XML Schema Infoset Model, Part 1

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

About this tutorial

The first of a two-part series, this tutorial gives you the building blocks you need to set up and work with the XML Schema Infoset Model. In this tutorial, you will learn:

- How to set up the development environment
- How to use the XML Schema Infoset Model classes
- How to load and create XML Schema models
- How to link XML Schema models together

This tutorial is for developers who are familiar with Java, XML, and XML Schema, and who are interested in combining these technologies using the XML Schema Infoset Model. You should therefore understand how to write Java code and understand how XML Schemas work. Some understanding of the Unified Modeling Language (UML) is helpful but not required. You can get an introduction to XML Schema fundamentals in the Resources on page 28 at the end of this tutorial, and an introduction to UML basics in How to read UML class diagrams on page 9 .

About the author

Dave Spriet is a software developer for the IBM Toronto Laboratory. He develops tools for the WebSphere Business Integration Message Broker (WBIMB) product. He specializes in object-oriented technologies, XML, XML Schema, middleware, and UML Modeling. He is also a committer for the *XML Schema Infoset Model* component. Dave has a Bachelor of Science degree (Honors) in Computer Science and Statistics from McMaster University. He welcomes any comments; please direct them to *spriet@ca.ibm.com*.

Section 2. XML Schema fundamentals

What is an XML schema?

An *XML schema*, like a DTD (Document Type Definition), defines the legal building blocks of an XML document. More and more companies are starting to use XML schemas to describe their documents shared between business partners because of a schema's wide range of features:

- Schemas define elements that can appear in a document
- Schemas define attributes that can appear in a document
- Schemas define which elements are child elements
- Schemas define the order of child elements
- Schemas define the number of child elements
- Schemas define whether an element is empty or can include text
- Schemas define data types for elements and attributes
- Schemas define default and fixed values for elements and attributes

So, are XML schemas the successors of DTDs? I think they will be soon, for the following reasons:

- Schemas are extensible to future additions
- Schemas are richer and more useful than DTDs
- Schemas are written in XML
- Schemas support data types
- Schemas support namespaces

Several XML Schema proposals exist, but the *XML Schema Infoset Model* is 100% compatible with the W3C XML Schema recommendation V 1.0, 2 May 2001. This version is described in: *XML Schema Part 0: Primer*, *XML Schema Part 1: Structures*, and *XML Schema Part 2: Datatypes*.

What is the XML Schema Infoset Model?

The *XML Schema Infoset Model* is a library for use with any code that examines, creates, or modifies XML Schemas. These XML Schemas can be stand-alone schemas or part of other artifacts, such as XForms or WSDL documents. The library provides an API for manipulating the components of an XML Schema as described by the W3C XML Schema specifications, as well as an API for manipulating the DOM-accessible representation of XML Schema as a series of XML documents, and for keeping these representations in agreement as schemas are modified.

This library includes APIs to serialize and deserialize XML Schema documents, and to do

integrity checking of XML Schemas. This integrity checking looks for things like element references that reference non-existent global elements invalid maximum values.

Section 3. Setting up the development environment

Setup overview

First familiarize yourself with the *Eclipse* environment. The main areas to focus on are:

- The Workbench
- The Workspace
- Perspectives usage
- Eclipse plug-ins
- Team support

Now we'll download and install the Eclipse Modeling Framework and the XML Schema components, and verify the installation.

Downloading and installing the EMF and XML Schema components

1. **Java Runtime Environment V 1.3**. Eclipse does not include a Java Runtime Environment (JRE), so you will need a 1.3-level Java runtime or Java Development Kit (JDK) installed on your machine in order to run Eclipse. I recommend using the IBM or the Sun JRE/JDK:
   - *IBM developer kit* (http://www.ibm.com/developerworks/java/jdk/)
   - *Sun developer kit* (http://java.sun.com/j2se/1.3/download.html)

2. **Eclipse Base V2.1.1**. After you read and understand the *Eclipse.org Software User Agreement*, then download the Eclipse base. For this tutorial, we are using the latest release, which is 2.1.1 at the time of writing, but almost any version will do. Download and unzip a copy of the *Eclipse base V2.1.1.*. Throughout this document we will refer to **%ECLIPSE_BASE%**, which is the base directory in which you unzipped the Eclipse base V2.1.1.

3. **Eclipse Modeling Framework, V1.1.0, Build 20030620_1105VL**. Next, download the Eclipse Modeling Framework component, which we will call **EMF** throughout this tutorial. For this tutorial, we are using version 1.1.0, Build 20030620_1105VL. You can either download EMF from the *EMF* Web site or directly download EMF Version 1.1.0 from the following links:
   - *EMF Runtime*
   - *EMF Documentation*
   - *EMF Source*

   After you have downloaded EMF V1.1.0, you should unzip the three zip files into the %ECLIPSE_BASE%\eclipse directory. Make sure that the directories line up so that the EMF plug-ins go into the %ECLIPSE_BASE%\eclipse\plugins directory.

As an alternative, you can keep these additional components separated from the base by unzipping the three zip files into a subdirectory and then using an Eclipse link file to point to where the additional plug-ins are. Remember that EMF and XML Schema components are just Eclipse plug-ins; they work the same way as any other plug-ins in Eclipse. To keep these other plug-ins separated from the base, you need to create a "links" directory directly under the %ECLIPSE_BASE%\eclipse directory. In the links directory, add a file that contains the following:

- Filename: org.eclipse.emf.link
- Contents: path=C:\\Dev\\eclipse\\emf110

NOTE: Remember that on windows you have to escape the backslashes in this file.

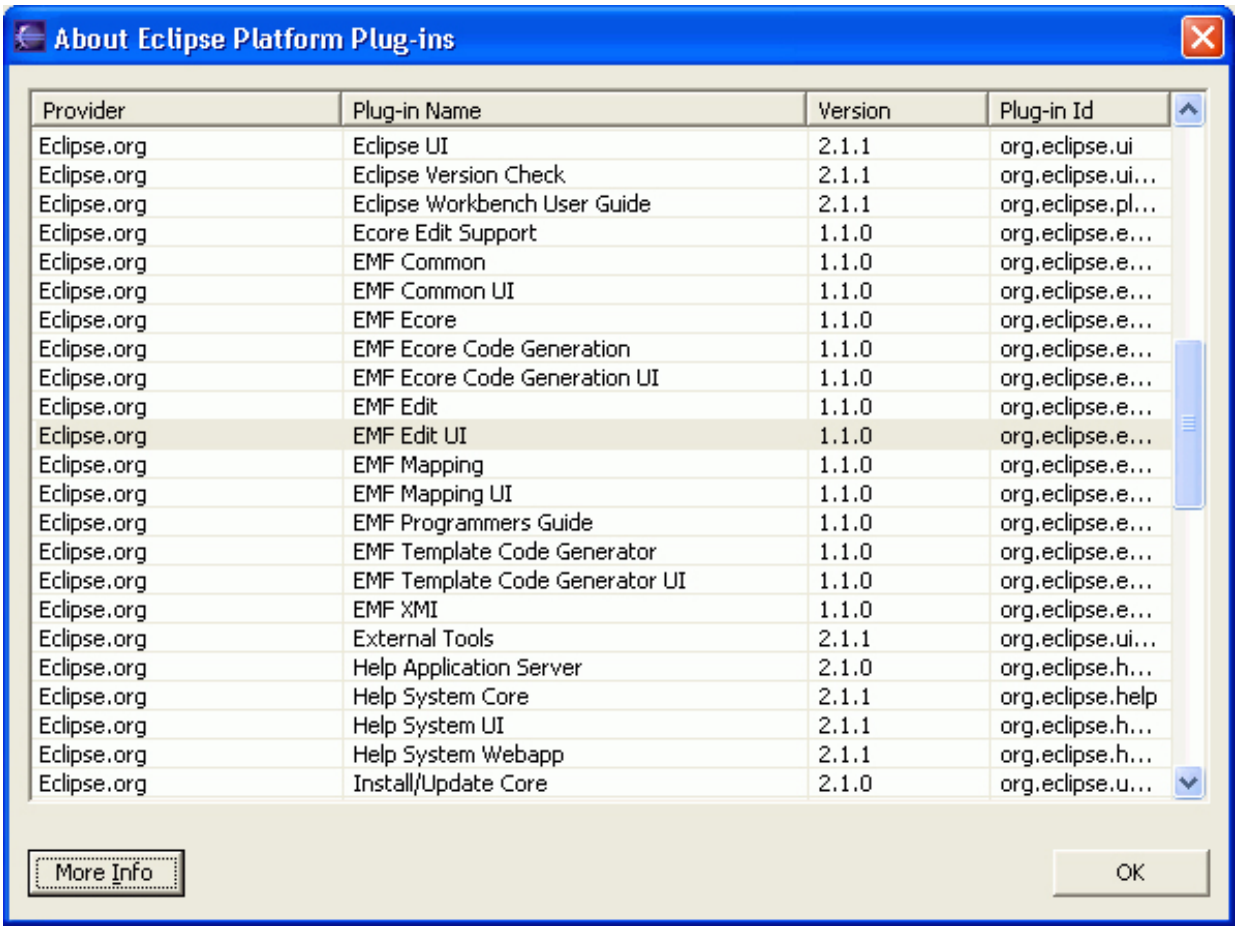
4. **XML Schema Infoset Model, V1.1.0, Build 20030620_1105VL**. Next, download the XML Schema Infoset component, which we will call **XML Schema model** throughout this tutorial. For this tutorial, we are using version 1.1.0, Build 20030620_1105VL. You can download XML Schema model from the *XML Schema Infoset Model* Web site or you can directly download XML Schema model Version 1.1.0 from the following links:
   - *XSD Runtime*
   - *XSD Documentation*
   - *XSD Source*

   After you have downloaded the XML Schema plug-ins, you can do the same thing that you did with the EMF plug-ins. Either unzip them into the %ECLIPSE_BASE%\eclipse directory or point to them using an eclipse link file.

Note: When using the EMF and XML Schema components, remember that the build date and build level for the XML Schema Runtime must match that of the EMF Runtime that you plan use. For example, you could use the XSD Runtime (xsd_1.1.0_20030620_1105VL.zip) with the EMF Runtime (emf_1.1.0_20030620_1105VL.zip), which is what we are using in this tutorial.

Verifying the installation

To verify that you have installed the EMF and the XML Schema components correctly, launch the %ECLIPSE_BASE%\eclipse\eclipse.exe executable. After the Eclipse Workbench comes up, select **Help > About Eclipse Platform**. Then click the "Plug-in Details" button. You should see a dialog like the one below that shows all of the plug-ins that are or could be loaded within that Eclipse environment.



If you installed EMF and the XML Schema components correctly, you should see these plug-ins:

Provider	Plug-in name	Version	Plug-in Id
Eclipse.org	ECore Edit Support	1.1.0	org.eclipse.emf.e
Eclipse.org	EMF Common	1.1.0	org.eclipse.emf.c
Eclipse.org	EMF Common UI	1.1.0	org.eclipse.emf.c
Eclipse.org	EMF ECore	1.1.0	org.eclipse.emf.e
Eclipse.org	EMF ECore Code Generation	1.1.0	org.eclipse.emf.c
Eclipse.org	EMF ECore Code Generation UI	1.1.0	org.eclipse.emf.c
Eclipse.org	EMF Edit	1.1.0	org.eclipse.emf.e
Eclipse.org	EMF Edit UI	1.1.0	org.eclipse.emf.e
Eclipse.org	EMF Mapping	1.1.0	org.eclipse.emf.m

Eclipse.org	EMF Mapping UI	1.1.0	org.eclipse.emf.m
Eclipse.org	EMF Programmers Guide	1.1.0	org.eclipse.emf.d
Eclipse.org	EMF Template Code Generator	1.1.0	org.eclipse.emf.c
Eclipse.org	EMF Template Code Generator UI	1.1.0	org.eclipse.emf.c
Eclipse.org	EMF XMI	1.1.0	org.eclipse.emf.e
Eclipse.org	XML Schema Edit Framework	1.1.0	org.eclipse.xsd.e
Eclipse.org	XML Schema Editor	1.1.0	org.eclipse.xsd.e
Eclipse.org	XML Schema Infoset Model	1.1.0	org.eclipse.xsd
Eclipse.org	XML Schema Infoset Model Source	1.1.0	org.eclipse.xsd.s

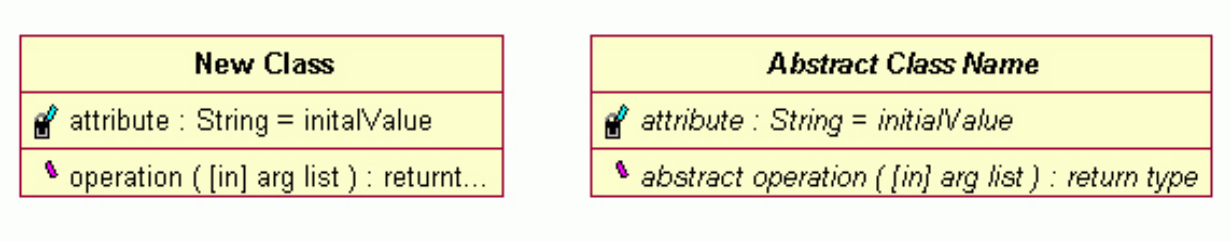
Section 4. Using XML Schema Infoset Model classes

How to read UML class diagrams

Class diagrams

Unified Modeling Language (UML) has become the developer's choice of tools for object-oriented design. Many types of diagrams are defined by UML, but here we'll only cover class diagrams and associations between classes. This tutorial will use IBM Rational XDE to display the different diagrams.

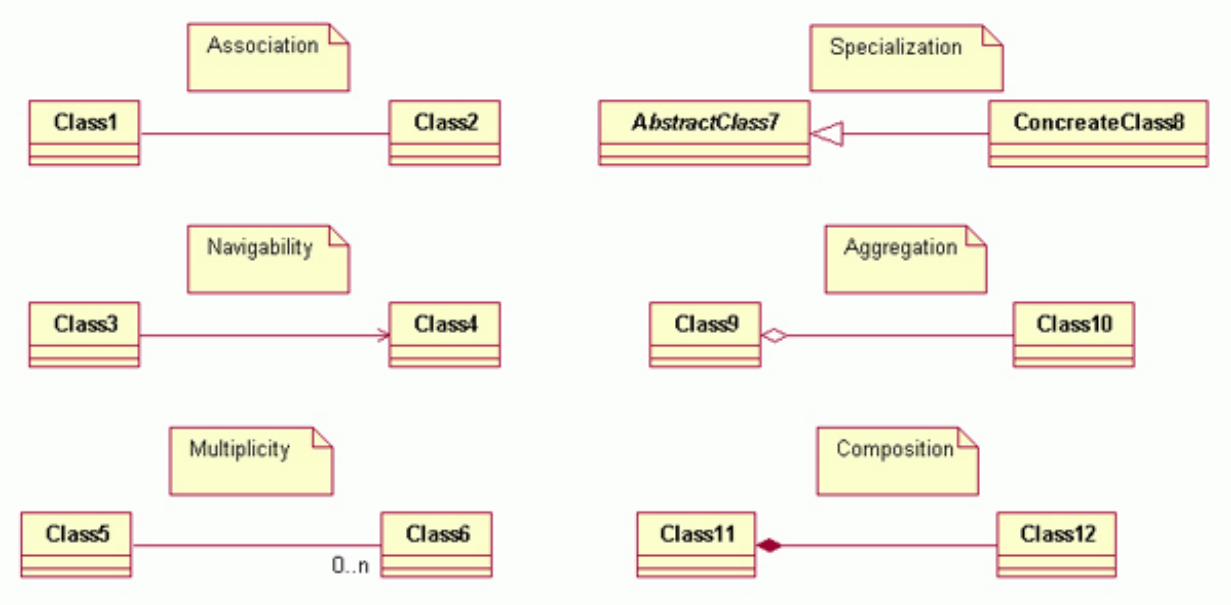
In class diagrams, classes are represented as boxes with three compartments:



The top section in a Class object contains the class name; if the class is abstract, then the name is italicized. The middle section contains the class attributes. The bottom section contains the class methods or operations. Like the class name, if a method is abstract, its name is italicized.

Associations between classes

Any interaction between classes is shown by a line drawn between the classes. A simple line indicates an association. The relationship between the classes can be modified to provide information about the association. The following diagram show the different types of associations that you will need to understand:



Understanding the XML Schema Infoset model diagrams

The next several panels show a set of UML diagrams that represent the XML Schema Infoset model, using the following notation:

- Each relationship or attribute Xyz, which we will refer to as a feature Xyz in a diagram, corresponds to a `getXyz()` method in Java for accessing that feature.
- If the feature is many-valued (in other words, (0..*)), the `getXyz()` method returns a *java.util.List* interface, which can be modified directly using standard java.util.List API.
- Otherwise, if the feature is single-valued, then a corresponding `setXyz()` method is available for modification.

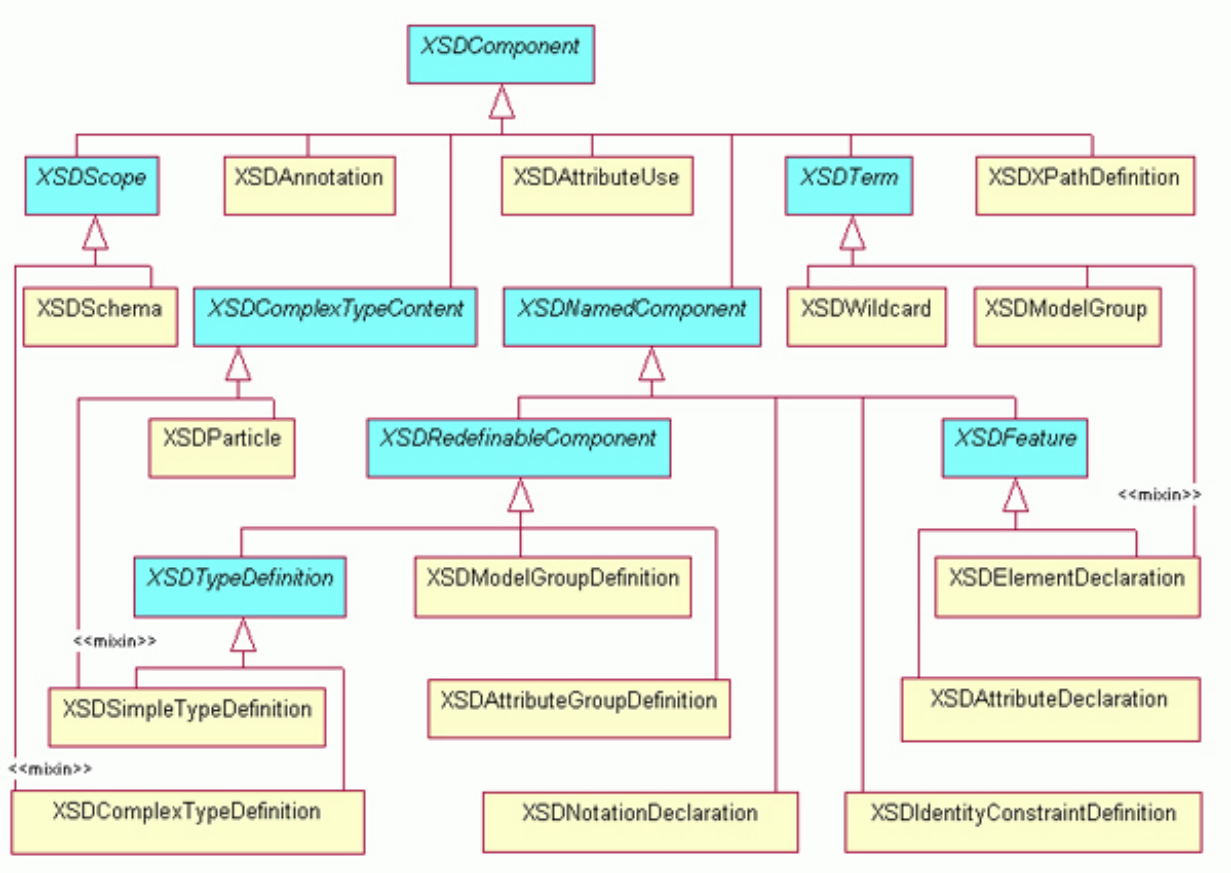
A relationship with a black diamond corresponds to a *containment relationship*. These form the basis for the concrete tree structure of the XML Schema Infoset model.

A relationship with a white diamond corresponds to a *shared pseudo-containment relationship* . It is used only for documentation purposes since it is not logically different from a relationship with no diamond. In general, the white diamond relationships represent relationships defined directly in the *XML Schema specification*. They are typically computed from other relations **and hence should not be modified directly**. The exception to this rule are relations involving types. These consist of base-type, member-type, item-type, and element- or attribute-type.

Note: The diagrams on the next several panels can also be found in the org.eclipse.xsd/src/model/XSD.mdl model file.

Component hierarchy

The abstract XML Schema components, as described in *XML Schema Part 1: Structures* of the Standard, are related according to the following hierarchy:



The figure above displays the hierarchy of the XML Schema Infoset model. The classes in light blue represent abstract classes, whereas the classes in light orange represent the concrete classes. Here is a short description of some of the classes that you will need to understand:

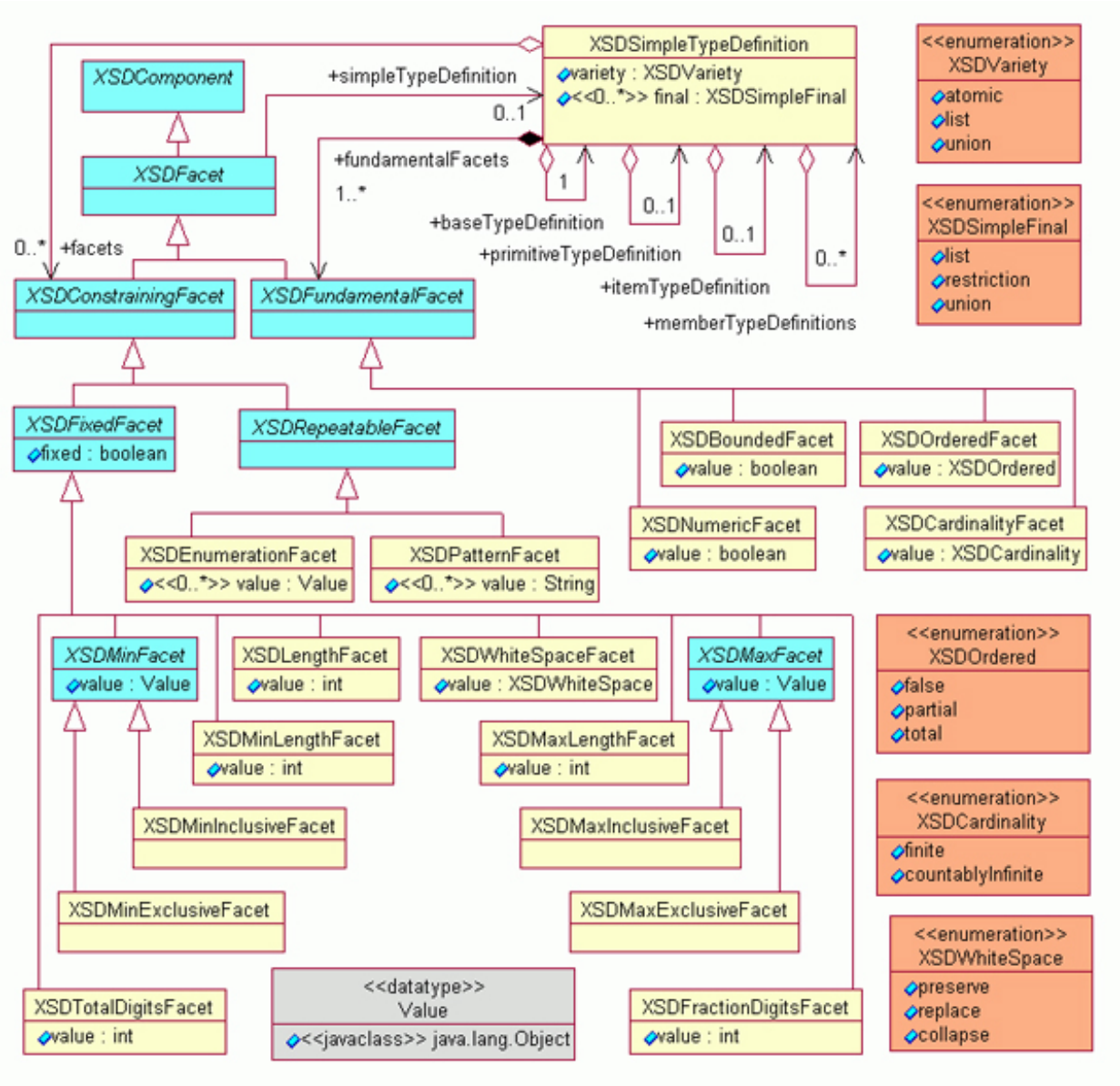
- **XSDComponent**: is the base class for all of the XML Schema Infoset model classes.
- **XSDSchema**: is the concrete class representing the xml schema root object (<xsd:schema ...>)
- **XSDAnnotation**: is the concrete class representing xml schema components annotations (<xsd:annotation>)
- **XSDSimpleTypeDefinition**: is the concrete class representing a simple type (<xsd:simpleType ...>)
- **XSDComplexTypeDefinition**: is the concrete class representing a complex type (<xsd:complexType ...>)
- **XSDElementDeclaration**: is the concrete class representing an element; either global, local or reference (<xsd:element ...>)
- **XSDAttributeDeclaration**: is the concrete class representing an attribute; either global,local

or reference (<xsd:attribute ...>)

- **XSDAttributeGroupDefinition**: is the concrete class representing an attribute group (<xsd:attributeGroup ...>)
- **XSDModelGroupDefinition**: is the concrete class representing a group; either global or reference (<xsd:group ...>)
- **XSDModelGroup**: is the concrete class representing a local group (<xsd:sequence>, <xsd:choice>or <xsd:all>)
- **XSDWildCard**: is the concrete class representing a wild card element or attribute (<xsd:any> or <xsd:anyAttribute>)

Component hierarchy, relations, and attributes

The abstract XML Schema components, as described in *XML Schema Part 2: Datatypes* of the Standard, are related according to the following hierarchy with these defined relationships and attributes:



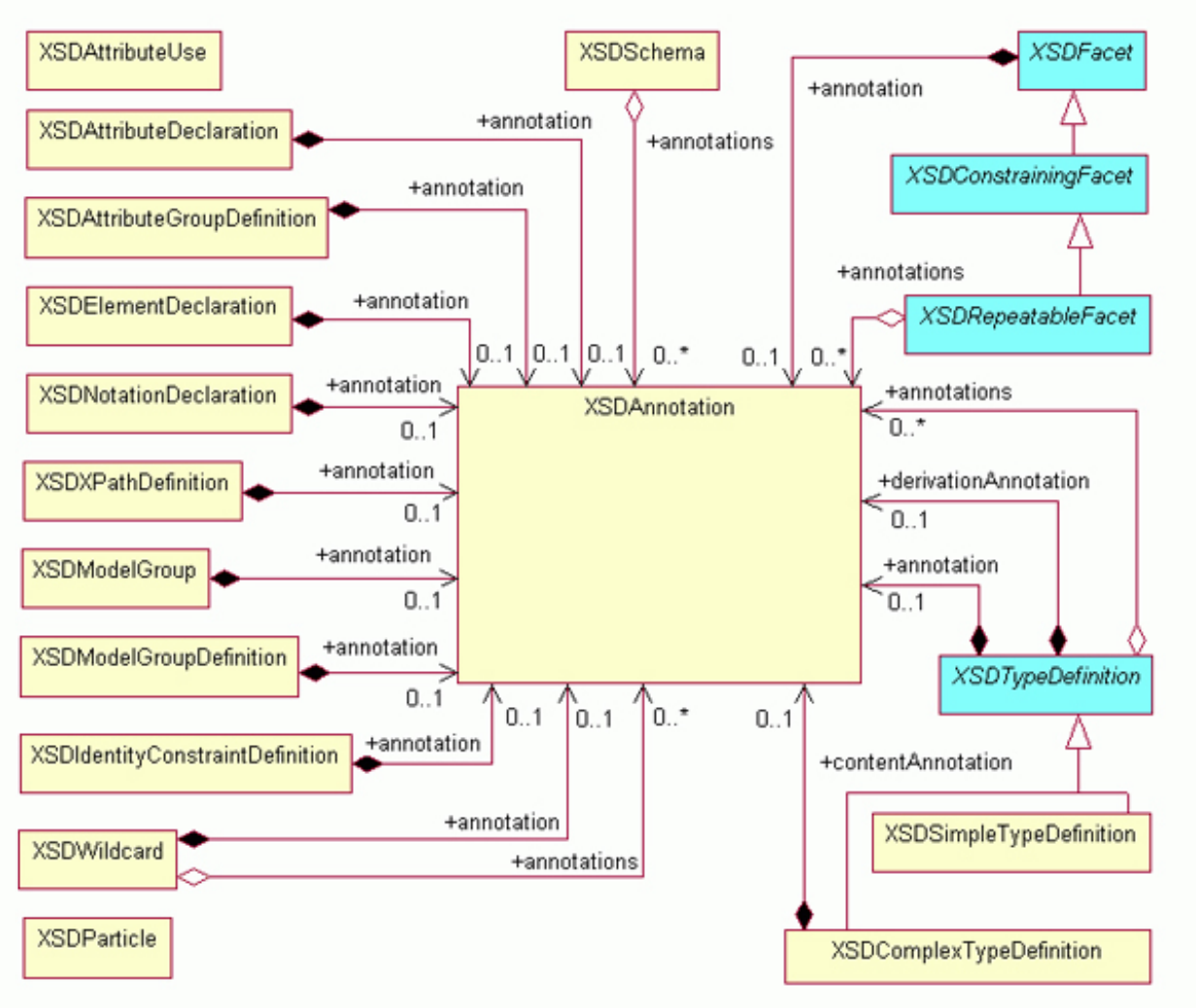
The figure above illustrates how the XML Schema Datatypes and facets are modeled:

- **XSDFacet**: is the base class for all of the XML Schema facets.
- **XSDConstrainingFacet**: is the base class for all of the XML Schema facets that provide constraints to the model, which include length facets, inclusive facets, exclusive facets, total digits facets, white space facets, fraction digits facets, enumeration facets, and pattern facets.
- **XSDFixedFacet**: is the base class for all of the XML Schema facets that provide constraints that can not be changed through type inheritance (in other words, must be fixed).
  - **XSDLengthFacet**: corresponds to the xml schema <xsd:length ...> facet.
  - **XSDMinLengthFacet**: corresponds to the xml schema <xsd:minLength ...> facet.
  - **XSDMaxLengthFacet**: corresponds to the xml schema <xsd:maxLength ...> facet.

  - **XSDMinInclusiveFacet**: corresponds to the xml schema <xsd:minInclusive ...> facet.
  - **XSDMaxInclusiveFacet**: corresponds to the xml schema <xsd:maxInclusive ...> facet.
  - **XSDMinExclusiveFacet**: corresponds to the xml schema <xsd:minExclusive ...> facet.
  - **XSDMaxExclusiveFacet**: corresponds to the xml schema <xsd:maxExclusive ...> facet.
  - **XSDWhiteSpaceFacet**: corresponds to the xml schema <xsd:whiteSpace ...> facet.
  - **XSDFractionDigitsFacet**: corresponds to the xml schema <xsd:fractionDigits ...> facet.
  - **XSDTotalDigits**: corresponds to the xml schema <xsd:totalDigits ...> facet.
- **XSDRepeatableFacet**: is the base class for all of the XML Schema facets that provide constraints that can repeat.
  - **XSDEnumeration**: corresponds to the xml schema <xsd:enumeration ...> facet.
  - **XSDPatternFacet**: corresponds to the xml schema <xsd:minLength ...> facet.

Component annotations

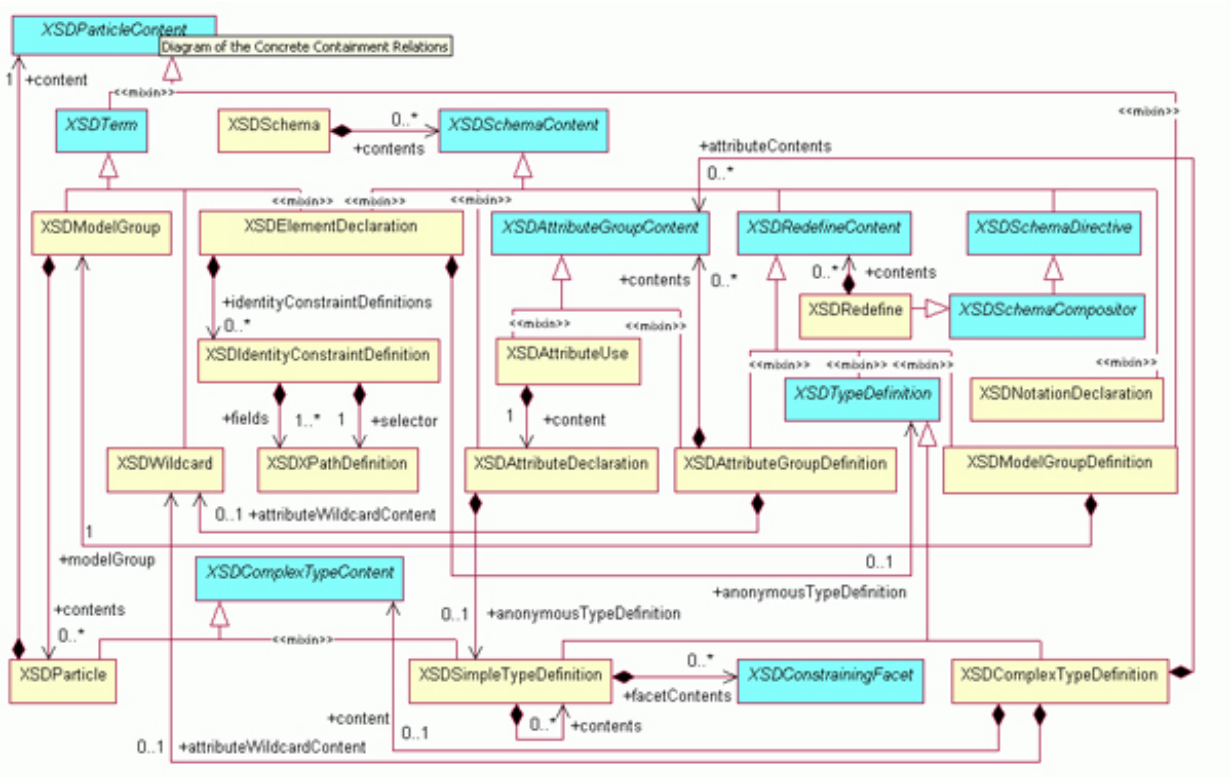
The abstract XML Schema Components, as described in Part 1 and Part 2 of the Standard, are annotated as follows:



The figure above illustrates how the XML Schema components are annotated, which correspond to xml schema <xsd:annotation ...> tag.

Concrete containment

The XML Schema Infoset model's concrete containment relationships are represented by the following diagram:



In the figure above, most of the relationships are black diamond relationships, which are the containment relationships. These form the basis for the majority of classes within the XML Schema Infoset model. Below is a short description of some of the more interesting relationships from different classes in the above diagram.

XSDSchema

The root XSDSchema object can contain (0..*) XSDSchemaContent objects. Each of these XSDSchemaContent objects will show up as xml elements under the root xml schema element.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.example.com/IPO"
 xmlns:ipo="http://www.example.com/IPO">
    <annotation>
        <documentation>
            Addresses for International Purchase Order Schema
        </documentation>
    </annotation>

    <complexType name="Address">
        ...
    </complexType>

    <simpleType name="USState">
```

```
        ...
    </simpleType>

    <element name="order" type="ipo:Address">

    <attribute name="order" type="ipo:USState">

...
</schema>
```

In the above code snippet, you can see that the root schema element contains an annotation, complexType, simpleType, element, and attribute.

The root XSDSchema object can contain any of the following model objects as children within the concrete tree structure of the XML Schema Infoset model:

- XSDImport
- XSDInclude
- XSDRedefine
- XSDAnnotation
- XSDElementDeclaration
- XSDAttributeDeclaration
- XSDComplexTypeDefinition
- XSDSimpleTypeDefinition
- XSDModelGroupDefinition
- XSDAttributeGroupDefinition
- XSDNotationDeclaration

XSDModelGroupDefinition

Model group definitions can either be global or references to global model group definitions. A global XSDModelGroupDefinition can contain a single XSDModelGroup.

XSDModelGroup

Model groups are also known as local groups, which have of one of the following compositions: sequence, choice, or all. An XSDModelGroup contains (0..*) XSDParticle objects.

XSDParticle

Particles contain the maxOccurs and minOccurs values for the content of the XSDParticle, which is known as the XSDParticleContent.

An XSDParticle contains a single XSDParticleContent, which can be one of the following objects:

- XSDModelGroup
- XSDWildCard

- XSDElementDeclaration
- XSDModelGroupDefinition

XSDComplexTypeDefinition

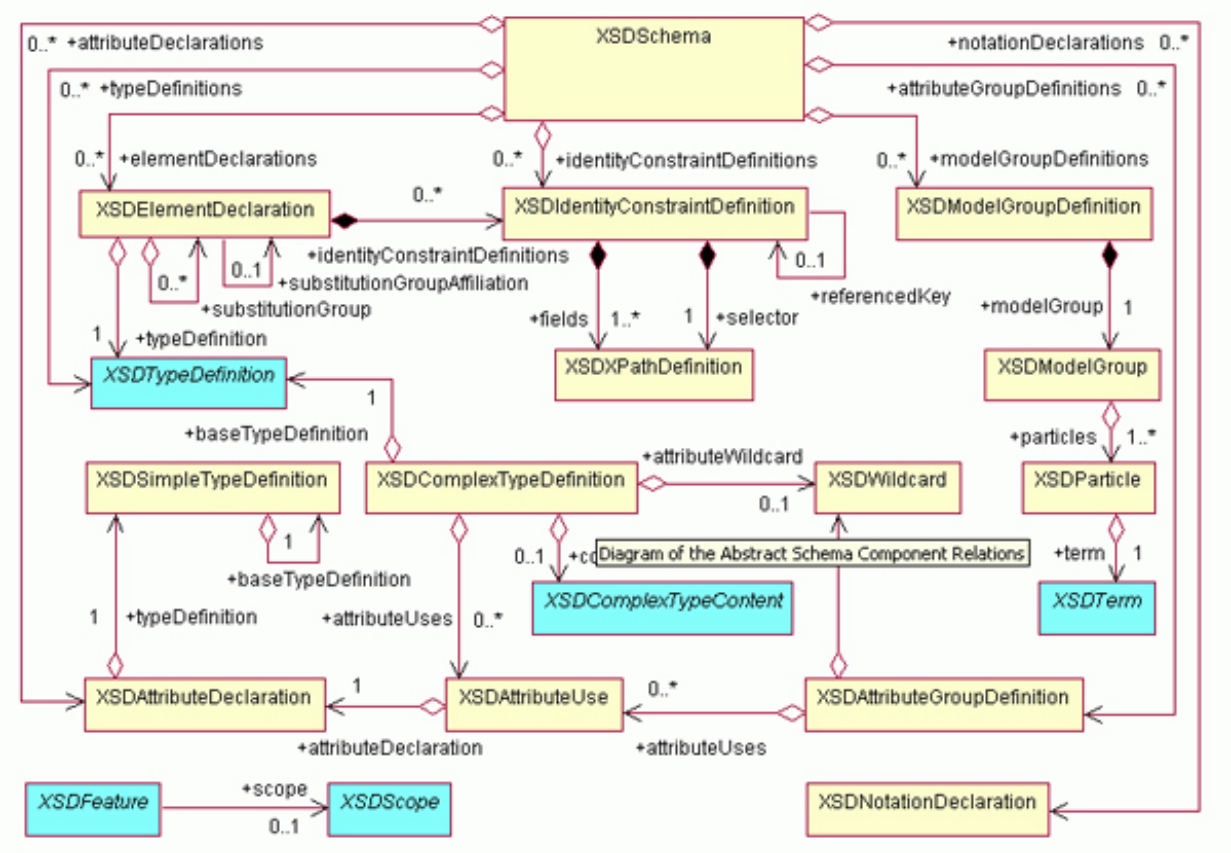
Complex type definitions can either be global or anonymous. If you are unfamiliar with the term anonymous, you can think of this as being local.

An XSDComplexTypeDefinition can contain a single attribute wildcard, (0..*) XSDAttributeGroupContent and a single XSDComplexTypeContent.

The XSDComplexTypeContent can either be an XSDParticle with an XSDModelGroup or XSDModelGroupDefinition as the particle's content or an XSDSimpleTypeDefinition, which means this complex type has simple content. If the XSDParticle's content is an XSDModelGroupDefinition, it will be a group reference to a global group.

Component relations

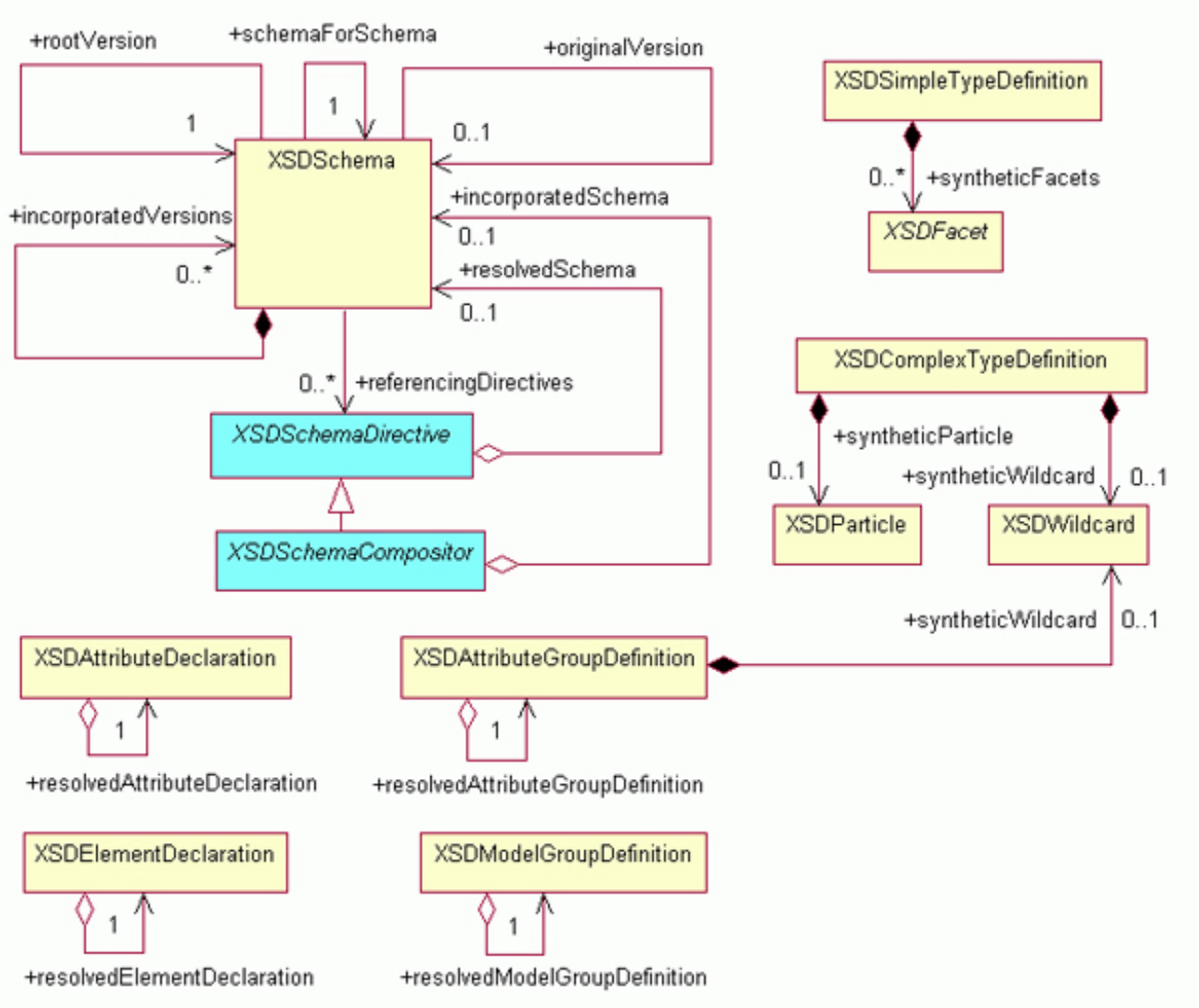
The abstract XML Schema components have the following defined relationships:



The figure above is very similar to the standard *non-normative Schema Components Diagram* . This diagram is very complex, so the main thing to notice are the white diamond relationships, which are the shared pseudo-containment relationships and are computed from other relations within the model.

Concrete Schema composition

The following concrete components resolve to the following abstract components:



The figure above illustrates how some concrete components resolve to the other concrete components:

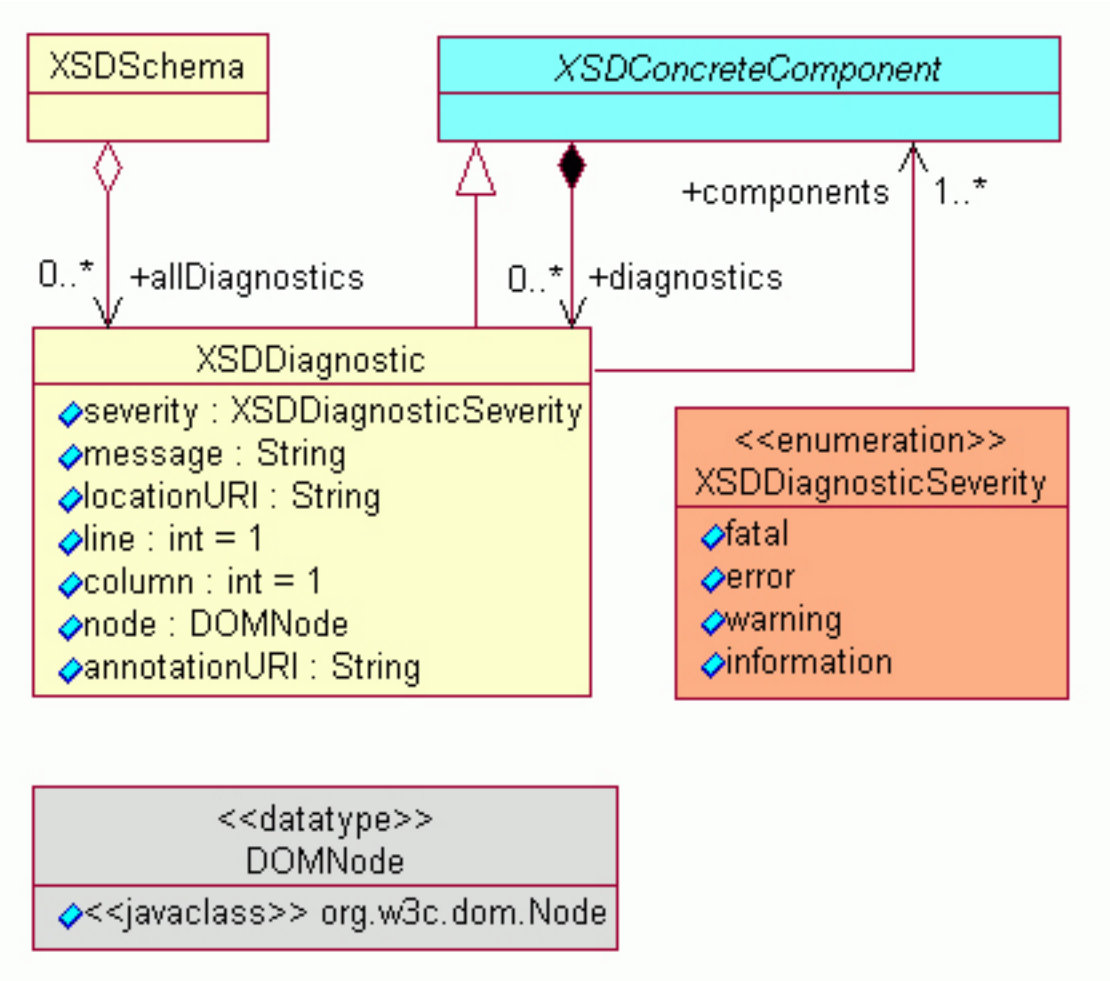
- **XSDModelGroupDefinition**: a group reference resolves to a global group
- **XSDAttributeGroupDefinition**: an attribute group reference resolves to a global attribute group
- **XSDElementDeclaration**: an element declaration reference resolves to a global element

declaration

- **XSDAttributeDeclaration**: an attribute declaration reference resolves to a global attribute declaration

Diagnostics diagram

The following diagnostics are produced by validation of the XML Schema Infoset model. More details on the validation of the XML Schema Infoset model will be in Part 2 of this tutorial series.



Section 5. Working with XML Schema resources in Eclipse

Creating an XML Schema file

Before you create complicated XML Schema structures, you should first create a skeleton XML Schema model and save it in the Eclipse workspace. You have the option of giving the schema a namespace or not.

Creating an XML Schema file with no target namespace

As shown in Listing 1 below, the steps for creating an XML Schema file with no target namespace are:

1. Create the org.eclipse.emf.ecore.resource.Resource using the new xml schema Uri
2. Create the XSDSchema root object
3. Set up the schema for schema namespace information on the root XSDSchema object
4. Call updateElement() on the root XSDSchema object
5. Add the root XSDSchema object to the resource you created in step 1
6. Save the resource

Listing 1. Creating an XML Schema file with no target namespace

```
org.eclipse.xsd.examples.command.CreateXSDWithNoTNSOperation
public XSDSchema createXSDSchema(IFile xsdFile)
{
  try
  {
    //Get the URI of the model file.
    URI fileURI = URI.createPlatformResourceURI(xsdFile.getFullPath().toString());

    //Create a resource set to manage the different resources
    ResourceSet resourceSet = new ResourceSetImpl();

    //Create a resource for this file.
    Resource resource = resourceSet.createResource(fileURI);

    //Create the root XSDSchema object
    XSDSchema xsdSchema = XSDFactory.eINSTANCE.createXSDSchema();

    //set the schema for schema QName prefix to "xsd"
    xsdSchema.setSchemaForSchemaQNamePrefix("xsd");
    java.util.Map qNamePrefixToNamespaceMap = xsdSchema.getQNamePrefixToNamespaceMap();

    //put the following namespace in the root schema namespace map
    //xsd:http://www.w3.org/2001/XMLSchema
    qNamePrefixToNamespaceMap.put(xsdSchema.getSchemaForSchemaQNamePrefix(),
      XSDConstants.SCHEMA_FOR_SCHEMA_URI_2001);

    //We call updateElement to synchronize the MOF model with the underlying DOM model
    //This should only have to be done after creating a new model
    xsdSchema.updateElement();
```

```
    //Add the root schema to the resource that was created above
    resource.getContents().add(xsdSchema);

    //Save the contents of the resource to the file system.
    resource.save(Collections.EMPTY_MAP);

    return xsdSchema;
  }
  catch (Exception exception)
  {
    exception.printStackTrace();
  }
  return null;
}
```

Creating an XML Schema file with target namespace "http://www.eclipse.org/xsd/examples/createxsd"

As shown in Listing 2, the basic steps in creating an XML Schema file with a target namespace are very similar to creating aa XML Schema file with no target namespace. The major difference is that you set the targetNamespace() on the XSDSchema root, and you also add a defined prefix to the root schema QName prefix namespace map on page 25 , which maps to the new target namespace.

Listing 2. Creating an XML Schema file with a target namespace

```
org.eclipse.xsd.examples.command.CreateXSDWithTNSOperation
public XSDSchema createXSDSchema(IFile xsdFile)
{
  try
  {
    //Get the URI of the model file.
    URI fileURI = URI.createPlatformResourceURI(xsdFile.getFullPath().toString());

    //Create a resource set to manage the different resources
    ResourceSet resourceSet = new ResourceSetImpl();

    //Create a resource for this file.
    Resource resource = resourceSet.createResource(fileURI);

    //Create the root XSDSchema object
    XSDSchema xsdSchema = XSDFactory.eINSTANCE.createXSDSchema();

    //Set the target namespace of the given schema document to
    //http://www.eclipse.org/xsd/examples/createxsd
    xsdSchema.setTargetNamespace("http://www.eclipse.org/xsd/examples/createxsd");

    java.util.Map qNamePrefixToNamespaceMap = xsdSchema.getQNamePrefixToNamespaceMap();
    qNamePrefixToNamespaceMap.put(xsdSchema.getSchemaForSchemaQNamePrefix(),
      XSDConstants.SCHEMA_FOR_SCHEMA_URI_2001);

    //put the following namespace in the root schema namespace map
    //createxsd:http://www.eclipse.org/xsd/examples/createxsd
    qNamePrefixToNamespaceMap.put("createxsd", xsdSchema.getTargetNamespace());
```

```
    //We call updateElement to synchronize the MOF model with the underlying DOM model
    //This should only have to be done after creating a new model
    xsdSchema.updateElement();

    //Add the root schema to the resource that was created above
    resource.getContents().add(xsdSchema);

    // Save the contents of the resource to the file system.
    resource.save(Collections.EMPTY_MAP);

  return xsdSchema;
  }
  catch (Exception exception)
  {
    exception.printStackTrace();
  }
  return null;
}
```

Loading an XML Schema file

Loading an existing XML Schema model is relatively simple if you use the default platform protocol resolver "platform:/resource". You may want to implement an org.eclipse.emf.ecore.resource.URIConverter to resolve resources that might not be found using the default platform protocol resolver. Some of these could include XML Schema files located in a plug-in directory or somewhere on the World Wide Web.

The following listing shows how to load any XML Schema file within the Eclipse workspace using the default platform protocol resolver.

```
public XSDSchema loadXSDSchema(IFile xsdFile)
{
  try
  {
    //Create a resource set to manage the resources
    ResourceSet resourceSet = new ResourceSetImpl();

    //Let the resource set load the resource from the given uri
    XSDResourceImpl xsdResource = (XSDResourceImpl) resourceSet.getResource(URI.
      createURI("platform:/resource" + xsdFile.getFullPath()), true);

    //The contents of the resource is the root XSDSchema object
    XSDSchema xsdSchema = xsdResource.getSchema();

    return xsdSchema;
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }

  return null;
```

```
}
```

XSDResourceFactory and XSDResource

The Eclipse Modeling Framework loads resources by first tying to find a ResourceFactory based on the file extension of the file being loaded. This is done by first looking in the org.eclipse.emf.ecore.resource.Resource.Factory.Registry, and if the extension is registered, then it uses that ResourceFactory.

If the extension is not found, then the default ResourceFactory is used. The default is XMIResourceFactory, which clearly would never load XML Schema models because XML Schema is not written in XMI.

So, what do you do if you want to load a resource that contains an XML Schema model that has an extension other than "xsd"?

Globally register the extension using either of these methods:

- org.eclipse.emf.ecore.resource.Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap(). put("myxsd", new org.eclipse.xsd.util.XSDResourceFactoryImpl());
- Add the following extension point in your plugin.xml file

  <extension point = "org.eclipse.emf.ecore.extension_parser">

  <parser type="myxsd" class="org.eclipse.xsd.util.XSDResourceFactoryImpl"/>

  </extension>

or

Locally register the extension using this method:

- org.eclipse.emf.ecore.resource.ResourceSet resourceSet = new org. eclipse.emf.ecore.resource.impl.ResourceSetImpl(); resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap(). put("myxsd", new org.eclipse.xsd.util.XSDResourceFactoryImpl());

  Then use the above resourceSet to load all of your XML Schema files.

Section 6. Working with namespaces

QName prefix namespace map

The *QName prefix namespace map* stores a prefix as the keys and namespace Uri as the values. This QName prefix namespace map is stored on the root XSDSchema object and is used when XSDNamedComponents need to be resolved.

If you want to add additional prefix-namespace pairs to a given XML Schema file, you can do the following:

```
//Add additional prefix-namespace pair xmlns:address="http://www.eclipse.org/xsd/examples/
qNamePrefixToNamespaceMap.put("address","http://www.eclipse.org/xsd/examples/address");
```

The above code snippet produces the following serialized XML Schema model:

```
<?xml version="1.0"?>
<schema targetNamespace="http://www.eclipse.org/xsd/examples/createxsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:address="http://www.eclipse.org/xsd/examples/address"
    xmlns:createxsd="http://www.eclipse.org/xsd/examples/createxsd"/>
```

Includes

An XML Schema model may contain any number of *<xsd:include>* elements. Their schemaLocation attributes consist of a URI reference that should be relative to the root XML Schema model document.

The XML Schema corresponding to the root XML Schema contains not only the components corresponding to its definition and declaration, but also all the components of all the XML Schemas corresponding to any <xsd:include>d schema documents.

Such included schema documents must either:

- Have the same targetNamespace as the <xsd:include>ing schema document
- Have no targetNamespace at all, in which case the <xsd:include>d schema document is converted to the <xsd:include>ing schema document's targetNamespace.

The code snippet below shows how you would create and add a <xsd:include> to a root schema. As soon as the xsdSchema.getContents().add(0,xsdInclude) method is called you will notice that the included schema will get loaded into the same resource set that the root schema is in.

```
//Create a XSDInclude
XSDInclude xsdInclude = XSDFactory.eINSTANCE.createXSDInclude();
```

```
//Set the uri to the xml schema file that you want to use the definitions from
xsdInclude.setSchemaLocation("address.xsd");

//we always add the new includes to the top of the root schemas content list
//the included XML Schema file will be loaded at this time
xsdSchema.getContents().add(0,xsdInclude);
```

Imports

An XML Schema model may contain any number of *<xsd:import>* elements. Their schemaLocation attributes consist of a URI reference that should be relative to the root XML Schema document. Unlike <xsd:include>, <xsd:import> has a namespace attribute that needs to match a namespace in the QName prefix namespace map on page 25 .

The <xsd:import> element tag identifies namespaces used in external references, in other words, those whose QNames identify them as coming from a different namespace than the root schema document's targetNamespace. The actual value of its namespace indicates that the containing schema document may contain qualified references to schema components in that namespace (via one or more prefixes declared with namespace declarations in the normal way).

The code snippet below shows how you would create and add a <xsd:import> to a root schema document. As soon as the `xsdSchema.getContents().add(0,xsdImport)` method is called, you might notice that the imported schema might **NOT** get loaded. To force the load on a <xsd:import> you need to call `((XSDImportImpl)xsdImport).importSchema();`.

```
//Create a XSDImport
XSDImport xsdImport = XSDFactory.eINSTANCE.createXSDImport();

//Set the uri to the xml schema file that you want to use the definitions from
xsdImport.setSchemaLocation("address.xsd");

//we always add the new imports to the top of the root schemas content list
xsdSchema.getContents().add(0,xsdImport);

//This will force load the imported schema
((XSDImportImpl)xsdImport).importSchema();
```

Section 7. You try it!

Copying the code

All of the code snippets used in this tutorial are bundled in this Eclipse plug-in: org.eclipse.xsd.examples.zip. To install this plug-in and examine the source code:

1. Unzip org.eclipse.xsd.examples.zip into a temporary directory.
2. Create a Simple project called org.eclipse.xsd.examples in the Eclipse Workbench.
3. Copy the contents of the org.eclipse.xsd.examples.zip file into the newly created eclipse project. Click Yes if a dialog asks if you want to override the .project file.
4. Expand the org.eclipse.xsd.examples/src directory to view the source.

If you have compiler errors, you can fix these by adjusting the "**Windows > Preferences > Plug-In Development > Target Platform**" setting and then updating the project's classpath.

Troubleshooting and bugs

Please report Eclipse bugs in the Eclipse Bugzilla server: *http://bugs.eclipse.org/bugs*.

Section 8. Summary and resources

Summary

As XML data is becoming more popular, more companies are depending on common definitions, which are being built using XML Schemas. The *XML Schema Infoset Model* provides a well-designed set of interfaces, implementations, and algorithms for representing and manipulating XML Schema models.

Now that you've completed Part 1 of this tutorial series, you'll be ready to continue with Part 2. Part 2 will walk you through the steps in creating almost every construct within the XML Schema Infoset model.

Resources

For more information on using XML Schema or the Eclipse Modeling Framework, see the following suggested material:

XML Schema resources

XML Schema Infoset Model

http://www.eclipse.org/xsd/

XML Schema Infoset Model FAQ

http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/xsd-home/faq.html

XML Schema Infoset Model Drivers

http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/xsd-home/downloads/dl.html

XML Schema Infoset Model newsgroup

news://news.eclipse.org/eclipse.technology.xsd

XML Schema Tutorial, Roger L. Costello, September 2001

http://www.xfront.com/

XML Schema Part 0: Primer

http://www.w3.org/TR/xmlschema-0/

XML Schema Part 1: Structures

http://www.w3.org/TR/xmlschema-1/

XML Schema Part 2: Datatypes

http://www.w3.org/TR/xmlschema-2/

Eclipse Modeling Framework resources

Eclipse Modeling Framework

http://www.eclipse.org/emf/

EMF Users' Guide

http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/emf-home/docs/emfug.pdf

Feedback

We welcome your feedback on this tutorial, and look forward to hearing from you. We'd also like to hear about other tutorial topics you'd like to see covered.

For questions about the content of this tutorial, contact the author, Dave Spriet, at *spriet@ca.ibm.com*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.