Creating a NIEM IEPD, Part 3: Extend NIEM

Design an XML information exchange between U.S. government entities

Priscilla Walmsley

May 18, 2010
(First published March 09, 2010)

In the first two articles of this series, you learned to model a NIEM exchange, map it to the NIEM base model, and create a subset of the NIEM model for use in your IEPD. Now explore what to do about the parts of your model that do not map directly to NIEM, as you create extension and exchange schemas to define your custom types and properties.

View more content in this series

18 May 2010: Added links to Part 4 of this series in Introduction, Conclusion, and Resources sections.

12 May 2010 - *As a followup to reader comments, the author changed one XML schema document in one download file. The error makes the schema invalid.*
In Listing 1, added '/niem-core.xsd' to end of line 15 so it now reads:

```
<xsd:import schemaLocation="../../niem/niem-core/2.0/niem-core.xsd"
```

Also, replaced the download file, niem3schemas.zip, with an updated version.

Frequently used acronyms

- **CMT:** Component Mapping Template
- **IEPD:** Information Exchange Package Documentation
- **NDR:** NIEM Naming and Design Rules
- **NIEM:** National Information Exchange Model
- **SSGT:** NIEM Schema Subset Generation Tool
- **XML:** Extensible Markup Language

The National Information Exchange Model (NIEM) is large—over 6000 elements—but it most likely does not contain everything you want to include in an XML exchange. It is not intended to cover every possible scenario but rather the most common information building blocks. In

© Copyright IBM Corporation 2010

most Information Exchange Package Documentations (IEPDs) you create, you will need to write an *extension schema* that adds types and properties that are unique to your exchange. NIEM provides detailed guidelines for how to extend the model in a way that maximizes interoperability among NIEM IEPDs.

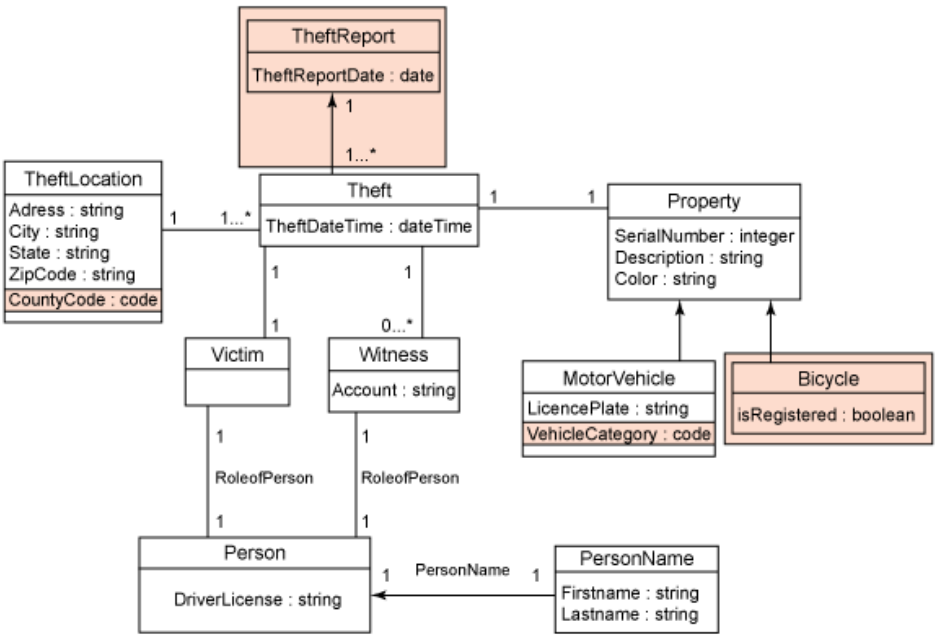
The NIEM model also does not define specific message types or structures for assembling all of the objects in an exchange. It is up to the creator of the IEPD to write an *exchange schema* to declare the root element and the basic structure of the messages.

Other articles in this series

- Part 1: Model your NIEM exchange
- Part 2: Map and subset NIEM
- Part 4: Assemble the IEPD

In Part 1 and Part 2 of this series, I worked through a simple example of a Theft Report (see Resources for links to the first two articles). Figure 1 shows the model I created for my exchange. The properties and types that I was unable to map to the base NIEM model include the `Bicycle` and `TheftReport` types plus the `IsRegistered`, `VehicleCategory`, and `CountyCode` properties. (View a text-only version of Figure 1.)

Figure 1. IEPD model showing extensions



In this case, NIEM met most of my needs. But I will need to create two new schemas:

- An extension schema to define the `Bicycle` type and the `IsRegistered`, `VehicleCategory`, and `CountyCode` properties
- An exchange schema to define the `TheftReport` type (because that is the root) and provide a structure that allows all of the other types to be included in the message

Writing NIEM schemas

NIEM extension and exchange schemas (as well as the generated subset schemas) are written in XML Schema. This article shows examples of NIEM-conformant schemas but does not provide a complete explanation of the XML Schema language. If you are a newcomer to schemas, I recommend the XML Schema Primer (see Resources) for a friendly introduction.

In addition to the constraints imposed by XML Schema, NIEM adds its own rules that are documented in the *NIEM Naming and Design Rules* (NDR) document (see Resources for a link). These rules cover, among other things, naming and documentation standards for NIEM components, the kinds of XML Schema constructs that are allowed and disallowed, and approved ways to use and extend NIEM. To be NIEM conformant, the schemas in an IEPD must follow the NDR rules.

Each schema document must have its own target namespace. For my example IEPD, I choose to use `http://www.datypic.com/theftreport/extension/1.0` (with the prefix `trext:`) as the namespace for the extension schema and `http://www.datypic.com/theftreport/exchange/1.0` (with the prefix `tr:`) for the exchange schema.

It is common practice to use a folder structure that reflects the namespace names. Inside the Theft Report IEPD, I will create folders called *extension* and *exchange* and within each one a subfolder named *1.0* in which I place the respective schema documents.

The beginning of a typical NIEM schema document—in this case, the extension schema for the Theft Report example—is in Listing 1.

Listing 1. Beginning of a NIEM-conformant schema

```
<xsd:schema targetNamespace="http://datypic.com/theftreport/extension/1.0"
            version="1.0"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:trext="http://datypic.com/theftreport/extension/1.0"
            xmlns:s="http://niem.gov/niem/structures/2.0"
            xmlns:nc="http://niem.gov/niem/niem-core/2.0"
            xmlns:niem-xsd="http://niem.gov/niem/proxy/xsd/2.0"
            xmlns:i="http://niem.gov/niem/appinfo/2.0">
  <xsd:annotation>
    <xsd:documentation>Theft Report extension schema</xsd:documentation>
    <xsd:appinfo>
      <i:ConformantIndicator>true</i:ConformantIndicator>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:import schemaLocation="../../niem/niem-core/2.0/niem-core.xsd"
              namespace="http://niem.gov/niem/niem-core/2.0"/>
  <xsd:import schemaLocation="../../niem/proxy/xsd/2.0/xsd.xsd"
              namespace="http://niem.gov/niem/proxy/xsd/2.0"/>
  <xsd:import schemaLocation="../../niem/structures/2.0/structures.xsd"
              namespace="http://niem.gov/niem/structures/2.0"/>
  <xsd:import schemaLocation="../../niem/appinfo/2.0/appinfo.xsd"
              namespace="http://niem.gov/niem/appinfo/2.0"/>
```

A NIEM schema document must contain an `xsd:annotation` element that has a description (in `xsd:documentation`) and an indicator that it is NIEM conformant (in `xsd:appinfo`).

As with any schema, it declares and imports all of the namespaces that it needs to reference directly. It is also required to import the appinfo schema on the last line of Listing 1, which declares the elements used inside the `xsd:appinfo` element.

Note: Complete extension and exchange schema documents that include all of the listings in this article are available in Downloads.

Extension schemas

Depending on the complexity of your IEPD, you might have one extension schema or many. Some IEPD developers choose to break extension schemas into multiple documents by subject area to allow them to reuse the schemas more granularly in various exchanges. Others choose to put components that might be versioned more frequently—for example, code lists—into a separate schema document.

For the Theft Report example, because it is simple, I choose to create one extension schema. After the beginning of the schema in Listing 1, I need to define types and declare elements for my custom components. There are several ways of extending NIEM, and I use a different method for each of my customizations.

Using substitution groups

Perhaps the easiest way to extend NIEM is through the use of *substitution groups,* which allow you to declare your own element and specify that it is substitutable for a NIEM element. This means that it can appear anywhere the NIEM element is allowed. You can use this method when there is a semantically equivalent element in the NIEM model, but it does not quite meet your needs. For example, in my model, I have a `CountyCode` property that is semantically equivalent to the NIEM abstract element `nc:LocationCounty` that appears inside an address. There are already two elements in the substitution group that are part of the NIEM core model, but they don't meet my needs: `nc:LocationCountyCode` uses a different code list, and `nc:LocationCountyName` is intended for a spelled-out name rather than a code. Instead, I declare a new element, `trext:LocationCountyCode`, that uses my own code list.

Listing 2 shows the element declaration for `trext:LocationCountyCode`. To indicate that it is substitutable for `nc:LocationCounty`, I use a `substitutionGroup` attribute.

Listing 2. Declaration of the trext:LocationCountyCode element and related types

```
<xsd:element name="LocationCountyCode" type="trext:CountyCodeType"
             substitutionGroup="nc:LocationCounty">
 <xsd:annotation>
  <xsd:documentation>A county code.</xsd:documentation>
 </xsd:annotation>
</xsd:element>

<xsd:simpleType name="CountyCodeSimpleType">
 <xsd:annotation>
  <xsd:documentation>A data type for a county code.</xsd:documentation>
  <xsd:appinfo>
    <i:Base i:namespace="http://niem.gov/niem/structures/2.0" i:name="Object"/>
  </xsd:appinfo>
```

```
 </xsd:annotation>
 <xsd:restriction base="xsd:token">
  <xsd:enumeration value="A">
    <xsd:annotation>
     <xsd:documentation>Ascot County</xsd:documentation>
    </xsd:annotation>
  </xsd:enumeration>
  <xsd:enumeration value="B">
    <xsd:annotation>
     <xsd:documentation>Burke County</xsd:documentation>
    </xsd:annotation>
  </xsd:enumeration>
  <xsd:enumeration value="C">
    <xsd:annotation>
     <xsd:documentation>Cross County</xsd:documentation>
    </xsd:annotation>
  </xsd:enumeration>
 </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="CountyCodeType">
 <xsd:annotation>
  <xsd:documentation>A data type for a county code.</xsd:documentation>
  <xsd:appinfo>
    <i:Base i:namespace="http://niem.gov/niem/structures/2.0" i:name="Object"/>
  </xsd:appinfo>
 </xsd:annotation>
 <xsd:simpleContent>
  <xsd:extension base="trext:CountyCodeSimpleType">
    <xsd:attributeGroup ref="s:SimpleObjectAttributeGroup"/>
  </xsd:extension>
 </xsd:simpleContent>
</xsd:complexType>
```

Listing 2 also shows the two type definitions that support the `trext:LocationCountyCode` element. First, a simple type is defined that has `xsd:enumeration` elements for each of the code values. Then, a complex type is defined based on the simple type. The complex type adds universal attributes like `s:id` that are allowed on all NIEM objects through a reference to `s:SimpleObjectAttributeGroup`.

Creating entirely new types

Another method of NIEM extension is to create a whole new type. In my model, `Bicycle` doesn't have an equivalent at all in the NIEM model, so I need to create a new element and a new corresponding complex type. Whenever you add a new type, you should consider whether it is a specialization of an existing NIEM type—for example, `nc:ActivityType`, `nc:PersonType`, or `nc:ItemType`. For `Bicycle`, I decide that it should be based on `nc:ConveyanceType`, because it represents a means of transportation, which is appropriate for a bicycle. Also, `nc:ConveyanceType` already has most of the properties I need, such as serial number and description.

As with the previous method of extension, I have to define both a new element—`trext:Bicycle`—and a type—`trext:BicycleType`. Listing 3 shows these definitions.

Listing 3. Declaration of the trext:Bicycle element and related type

```
<xsd:element name="Bicycle" type="trext:BicycleType">
 <xsd:annotation>
  <xsd:documentation>A bicycle.</xsd:documentation>
 </xsd:annotation>
</xsd:element>

<xsd:complexType name="BicycleType">
 <xsd:annotation>
  <xsd:documentation>A data type for a bicycle.</xsd:documentation>
 </xsd:annotation>
 <xsd:complexContent>
  <xsd:extension base="nc:ConveyanceType">
    <xsd:sequence>
     <xsd:element ref="trext:BicycleRegisteredIndicator" minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

The type definition for `trext:BicycleType` indicates that it extends `nc:ConveyanceType`. Note that if you are creating a type that is truly new—that is, not based on any concept already in NIEM—you must base your type on `s:ComplexObjectType`, which is the root of all complex types in NIEM.

In `trext:BicycleType`, I reference a `trext:BicycleRegisteredIndicator` element that I have to declare separately. All elements, attributes, and types in NIEM schemas are global, uniquely named, top-level components. Listing 4 shows the declaration of the `trext:BicycleRegisteredIndicator` element.

Listing 4. Declaration of the trext:BicycleRegisteredIndicator element

```
<xsd:element name="BicycleRegisteredIndicator" type="niem-xsd:boolean">
 <xsd:annotation>
  <xsd:documentation>Whether a bicycle is registered.</xsd:documentation>
 </xsd:annotation>
</xsd:element>
```

Unlike `trext:LocationCountyCode`, which had its own code list type, `trext:BicycleRegisteredIndicator` has a type that corresponds to one of the XML Schema built-in types, `boolean`. However, instead of giving it the built-in type `xsd:boolean`, I use `niem-xsd:boolean`. This complex type, defined in the "proxy" schema `xsd.xsd`, specifies that the element contains an `xsd:boolean` value but also allows the universal NIEM attributes like `s:id`.

Adding properties to existing types

Another extension situation is where you have a complex type that is semantically equivalent to a NIEM type but you need to alter or add to it in some way. In my model, the `MotorVehicle` class is equivalent to the NIEM `nc:VehicleType` but needs an extra property, `VehicleCategoryCode`. When doing the mapping, I looked at `nc:ItemCategoryText` as a possible mapping candidate but decided that it was too general. In fact, the `VehicleCategoryCode` property represents a classification of vehicles used for tax purposes, so I decide to call the element `trext:VehicleTaxClassCode`.

The required XML Schema definitions are similar to the `Bicycle` extension. Listing 5 shows how I declare a new element—`trext:Vehicle`—and a new complex type—`trext:VehicleType`—that extends `nc:VehicleType`.

Listing 5. Declaration of the trext:Vehicle element and related type

```
<xsd:element name="Vehicle" type="trext:VehicleType">
 <xsd:annotation>
  <xsd:documentation>A motor vehicle.</xsd:documentation>
 </xsd:annotation>
</xsd:element>

<xsd:complexType name="VehicleType">
 <xsd:annotation>
  <xsd:documentation>A data type for a motor vehicle.</xsd:documentation>
 </xsd:annotation>
 <xsd:complexContent>
  <xsd:extension base="nc:VehicleType">
    <xsd:sequence>
     <xsd:element ref="trext:VehicleTaxClassCode" minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Adding new objects with existing types

Sometimes, you are happy with the NIEM types, but you want to use names that are more specific or relevant to your exchange. In my model, I decided that the `Theft` class corresponded to the NIEM `nc:ActivityType`. However, I'm not completely satisfied with calling my element `nc:Activity`, because it is too general and not descriptive enough. In this case, I choose to declare a new element, named `trext:Theft`, but give it the existing type `nc:ActivityType` rather than define a new type. Listing 6 shows the element declaration.

Listing 6. Declaration of the trext:Theft element

```
<xsd:element name="Theft" type="nc:ActivityType">
 <xsd:annotation>
  <xsd:documentation>A theft incident.</xsd:documentation>
 </xsd:annotation>
</xsd:element>
```

Exchange schemas

Exchange schemas contain definitions that are unique to a message type or group of message types. This generally includes only the root element and its type and possibly some structural elements that form the basic framework of the message. Typically, an exchange schema is IEPD specific, while an extension schema might be shared across several IEPDs.

You are not required to have separate exchange and extension schemas; you can put all of your extensions in the same schema document. You can also have multiple exchange schemas in order to represent different message types or groups of different message types.

Exchange schemas follow all of the same rules described previously for extension schemas. For example, they must have their own target namespace and must have annotations.

In the Theft Report example, the exchange schema will contain the `tr:TheftReport` element, because that is the root, and its type. It will contain a `TheftReportDate`, which is shown in the model. But more importantly, the `tr:TheftReport` element will be what brings together all of the objects and associations defined in the exchange. The element and type for `TheftReport` are in Listing 7.

Listing 7. Declaration of the tr:TheftReport element and related type

```
<xsd:element name="TheftReport" type="tr:TheftReportType">
 <xsd:annotation>
  <xsd:documentation>A theft report.</xsd:documentation>
 </xsd:annotation>
</xsd:element>

<xsd:complexType name="TheftReportType">
 <xsd:annotation>
  <xsd:documentation>A data type for a theft report.</xsd:documentation>
  <xsd:appinfo>
    <i:Base i:namespace="http://niem.gov/niem/structures/2.0" i:name="Object"/>
  </xsd:appinfo>
 </xsd:annotation>
 <xsd:complexContent>
  <xsd:extension base="s:ComplexObjectType">
    <xsd:sequence>
     <xsd:element ref="tr:TheftReportDate" minOccurs="1" maxOccurs="1"/>
     <xsd:element ref="trext:Theft" minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="nc:ActivityConveyanceAssociation"
                                    minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="trext:Vehicle" minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="trext:Bicycle" minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="j:ActivityLocationAssociation"
                                    minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="nc:Location" minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="j:ActivityVictimAssociation"
                                    minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="j:Victim" minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="j:ActivityWitnessAssociation"
                                    minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="j:Witness" minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="nc:Person" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Note that the object and association elements are all siblings of each other. This is typical of a NIEM message, where associations between objects are separate components that reference the related objects through `s:ref` attributes.

A snippet of a message that shows the association between a theft and its location is in Listing 8. The objects `trext:Theft` and `nc:Location` are siblings, and each has an `s:id` attribute giving it a unique identifier. The association, `j:ActivityLocationAssociation`, is another sibling that links the two objects using child elements with `s:ref` attributes.

Listing 8. Sample instance showing association

```
<trext:Theft s:id="T1">
  <nc:ActivityDate>
      <nc:DateTime>2006-05-04T08:15:00</nc:DateTime>
```

```
  </nc:ActivityDate>
</trext:Theft>

<j:ActivityLocationAssociation>
   <nc:ActivityReference s:ref="T1"/>
   <nc:LocationReference s:ref="L1"/>
</j:ActivityLocationAssociation>

<nc:Location s:id="L1">
  <nc:LocationAddress>
   <nc:StructuredAddress>
     <nc:LocationStreet>
       <nc:StreetFullText>123 Main Street</nc:StreetFullText>
     </nc:LocationStreet>
     <nc:LocationCityName>Big City</nc:LocationCityName>
     <trext:LocationCountyCode>A</trext:LocationCountyCode>
     <nc:LocationStateUSPostalServiceCode>MI</nc:LocationStateUSPostalServiceCode>
     <nc:LocationPostalCode>49684</nc:LocationPostalCode>
   </nc:StructuredAddress>
  </nc:LocationAddress>
</nc:Location>
```

Another option for expressing relationships among objects is *containment,* where one object is the parent of another object. For example, it is hypothetically possible to create a brand new `TheftType` that contains within it a person and a location or a reference to a person or a location. However, this is not the recommended approach to using NIEM. Separating associations makes the delineation of objects clearer, reduces problems with recursion, and is more adapted to many-to-many relationships.

Naming and documenting NIEM components

You might have noticed some consistency in the names used in the examples. NIEM imposes certain rules for names:

- A name has an *object term* and a *property term,* and, if it is a simple element, a *representation term.* For example, in the name *BicycleRegisteredIndicator,Bicycle* is the object term, *Registered* is the property term, and *Indicator* is the representation term. It can also have optional qualifier terms.
- There is a specific set of approved representation terms, among which are *Indicator, Code, Date, Text, Value,* and *Quantity.*
- All names use camel case (uppercasing the first letter of each word) rather than separator characters.
- Attribute names start with a lowercase letter, while element and type names start with an uppercase letter.
- All types have the word *Type* at the end of their name.

There are also rules that govern the documentation of NIEM components. All schemas, elements, attributes, types, and enumerations must have definitions, and they must start with one of an approved set of beginning phrases, such as *A name of* or *A relationship.*

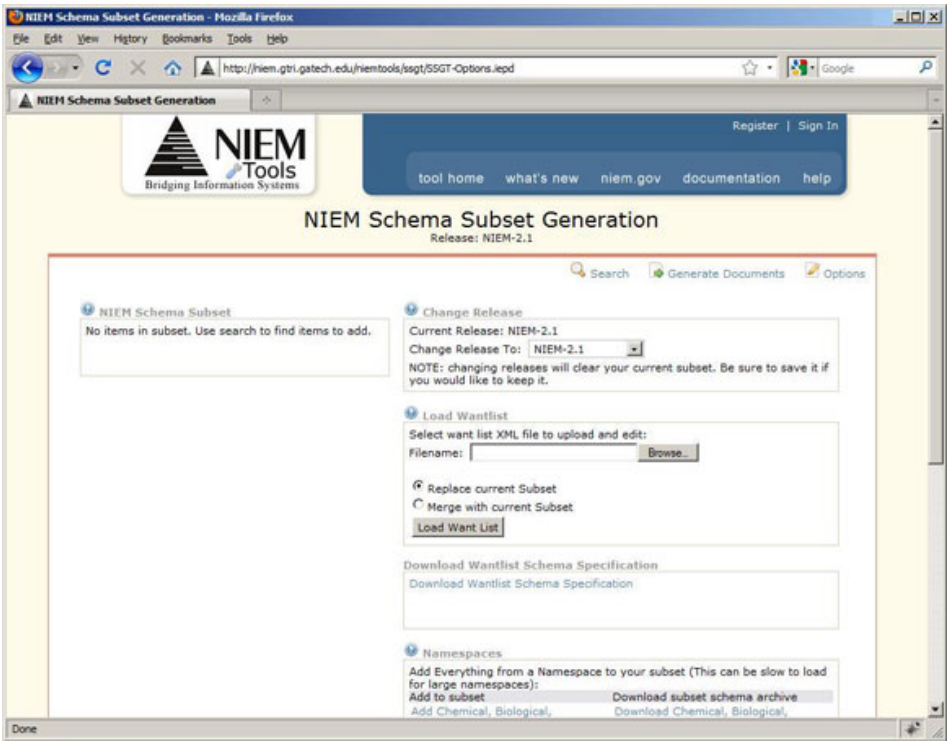
This is just a sampling of the rules; you can find a complete listing of the NIEM rules in the NDR.

Modifying the subset

As you build your extension and exchange schemas, you might need additional components from NIEM that you did not include in your subset. For example, `trext:BicycleRegisteredIndicator` is of type `niem-xsd:boolean`, a type that was not in my original subset.

Fortunately, it is easy to modify your subset using the Schema Subset Generation Tool (SSGT). From the main page of the SSGT (see Resources for a link), click **Options** in the upper right corner. This brings up the page in Figure 2. (View a larger version of Figure 2.)

Figure 2. SSGT Options page



In the section called **Load Wantlist**, fill in (or browse for) your wantlist.xml file name, and then click **Load Want List**. Doing so brings up your subset in the left pane. You can then click **Search** and use the right pane to search and add components to your NIEM subset. When you are done, regenerate the subset.

How to regenerate a subset

For details on how to regenerate a subset, see Generating your NIEM subset in Part 2 of this article series.

When working with extensions, you sometimes want to use the SSGT to look for types rather than properties. To find `niem-xsd:boolean`, I can't use the default search on properties, because that only finds element and attribute names, not type names. To specifically look for types, choose **Types** from the **Search for a** drop-down menu on the search page of the SSGT.

Documenting your mapping in the CMT

Be sure to document your extensions in the Component Mapping Template (CMT) that was described in Part 2 of this article series. At a minimum, you should fill in the XPath expressions for your extension elements. Table 1 shows the extensions I filled in for the `MotorVehicle` and `Bicycle` classes.

Table 1. XPath mapping for extensions

Source type	Source property	...	Ext?	XPath
`MotorVehicle`		...	Y	`trext:Vehicle`
`MotorVehicle`	`LicensePlate`	...	Y	`trext:Vehicle/ nc:ConveyanceRegistrationPlateIdentification/ nc:IdentificationID`
`MotorVehicle`	`VehicleCategory`	...	Y	`trext:Vehicle/ trext:VehicleTaxClassCode`
`Bicycle`		...	Y	`trext:Bicycle`
`Bicycle`	`IsRegistered`	...	Y	`trext:Bicycle/ trext:BicycleRegisteredIndicator`

Some NIEM practitioners create more formal CMTs that have separate columns indicating the kind of extension, the base types and elements, and the level of semantic alignment. For my CMT, I chose to take a looser approach to defining the dependencies by including this information in a **Comments** column. The final Theft Report CMT is available from Downloads.

Conclusion

Other articles in this series

- Part 1: Model your NIEM exchange
- Part 2: Map and subset NIEM
- Part 4: Assemble the IEPD

In this article, I described the process of extending NIEM. I explained the role of extension and exchange schemas and showed the various methods of adding new elements and types based on NIEM components. The majority of the work in creating a NIEM IEPD is now complete. Part 4 will describe the last and final step, assembling the IEPD. I will take you through the various artifacts that make up a IEPD. The result will be a complete, NIEM-conformant Theft Report IEPD.

Downloadable resources

Description	Name	Size
Component Mapping Template (CMT)	niem3mapping.zip	26KB
NIEM Exchange, Extension and Subset Schemas	niem3schemas.zip	18KB

© Copyright IBM Corporation 2010
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)