heidi</FirstName>
   <LastName>Smith</LastName>
</Name>
```

Maximum Length Values

XML does not require text data to occupy the maximum length specified for a tag. In the following example the FirstName element has a maximum length of 12; however, in the correct example, only 5 characters are included between the start and end tags.

Element Definition

```
<xsd:element name="FirstName" nillable="true" minOccurs="0">
      <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                  <xsd:minLength value="0"/>
                  <xsd:maxLength value="12"/>
            </xsd:restriction>
      </xsd:simpleType>
</xsd:element>
```

Incorrect Example

```
<FirstName>Heidi          </FirstName>
```

Correct Example

```
<FirstName>Heidi</FirstName>
```

Leading Zeros

XML does not require numeric data to include leading zeros to fill out the maximum value specified for a tag. In the following example, the TotalCount element has a maximum value of 999,999,999; however, in the correct example, no leading zeros are included in the instance document.

Element Definition

```
   <xsd:element name="TotalCount" nillable="true" minOccurs="0">
      <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
                  <xsd:minInclusive value="0"/>
                  <xsd:maxInclusive value="999999999"/>
            </xsd:restriction>
      </xsd:simpleType>
</xsd:element>
```

Incorrect Example

```
   <TotalCount>000000099</TotalCount>
```

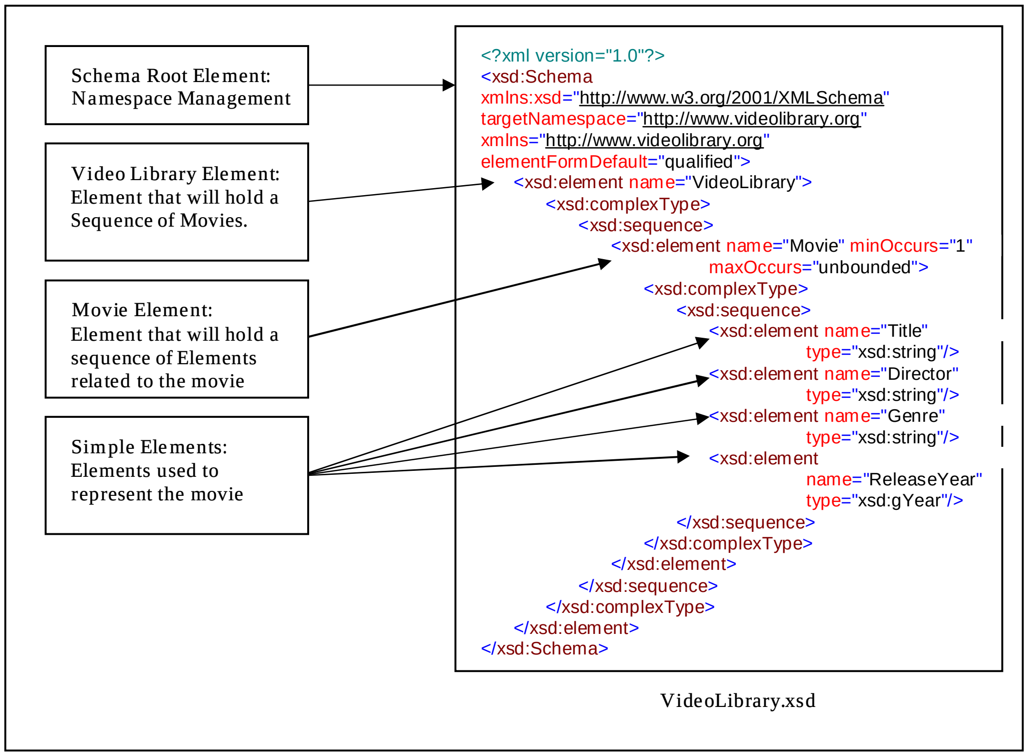
Correct Example

```
   <TotalCount>99</TotalCount>
```

2.2 XML Schemas

An XML Schema is a collection of element definitions, using an XML format, that specifies the rules surrounding the logical structure of some collection of data. In essence, it is a vocabulary that defines the allowed content of XML documents that are based upon it. It defines the elements present in the document and the order in which they appear, as well as any attributes that may be associated with an element.

The following example shows a very simple way a Schema designer might implement a Schema for a movie library.


Schema Root Element: Namespace Management
Video Library Element: Element that will hold a Sequence of Movies.
Movie Element: Element that will hold a sequence of Elements related to the movie
Simple Elements: Elements used to represent the movie
VideoLibrary.xsd

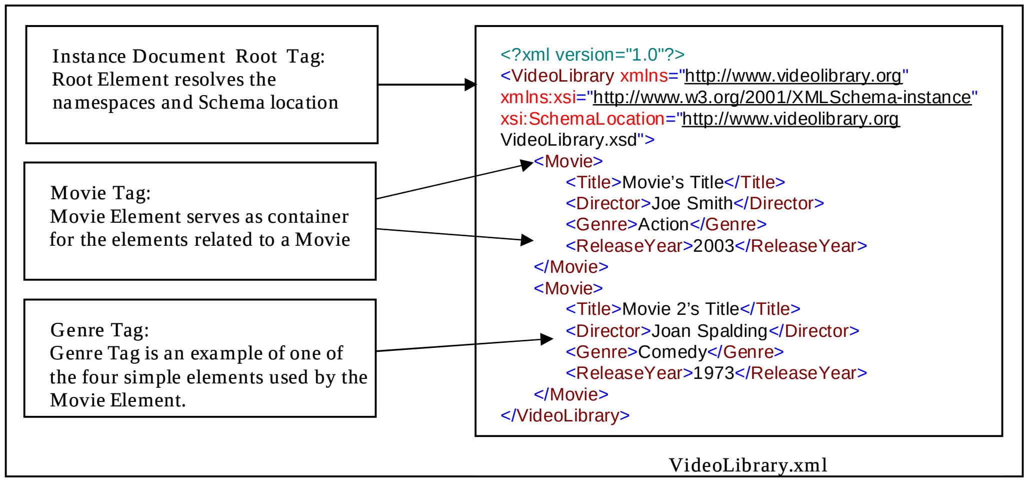
```
<?xml version="1.0"?>
<xsd:Schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.videolibrary.org"
xmlns="http://www.videolibrary.org"
elementFormDefault="qualified">
    <xsd:element name="VideoLibrary">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Movie" minOccurs="1"
                        maxOccurs="unbounded">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="Title"
                                    type="xsd:string"/>
                            <xsd:element name="Director"
                                    type="xsd:string"/>
                            <xsd:element name="Genre"
                                    type="xsd:string"/>
                            <xsd:element
                                    name="ReleaseYear"
                                    type="xsd:gYear"/>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:Schema>
```

Figure 2.1 – Basic XML Schema Example

2.3 XML Instance Documents

Once a Schema is defined, instance documents based on the specifications in the Schema can be created. An XML instance document represents one possible set of data for a particular markup language or XML vocabulary. It contains declaration, elements, attributes, and, most importantly, data that needs to be transferred between applications. It might be saved as a file or sent over the internet as the payload of a message.

An example instance document based on the Movie Library Schema illustrated in Figure 2.1 is depicted below.


Instance Document Root Tag:
Root Element resolves the namespaces and Schema location
Movie Tag:
Movie Element serves as container for the elements related to a Movie
Genre Tag:
Genre Tag is an example of one of the four simple elements used by the Movie Element.
VideoLibrary.xml

```
<?xml version="1.0"?>
<VideoLibrary xmlns="http://www.videolibrary.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:SchemaLocation="http://www.videolibrary.org
VideoLibrary.xsd">
    <Movie>
        <Title>Movie’s Title</Title>
        <Director>Joe Smith</Director>
        <Genre>Action</Genre>
        <ReleaseYear>2003</ReleaseYear>
    </Movie>
    <Movie>
        <Title>Movie 2's Title</Title>
        <Director>Joan Spalding</Director>
        <Genre>Comedy</Genre>
        <ReleaseYear>1973</ReleaseYear>
    </Movie>
</VideoLibrary>
```

Figure 2.2 – Basic XML Instance Document Example

2.4 Summary

XML Schema provides a standard typing system for defining markup languages and validating XML documents. The following sections address best practices and design principles that should be employed in developing PESC-compliant XML Schemas.

3 XML Schema Design Best Practices

XML Schemas enforce rules around the content of XML instance documents. The W3C XML Schema language specification describes a fairly complicated syntax for defining these rules. The complexity is not far different from that of a programming language. It is not surprising, then, that in many cases, there are several ways to accomplish the same basic goal. There are a number of different ways a schema can create a framework for identical XML instance documents.

This section will give schema developers a set of schema design best practices that will be used in future schema design projects. The best practices will allow XML Schema developers to develop schemas that employ the following concepts:

- Organization – Concept that Schemas will follow similar organization patterns. This applies to the physical Schema document, as well as the namespaces where the Schema objects will reside.
- Consistency – Concept that Schemas should not only have a consistent design pattern, but that the XML instance documents created should also have a consistent design pattern.
- Extensibility – Concept that Schemas should be designed to easily allow for additions at a later date.
- Flexibility – Concept that Schemas should enable and facilitate growth and change as data transfer requirements change.
- Reuse – Concept that new Schemas should not be built from scratch, but should be able to leverage previously developed Schemas.

3.1 Developing Reusable XML Structures

In XML instance documents, viewers see XML tags that have data stored between them. These XML Tags are defined as elements in XML Schemas. The allowable content for an element can be based on the built-in types provided by XML, such as date, string, and integer.

Example Element #1

```
<xsd:element name=”BirthDate” type="xsd:date">
```

Example Tag #1

```
<BirthDate>1980-04-26</BirthDate>
```

Example Element #2

```
<xsd:element name=”PhoneNumber” type="xsd:string">
```

Example Tag #2

```
<PhoneNumber>703-292-0694</PhoneNumber>
```

XML's built-in types alone are not sufficient to adequately describe Core Components, however. In the example above, saying that a phone number is a string does not adequately describe a data type for phone number. For example, according to the definition above for "PhoneNumber", data that is not a valid Phone Number may be placed between the "PhoneNumber" tags without causing a validation error.

Example Tag #2

```
<PhoneNumber>ThisIsValidInXML,ButNotAValidPhoneNumber</PhoneNumber>
```

Fortunately, XML Schema provides designers a methodology to extend the base types. Each primitive data type has a set of optional facets that can be used to describe the valid data for a particular element. For example, the primitive data type String has the following optional facets which a Schema designer can use to modify the string data type:

- length
- minLength
- maxLength
- pattern
- enumeration
- whitespace (legal values: preserve, replace, collapse)

In the following example we will create a simple type that can be used to store a phone number in the following pattern: "###-###-####" (# is a digit 0-9). We will use one of the optional facets that the string datatype provides to create a simple type for phone number.

Example Simple Type

```
<xsd:element name="PhoneNumber">
   <xsd:simpleType>
         <xsd:restriction base="xsd:string">
               <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
         </xsd:restriction>
   </xsd:simpleType>
</xsd:element>
```

In this example we use the pattern facet to limit the string datatype. This facet uses regular expressions. In this example, the pattern is three digits, followed by a "-", followed by three digits, followed by a "-", followed by 4 digits. References for more information on regular expressions can be found in Appendix A.

Complex Types

Complex types provide designers an additional set of tools to use for building schemas. Complex types allow schema designers to define child elements and to define attributes for elements. Attributes allow a schema designer to add an additional level of detail to an element. Attributes will be discussed in the Attribute Construction section. Child elements allow

Schema designers to build object hierarchy into their Schema designs. Say that a Schema designer wanted to model a Movie. Specifically we want to model the following characteristics:

- Title
- Director
- Genre
- Release Year

We would want to create a complex type containing those elements. One way to do this is to create a Complex Type with a "sequence" of elements.

Example Complex Type Schema Snippet

```
<xsd:element name="Movie">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="Title" type="xsd:string"/>
                  <xsd:element name="Director" type="xsd:string"/>
                  <xsd:element name="Genre" type="xsd:string"/>
                  <xsd:element name="ReleaseYear" type="xsd:gYear"/>
             </xsd:sequence>
      </xsd:complexType>
</xsd:element>
```

The Schema snippet above shows the creation of an element "Movie" that has four elements. The first three elements are of type string, and the last is made of type gYear, Gregorian Year. The instance of this Schema snippet would look like the following:

Example Complex Type Instance Snippet

```
   <Movie>
      <Title>Movie's Title</Title>
      <Director>Joe Smith</Director>
      <Genre>Action</Genre>
      <ReleaseYear>2003</ReleaseYear>
   </Movie>
```

Complex Types and Simple Types can be used in combination. Assume that the "Genre" element only has the following valid values:

- Action
- Comedy
- Drama
- Mystery

Example Complex Type and Simple Type

```
<xsd:element name="Movie">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="Title" type="xsd:string"/>
                  <xsd:element name="Director" type="xsd:string"/>
                  <xsd:element name="Genre">
                        <xsd:simpleType>
                              <xsd:restriction base="xsd:string">
                                    <xsd:enumeration value="Action"/>
                                    <xsd:enumeration value="Comedy"/>
                                    <xsd:enumeration value="Drama"/>
                                    <xsd:enumeration value="Mystery"/>
                              </xsd:restriction>
                        </xsd:simpleType>
                  </xsd:element>
                  <xsd:element name="ReleaseYear" type="xsd:gYear"/>
             </xsd:sequence>
      </xsd:complexType>
</xsd:element>
```

The previous example instance document would still be valid under this new Schema. However, the following example would not, because “Science Fiction” is not listed as a possible enumeration value.

Example Invalid Instance Snippet

```
<Movie>
      <Title>Movie’s Title</Title>
      <Director>Joe Smith</Director>
      <Genre>Science Fiction</Genre>
      <ReleaseYear>2003</ReleaseYear>
</Movie>
```

Named Types

In the previous examples, the schemas have defined types only when they are declaring elements. It is also possible to define a type outside of an element declaration. When this is done, a Schema designer will give the Complex Type or Simple Type a name. This name can be referenced in any future types or elements that want to use this type.

Example Named Simple Type

```
<xsd:simpleType name=”PhoneNumberType”>
      <xsd:restriction base="xsd:string">
            <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
      </xsd:restriction>
</xsd:simpleType>
<xsd:element name=”PhoneNumber” type=”PhoneNumberType/>
```

In this example, a “PhoneNumberType” is defined. This definition is used as the type by the “PhoneNumber” declaration. Excluding namespace consideration, this user defined type operates the same way that XML’s primitive types operate.

The use of named simple and complex types is the preferred method for defining types in PESC-compliant XML Schemas. Named types provide for reusability and common definitions throughout the higher education domain.

Groups

Groups allow Schema designers to create an association between elements or an association between attributes. The following example demonstrates how to create a Complex Type by using a group.

Example Group and Complex Type

```
<xsd:group name="MovieSupplementalElements">
      <xsd:sequence>
            <xsd:element name="Director" type="xsd:string"/>
            <xsd:element name="Genre">
                  <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                              <xsd:enumeration value="Action"/>
                              <xsd:enumeration value="Comedy"/>
                              <xsd:enumeration value="Drama"/>
                              <xsd:enumeration value="Mystery"/>
                        </xsd:restriction>
                  </xsd:simpleType>
            </xsd:element>
            <xsd:element name="ReleaseYear" type="xsd:gYear"/>
      </xsd:sequence>
</xsd:group>

<xsd:element name=”Movie”>
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="Title" type="xsd:string"/>
                  <xsd:group ref=”movieSupplementalElements”/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>
```

In this example the Schema designer creates a group of elements related to the “Movie” Object. When defining the “Movie” element, the Schema designer has defined the element “Title”, and added the elements in the “movieSupplementalElements” group. This Schema snippet produces a “Movie” element which is equivalent to “Movie” element created in the previous Schema snippet.

3.2 Hide vs. Expose Namespaces

A typical Schema will use types and elements from multiple Schemas, each with different namespaces. These namespaces can be hidden (also called localized) or they may be exposed.

The Schema attributes elementFormDefault and attributeFormDefault are the mechanisms for hiding or exposing namespaces. The attribute value of “unqualified” will hide the namespaces from the instance documents while the attribute value of “qualified” will expose the underlying namespaces to the instance documents.

It is important to note that all schemas must have a consistent use of elementFormDefault and attributeFormDefault. These values only apply to the Schema that they are in and do not apply to Schemas that are imported. If there are mixed uses of these attributes then the instance document will have to include the namespace qualifiers for some elements and not include it for other elements.

The following examples show the difference between a sample XML snippet in which namespaces are qualified or exposed, and one in which namespaces are unqualified or hidden.

Qualified:

```
<?xml version="1.0" encoding="UTF-8"?>
<CRCRequest:CommonRecordCommonline
   xmlns:CRCRequest="urn:org:pesc:message:CommonLineRequest:v1.0.2"
   xmlns:FFEL="urn:org:pesc:sector:Aid-Delivery-FFEL:v1.0.2"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:org:pesc:message:CommonLineRequest:v1.0.2
CommonLineRequest_v1.0.2.xsd
   urn:org:pesc:sector:Aid-Delivery-FFEL:v1.0.2
FFELAlternative_v1.0.2.xsd"
   DocumentProcessCode="TEST">
  <FFEL:TransmissionData>
    <FFEL:CreatedDateTime>2001-12-31T12:00:00</FFEL:CreatedDateTime>
    <FFEL:DocumentTypeCode>Request</FFEL:DocumentTypeCode>
    <FFEL:Source>
      <FFEL:Lender>
        <FFEL:OPEID>123</FFEL:OPEID>
        <FFEL:NonEDBranchID>456</FFEL:NonEDBranchID>
        <FFEL:OrganizationName>My University</FFEL:OrganizationName>
      </FFEL:Lender>
    </FFEL:Source>
```

Unqualified:

```
<?xml version="1.0" encoding="UTF-8"?>
<CRCRequest:CommonRecordCommonline
   xmlns:CRCRequest="urn:org:pesc:message:CommonLineRequest:v1.0.2"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:org:pesc:message:CommonLineRequest:v1.0.2
CommonLineRequest_v1.0.2.xsd
   DocumentProcessCode="TEST">
  <TransmissionData>
    <CreatedDateTime>2001-12-31T12:00:00</CreatedDateTime>
    <DocumentTypeCode>Request</DocumentTypeCode>
    <Source>
      <Lender>
        <OPEID>123</OPEID>
        <NonEDBranchID>456</NonEDBranchID>
        <OrganizationName>My University</OrganizationName>
      </Lender>
    </Source>
```

Schemas should be designed to hide Namespaces[1]. Hiding namespaces provides for XML instance documents that are easier to read and understand, most notably when Schemas import definitions from another namespace. Hiding namespaces moves the complexity of a document's framework to the Schema level. Restricting instance documents to a single namespace qualifier at the root level follows the recommendation of the ASC X12 Reference Model for XML Design.

Additionally, maintenance is easier when hiding namespaces as it is possible to change a Schema without impact to instance documents.

3.3 Element Construction

Overview

When an element is created, it is created with either local scope or global scope. Local scope means that the element is only able to be used within the object under which it is created. Global scope means that the element is able to be reused.

In the “Movie” Schema snippet, the Movie object was created in global scope while the other elements were created in local scope. This means that only the “Movie” element would be able to be reused by another element in the Schema.

Schema designers can use the following three element construction patterns:

- Russian Doll Design
- Salami Slice Design
- Venetian Blind Design (Strongly Recommended)

Each of these designs has its advantages and disadvantages. Each methodology of constructing complex element constructs can yield identical instance documents. Each of the Schemas in the Element Construction section will produce instance snippets that look like the following:

Example Element Construction Instance Snippet

```
<Movie>
   <Title>Movie’s Title</Title>
   <Director>Joe Smith</Director>
   <Genre>Action</Genre>
   <ReleaseYear>2003</ReleaseYear>
</Movie>
```

Some approaches, however, offer far more flexibility and reusability than others. The modeling approach strongly recommended for PESC XML Schema is the Venetian Blind Design, because it offers maximum reusability with the simplest management of namespace issues.

[1] This has been standard practice for XML Schema development. It is not the standard practice for XML-based web services, however. Consequently, this statement will be revisited by the PESC Technical Advisory Board in 2005.

Russian Doll Design

The Russian Doll Design defines objects in local scope. When elements are created using this methodology, a Schema will look very similar to the instance document. However, this limits the reusability of Schema designs. The Russian Doll Design does facilitate the hiding of namespaces, which can prevent certain namespace issues like name collisions.

Example Russian Doll Design Schema Snippet

```
<xsd:element name=”Movie”>
   <xsd:complexType>
         <xsd:sequence>
               <xsd:element name="Title" type="xsd:string"/>
               <xsd:element name="Director" type="xsd:string"/>
               <xsd:element name="Genre" type="xsd:string"/>
               <xsd:element name="ReleaseYear" type="xsd:gYear"/>
         </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

Salami Slice Design

The Salami Slice Design defines all objects in the global scope. When elements are created using this methodology, object reuse is very easy. However, a user’s mapping between the Schema and an instance document will not be as straight forward. It should be noted that this limitation does not carry over to automated validation of instance documents against Schemas. Since the Salami Slice Methodology allows for the reuse of elements, Schema designers must be cognizant of possible issues like name collisions. Name collisions occur when an object with a unique definition uses the same element type or attribute name as a previously used object. Because the Salami Slice Design pattern defines objects in a global namespace, name collisions are more likely to occur. Refer to Section 3.5 for more information on Namespaces.

Example Salami Slice Design Schema Snippet

```
<xsd:element name=”Movie”>
   <xsd:complexType>
         <xsd:sequence>
               <xsd:element ref="Title"/>
               <xsd:element ref="Director"/>
               <xsd:element ref="Genre"/>
               <xsd:element ref="ReleaseYear"/>
         </xsd:sequence>
   </xsd:complexType>
</xsd:element>
<xsd:element name="Title" type="xsd:string"/>
<xsd:element name="Director" type="xsd:string"/>
<xsd:element name="Genre" type="xsd:string"/>
<xsd:element name="ReleaseYear" type="xsd:gYear"/>
```

With the Salami Slice Design, if a Schema designer also needs to create an element “Book”, the Schema designer could reuse any of the previously created elements (e.g. “Title” and “ReleaseYear”) in the creation of the “Book” element.

Venetian Blind Design (Strongly Recommended)

The Venetian Blind Design leverages the design advantages of both the Russian Doll Design and the Salami Slice Design. It facilitates reuse while also hiding namespace complexities. It does so by creating type definitions. Instead of actually creating elements and referencing them, a Schema designer would create a type, and reference that when creating their elements.

Example Venetian Blind Design Schema Snippet

```
<xsd:simpleType name=”TitleType”>
   <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
<xsd:simpleType name=” DirectorType”>
   <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
<xsd:simpleType name=”GenreType”>
   <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
<xsd:simpleType name=”ReleaseYearType”>
   <xsd:restriction base="xsd:gYear"/>
</xsd:simpleType>
<xsd:complexType name=”MovieType”>
   <xsd:sequence>
      <xsd:element name="Title" type=”TitleType”/>
      <xsd:element name="Director" type=”DirectorType”/>
      <xsd:element name="Genre" type=”GenreType”/>
      <xsd:element name="ReleaseYear" type=”ReleaseYearType”/>
   </xsd:sequence>
</xsd:complexType>
<xsd:element name=”Movie” type=”MovieType”/>
```

Conclusion

It is possible to create identical instance documents using any of the three different element design patterns discussed in this section. However, if an enterprise is striving for data consistency, a Schema designer must choose one of the element design patterns that facilitate the reuse of data definitions. This design principle is critical since PESC’s Schemas will be based on the Core Components stored in the XML Registry and Repository. Since element reuse is an important design principle for PESC’s Schemas, only Salami Slice Design and the Venetian Blind Design offer this capability and can be considered. However, when comparing the Venetian Blind Design to the Salami Slice design, the Venetian Blind Design has the advantage of hiding namespace complexities. Therefore, the Venetian Blind Design is the choice most strongly recommended for PESC’s element design pattern.

3.4 Explicit Versus Generic Names for Elements

The Standards Forum has had ongoing discussions regarding the best approach for modeling data elements or groups of data that are similar in concept but have different data requirements or meanings in the context of a business document. Examples include differentiating between a home address and a business address, a domestic address and a foreign address, or various types of financial awards.

Essentially, there are two methods for representing these semantics in XML: 1) by defining a unique element for each variation on a general type – e.g., ShippingAddress, MailingAddress, etc. could all be child elements of a Person element or complex type, the structure of each being defined by a single AddressType complex type – and 2) by defining a generic complex element that has, as a child, a "type" element or attribute that indicates context or usage and, if necessary, constraining the element or attribute by an enumerated list of acceptable values – e.g. include a Type attribute as part of the AddressType complex type, which has an enumerated list of values like Home, Work, etc.

Both of these methods are valid and advantageous in specific situations. Defining unique elements, for example, tightly constrains the schema, allowing business rules to be validated by the parser. The advantage of the second method is a generic and flexible structure that can be used in a number of instances.

Since each of these methods has its place, knowing when to employ the proper technique is largely a modeling decision requiring thorough analysis to understand the business use of the element(s). Specifically, modelers should consider whether or not the similar elements have structural differences or differences in data requirements between them and the business semantics of the element(s) vis-à-vis the parent element.

These points are covered in more detail in the sections that follow. Included are guidelines regarding which XML technique to employ based on specific requirements.

Elements with Structural Differences

When similar objects have structural differences, preference should be given to defining each object separately. The rationale for this approach is that they are separate entities despite some common elements or usage. From an object-oriented perspective, these may be subclasses of a common super class, or they may be separate classes that implement the same interface. In either case, they are separate classes that should be modeled as such.

In the financial aid domain, for example, a student could be applying for multiple financial aid awards. Since each of these awards has its own data requirements and processing logic, each is modeled as a separate child element, as the following XML representation reflects.

```
<xsd:element name="Student" type="StudentType"/>

<xsd:complexType name="StudentType">
   <xsd:sequence>
         <xsd:element name="DLAwardSub" type=" DLAwardSubType"
         minOccurs="0"/>
         <xsd:element name="DLAwardPLUS" type=" DLAwardPLUSType"
         minOccurs="0"/>
         <xsd:element name="PellAward" type=" PellAwardType"
         minOccurs="0"/>
```

```
            <xsd:element name="CampusBasedAward" type=" CampusBasedAward
            Type"
      minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>
```

Understand Child Element Semantics

Child elements should be defined in a manner that preserves their business semantics vis-à-vis their parent element. For example, while all addresses are essentially the same from a conceptual and structural standpoint, when defined as a child of another element, such as a student or a loan guarantor, an address may acquire additional semantics that must be captured. Understanding how the child element relates to its parent, both structurally and within the context of a business transaction, will help in determining what additional information, if any, needs to be captured, as well as the proper XML modeling technique to employ.

Tightly Constrained Relationships

If the relationship between a child element and its parent needs to be constrained in terms of cardinality or to enforce business rules, explicit child elements should be used whose names convey the nature of the relationship.

For example, suppose that a Business may have multiple addresses, but that each address is always either a Billing Address, a Shipping Address, or a Mailing Address. Furthermore, suppose that a Mailing Address is always required.

This can be represented most accurately and efficiently with 3 separate child elements of Business – MailingAddress, BillingAddress, and ShippingAddress – each of which is of type AddressType, as reflected below.

```
   <xsd:element name="Business" type="BusinessType"/>

   <xsd:complexType name="BusinessType">
      <xsd:sequence>
            <xsd:element name="Name" type="xsd:string"/>
            <xsd:element name="MailingAddress" type="AddressType"/>
            <xsd:element name="BillingAddress" type="AddressType"
            minOccurs="0"/>
            <xsd:element name="ShippingAddress" type="AddressType"
            minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="AddressType">
      <xsd:sequence>
            <xsd:element name="Street" type="xsd:string" maxOccurs="3"/>
            <xsd:element name="City" type="xsd:string"/>
            <xsd:element name="State" type="xsd:string"/>
            <xsd:element name="PostalCode" type="xsd:string"/>
      </xsd:sequence>
   </xsd:complexType>
```

This representation is superior to having a single repeatable Address element that includes a "type" element since the context or use of each address is clearly indicated. Furthermore, the business rule that at least one MailingAddress is required for a business can be constrained in the schema. Reusability of the address structure is still accomplished through the use of a complex type that is used for each named element.

Conveying Ancillary Information

If the data that a child element conveys primarily refers to usage, description, or other ancillary information that does not alter the basic relationship of the element to its parent, a child, "type" element, constrained by an enumeration of acceptable values, should be used.

For example, suppose that a Person may have multiple addresses, exactly as the Business in the example above. In addition, a requirement exists that the schema capture whether the address is a business or personal address. Suppose further that this distinction does not alter the relationship of the address to the person from the standpoint of the schema designer – i.e. an address still will be a MailingAddress, ShippingAddress, or BillingAddress, and whether each address is personal or business is merely ancillary information. Given these requirements, the AddressType complex type could be enhanced to include a Type element (or attribute) that would be constrained by an enumerated list of values indicating whether the address was Personal or Business.

3.5 Use of Elements vs. Attributes

Deciding whether to model information using an element or an attribute is difficult. No universal rule exists and the debate often takes on esoteric rather than technical overtones. Nevertheless, there are a number of facts about attributes and elements that must be considered in the decision-making process:

- Attributes cannot contain any complex structures, and therefore are not extendable. As a result, if a concept that holds a simple element becomes more complex, it must be remodeled as an element.
- Attributes cannot be repeated under an element. If the data under an element repeats, such as address lines under a street address, an attribute cannot be used.
- Attributes can only be declared as required or optional. Elements not only can be declared as required or optional, they can also be part of more expressive constructions. For example, they can be given cardinality constraints (minOccurs and maxOccurs), be part of a choice group, or be part of a substitution group.

For these reasons, elements should be used in the design of PESC Standards Forum schemas in the majority of circumstances; especially since these schemas are oriented towards data exchange. Institutions need a form of exchange that can be understood by computers and humans alike, which elements, with their ability to reflect hierarchical structure, order, and cardinality, more readily provide.

Attributes may be used to capture information that describes an element but is not a constituent part of that element. Used in this manner, attributes capture metadata – information that describes an element, such as a ID numbers, URLs, types, and other references.

The following guidelines have been compiled from works by Eliot Kimber and C. M. Sperberg-McQueen in order to assist in determining when to use an element and when to use an attribute.

1. Determine if the data in question is fundamentally metadata or content. Metadata is information that describes the container while content is the information the container conveys.
    a. Use an embedded element when the information you are recording is a constituent part of the parent element.
    b. Use an attribute when the information is inherent to the parent but not a constituent part (one's head and one's height are both inherent to a human being, but one's head is a constituent part and one's height isn't – you can cut off a person's head, but not their height).
2. Determine the structural requirements for the data: does the data item syntactically conform to the rules for attributes or does it require more structuring?
    a. Use embedded elements for complex structure validation.
    b. Use attributes for simple data type validation.
3. Determine how the data is intended to be used – i.e. primarily for the conveyance of domain information or for the processing of information.
    a. Use elements to capture domain information since they can have substructures, order, and are more readily extensible.
    b. Use attributes for data processing information such as IDs or "key" data since they can be easily located and processed.
4. Use attributes to stress the one-to-one relationship among pieces of information, i.e., to stress that the element represents a tuple of information.

To illustrate the recommended use of elements and attributes, consider a student for which an id, name, and email address must be captured.

This information could be modeled using either:

- all Elements (see Example-1A.xsd and Example-1A.xml)
- all Attributes (see Example-1B.xsd and Example-1B.xml)
- a combination of the two (see Example-2.xsd and Example-2.xml)

While all of these capture the same information, the structure used in the last method is preferred since it incorporates several of the guidelines listed above.

The student's name and email address are treated as content – "information the container conveys" – since this is the domain information to be captured and communicated. Both are modeled as child elements, providing a logical, extensible structure. The NameType, for example, only contains a FirstName element. This could later be expanded to include LastName and MiddleName elements as well as attributes that capture metadata about the name – e.g. whether or not the name is preferred, the name type, such as legal name, maiden name, etc.

The id, on the other hand, is modeled as an attribute since, in this example, at least, it is considered “key” data; making it more important for data processing than for conveying information about the student.

Example-1A.xsd - (Use of Elements vs. Attributes)

```
<?xml version=”1.0”?>
<xsd:schema
   targetNamespace=”urn:org:pesc:sector:example1A”
   xmlns:Example-1A=”urn:org:pesc:sector:example1A”
   xmlns:xsd=”http://www.w3.org/2001/XMLSchema”
   elementFormDefault=”unqualified”
   attributeFormDefault=”unqualified”>
  <xsd:element name=”Student” type=”Example-1A:StudentType”/>
    <xsd:complexType name=”StudentType”>
      <xsd:sequence>
        <xsd:element name=”ID” type=”xsd:string”/>
        <xsd:element name=”Name” type=”xsd:string”/>
        <xsd:element name=”Email” type=”xsd:string”/>
      </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Example-1A.xml – (Use of Elements vs. Attributes)

```
<?xml version=”1.0”?>
<Example-1A:Student
   xmlns:Example-1A=”urn:org:pesc:sector:example1A”
   xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
   xsi:schemaLocation=”urn:org:pesc:sector:example1A Example-1A.xsd”>
  <ID>9906789</ID>
  <Name>Adam</Name>
  <Email>adam@smplu.edu</Email>
</Example-1A:Student>
```

Example-1B.xsd – (Use of Elements vs. Attributes)

```
<?xml version=”1.0”?>
<xsd:schema
   targetNamespace=”urn:org:pesc:sector:example1B”
   xmlns:Example-1B=”urn:org:pesc:sector:example1B”
   xmlns:xsd=”http://www.w3.org/2001/XMLSchema”
   elementFormDefault=”unqualified”
   attributeFormDefault=”unqualified”>
  <xsd:element name=”Student” type=”Example-1B:StudentType”/>
  <xsd:complexType name=”StudentType”>
    <xsd:attribute name=”id” type=”xsd:string” use=”required”/>
    <xsd:attribute name=”name” type=”xsd:string” use=”required”/>
    <xsd:attribute name=”email” type=”xsd:string” use=”required”/>
  </xsd:complexType>
</xsd:schema>
```

Example-1B.xml – (Use of Elements vs. Attributes)

```
<?xml version="1.0"?>
<Example-1B:Student
   id="9906789"
   name="Adam"
   email="adam@smplu.edu"
   xmlns:Example-1B="urn:org:pesc:sector:example1B"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:org:pesc:sector:example1B Example-1B.xsd"/>
```

Example-2.xsd – (Use of Elements vs. Attributes)

```
<?xml version="1.0"?>
<xsd:schema
   targetNamespace="urn:org:pesc:sector:example2"
   xmlns:Example-2="urn:org:pesc:sector:example2"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="unqualified"
   attributeFormDefault="unqualified">
  <xsd:element name="Student" type="Example-2:StudentType"/>
  <xsd:complexType name="StudentType">
    <xsd:sequence>
      <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="Email" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:schema>
```

Example-2.xml – (Use of Elements vs. Attributes)

```
<?xml version="1.0"?>
<Example-2:Student
   id="9906789"
   xmlns:Example-2="urn:org:pesc:sector:example2"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:org:pesc:sector:example2 Example-2.xsd">
  <Name>Adam</Name>
  <Email>adam@smplu.edu</Email>
</Example-2:Student>
```

Another instance where attributes can and should be used is as an index for each item of a list of items. The name of the attribute in this case should almost always logically follow to be the word "Number".

The criteria for when to use the "Number" attribute would be:

- There is a group of items that naturally fall into an ordered list. That is, a group of similar items is not enough; these items also have an order to them.

- The list can conceptually be represented with only some of the members present. That is, the first and third items might usefully be sent without the second. For this reason, the position in the list would not alone be enough to distinguish the index of each item.

An example of an improper use of the "Number" attribute is address lines. In an address, often one may include more than one AddressLine sub-element to fully list the address. However, the second AddressLine can really not logically exist without the first AddressLine. Therefore, it is not necessary to tag each line with a "Number" attribute:

```
<Address>
   <AddressLine>10 Main Street</AddressLine>
   <AddressLine>Apt 202</AddressLine>  <!--No need for Number attribute-->
   <!-- Other address fields -->
</Address>
```

An example for the proper use of the "Number" attribute is a set of disbursements for loans. In a list of disbursements, an attribute of Number helps identify each disbursement block. Note that the attribute allows the listing of disbursements 1 and 3 without including 2.

```
<Disbursement Number="1">
...      <!-- Disbursement Data -->
</Disbursement>
<Disbursement Number="3">
...      <!-- Disbursement Data -->
</Disbursement>
```

3.6 Object – Oriented Design

Subclassing and Composition

When building aggregate objects, there are several methods that can be followed. One option is called Subclassing. This design methodology builds a hierarchy of type definitions that creates an aggregate object. The second option is called Composition. This methodology combines objects into an aggregate object by referencing groups of objects.

In order to explain the differences between these two methods, consider how one would want to model Pell Grants, Unsubsidized Direct Loans, PLUS Loans, and Subsidized Stafford Loans. Assume that these award types have some commonly shared elements.

Design by Subclassing

The Subclassing method would require a Schema designer to create a base FinancialAwardType that could be defined for all shared elements. Then the Schema designer would have to extend that FinancialAwardType to create a PellType. This new type would include any Pell Grant specific objects. The Schema designer would also have to extend FinancialAwardType to create a DLType to include any Direct Loan specific objects. Finally, the Schema designer would have to extend the DLType to include any specific objects for each of the three Direct Loan types (DLPLUSType, DLSubsidizedType, and DLUnsubsidizedType). The figure below illustrates this example.

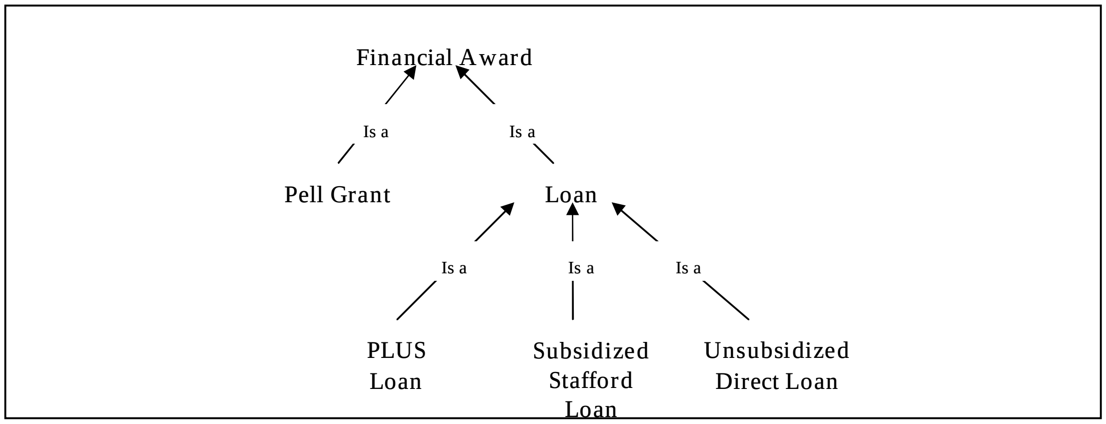


Figure 3.1 – Design by Subclassing Diagram

Example Subclassing Schema Snippet

```
<xs:complexType name="FinancialAwardType">
...
</xs:complexType>

<xs:complexType name=”LoanType”>
   <xs:extension base=”FinancialAwardType”>
   ...
   </xs:extension>
</xs:complexType>

<xs:complexType name=”PLUSType”>
   <xs:extension base=”LoanType”>
   ...
   </xs:extension>
</xs:complexType>
```

The difficulty with this technique for assembling complex types is that an individual who is trying to understand a PLUS Loan, needs to understand the LoanType, and the FinancialAwardType. While this is possible to do, it can become difficult as the number of levels of inheritance increases. Another problem with this method is that a Schema designer may make a change to the FinancialAwardType, and not realize how those changes have downstream ramifications on all of the types which inherit from FinancialAwardType both directly and indirectly.

Design by Composition

The Composition method does not rely on this inheritance hierarchy to create complex types. It relies on the referencing of groups of elements. Instead of a PLUS Direct Loan being a Direct Loan, the PLUS Direct Loan uses the elements from the Direct Loan Group of elements. The Direct Loan Group of elements uses the elements form the Financial Award Group of elements.

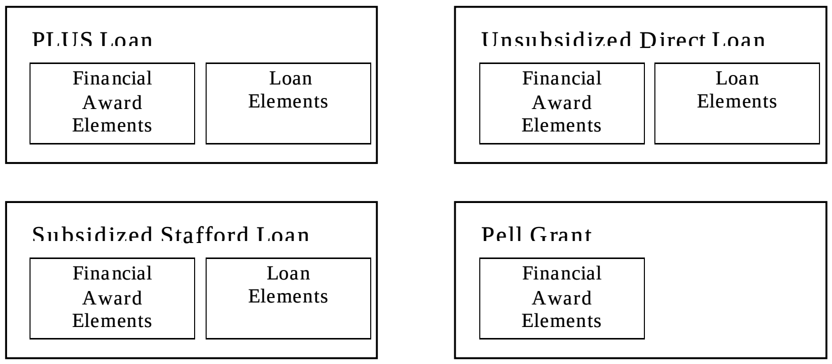


Figure 3.2 – Design by Composition Diagram

Example Subclassing Schema Snippet

```
<xs:group name="FinancialAwardGroup">
   <xs:sequence>
   ...
   </xs:sequence>
</xs:group>

<xs:group name="LoanGroup">
   <xs:sequence>
   ...
   </xs:sequence>
</xs:group>

<xs:complexType name="PLUSType">
   <xs:sequence>
         <xs:element name="PLUSElement1" type="xs:string"/>
         <xs:group ref="FinancialAwardGroup"/>
         <xs:group ref="LoanGroup"/>
         <xs:element name="PLUSElement2" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

There are several advantages for designing Schemas by composition. The first is that elements can be included in any order. For example, in the subclassing design pattern, when PLUS base extends LoanType, the elements from the LoanType complex type must be the first elements in the PLUS complex type. However, when including a group of elements, the group of elements can be referenced anywhere in the type definition. In addition, Schema designers can reference more than one group, as opposed to only being able to extend one type. This gives Schema designers greater flexibility with regard to how aggregate objects will be formed. The second is that the composition design pattern creates loose collections of elements that can be changed with fewer downstream considerations.

Derivations

Derivation by Extension

XML allows Schema designers to do more than simply reuse types and elements. It allows users to extend them. This is call derivation by extension. For example, a Schema designer can take a base type and add an additional element to a new type. This technique facilitates the reuse of type definitions. Assume that a Schema designer had specifications that required one object type identical to the "MovieType" example, and another object type that also had an element to store the language of the Movie.

MovieType Example

```
<xsd:complexType name=”MovieType”>
   <xsd:sequence>
         <xsd:element name="Title" type=”xsd:string”/>
         <xsd:element name="Director" type=”xsd:string”/>
         <xsd:element name="Genre" type=”xsd:string”/>
         <xsd:element name="ReleaseYear" type=”xsd:gYear”/>
   </xsd:sequence>
</xsd:complexType>
```

In order to derive by extension, a Schema designer would take a base type and extend it with whatever additional elements are necessary. As the example instance snippet shows, any new elements would be placed after all of the elements from the base type.

Example Derivation by Extension

```
<xsd:complexType name=”MovieLanguageType”>
   <xsd:extension base=”MovieType”>
         <xsd:sequence>
               <xsd:element name="Language" type=”xsd:string”/>
         </xsd:sequence>
   </xsd:extension>
</xsd:complexType>
```

Example Instance Snippet for Derivation by Extension

```
<Movie>
   <Title>Movie’s Title</Title>
   <Director>Joe Smith</Director>
   <Genre>Action</Genre>
   <ReleaseYear>2003</ReleaseYear>
   <Language>English</Language>
</Movie>
```

Derivation by Restriction

XML also provides for a technique that allows Schema designers the ability to limit a particular data definition. In the phone number example, the string datatype was restricted down to a pattern that only allowed a valid phone number format to be included in that tag.

Example Derivation by Restriction

```
<xsd:element name="PhoneNumber">
   <xsd:simpleType>
         <xsd:restriction base="xsd:string">
               <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
         </xsd:restriction>
</xsd:simpleType>
</xsd:elemnt>
```

This technique can also be applied to Complex Types. In the modified MovieType, the element title can be included an unlimited number of times (remember that the default for minOccurs and max Occurs is one for both). Also the Genre element is listed as optional.

Modified MovieType Example

```
<xsd:complexType name="MovieType">
<xsd:sequence>
         <xsd:element name="Title" type="xsd:string"
                      maxOccurs="unbounded"/>
   <xsd:element name="Director" type="xsd:string"/>
   <xsd:element name="Genre" type="xsd:string" minOccurs="0"/>
   <xsd:element name="ReleaseYear" type="xsd:gYear"/>
</xsd:sequence>
</xsd:complexType>
```

The following example restricts the modified MoveType so that the “Title” element can only be included once, and it removes the “Genre” element completely.

Example Derivation by Restriction ComplexType

```
<xsd:complexType name="MovieOneTitleNoGenreType">
   <xsd:complexContent>
         <xsd:restriction base="MovieType">
         <xsd:sequence>
         <xsd:element name="Title" type="xsd:string"/>
               <xsd:element name="Director" type="xsd:string"/>
               <xsd:element name="ReleaseYear" type="xsd:gYear"/>
         </xsd:sequence>
         </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

Extension by restriction allows a Schema designer to limit the number of occurrences of an element, and remove optional elements. This technique is limited when used with complex types because it requires a Schema designer to list elements over again. Because of this redundancy, PESC schema designers should not use derivation by restriction when creating complex types.

Limiting Derivations

Schema designers can also limit the ability to derive by restriction, derive by extension, or to derive by either restriction or extension.

The following example creates a complex type which can not be used in a derivation by restriction.

Limit Restriction

```
<xsd:complexType name="MovieType" type=”xsd:string” final="restriction">
```

The following example creates a complex type which can not be used in a derivation by extension.

Limit Extension

```
<xsd:complexType name="MovieType" type=”xsd:string” final="extension">
```

The following example creates a complex type which can not be used in a derivation by any method.

Limit Extension and Restriction

```
<xsd:complexType name="MovieType" type=”xsd:string” final="#all">
```

Redefines

The Standards Forum should not use the `redefine` option when defining its schemas. A schema `redefine` operation performs an implicit `include` operation. All of the components in the schema that are the object of the `redefine` are included in the schema performing the `redefine`. However, `redefine` takes things farther than `include` by allowing the schema performing the `redefine` to extend or restrict components in the redefined schema. Most likely this will not be necessary for the generic PESC definitions.

Use of `redefine`, however, might be advantageous for use by an organization that has additional requirements for a data item which fall outside the requirements defined in the PESC Schema. Like `include`, `redefine` can be used for any schema that does not have a `targetNamespace`. This allows an entity to `redefine` (i.e., `include`) a PESC component schema, but modify that schema with its own extensions/requirements.

3.7 Flexible Architectures

There are several techniques Schema designers can employ to make XML Schemas more flexible. The following techniques will be demonstrated in this section:

- Choice Groups
- Substitution Groups
- Abstract Types with Type Substitution

Choice Groups

In most of the previous Schema snippets complex types were built using a “sequence” of elements. It is also possible to build a complex type using “choice”.

Choice Groups Schema Snippet

```
<xsd:complexType name=”MovieType”>
   <xsd:choice>
         <xsd:element name="Title" type=”xsd:string”/>
         <xsd:element name="Director" type=”xsd:string”/>
         <xsd:element name="Genre" type=”xsd:string”/>
         <xsd:element name="ReleaseYear" type=”xsd:gYear”/>
   </xsd:choice>
</xsd:complexType>
```

The default for choice is that it contains one and only one of the elements included within a choice block. This means that only one “Title”, or one “Director”, or one “Genre”, or one “ReleaseYear” element could be included under an element based on the “MovieType”. However the defaults can be overridden.

Choice Groups Defaults Overridden Schema Snippet

```
<xsd:complexType name=”MovieType”>
   <xsd:choice minOccurs=”0” maxOccurs=”unbounded”>
         <xsd:element name="Title" type=”xsd:string”/>
         <xsd:element name="Director" type=”xsd:string”/>
         <xsd:element name="Genre" type=”xsd:string”/>
         <xsd:element name="ReleaseYear" type=”xsd:gYear”/>
   </xsd:choice>
</xsd:complexType>
```

This example would allow any number of the elements included with in the choice group to be included under an element based on the “MovieType”.

There are two considerations that Schema designers need to remember when choosing to use “choice”. These considerations will be contrasted with Substitution Groups.

- Central location of all valid elements means that the addition or removal of an element from a choice group will require an update to that choice group.
- Central location of all valid elements makes Schema documents easier to understand.

Substitution Groups

A Substitution Group allows a Schema designer to declare one element and use that element throughout a Schema. This is referred to as Element Substitution. Then a Schema designer can create an additional element that is in a substitution group with the original element. This allows the second element to substitute for the first element anywhere the first element is used.

Substitution Group Schema Snippet

```
<xsd:element name=”Genre” type=”xsd:string”/>
<xsd:element name=”Category” substitutionGroup=”Genre” type=”xsd:string”/>
<xsd:element name="Movie">
   <xsd:complexType>
         <xsd:sequence>
               <xsd:element name=”Title” type”xsd:string”/>
               <xsd:element ref="Genre"/>
```

```
            </xsd:sequence>
      /xsd:complexType>
   </xsd:element>
```

Substitution Group Instance Snippet #1

```
   <Movie>
      <Title>Movie’s Title</Title>
      <Genre>Action</Genre>
   </Movie>
```

Substitution Group Instance Snippet #2

```
   <Movie>
      <Title>Movie’s Title</Title>
      <Category>Action</Category>
   </Movie>
```

Because of the creation of the substitution group, both of the Instance Snippets are valid according to the Substitution Group Schema Snippet. In this example “Genre” is called the Head Element, and “Category” is declared to be an element that can replace “Genre”. It is important to note that an element only be in a substitution group with a head element, if the element joining a substitution group is of the same type as the head element, or derived from the same element as the derived type.

Schema designers also have the option to block Element Substitution.

Limit Element Substitution

```
<xsd:element name=”Genre” type=”xsd:string” block=”substitution”/>
```

There are two considerations that Schema designers need to remember when choosing to use Substitution Groups.

- Decentralized location of all valid elements allows for the addition or removal of a valid element without the remaining valid elements.
- Decentralized location of all valid elements makes Schema documents more difficult to understand.

Abstract Type and Type Substitution

Schema designers also have the ability to create an abstract complex type. Abstract complex types can be extended or restricted in the same manner as any non-abstract complex type can be extended or restricted. However, when an abstract complex type is included in another complex type, an instance document can include any non-abstract complex type that is based off of the original abstract complex type. The following example shows how to implement this.

Abstract Type and Type Substitution Snippet

```
      <xs:complexType name="AudioVisualType" abstract="true">
            <xs:sequence>
```

```
                  <xs:element name="Title" type="xs:string"/>
                  <xs:element name="Director" type="xs:string"/>
                  <xs:element name="Genre" type="xs:string"/>
                  <xs:element name="ReleaseYear" type="xs:gYear"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="MovieType">
            <xs:complexContent>
                  <xs:extension base="AudioVisualType"/>
            </xs:complexContent>
      </xs:complexType>
      <xs:complexType name="TelevisionType">
            <xs:complexContent>
                  <xs:extension base="AudioVisualType">
                        <xs:sequence>
                              <xs:element name="Season" type="xs:string"/>
                        </xs:sequence>
                  </xs:extension>
            </xs:complexContent>
      </xs:complexType>
      <xs:element name="AudioVisualLibrary">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element name="AudioVisual" type="AudioVisualType"
   maxOccurs="unbounded"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
```

Abstract Type and Type Substitution Instance Snippet

```
   <AudioVisualLibrary>
      <AudioVisual xsi:type="TelevisionType">
            <Title>Show's Title</Title>
            <Director>Joe Wilson</Director>
            <Genre>Comedy</Genre>
            <ReleaseYear>2002</ReleaseYear>
            <Season>3</Season>
      </AudioVisual>
      <AudioVisual xsi:type="MovieType">
            <Title>Movie's Title</Title>
            <Director>Joe Smith</Director>
            <Genre>Action</Genre>
            <ReleaseYear>2003</ReleaseYear>
      </AudioVisual>
   </AudioVisualLibrary>
```

In this example the abstract type “AudioVisualType” is substituted for by the “TelevisionType” and the “MovieType”. This method produces awkward implementation documents, because the “AudioVisual” element needs to have an attribute to qualify what type of non-abstract complex type will be used. The fact that the Schema implementer must qualify the element with an attribute describing the element type places an unnecessary burden on the XML implementer; hence, this method is not recommended for PESC use.

3.8 Null Values vs. Empty Strings

XML allows for three different types of non-value data submission. Application teams need to decide how to handle each of these scenarios, with either or both application logic and default values. However, there are some suggestions for how to handle each type of non-value data.

- Absent Element – Make no change to the target system's stored value.
- Empty Element – Change the target system's stored value to "Blank".
- Nil Element – Change the target system's stored value to "Null".

The following Schema will be used to demonstrate each of these scenarios.

Example Absent, Empty and Nillable Schema Snippet

```
<xsd:element name="X">
   <xsd:complexType>
         <xsd:sequence>
               <xsd:element name="Y" type="xsd:string" nillable="true"
               minOccurs="0"/>
         </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

This Schema allows for the following Instance Documents

Example Instance Snippet

```
<X><Y>Data</Y></X>
```

Example Absent Instance Snippet

```
<X></X>
```

Example Empty Instance Snippet

```
<X></Y></X> or <X><Y></Y></X> or <X><Y>  </Y></X>
```

Example Nillable Instance Snippet

```
<X><Y xsi:nill="true"></X>
```

3.9 Enumerations and Code Lists

When modeling data in XML, there are occasions when the value of an element or attribute can be constrained to a finite list. Several approaches to capturing and conveying this information can be applied:

- The list of values may be captured as an enumeration within a defined simple type;
- The list of values may be encoded and subsequently captured as an enumeration within a defined simple type;
- The list of values simply may be documented without implementation in XML Schema.

Regardless of the approach, schema designers will need to create the code list themselves or, preferably, base it on a pre-existing code list[2].

The following guidelines apply to the development and deployment of code lists:

- For any code list that is created and maintained by the Standards Forum, permitted values should be listed in the data dictionary and as documentation in document schemas.
- The code list may be implemented as an enumeration in a defined simple type – providing for the validation of codes within the schema itself – if the permitted values in the list change infrequently. In general, implementations should not be delayed by administrative and procedural delays in adding codes to schemas. If the potential for such delays exist due to the volatility of the code list, enumerations within a simple type should not be used and business applications should be expected to perform their own code value checking.
- For code lists that are created and maintained by organizations other than the Standards Forum, the Forum should determine whether or not schema validation is to be supported. The team should make this decision based on factors such as the stability of the code list, size of the code list, and copyright status. Schemas should not import or include schemas from other organizations for the purpose of code list validation.

Each Code List is made up of Code List Items, each of which has a name and a definition. The Code List Item Name will be the value referenced in the code list. The following conventions should be used when developing Code List Item Names.

- The ***Code List Item Name*** shall be unique within the Code List.
- The ***Code List Item Name*** should be extracted from the Code List Item Definition.
- ***Code List Item Names*** are case sensitive. The first letter of each concatenated word should be in uppercase, and the rest of the letters should be lowercase.
- The ***Code List Item Name*** shall be concise and shall not contain redundant words.
- The ***Code List Item Name*** shall be in singular form unless the concept itself is plural.
- The ***Code List Item Name*** shall not use non-letter characters unless required by language rules.
- The ***Code List Item Name*** shall only contain verbs, nouns and adjectives (i.e. no words like ***and, of, the***, etc.). This rule shall be applied to the English language, and may be applied to other languages as appropriate.
- Abbreviations and acronyms that are part of the ***Code List Item Name*** shall be expanded or explained in the definition. Abbreviations shall only be used if commonly known and context is not lost.

[2] The term “code list” is used throughout despite the fact that one- or two-word literal values are preferred to actual codes.

- ***Code List Item Names*** should be made up of whole words or industry-recognized fragments when text values are of reasonable length and are common to all participants. This same approach has been adopted by the Association of Retail Technology Standards (ARTS) .
- If a ***Code List Item Name*** can not be simplified to 3 words, schema designers may consider using short, two or three character codes as ***Code List Item Names***.

In the following example, a code list is created by creating an enumerated list of valid values. The valid values ***are Code List Item Names***. Each of the Code List Items will also need to have a matching definition

Enumeration Example

```
<xsd:element name="Genre">
      <xsd:simpleType>
            <xsd:restriction base="xsd:GenreType"/>
      </xsd:simpleType>
</xsd:element>

<xsd:simpleType name="GenreType">
      <xsd:restriction base="xsd:string">
            <xsd:enumeration value="Action"/>
            <xsd:enumeration value="Comedy"/>
            <xsd:enumeration value="Drama"/>
            <xsd:enumeration value="Mystery"/>
      </xsd:restriction>
</xsd:simpleType>
```

The following example demonstrates how a Schema designer can create a Code List based on the union of two sets of enumerated values. XML allows Schema designers to combine multiple Code Lists into one Code List. This is accomplished by creating a type that is a "union" of two of more existing types. In this example the "Genre" element would have the following valid values:

- Action
- Comedy
- Drama
- Mystery
- Satire
- BlackComedy
- ScienceFiction

Union Example

```
<xsd:simpleType name="AlternateGenreType">
      <xsd:restriction base="xsd:string">
            <xsd:enumeration value="Satire"/>
            <xsd:enumeration value="BlackComedy"/>
            <xsd:enumeration value="ScienceFiction"/>
      </xsd:restriction>
```

```
</xsd:simpleType>

<xsd:element name="Genre">
      <xsd:simpleType>
            <xsd:union memberTypes="xsd:GenreType xsd:AlternateGenreType "/>
      </xsd:simpleType>
</xsd:element>
```

4 PESC XML Schema Structure

PESC XML Schema Development shall be done in a tiered manner to obtain maximum reusability. The tiers correspond to the scoping of the elements, from general use across all systems, to specific use for a particular message. The three tiers for development are:

- Core Component Library
- Sector Library
- Message Specification

These tiers, or levels, are illustrated in the following diagram:

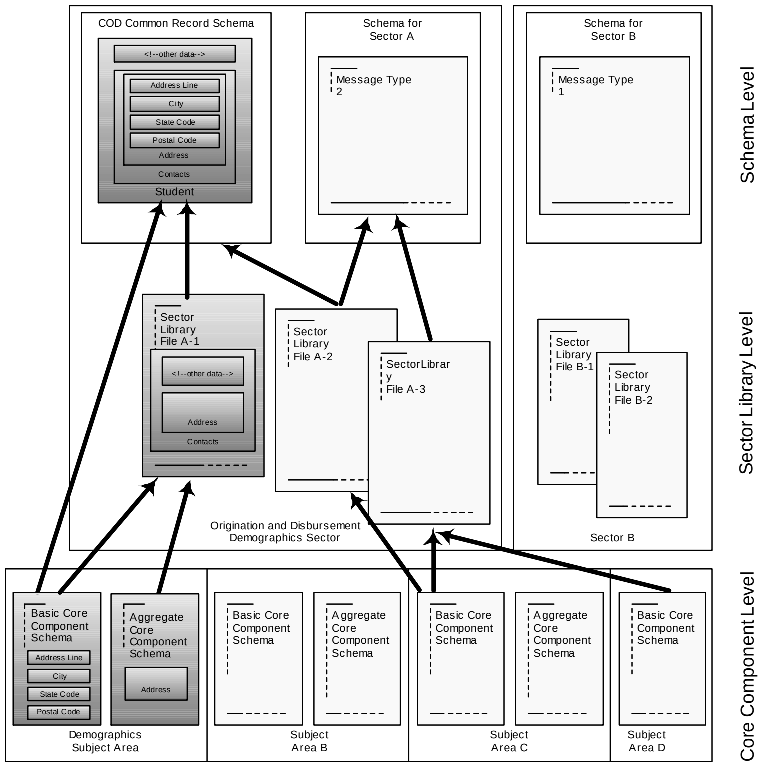


Figure 4.1 – Address Line Core Component Example

The diagram traces the use of a representative data entity, Address, through the levels of the XML modeling architecture, from its definition as a Core Component (bottom level), through its refinement (through the addition of context) into a specific component on the Sector Library Level, to its ultimate inclusion in a Message Specification Schema as part of a Contacts Block.

4.1 Naming Conventions

Overview

PESC's Core Component Naming Convention will be applied to three entities:

- Core Component Dictionary Entry Names
- XML Type Definition Names
- XML Tag Names

Each of these entities will have slight differences in their naming conventions; however they will be related to each other. The Core Component Dictionary Entry Name and the XML Type Definition Name will have a naming convention that will ensure each object has a unique name. XML Tag names will be simpler than the Dictionary Entry Names and XML Type Definition Names. This is because XML Tags will rely on their own name, and the context of the Aggregate the XML Tag is located within.

Naming Standards

PESC's Core Component Naming convention is based on the Core Component naming convention described in the UNCEFACT Core Component Technical Specification. The UNCEFACT Core Component Technical Specification's naming convention is based on the standards outlined ISO 11179 Part 5 – Naming and Identification Principles for Data Elements. The Core Component Technical Specification expands upon the ISO 11179 naming convention standards to include Core Component Types and Business Information Entities.

Core Component Dictionary Entry Name Convention

The name a Core Component is referenced by in the Core Component Dictionary is the Core Component Dictionary Entry Name. This Dictionary Entry Name is based on the summation of its named parts.

- Object Class Terms
- Property Terms
- Representation Terms
- Qualifier Terms

Name uniqueness is guaranteed because the name is based on the combination of terms that make each Core Component Unique.

PESC's Core Component Naming Convention uses the following rules to produce the Dictionary Entry Names for Core Components:

- The name of an ***Object Class*** shall be unique throughout the dictionary and may consist of more than one word. The name of a ***Property Term*** shall occur naturally in the definition and may consist of more than one word. A name of a ***Property Term*** shall be unique within the context of an ***Object Class*** but may be reused across different ***Object Classes***.

- If the name of the ***Property Term*** uses the same word as the ***Representation Term*** (or an equivalent word), this ***Property Term*** shall be removed from ***Dictionary Entry Name***. The ***Representation Term*** word in this case only will remain.
- The name of the ***Representation Term*** shall be one of the terms specified in the ***List of Representation Terms*** as included in this document.
- The name of the ***Representation Term*** shall not be truncated in the ***Dictionary Entry Name***.
- The ***Dictionary Entry Name*** shall be unique.
- The ***Dictionary Entry Name*** shall be extracted from the ***Core Component*** definition.
- The ***Dictionary Entry Name*** shall be concise and shall not contain consecutive redundant words.
- The ***Dictionary Entry Name*** and all its components shall be in singular form unless the concept itself is plural.
- The ***Dictionary Entry Name*** shall not use non-letter characters unless required by language rules.
- The ***Dictionary Entry Name*** shall only contain verbs, nouns and adjectives (i.e. no words like ***and, of, the***, etc.). This rule shall be applied to the English language, and may be applied to other languages as appropriate.
- Abbreviations and acronyms that are part of the ***Dictionary Entry Name*** shall be expanded or explained in the definition.
- The ***Dictionary Entry Name*** of a ***Basic Core Component*** shall consist of the name of an ***Object Class,*** the name of a ***Property Term*** and the name of a ***Representation Term***
- The components of a ***Dictionary Entry Name*** shall be separated by dots. The space character shall separate words in multi-word ***Object Classes*** and/or multiword ***Property Terms***. Every word shall start with a capital letter. To allow spell checking of the ***Directory Entry Names***' words, the dots after
- ***Object Class*** and ***Property Terms*** shall be followed by a space character.
- The ***Dictionary Entry Name*** of a ***Core Component Type*** shall consist of a meaningful type name followed by a dot, a space character, and the term ***Type***.
- The ***Dictionary Entry Name*** of an ***Aggregate Core Component*** shall consist of a meaningful ***Object Class*** followed by a dot, a space character, and the term ***Details***. The ***Object Class*** may consist of more than one word.
- If the ***Object Class*** of a ***Core Component*** is Global, this ***Object Class*** is not included in the ***Dictionary Entry Name.***

For example, if a Basic Core Component had the Object Class of "Person" and the Property Term of "Birth Date", then the Core Component Dictionary Entry Name would be "Person. Birth Date".

XML Types and XML Tags

Since PESC's XML Schemas will be based on Core Components, schema designers will be using types that have been previously defined. Both XML Type Definitions and XML Tag Definitions have names. PESC has a set of standards for creating both types of names from the Core Component Dictionary Entry Name. The following example shows the definition of an XML Type and an XML Tag which is based on the defined XML Type for the "Person. Birth Date" Core Component.

Example XML Type Definition

```
<xsd:simpleType name=”PersonBirthDateType“>
   <xsd:restriction base="xsd:date"/>
</xsd:simpleType>
```

Example XML Tag Definition

```
<xsd:element name="BirthDate" type="PersonBirthDateType"/>
```

Please note the following information on XML Tag Names and XML Type Definition Names:

- XML Tag Names and XML Type Definition Names are case sensitive. The first letter of each concatenated word should be in uppercase, and the rest of the letters should be lowercase.
- Length should be considered when creating these names (especially for XML Tag Names), but not to the point of sacrificing context. Well known abbreviations and acronyms may be used, but only if commonly known and context is not lost.
- Representation terms are not necessary in XML Tag Names or XML Type Definition Names, if the representation term for a specific Core Component is not one of the following:
  - Date
  - Indicator
  - Code

XML Type Definition Name Convention

During the creation of a Core Component, an XML type definition is created. This type definition is the XML representation of a Core Component. The name of this XML type definition is based on the Dictionary Entry Name. Three steps need to occur to create an XML Type Definition Name out of a Core Component Dictionary Entry Name.

1. Remove Space Characters and Dots from the Dictionary Entry Name.
2. If the Representation Term is anything other than Date, Code, or Indicator, remove the Representation Term from the modified Dictionary Entry Name.
3. Add the following characters to the end of the modified Dictionary Entry Name “Type”.

For example, if a Core Component Dictionary Entry Name was “Person. Birth Date”, then the XML Type Definition would be “PersonBirthDateType”.

XML Tag Name Convention

When designing an XML Schema, XML Type Definitions are given XML Tag Names. This tag name will provide the description of the data stored in an XML Document. It is critical that this name, as with the other names, be descriptive and concise. Abbreviations and acronyms may be used in XML Tag Names only if the abbreviation or acronyms is commonly known and context is not lost. The use of an abbreviation or acronym should not limit the descriptiveness of a tag name. There are two steps that need to happen to create a XML Tag Name out of a XML Type Definition Name.

1. Remove the characters "Type" from the end of the XML Type Definition Name.
2. If the XML Tag is going to be used within an Aggregate, then the Object Class can be removed from the front of the modified XML Type Definition Name. This can be done because the context for this element can be found from the Aggregate itself.

For example, if an XML Type Definition Name was "PersonBirthDateType", and the XML Tag is going to be used in a Student Block, then the XML Tag Name would be "BirthDate".

Schema Document Root Element Naming Convention

Schema document names (the root element of a schema) should be based on the business purpose of the document.

Example Application of Naming Convention

The following table shows an additional example of how Core Component metadata is combined to form a Dictionary Entry Name, an XML Type Definition Name, and an XML Tag Name.

Metadata Field	Metadata Value
Object Class	Address
Property Term	City
Representation Term	Text
Core Component Dictionary Entry Name	Address. City. Text
XML Type Name	AddressCityType
XML Tag Name	City

Table 4.1 – Example Application of Naming Convention

4.2 Namespace Conventions

Description

According to the World Wide Web Consortium (W3C), the purpose of XML namespaces is to "provide a simple method for qualifying element and attribute names used in Extensible Markup Language documents by associating them with namespaces identified in Uniform Resource Identifiers (URI) references."

In other words, XML Namespaces allow an organization to group their element names and attribute names in such a way to prevent conflicts in like named elements and attributes. This is done to manage element names and attribute names within a single organization and between multiple organizations.

PESC has developed an approach for the namespaces it will use, and it is defined in the following section.

Naming of Namespaces

The general approach for PESC namespaces is based on the guidelines in XML.gov, as well as the ASC X12 Reference Model for XML Design. The root of all namespaces PESC will use in its XML Libraries will begin with the organizational identifier:

urn:org:pesc:

After the organizational identifier, additional terms will be used to indicate the node of the classification scheme under which the document falls.

The terms for XML Namespaces are defined in accordance with the standard PESC XML Classification System outlined in Section 4.2. The last part of the namespace consists of the version indicator, as outlined in Section 4.3 The entire Core Component library namespace structure, along with a base version of 1.0.0, is listed as follows:

- I. Entity
  - § Address and Contact
    - urn:org:pesc:core:organization-demographic:v1.0.0
- II. Person
  - § Identification
    - urn:org:pesc:core:person-identification:v1.0.0
  - § Demographics
    - urn:org:pesc:core:person-demographic:v1.0.0
  - § Financial
    - urn:org:pesc:core:person-financial:v1.0.0
- III. School
  - § Demographics
    - urn:org:pesc:core:school-demographic:v1.0.0
  - § Participation
    - urn:org:pesc:core:school-participation:v1.0.0
- IV. Financial Partner (FP)
  - § Demographics
    - urn:org:pesc:core:financialpartner-demographic:v1.0.0
  - § Participation
    - urn:org:pesc:core:financialpartner-participation:v1.0.0
- V. Aid
  - § Loans and Grants
    - urn:org:pesc:core:aid-loanandgrant:v1.0.0
  - § Disbursements
    - urn:org:pesc:core:aid-disbursement:v1.0.0

Application of Namespaces

Description

Before discussing namespace usage specifically, consider a situation with four documents:

1. A Base Schema file, holding Core Component Definitions
2. A Sector Library Schema file, holding sector-level XML Schema objects

3. A Message Specification Schema file, holding the final XML message definition
4. An XML "Instance Document", holding a file that conforms to the Message Specification

Each of the first three documents contains XML Schema objects, which may be part of different namespaces. The elements need to be used in the XML Instance Document, with minimal complexity, but with having their unique locations (through namespace identification) preserved. This section describes exactly how each XML Schema document should be constructed to meet these goals.

There are several XML Schema mechanisms that work together to support namespace usage in XML documents. These mechanisms include:

- Core Component XML Schema:
  - Setting the targetNamespace attribute
  - Setting the elementFormDefault and attributeFormDefault attributes to indicate how elements will be qualified in an instance document using this schema
- Sector Library Schema
  - Using xs:import or xs:include, as appropriate, to include the Core Component XML Schema
  - Setting the targetNamespace attribute
  - Setting the elementFormDefault and attributeFormDefault attributes to indicate how elements will be qualified in an instance document using this schema
- Message Specification XML Schema
  - Using xs:import or xs:include, as appropriate, to include the Sector Library XML Schema
  - Setting the targetNamespace attribute
  - Setting the elementFormDefault and attributeFormDefault attributes to indicate how elements will be qualified in an instance document using this schema
- XML Instance Document
  - Setting the xmlns:[prefix] attribute to reference the Message Specification
  - Setting the schemaLocation attribute to reference the Message Specification
  - Setting the namespace prefix on the elements of the Instance Document

Approach

There are several different ways to use these different settings within a set of included/including schemas, and the instance document that refers to them. Rather than list all of the combinations possible, we will focus on the recommended approach for PESC XML development. The approach is as follows:

- Core Component XML Schema:
  - Set the targetNamespace attribute to a value appropriate for the Core Component Library, such as:

    `<xs:schema targetNamespace="urn:org:pesc:core:aid-loanandgrant:v1.0.0″`
  - Set the attributes elementFormDefault="unqualified" and attributeFormDefault="unqualified"
- Sector Library Schema
  - Use the `xs:import` directive to import the Core Component XML Schema
  - Set the `xmlns:[prefix]` attribute to reference the Core Component Library
  - Set the `targetNamespace` attribute to a value appropriate for the Sector Library, such as:
    `<xs:schema targetNamespace="urn:org:pesc:thissector:v1.0.0″`
  - Set the attributes elementFormDefault="unqualified" and attributeFormDefault="unqualified"
  - Create all element objects in this schema (do not use ones declared in the Core Component Library) - use only the `simpleType` and `complexType` objects from the Core Component Library, using the namespace prefix to qualify them.
- Message Specification XML Schema
  - Use the `xs:import` directive to import the Sector Library XML Schema
  - Set the `xmlns:[prefix]` attribute to reference the Sector Library
  - Set the `targetNamespace` attribute to a value appropriate for the Message Specification, such as:
    `<xs:schema targetNamespace="urn:org:pesc:thismessage:v1.0.0″`
  - Set the attributes elementFormDefault="unqualified" and attributeFormDefault="unqualified"
  - Create all top-level element objects in this schema (do not use ones declared in the Sector Library) - use only the `simpleType` and `complexType` objects from the Sector Library, using the namespace prefix to qualify them.
- XML Instance Document
  - Set the `xmlns:[prefix]` attribute to reference the Message Specification
  - Set the `schemaLocation` attribute to reference the Message Specification
  - Set the namespace prefix on the root element of the Instance Document to the one set for the `xmlns:[prefix]` attribute.

This approach will preserve unique traceability of types and elements through the different layers of the PESC XML Libraries, but require few steps on the part of the Instance Document authors in order to be compliant. A sample instance document would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<root:RootTagOfDocument xmlns:root="urn:org:pesc:thismessage:v1.0.0"
xmlns:sector="urn:org:pesc:thissector:v1.0.0"
xmlns:core="urn:org:pesc:aid:loangrant:v1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:org:pesc:thismessage:v1.0.0 MessageSpec.xsd">
    <DocumentID>2003-08-17T09:30:47-05:00BatchID</DocumentID>
<!-- all other elements do not need to be qualified except for the root --
>
</root:RootTagOfDocument>
```

5 Design Patterns for Core Components

The purpose of component design patterns is to show specific XML Schema coding solutions for some of the more common data structures that will be needed for XML message specifications. They are specific pieces of XML code, along with guidelines on usage, that can be directly implemented in an XML Schema document.

5.1 Entity Identifiers

Description

The Entity Identifiers Design Pattern addresses the issue that many systems have when communicating an identifier code for a particular entity, for example a school. A school can be identified according to one of several well-known identification systems, such as the OPEID, NCHELPID, or the IPEDS number. Each number has its own format, which the Schema should enforce.

All of these known systems are captured in the Core Component OrganizationIDGroup. It lists many different types under an xs:group compositor. The group allows for the choice of one and only one ID element to use for the identifier data. However, a particular message might want to restrict the list to only use one or two items from the choice group. In that case it would need to recreate this group in its own sector library with only the choice elements that are relevant.

Code

The following XML Schema code defines the base OrganizationIDGroup:

```
<xs:group name="OrganizationIDGroup">
   <xs:annotation>
         <xs:documentation>Allowable Entity IDs – Exclusive
choice</xs:documentation>
     </xs:annotation>
   <xs:choice>
         <xs:element name="OPEID" type="core:OPEIDType"/>
         <xs:element name="NCHELPID" type="core:NCHELPIDType"/>
         <xs:element name="IPEDS" type="core:IPEDSType"/>
         <xs:element name="ATP" type="core:ATPType"/>
         <xs:element name="FICE" type="core:FICEType"/>
         <xs:element name="ACT" type="core:ACTType"/>
         <xs:element name="CCD" type="core:CCDType"/>
         <xs:element name="CEEBACT" type="core:CEEBACTType"/>
         <xs:element name="CSIS" type="core:CSISType"/>
         <xs:element name="USIS" type="core:USISType"/>
         <xs:element name="ESIS" type="core:ESISType"/>
         <xs:element name="DUNS" type="core:ESISType"/>
   </xs:choice>
</xs:group>
```

These are the base type definitions for some of the OrganizationID types:

```
    <xs:simpleType name="OPEIDType">
        <xs:annotation>
```

```
            <xs:documentation>The unique identifier assigned by the Office
of Postsecondary Education for each data exchange
partner.</xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:minLength value="8"/>
            <xs:maxLength value="8"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="NCHELPIDType">
        <xs:restriction base="xs:string">
            <xs:minLength value="3"/>
            <xs:maxLength value="8"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="IPEDSType">
        <xs:annotation>
            <xs:documentation>The unique identifier assigned by National
Center for Education Statistics for each data exchange
partner.</xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:minLength value="6"/>
            <xs:maxLength value="6"/>
        </xs:restriction>
    </xs:simpleType>
```

The choice group can be restricted by copying the base Core Component definition into a Sector Library, and then using that group under a particular entity role tag (for example Source):

```
<xs:element name="Source">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="core:OrganizationIDGroup"/>
   <!–other fields -->
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:group name="OrganizationIDGroup">
  <xs:annotation>
    <xs:documentation>Allowable Entity Ids for this sector - Exclusive
choice</xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element name="OPEID" type="core:OPEIDType"/>
    <xs:element name="NCHELPID" type="core:NCHELPIDType"/>
    </xs:choice>
</xs:group>
```

Example

The following XML shows a document snipped that conforms to the Schema above:

```
<Source>
  <OPEID>12345678</OPEID>
</Source>
```

5.2 Person Identifiers

Description

The Person Identifiers Design Pattern has been developed to support the data requirements of the systems currently implementing an XML Schema for messages. The concepts for the Person Identifiers are as follows:

- The Person Identifiers Design Pattern incorporates four fields into a single block, called Index. The fields are the Social Security Number, Birth Date, First Name, and Last Name.
- All person-derived entities that serve as the core system data entity should use the Person Identifiers (Index block). This entity is usually the main Student block.
- Note that the Index block consists of Elements, rather than Attributes. This follows from the best practice that Elements should almost always be used instead of Attributes, because they allow for more flexible XML Schema designs.
- Only the SSN element is mandatory in the core pattern Schema for IndexType.
- The IndexType and its sub-elements are true PESC Core Components and are stored as such in the PESC XML Registry and Repository.

Code

The following is the XML Schema code that defines the Person Identifiers Pattern:

```
<xs:complexType name="IndexType">
  <xs:annotation>
    <xs:documentation>System identifier or system key</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="SSN" type="core:SSNType"/>
    <xs:element name="BirthDate" type="core:BirthDateType" minOccurs="0"/>
    <xs:element name="LastName" type="core:LastNameType" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

The following code shows a Student object definition that implements the Person Identifiers Pattern:

```
<xs:complexType name="PersonType">
    <xs:sequence>
        <xs:element name="Index" type="core:IndexType"/>
    <!–other person data fields -->
</xs:complexType>
```

Example

The following code illustrates an XML Block for a Student (of PersonType) that conforms to the XML Schema above:

```
<Student>
   <Index>
```

```
            <SSN>111223333</SSN>
            <BirthDate>1980-01-01</BirthDate>
            <LastName>Smith</LastName>
      </Index>
      <!—other information for the Student -->
   </Student>
```

5.3 *User-Defined Extensions*

Description

The User-Defined Extension Design Pattern is intended to address situations where an XML Schema message specification may have to carry sender-specific data that cannot be defined at the time the message specification is designed. The Schema has to allow for additional elements to be defined and used at a later date. The User-Defined Extensions Pattern serves as a placeholder for these to-be-defined fields. However, it can require that these fields are defined in a Schema by the organization that wants to use the extensions area.

Care should be taken not to use the user-defined extensions as a fall-back for doing appropriate research and design. It should only be used when in actuality, the organization defining the base Schema cannot define the additional elements that other organizations may need, and furthermore, is not interested in the data these other organizations want to exchange in the user-defined area.

Code

The following is an example of a small XML Schema document that implements a User-Defined Extensions Pattern:

```
   <?xml version="1.0" encoding="UTF-8"?>
   <xs:schema
      targetNamespace="urn:org:pesc:extensions-allowed"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:ExtAllow="urn:org:pesc:extensions-allowed"
      elementFormDefault="unqualified"
      attributeFormDefault="unqualified">
      <xs:element name="Document" type="ExtAllow:DocumentType"/>
      <xs:complexType name="DocumentType">
            <xs:sequence>
                  <xs:element name="DataField1" type="xs:string"/>
                  <xs:element name="DataField2" type="xs:string"/>
                  <xs:element name="DataField3" type="xs:string"/>
                  <xs:element name="UserDefinedExtensions"
   type="ExtAllow:UserDefinedExtensionsType" minOccurs="0"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="UserDefinedExtensionsType">
            <xs:sequence>
                  <xs:any namespace="##other" processContents="lax"/>
            </xs:sequence>
      </xs:complexType>
   </xs:schema>
```

A User-Defined Extension is accomplished with the use of the xs: any XML Schema construct. It allows the inclusion of content in an XML document that is not defined in that document's Schema. There are two attributes for this construct to discuss:

- The namespace attribute indicates whether the elements in the user-defined space can be taken from the original Schema ("##local"), if they have to have been defined in another Schema ("##other"), or can be taken from anywhere ("##any"). Since the point of allowing user-defined extensions is that the elements currently aren't known, "##other" is a logical choice; it requires the organization extending the Schema to design the extensions.
- The processContents attribute in the example above can be set to "none", to do no Schema validation; "lax", to do Schema validation if a Schema can be found; and "strict", to enforce Schema validation always (fail validation if no Schema is found). The value "lax" allows validation without bringing things to a halt if for some reason the Schema can't be found.

Next, a XML Schema document that defines fields to use in the User-Defined extensions area follows. Note the different targetNamespace that is defined.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   targetNamespace="urn:org:pesc:useextensions"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:UseExt="urn:org:pesc:useextensions"
   elementFormDefault="unqualified"
   attributeFormDefault="unqualified">
   <xs:element name="CustomData" type="UseExt:CustomDataType"/>
   <xs:complexType name="CustomDataType">
         <xs:sequence>
               <xs:element name="ContentString" type="xs:string"/>
               <xs:element name="ContentNumber" type="xs:integer"/>
               <xs:element name="ContentDate" type="xs:date"/>
         </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Example

Based on the XML Schema code defined above, a snippet of a sample document would appear as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtAllow:Document
   xmlns:ExtAllow="urn:org:pesc:extensions-allowed"
   xmlns:UseExt="urn:org:pesc:useextensions"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:org:pesc:extensions-allowed
BaseAllowUserDefinedExtensions.xsd urn:org:pesc:useextensions
UserDefinedExtensionsSchema.xsd">
   <DataField1>test1</DataField1>
   <DataField2>test2</DataField2>
   <DataField3>test3</DataField3>
```

```
      <UserDefinedExtensions>
            <UseExt:CustomData>
                  <ContentString>YourNameHere</ContentString>
                  <ContentNumber>6</ContentNumber>
                  <ContentDate>2003-01-01</ContentDate>
            </UseExt:CustomData>
      </UserDefinedExtensions>
   </ExtAllow:Document>
```

5.4 Name/Value Pairs

Description

The Name/Value Pair Design Pattern is intended to handle data that is structured as a group of names (or fields) and corresponding values. The key step in implementing this pattern is recognizing the appropriate situations where it should be used. Many pieces of data can be considered to be field/value pairs; in fact any tag in an XML document can be viewed as a field and value. For example, in ***an incorrect usage*** a name can be represented as:

```
   <TagName>FirstName</TagName>
   <TagValue>John</TagValue>
   <TagName>LastName</TagName>
   <TagValue>Smith</TagValue>
```

However, a whole document should not be constructed in this manner; in general specific tag names should be used.

The key to recognizing where to use a Name/Value Pair Pattern is to identify:

- That the data is related in some distinct manner
- That the set of fields and values is fairly flexible in two ways: Only a subset of the pairs is ever sent in a particular document, and the set of allowable fields may change fairly frequently.

Some types of data that meet these criteria would be Reject Fields, Assumed Fields, or Verified Fields. For example, Assumed Fields consist of the name of the field on which an assumption was made, and the value to which that field was assumed. These pairs are all related because they all represent assumptions. The list of assumed fields varies greatly from document to document; some documents might have many assumptions made, some very little.

Code

The following is an example of the XML Schema code required to implement a Name/Value Pair Pattern:

```
   <xs:element name="Assumptions" type="AssumptionsType" minOccurs="0"/>
   <xs:complexType name="AssumptionsType">
      <xs:sequence>
            <xs:element name="Assumption" type="AssumptionType"
   maxOccurs="24"/>
      </xs:sequence>
```

```
</xs:complexType>
<xs:complexType name="AssumptionType">
   <xs:sequence>
         <xs:element name="FieldName" type="FieldNameType"/>
         <xs:element name="FieldValue" type="FieldValueType"/>
   </xs:sequence>
</xs:complexType>
```

Example

Based on the XML Schema code defined above, a snippet of a sample document would appear as the following:

```
<Assumptions>
   <Assumption>
         <FieldName>LastName</FieldName>
         <FieldValue>Smith</FieldValue>
   </Assumption>
   <Assumption>
         <FieldName>MaritalStatus</FieldName>
         <FieldValue>Single</FieldValue>
   </Assumption>
</Assumptions>
```

6 XML Schema Development Methodology

6.1 Overview

Designing an XML Schema is like designing an application, and it should follow a design, build, test life cycle. The basics steps are as follows:

- Gather Requirements and Data Definitions
- Look up Data Definitions in Core Component Registry and Repository
- Assemble Schema
- Test Schema
- Deploy Schema

6.2 Gather Requirements and Data Definitions

Just like building any new system, building an XML Schema starts with good requirements. This includes understanding the business process that will be modeled in XML, knowing the different business scenarios, and what data is modeled in each of the scenarios. It is important for the Schema designer to understand what data will be needed and how the data is going to be used.

6.3 Look up Data Definitions in Core Component Registry and Repository

During this phase of Schema development, a Schema designer will search the XML Registry and Repository for the data definitions gathered during the requirements gathering phase. For each data element that a Schema designer searches for, there are three possibilities.

- Exact Match – The Schema designer will use the Core Component, as is, to model this piece of data in XML.
- Close Match – The Schema designer will evaluate if a modification can be made to the Core Component that will allow this core component to be used in this Schema. If a modification can be made to update a Core Component, then the Schema designer must go through the change process outlined in the PESC Policies and Procedures manual. If this change cannot be made, then the Schema designer must follow the Enhancement Request Management process to create a new Core Component.
- No Match – The Schema designer must go through the Enhancement Request Management process to create a new Core Component, and get it stored into the XML Registry and Repository.

Comment [MMB1]: Need to provide actual location in the document.

Metadata Essential for XML Syntax

To facilitate creation of schemas, the following metadata items should be recorded (these should not be considered a limit) in the data dictionary for each element.

- Simple Types
    - Element name
    - Data type (string, date, number, etc)
    - Cardinality rules

  - Element description
  - Element equivalence in other transaction(s)
  -

  Element facets such as minimum length, maximum length, minimum values, maximum values, or patterns depending on the data type.
  - Values of code elements
- Aggregate Items (Complex Types)
  - Element name
  - Element sequence
  - Cardinality rules
  - Element description
  - Element equivalence in other transaction(s)

Data Types

The following simplified list of datatypes should be used for core component analysis, instead of the full set supported by XML schemas. Each type has several optional attributes that may be specified, as needed, for a particular data item.

- ***Number*** - precision (number of decimal places), minimum value, maximum value
- ***String*** (as defined by the W3C in ***XML Schema Part 2: Datatypes***) - minimum length, maximum length, and pattern facets (such as NNN-NN-NNNN for Social Security Numbers). Patterns, if used, must be specified using a regular expression language as defined by the W3C in ***XML Schema Part 2: DataTypes Regular Expressions***. If a pattern facet is specified in the Core schema it may be modified by a Sector schema as long as that modification is a subset of the Core schema pattern. If an element contains a member of a list, all potential list values must be specified.
- NOTE: If a string item is specified as mandatory in an aggregate item, it is recommended to have a minimum length of 1.
- ***Date***
- ***Time***
- ***DateTime***
- ***Boolean*** - 0,1,true,false

When a data item is defined, it must be assigned a type from this set. The attributes listed should be used to place restrictions on the allowed values. If the attributes are not listed in the data item's definition, then there are no restrictions beyond the general restrictions implied by the datatype.

Aggregate Items

Specification of Aggregates

Aggregate data items are composed of two or more data items. For aggregates the following apply.

- The included elements must be specified in sequence. The core dictionary should specify the common elements.
- Sector dictionaries may restrict included elements, and may add additional elements.
- Cardinality (how many times an included element may occur in the aggregate) shall be specified for aggregates in the core dictionary. The widest common range of cardinality shall be expressed in the Core. The cardinality of elements within aggregates in the Core is defined as that which is most applicable to the widest range of uses, with a goal of minimizing the need for modification in sector or document schemas. The defaults in most cases will be 0..1 or 1..1).
- The cardinality shall be expressed as l..u where l is the lower number of occurrences and u is the upper number of occurrences. A wild card of "*" shall be used to indicate no upper limit. (For example, a cardinality of 1..1 means that the data item is mandatory in the aggregate and can occur only once. 0..1 means that the data item is optional, and can occur no more than once. 0..* means that it is optional and if it does occur there are no limits on how many times it can occur.)

  NOTE: It is recommended that judicious consideration be given before specifying an item in an aggregate as mandatory (minimum cardinality of 1).

Issues Concerning Aggregates

The following recommendations are made for addressing issues regarding aggregates.

- ***<u>Over-riding the cardinality of an item in an aggregate on a per document basis</u>***
  (***example***: a street address is mandatory in a reissue but is not mandatory in an adjustment.)
  It is recommended that this type of definition not be supported since it makes defining reusable aggregates more complex. One recommended approach is to define street address with a cardinality 0..2 in an "address" aggregate, but define address 1..1 in the reissue and 0..1 in the adjustment.
- ***<u>Conditional use of items in an aggregate</u>*** – As in the case of X12 EDI, these are the relational conditions often imposed on elements in segments.
  (***examples***: Use "a" or "b" but not both;
  if "a" then use "b", else use "c".)
  It is recommended that conditionals not be supported since it adds complexity to the analysis and construction of the schemas. Use of such conditional restrictions and edits, not being supported in the schemas should be the responsibility of the business applications that use the data.

Analysis Orientation

It is recommended that the data dictionary use the core components as "abstract" items or types rather than the full set of all particular items.

> (***example***: a general "entity" is defined rather than specifying "student", "lender", or "guarantor" separately.)

This approach enhances reusability and simplifies maintenance.

6.4 Assemble Schema

Once the Schema designer has located the entire set of core components needed to model this interface, the Schema designer is now ready to combine these core components into a Schema. The Schema designer should attempt to leverage existing sector libraries, and existing message specifications stored in the XLM Registry and Repository when building a new schema. These existing resources will save the Schema designer time, and limit mistakes. The Schema designer should follow the design patterns and best practices outlined in this document, XML Technical Reference and Usage Guideline.

6.5 Test Schema

A Schema should go through two phases of testing. The first phase of testing is to try and model each of the required business scenarios. If any issues are found in this phase of testing, then the Schema designer must correct them by either going back to the Core Component Registry and Repository, or correcting the assembly of the Schema. The second phase of testing is Inter System Testing. The Schema should be used during a System's IST to test out the interface.

6.6 Deploy Schema

Once a Schema has completed the testing phase of Schema development, it is ready to be deployed. A Schema's deployment process should follow the same deployment process that application code goes through.

7 XML Schema Object Management

7.1 Overview

The PESC XML Core Component Registry and Repository will store a large amount of XML artifacts, such as Core Components, Sector Libraries (and their constituent components), and complete Message Specifications. This vast amount of information objects needs to be appropriately managed in order to be effective and useful for PESC. There are two components to the management of these objects:

- Classification of Objects
- Versioning of Objects

The first component, classification, focuses on how to make XML artifacts easy to find and understand. The second component, versioning, focuses on how to manage change in an XML artifact, and across XML artifacts. The following sections describe these components in detail.

7.2 Classification of Objects

Overview

PESC will use a classification scheme to group Core Components into manageable sets. A classification scheme, also known as taxonomy, attempts to divide the objects across a given information domain into two or more distinct groups. There are various ways for determining the groups to use in a classification scheme. Generally speaking, groupings can be constructed along one or more facets. Examples of general facets include:

- Audiences / Actors
- Projects
- Services
- Locations
- Functions
- Disciplines
- Chronology

Data can be grouped by more than one of these classifications. For example, for fruit:

- Fruit Types: Citric and Non-Citric
- Fruit Colors: Orange, Red, Yellow, Green

In this manner the following fruit can be classified along the following facets:

- Banana: Non-Citric and Yellow / Green
- Orange: Citric and Orange

Only as many classification facets as necessary should be used to construct a classification scheme for an information domain. Thought should be given to the future elements that may be added to the information domain, as well as the current set of elements. The complete set of classifications is called a Classification Scheme.

PESC XML Registry Classification Scheme

As stated above, a classification scheme is a key component to the PESC XML Registry, since it will greatly increase the usability of the repository by making the elements far easier to access and understand. The classification scheme developed will be built specifically to maximize ease of access to PESC XML artifacts.

Analysis of the data elements reveals that the following classification scheme most naturally fits the data elements within the structures and characteristics of PESC:

I. Organization
    - § Demographics

II. Person
    - § Identification
    - § Demographics
    - § Financial

III. School
    - § Demographics
    - § Participation

IV. Financial Partner (FP)
    - § Demographics
    - § Participation

V. Aid
    - § Loans and Grants
    - § Disbursements

A brief review of this scheme: The top level of the classifications is based on a union of two facets within PESC, namely:

- The principal Actors involved in PESC's business processes: Entity, Person, School, Organization
- The central Object (or Service) provided by PESC: Aid

The classifications on the next level consist of different types of Characteristics of the Actors, and a further refinement of the different types of Services under the top level Service (Aid).

The Core Components will be managed along these classifications. Each Core Components library file is a collection of the components; likewise, the libraries will also be managed along these categories. The main technical mechanism for indication of groupings is the Namespace. Namespaces as they are defined along the categories, are discussed in Section 3.5.

7.3 Versioning of Objects

Overview

Each Core Component, Core Component Library, Sector Library, and XML Schema will have a Major Version, Minor Version, and Micro Version. A change in the Major Version will occur if the change in the object is not backward compatible. A change in the Minor Version occurs only when a change is made to the XML representation, but the change is backward compatible with all previous versions for that major version. A change in the Micro Version occurs when there is a change to something other than the XML representation of an object. Specific descriptions for each object type will explain what makes an object change backward compatible. All object types will follow the same representation pattern for versioning. Versions will be represented by three integers separated by periods.

Example Version Representation
Major Version: 2
Minor Version: 1
Micro Version: 3
Version Representation: 2.1.3

Each new component is created with a version of “1.0.0”. When a change is made to the Minor Version, the Micro Version resets to “0”. For example, if the original version was “2.1.3”, and there was an update to the Minor Version, then the new version would be “2.2.0”. If there is a change to the Major Version, then the Minor Version and the Micro Version are both reset to “0”. For example if the original version was “2.1.3”, and there was an update to the Major Version, then the new version would be “3.0.0”.

Versioning Techniques

XML Schemas can be versioned using the following techniques:

1. Change the (internal) Schema version attribute.
2. Create a Schema Version attribute on the root element.
3. Change the Schema’s targetNamespace
4. Change the name/location of the Schema.

Core Component Versioning

The Core Component Version is stored in a field in the Registry and Repository. The version is also noted in the Annotation section of the XML definition of a Core Component. The following types of changes to Core components represent modifications that will result in a Major Version change for a Core Component:

- Change to Base Type such that all previously accepted values would no longer be valid
- Change to a more Restrictive XML Representation such that all previously accepted values would no longer be valid.
- Change to the meaning of a Core Component Definition (A change can be made to the definition of a Core Component without changing the meaning, these types of changes would be referred to as descriptive changes.)

- Change to the XML Tag Name.
- Change to the Type Name.
- Change to any other part of the XML representation that makes the XML definition more restrictive.

The following represents a list of changes that would result in a Minor Version change. This list is not meant to be all encompassing. In general, a Minor Version change can be thought of as a change to the XML representation that does not invalidate any previously valid values.

- Add a new valid value.
- Add a new valid format.
- Change to a Base Type that does not remove previously valid values.

The following represents a list of changes that would result in a Micro Version change. This list is not meant to be all encompassing. In general, a Micro Version change can be thought of as a change to anything other than the XML representation of a Core component.

- Add or remove a “Related Term”.
- Change to the “Definition”.
- Change to the “Stability Indicator”.

It should be noted that a change to the “Status” of a Core Component does not result in a version change to the actual Core Component. However, this change will result in version changes in the Core Component Library as well as the Sector Library.

Core Component Library Versioning

The Core Component Library Version is stored in the following places:

- As a field in the Registry and Repository.
- As the last characters in the Name/Location of the Core Component library.
- As the last characters in the Target Namespace (targetNamespace="…v1.1.3").

The following types of changes represent modifications that will result in a Major Version change for a Core Component Library:

- Make a Major Version change to a Core Component in the Sector Library.
- Add a new required element in an Aggregate.
- Make an element in an aggregate that was previously optional required.
- Reduce the maximum number of times an element can occur in an aggregate.
- Remove an element from an Aggregate.
- Change the “Status” of a Core Component to “Withdrawn”.
- Change the Core Component Library’s Namespace.

The following represents a list of changes that would result in a Minor Version change to the Core Component Library. This list is not meant to be all encompassing. In general, a Minor

Version change can be thought of as a change to the XML representation that does not invalidate any previously valid structures or previously valid values.

- Make a Minor Version change to a Core Component.
- Add a new optional element to an aggregate.
- Make a previously required element in an aggregate optional.
- Increase the maximum number of times an element can occur in an aggregate.
- Change the “Status” of a Core Component to anything other than “Withdrawn”.

The following represents a list of changes that would result in a Micro Version change. This list is not meant to be all encompassing. In general, a Micro Version change can be thought of as a change to the anything other than the XML representation.

- Make a Micro change to a Core Component.

Sector Library Versioning

The Sector Library Version is stored in the following places:

- As a field in the Registry and Repository.
- As the last characters in the Name/Location of the Sector library.
- As the last characters in the Target Namespace (targetNamespace="…v1.1.3").

The following types of changes represent modifications that will result in a Major Version change for a Sector Library:

- Make a Major Version change to a Core Component Library reference by the Sector Library.
- Make a Major Version change to a Core Component in the Sector Library.
- Add a new required element in an Aggregate.
- Make an element in an aggregate that was previously optional required.
- Reduce the maximum number of times an element can occur in an aggregate.
- Remove an element from an Aggregate.
- Change the “Status” of a Core Component to “Withdrawn”.
- Change the Sector Library’s Namespace.

The following represents a listing of changes that would result in a Minor Version change to the Sector Library. This list is not meant to be all encompassing. In general, a Minor Version change can be thought of as a change to the XML representation that does not invalidate any previously valid structures or previously valid values.

- Make a Minor Version change to a Core Component Library reference by the Sector Library.
- Make a Minor Version change to a Core Component.
- Add a new optional element to an aggregate.
- Make a previously required element in an aggregate optional.
- Increase the maximum number of times an element can occur in an aggregate.

- Change the "Status" of a Core Component to anything other than "Withdrawn".

The following represents a listing of changes that would result in a Micro Version change. This list is not meant to be all encompassing. In general, a Micro Version change can be thought of as a change to the anything other than the XML representation.

- Make a Micro change to a Core Component Library referenced by the Sector Library.
- Make a Micro change to a Core Component.

XML Schema Message Specification Versioning

The XML Schema Version is stored in the following places:

- As a field in the Registry and Repository.
- As the last characters in the Name/Location of the Sector library.
- As the last characters in the Target Namespace (targetNamespace="…v1.1.3").

The following types of changes represent changes that will result in a Major Version change for an XML Schema:

- Make a Major Version change to a Sector Library reference by the XML Schema.
- Make a Major Version change to a Core Component Library reference by the XML Schema.
- Add a new required element in an Aggregate.
- Make an element in an aggregate that was previously optional required.
- Reduce the maximum number of times an element can occur in an aggregate.
- Remove an element from an Aggregate.
- Make a change to the XML Schema's Namespace.

The following represents a listing of changes that would result in a Minor Version change to the XML Schema. This list is not meant to be all encompassing. In general, a Minor Version change can be thought of as a change to the XML representation that does not invalidate any previously valid structures or previously valid values.

- Make a Minor Version change to a Sector Library reference by the XML Schema.
- Make a Minor Version change to a Core Component Library reference by the XML Schema.
- Add a new optional element to an aggregate.
- Make a previously required element in an aggregate optional.
- Increase the maximum number of times an element can occur in an aggregate.

The following represents a listing of changes that would result in a Micro Version change. This list is not meant to be all encompassing. In general, a Micro Version change can be thought of as a change to the anything other than the XML representation, such as a non-meaning-changing adjustment to the definition text, or an addition or change to the related terms.

- Make a Micro change to a Sector Library reference by the Sector Library.
- Make a Micro change to a Core Component Library reference by the Sector Library.

Appendix A: Revision History

DATE	SECTION/ PAGE	DESCRIPTION	REQUESTED BY	MADE BY
2/20/05	Whole Document	Initial Revision		M. Bolembach