ALTOVA®

WhitePaper

Enterprise Data Modeling Using XML Schema

Investigating an emerging paradigm using components of Altova's MissionKit™ for Software Architects

Altova GmbH • Rudolfsplatz 13a/9 • A-1010 Wien, Austria/EU • Tel: 545 5155 - 0 • Fax: 545 5155 - 9 • Web: www.altova.com

Enterprise Data Modeling Using XML Schema

Investigating an emerging paradigm using components of Altova's MissionKit™ for Software Architects

Nick Nagel, Ph.D.
Altova Inc.

1. Introduction 4
2. What Exactly is a Formal Specification Anyway? 5
 2.1 Data Modeling with XSD 6
 2.2 Specifying the Model 6
 2.2 A Short List of XSD Benefits 11
 2.3 Composition vs. Aggregation 11
3. Generating Distributed System Components from an XSD 14
 3.1 Generating a Persistent Store 14
 3.2 Front-end Generation 16
 3.3 Information Asset Management 18
  3.3.1 Creating the User Interface 20
  3.3.2 Mapping XML to RDBMS 24
4. Charting the Model with UML 25
5. Conclusion 28
References 30

© Copyright 2007 Altova® GmbH. All rights reserved.

Executive Summary

With its rich support for associating programmatic data types with information items, and its complete implementation of the object oriented data modeling framework, XSD stands poised to pave the way for a paradigm shift in enterprise application development. In recent years, there has been an ever increasing demand for the definition of business entities placed on XML across multiple platforms. This reflects a natural division along which to achieve a separation of concerns in enterprise development – between the declarative data model and the associated business rules and business process logic that encapsulate operations performed on instances of the model. The declarative nature of data modeling readily lends itself to expression using XSD.

A new paradigm will emerge in coming years where the XSD-specified data model will become the focal point in enterprise development. XSD is sufficiently formalized to enable the auto-generation of a myriad of components from end-to-end along distributed systems, ranging from relational entities optimized for storage and retrieval on the back end, to language-independent RAM-resident object representations (generated by XML binding) in middleware, to document-style, form-based user-interfaces on the client side. All along the way the constraints inherent to a schema-specified architectural model enforce business requirements, contributing at every stage to the robustness of the overarching distributed system.

This whitepaper is intended to explicitly articulate this insight and demonstrate by way of example how the XSD provides just the right level of abstraction, and just the right degree of precision, to generate a highly constrained and formal data model that can be used to propagate business rules end-to-end in the enterprise system.

Components of Altova's MissionKit for Software Architects (XMLSpy®, MapForce®, StyleVision®, UModel®, DatabaseSpy™, and SchemaAgent®), are uniquely situated to provide comprehensive support for the enterprise data modeling process from start to finish.

© Copyright 2007 Altova® GmbH. All rights reserved.

1. Introduction

This whitepaper is an exploration of the possibilities engendered by rethinking the role of the XML Schema Definition (XSD) language in the development of enterprise information systems. In particular, it might be argued that current trends in system development reflect a paradigm shift toward XML-centric data processing with XSD playing an increasingly critical role in the specification of the data model. The purpose of this paper is to explicitly state the potential role of XSD as a data modeling language, and then to show some of the benefits that a formal XSD specification can bring.

© Copyright 2007 Altova® GmbH. All rights reserved.

2. What Exactly is a Formal Specification Anyway?

Not too long ago, much ado was made in the world of data modeling about formalism, notational schemes, and the “correct” means of achieving a formal description of a domain model. But when it comes right down to it, much of the effort toward formalizing a model amounts to devising a system to specify the *language* of some specific domain(s) of interest. In other words, data modeling is largely comprised of the identification of the vocabulary used to describe domain-specific entities (the objects that will be acting or acted upon in a system) and identifying the relationships between these entities. How you go about specifying the model is – in the end – largely irrelevant. What matters is that you make the time to do so and do it in a formal and explicit way. Creating a specification and documenting your efforts may cost a bit up front, but in the end the benefits of doing so can be well worth it.

Data modeling today is a brave new world with a varied and colorful palette to choose from for sketching out your system. Certainly a wide range of more or less formal systems has been proposed, with the most notable culmination of approaches leading to the development of UML (the Unified Modeling Language) [Booch et. al., 2005]. But UML is not the only formal system available for describing a model (compare, e.g., UML vis-à-vis Entity Relational Diagrams (ERD)). Indeed, at the finest level of analysis, the implementation of an information processing system is a formal description of the data model (or, more precisely, contains the model). But working toward the implementation of a complex system it is often useful (many would argue necessary) to derive formal abstractions of aspects of the system to serve as “roadmaps”, or guides to development.

XSD is an excellent way to achieve precisely such a formal system description. I hope to show that XSD can be used together with other languages and technologies in the developer's toolkit to guide and facilitate the development of complex information processing systems.

© Copyright 2007 Altova® GmbH. All rights reserved.

2.1 Data Modeling with XSD

Arguably the greatest benefit of using XSD to specify your data model is that it provides a formal description that can contribute at every step of the way toward the development of an end-to-end solution to a broad class of problems. XSD provides a method to create a description that is precise and follows a prescribed set of rules to generate an unambiguous declaration of the set of entities and associated features of the model. Such a description vastly benefits distributed information processing development by serving as a standard against which members of disparate teams can develop compatible processing components. Also, XSD affords a high degree of precision in creating the model, which ultimately contributes to the robustness of overarching processing systems through its inherent ability to validate infosets.

2.2 Specifying the Model

Let's consider a hypothetical scenario in which we have to represent information associated with educational resources – books, articles, Web pages and such – that might be used in the creation of literary works. At some point between this loose conceptualization of a requirement and the implementation of our hypothetical processing system, we'll have to develop a vocabulary to refer to the entities in the targeted domain. Of course, there are several different approaches we can take to create this vocabulary, but for our purposes let's take as our starting point the specification of the data model using XSD.

An inherent issue that we come across when using XSD is that XSD is XML, and therefore it is verbose by design. Consequently, it's easy to get bogged down in the syntactical details of the modeling language and overlook the big picture. This problem can be readily solved, however, by the adoption of visual guides to schema development and the autogeneration of the underlying XSD code. The visual XML Schema editor in Altova XMLSpy facilitates precisely this approach, allowing us to take a “top-down” view of schema development and begin creating the data model from the blank page.

© Copyright 2007 Altova® GmbH. All rights reserved.

We can start our modeling project by breaking the top level component of our example scenario, *Resource,* into composite parts. Here, *Resource* comprises a *Title,* perhaps a *URL* (for Web pages), an associated *Author* or *Organization,* and perhaps a brief *Description* (which we might like to include in a resource database). Another component, *Articles*, might include an associated *Journal*, and *Books* might be associated with *Publishers, ISBNs,* etc. This preliminary analysis might lead us to construct an object oriented data model with definition reuse by inheritance. The good news is that a tremendous benefit of working with XSD is that it provides full support for object oriented data modeling. Indeed the Datatypes section (Part 2) of the XML Schema 1.0 specification (2005) defines a complete object oriented framework for built-in and user-defined types, allowing developers to construct a data model that is robust and compatible across a wide range of programming languages and platforms.

For the purposes of this discussion, let's keep things simple and limit the model to describing a Web resource *(Resource)*. The screenshot below shows a first-pass sketch of a *Resource* element created using the XMLSpy schema design view.

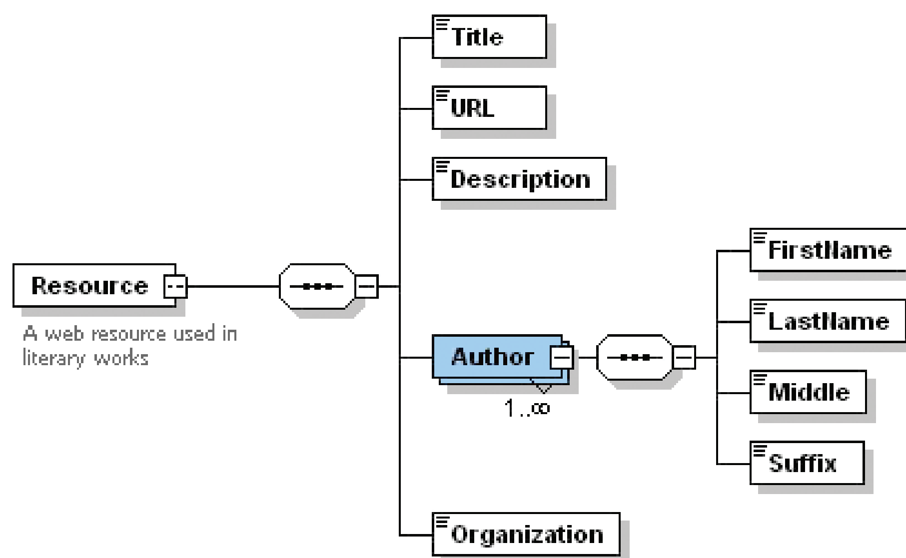

Figure 1. A schema diagram developed using the XMLSpy visual XML Schema editor

© Copyright 2007 Altova® GmbH. All rights reserved.

This diagram shows the *Resource* component modeled as a composite containing many of the elements identified above. The diagram also shows that a *Resource* might be associated with a multiplicity of authors (many authors might contribute to the development of a *Resource*), and that each *Author* breaks down into *FirstName, LastName, Middle* and *Suffix* fields. (Of course, the *Organization* and other elements might also be further decomposed in a more complex example.)

The raw XSD code autogenerated from the graphical representation is provided below.

```
---------------------------------------
<xs:schema ...>
  <xs:element name="Resource">
    <xs:annotation>
      <xs:documentation>
        A web resource used in
        literary works
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Title" type="xs:string"/>
        <xs:element name="URL" type="xs:string"/>
        <xs:element name="Description" type="xs:string"/>
        <xs:element name="Author"
              minOccurs="0"
              maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="FirstName" type="xs:string"/>
              <xs:element name="LastName" type="xs:string"/>
              <xs:element name="Middle" type="xs:string"/>
              <xs:element name="Suffix" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Organization"
              type="xs:string"
              minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
---------------------------------------
```

*Figure 2. The **Resource** schema code*

Note: This example shows the raw code generated up to this point. Code fragments not relevant to the discussion have been removed for clarity.

© Copyright 2007 Altova® GmbH. All rights reserved.

A cursory examination of the code reveals a “Russian doll” pattern in which the entire model definition is nested locally under the *Resource* element. The next step in fleshing out the model might be to abstract some of the components into global, named type definitions. The prime candidates for this sort of abstraction are the *Author* component and the *Resource* component itself. Below is a diagram of the revised schema after abstraction of the *Author* and *Resource* type definitions.

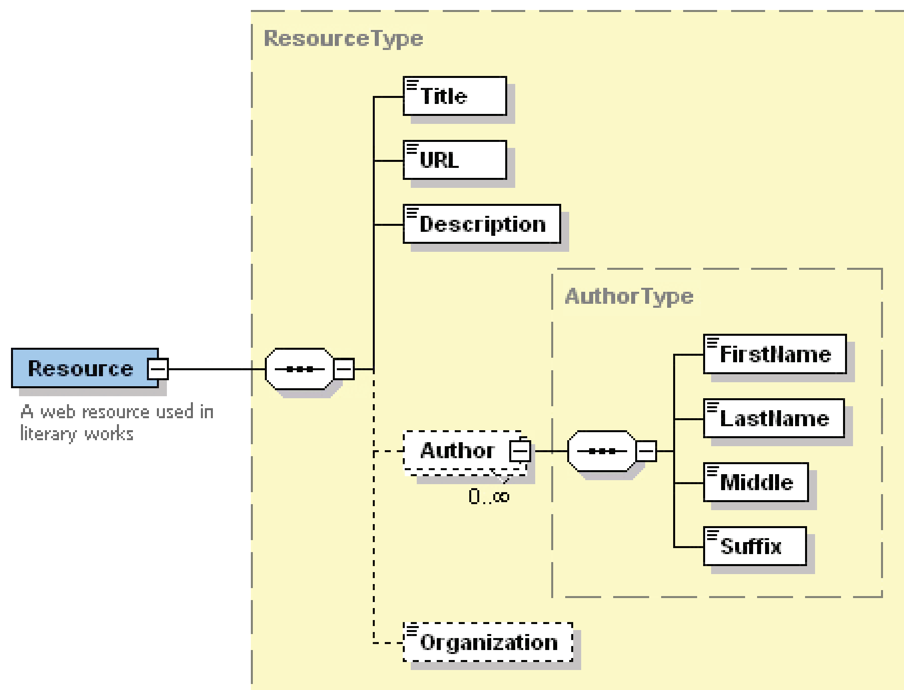

*Figure 3. The **Resource** schema diagram revised to define the **Resource** and **Author** components as named global types*

This example shows how XSD definitions can be developed incrementally – an approach with which it's possible to massage a model into shape over a successive number of developmental iterations.

Let's take a closer look at the *Author* type definition. Notice that at this point it's defined using simple fields typed as generic strings. Under some systems this might be about the finest level of precision we might achieve in constraining our data model (although Java now supports enumerated types and some relational databases provide non-standard mechanisms to make use of, e.g., regular expressions,

© Copyright 2007 Altova® GmbH. All rights reserved.

to constrain data fields). But with XSD we have built-in support for the use of *facets* – features specifically designed to constrain the value/lexical space of any simple information item.

For example, we might constrain the maximum length of the *FirstName* and *LastName* fields to 80 characters using the *max-length* facet. We might further constrain our *Middle* field to hold only a middle initial (we can impose such constraints using regular expressions). And we might permit only a small set of values for the suffix using the enumeration facet. These constraints would render instance data, such as the following, valid:

```
<Author>
     <FirstName>Ronald</FirstName>
     <LastName>Hornblatt</LastName>
     <Middle>R.</Middle>
     <Suffix>Ph.D.</Suffix>
   </Author>
```

While the following would not be valid:

```
<Author>
     <FirstName>Ronald</FirstName>
     <LastName>Hornblatt</LastName>
     <Middle>Rondolet</Middle>
     <Suffix>Jr.</Suffix>
   </Author>
```

This exposes yet another benefit of data modeling using XSD: its inherent support for validating the infoset. The formal definition of the model using XSD allows developers to take advantage of validation functionality abstracted into schema-aware XML parsers. Most information processing frameworks in use today now have schema-aware XML parsers built in and readily available for access by application developers.

© Copyright 2007 Altova® GmbH. All rights reserved.

2.3 A Short List of XSD Benefits

A few of the benefits of using XSD revealed by our simple example include the following:

1. XSD offers a platform agnostic approach to data modeling with many features that support a wide range of programming languages and platforms
2. XSD is object oriented and provides a well-specified framework for object oriented development
3. A wide range of pre-defined, built-in simple types (attributes and elements with text only content) provide rich support for constraining the data model to conform to multiple platform requirements
4. The highest levels of precision can be achieved using facets which are supported by XSD
5. Visual XSD development fosters an incremental and iterative approach to modeling in keeping with many of the methodologies applied toward software development today

Of course, there are numerous other benefits offered by XSD in addition to those listed above.

Next, we'll see how we can bridge the gap between XSD and schemas developed for relational database management systems (RDBMS).

2.4 Composition vs. Aggregation

The data model described in our example is compositional in nature. The *Resource* is a composite entity defined as the sum of its parts where each part belongs to a whole and the parts live and die with the whole. Indeed, the XSD approach to defining elements in terms of content models fosters the establishment of compositional relationships. However, this doesn't necessarily have to be the case. Figure 4 shows an alternative model for our *Resource* entity that establishes an aggregate relationship between the *Resource* and *Author* components.

© Copyright 2007 Altova® GmbH. All rights reserved.

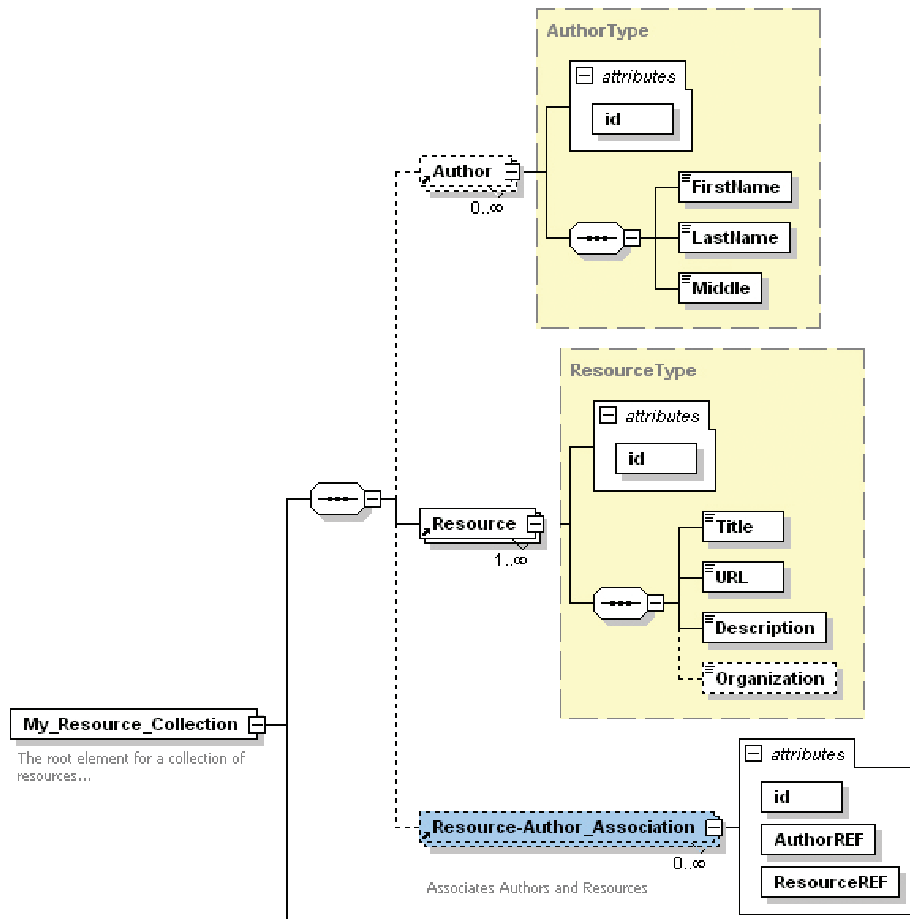

*Figure 4. The **Resource/Author** aggregate model in XSD*

To relate *Author* instances to *Resources* instances in this aggregate schema, we simply add *id* attributes to the *Author* and *Resource* types. This allows us to set up an *association element* (analogous to an *association class* or a *join table* in object oriented programming). The association element (*Resource-Author_Association* in this example) has two attributes, *AuthorREF* and *ResourceREF,* which point to a unique *Author id* and *Resource id* respectively. The explicit ids and references can be used to associate instances of the *Resource* and *Author* entities. We can take this a step further and enforce these referential dependencies in XSD using *key* and *keyref* constraints. Figure 5 shows a set of constraints specified to enforce uniqueness and referential integrity in the model.

© Copyright 2007 Altova® GmbH. All rights reserved.

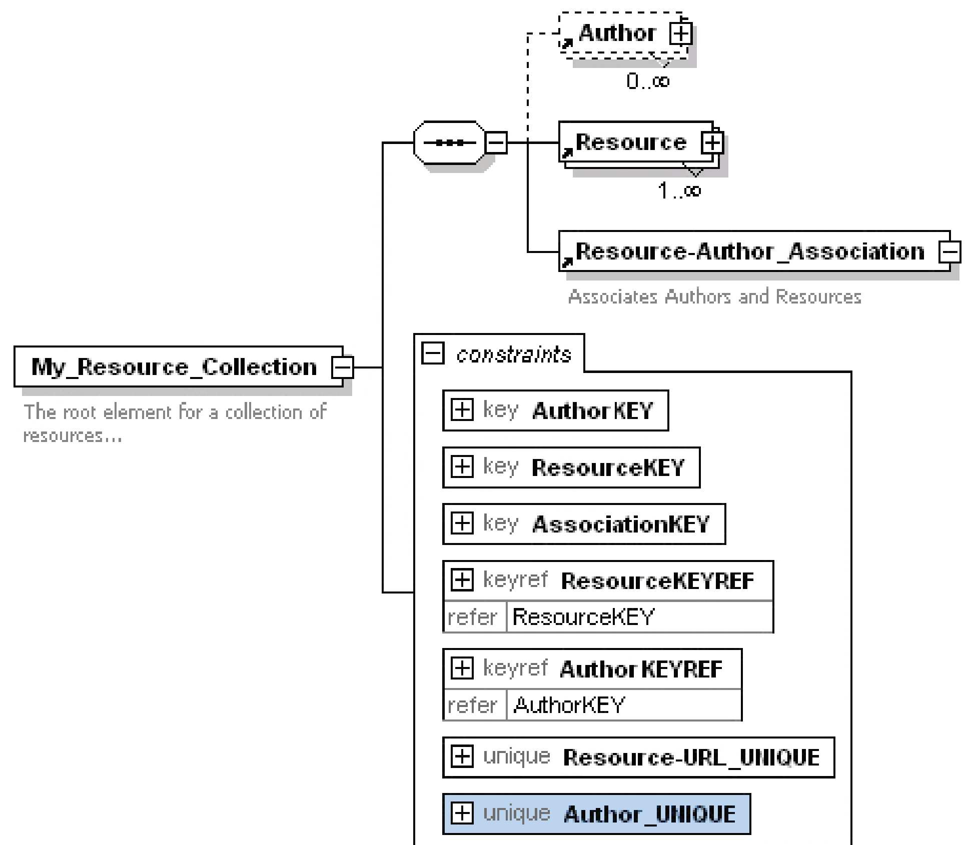

Figure 5. Enforcing uniqueness and referential integrity in XSD

Among the many implications that follow from this is that we can represent many-to-many relationships in XSD and define constraints over these relationships that would be difficult or impossible to represent using hierarchical XML. These features allow us to create XML structures that readily map into relational back-ends. Indeed, that's exactly what we'll demonstrate next with our model.

© Copyright 2007 Altova® GmbH. All rights reserved.

3. Generating Distributed System Components from an XSD

The beauty of data modeling with XSD is that the XML Schema can be made sufficiently precise to enable the autogeneration of distributed information processing components on all the tiers of a distributed system. The data model can be specified once, and used to generate model components system-wide to meet a wide range of needs, rather than having to manually map components on all the tiers of an application.

3.1 Generating a Persistent Store

To illustrate model component generation, I'll now discuss how to create a data entry system for the *Resource* component. Armed with our XSD data model, we can start by generating a persistent store for collections of *Resource* instance data. The constraints we've built into the model currently provide sufficient information to autogenerate the DDL (Database Definition Language) necessary to create a database backend. Figure 6 shows the DDL for the *Resource* component, which has been generated for this example using XMLSpy. We can use this DDL to create a relational database which will be built on a Microsoft® Access database.

© Copyright 2007 Altova® GmbH. All rights reserved.

```
1  CREATE TABLE [Author]
2  (
3    [FirstName] TEXT (255) NOT NULL ,
4    [LastName] TEXT (255) NOT NULL ,
5    [Middle] TEXT (255) NOT NULL ,
6    [id] TEXT (255) NOT NULL ,
7    CONSTRAINT [AuthorKEY] PRIMARY KEY ( [id] )
8  );
9
10 CREATE TABLE [Resource]
11 (
12   [Description] TEXT (255) NOT NULL ,
13   [Organization] TEXT (255) NOT NULL ,
14   [Title] TEXT (255) NOT NULL ,
15   [URL] TEXT (255) NOT NULL ,
16   [id] TEXT (255) NOT NULL ,
17   CONSTRAINT [ResourceKEY] PRIMARY KEY ( [id] )
18 );
19
20 CREATE TABLE [Resource-Author_Association]
21 (
22   [AuthorREF] TEXT (255) NOT NULL ,
23   [ResourceREF] TEXT (255) NOT NULL ,
24   [id] TEXT (255) NOT NULL ,
25   CONSTRAINT [AssociationKEY] PRIMARY KEY ( [id] )
26 ) ;
27
28 ALTER TABLE [Resource-Author_Association]
29   ADD CONSTRAINT [FK_TO_Author] FOREIGN KEY ( [AuthorREF] )
30   REFERENCES [Author] ( [id] ) ;
31
32 ALTER TABLE [Resource-Author_Association]
33   ADD CONSTRAINT [FK_TO_Resource] FOREIGN KEY ( [ResourceREF] )
34   REFERENCES [Resource] ( [id] ) ;

Message
Summary:
Success: 5
SQL1 *
```

Figure 6. Autogenerated Data Definition Language DDL for Microsoft Access (from ResourceSchema.xsd)

This listing shows the SQL generated to create an MS Access database for the *Resource* example.

Note: I modified the *alter* statements slightly to establish out-of-scope relationships.

© Copyright 2007 Altova® GmbH. All rights reserved.

Figure 7 shows the generated tables illustrating the many-to-many relationship between the *Author* and *Resource* elements established from the XSD data model as seen in the database design view of Altova DatabaseSpy.

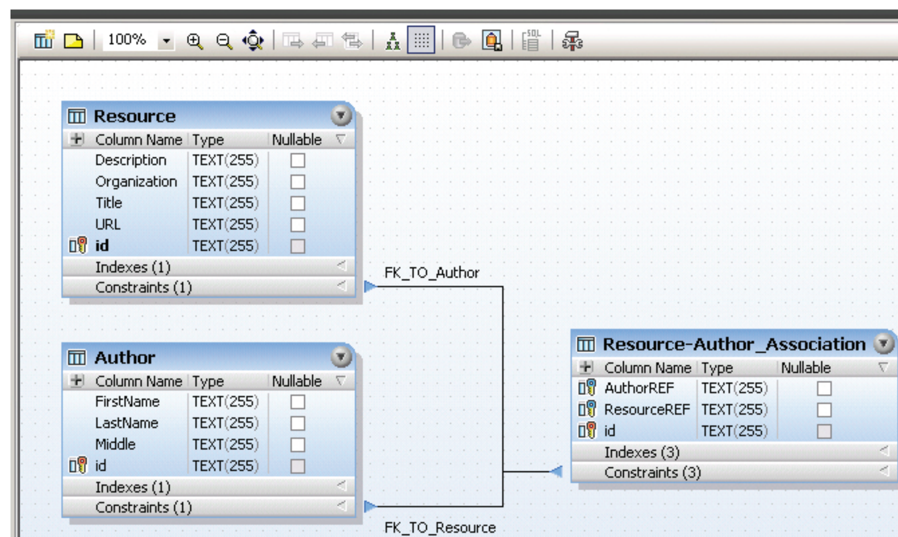

Figure 7. The newly generated database schema as viewed from the Altova DatabaseSpy database design editor

This example highlights the relative ease with which we can generate a database from the XSD-specified model. Next, we'll explore how the XSD can help generate a front-end for editing the database.

3.2 Front-end Generation

We will now generate a user interface that will enable data entry and editing. Again, we'll drive the workflow using our XSD model. To facilitate the development of the user interface, we'll specify a hierarchical XML format using the components defined in our existing data model. We'll start by creating a new schema which includes the *Resource* component model. Including the model allows us to re-use our type definitions to create a new structure for data entry. Figure 8 shows the hierarchical structure we'll use to define the data entry format.

© Copyright 2007 Altova® GmbH. All rights reserved.

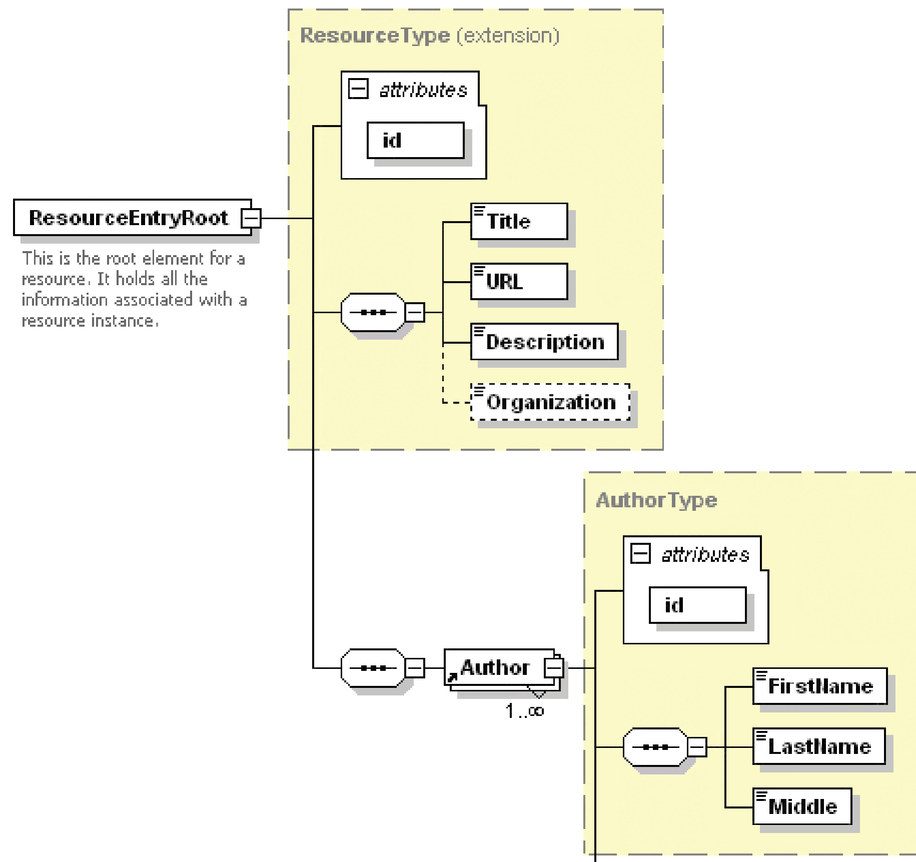

*Figure 8. The **Resource** data entry structure*

To create the new structure we simply add a global element to serve as the structural root for a resource (the *ResourceEntryRoot* element) and type it using our user-specified *ResourceType*. Next we add a sequence permitting the occurrence of one-to-many *Author* elements. The process we've just walked through is actually a form of object oriented type-derivation. To define the *ResourceEntryRoot* element we extend the *ResourceType* to permit the *Author* elements as content. The *Author* element is already defined in our model and available as a global named type. So we're done. It's that easy.

© Copyright 2007 Altova® GmbH. All rights reserved.

3.3 Information Asset Management

The preceding discussion exemplifies another huge benefit of data modeling with XSD: namely information asset management. It is a truism that here in the information age, information assets – information processing components that can be defined, scoped, and managed for reuse – are of enormous value to any organization. But in order for the potential value of such assets to be maximized and efficiently utilized, proper management is essential. XSD readily lends itself to component integration and reuse and, consequently, has unlimited potential to facilitate information asset management.

To illustrate the use of XSD for information asset management, consider the *Resource* data entry schema we just created. To generate that schema we included existing components, i.e., existing assets, into a new schema defining a data entry component structure. These sorts of relationships, inclusions and importations, can be readily visualized and managed through Altova SchemaAgent, a visual XML file management tool with a graphical interface designed to overcome the complexity inherent in managing large-scale XML component design and integration. Figure 9 clearly illustrates the ease with which we can view and manage heterogeneous schema relationships using SchemaAgent.

© Copyright 2007 Altova® GmbH. All rights reserved.

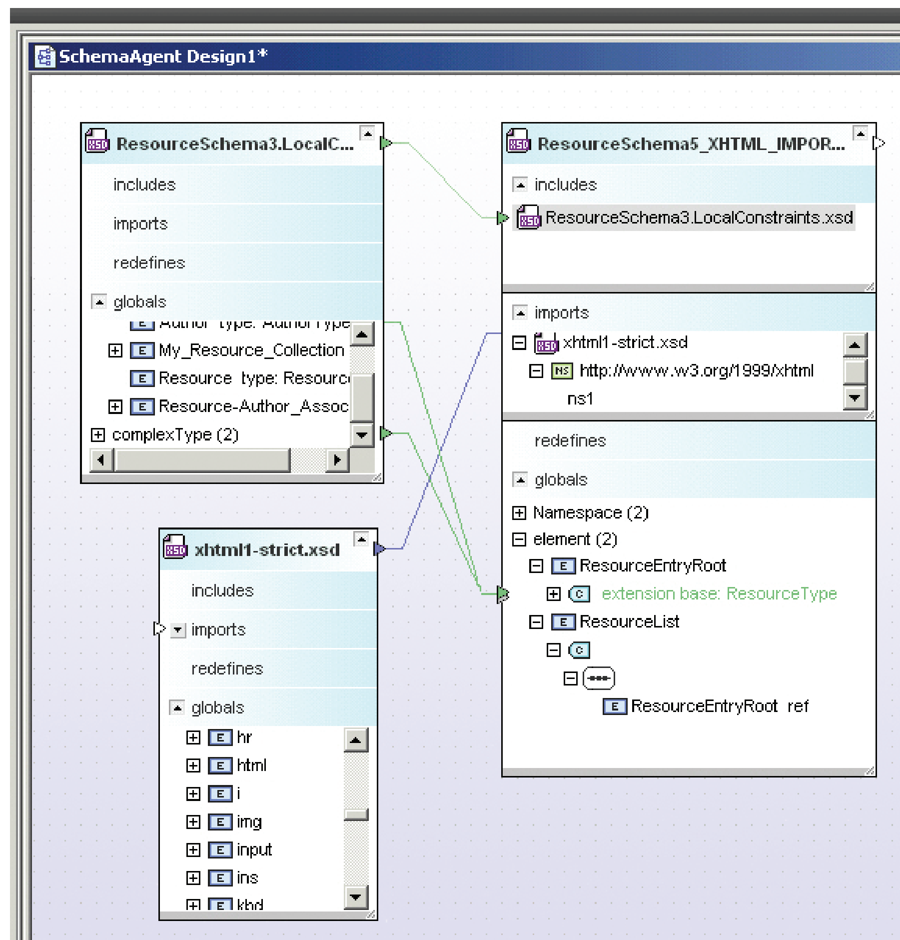

Figure 9. The Altova SchemaAgent view of our ***Resource*** *data entry schema showing the inclusion of our pre-existing* ***Resource*** *schema and the importation of the W3C defined XHTML XML language*

Using the drag-and-drop functionality in SchemaAgent, I've imported the XSD that defines the XHTML 1.0 vocabulary, the XML markup language that underlies virtually all WWW document creation today (see the XHTML Standard, 2002). This import renders all of the globally-scoped XHTML components available for use in our user-defined *Resource* specification. A cursory examination of the figure reveals the key relationships involved in the development and management of these resources and some of the globally available elements (e.g., the *html* root, the *img* element, etc.) now available for reuse.

© Copyright 2007 Altova® GmbH. All rights reserved.

This example highlights the power of XSD for enabling resource reuse and information asset management. With its document-centric approach to resource integration and management, XSD provides an optimal tailor-made system for defining and scoping components for reuse and integration. Embracing XSD for model specification provides a platform agnostic, universally adopted system for creating model components that can be readily integrated and managed throughout systems of any degree of complexity.

3.3.1 Creating the User Interface

Armed with our data format, we're now ready to create the user interface (UI). We'll use Altova StyleVision to create a stylesheet that can be used to create an electronic form in Authentic, Altova's freeform data entry/editor application (a free tool). StyleVision uses an XSD-defined data model along with a sample XML instance document to enable developers to rapidly and easily create electronic forms, which, in turn, enable end users to view and edit subsequent XML instances without being exposed to the underlying XML technology. Figure 10 shows the design of a user interface based on the *Resource* data entry model of our example.

© Copyright 2007 Altova® GmbH. All rights reserved.

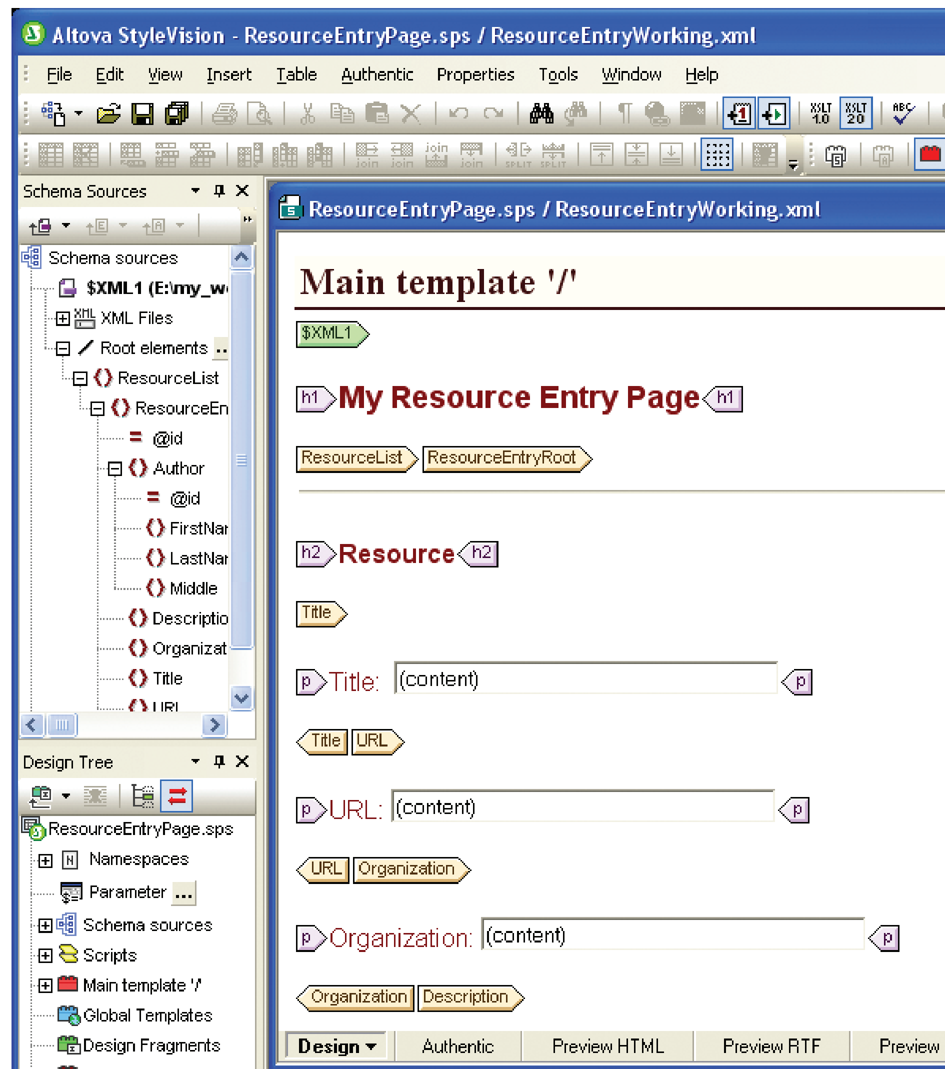

*Figure 10. The **Resource** data entry page design*

The model structure is visible as a tree in the Schema Sources helper window to the left of the graphical user interface (GUI), while the main display area shows the electronic form design achieved by dragging and dropping elements from the data model onto the page.

© Copyright 2007 Altova® GmbH. All rights reserved.

Figure 11 shows XML content being edited in the Authentic electronic form.

*Figure 11. The **Resource** data entry electronic form in action*

© Copyright 2007 Altova® GmbH. All rights reserved.

Once the *Resource* data has been entered into the Authentic electronic form, it is automatically formatted into an XML instance document according to its associated *Resource* data entry model. We now have a sample listing of the XML autogenerated from the data entered in the Authentic electronic form.

```
--------------------------------------
<ResourceList ...>
        <ResourceEntryRoot id="BLANK">
                <Title>3D Application Development
                        with J2ME</Title>
                <URL>
                  http://www.mobilizedsoftware.com/
                </URL>
                <Description>
                  Article describing the
                  framework for 3D application
                  development using J2ME.
                </Description>
                <Author id="BLANK">
                        <FirstName>Aaron</FirstName>
                        <LastName>Walsh</LastName>
                        <Middle>A.</Middle>
                </Author>
                <Author id="BLANK">
                        <FirstName>Harold</FirstName>
                        <LastName>Nagel</LastName>
                        <Middle>N.</Middle>
                </Author>
        </ResourceEntryRoot>
        <ResourceEntryRoot id="">
                <Title>Cheeses of the World</Title>
--------------------------------------
```

Figure 12. *An XML fragment from a sample Resource instance*

This example shows a sample of the XML data autogenerated during a test run of the *Resource* data entry page.

© Copyright 2007 Altova® GmbH. All rights reserved.

3.3.2 Mapping XML to RDBMS

Up to this point we've gotten quite a bit of mileage out of our XSD data model. We've generated a database schema from the model, and also used it to create a user interface to facilitate data entry. All that remains in generating an end-to-end solution is to map the XML output from our entry page to the database. We can easily accomplish this using Altova MapForce. MapForce provides a two-dimensional visual interface that, among other things, permits the use of XSD-specified models to quickly and intuitively create solutions enabling the mapping and/or migration of data from any number of different source objects to any number of targets. Figure 13 illustrates the mapping between our *Resource* XML and the database we created earlier.

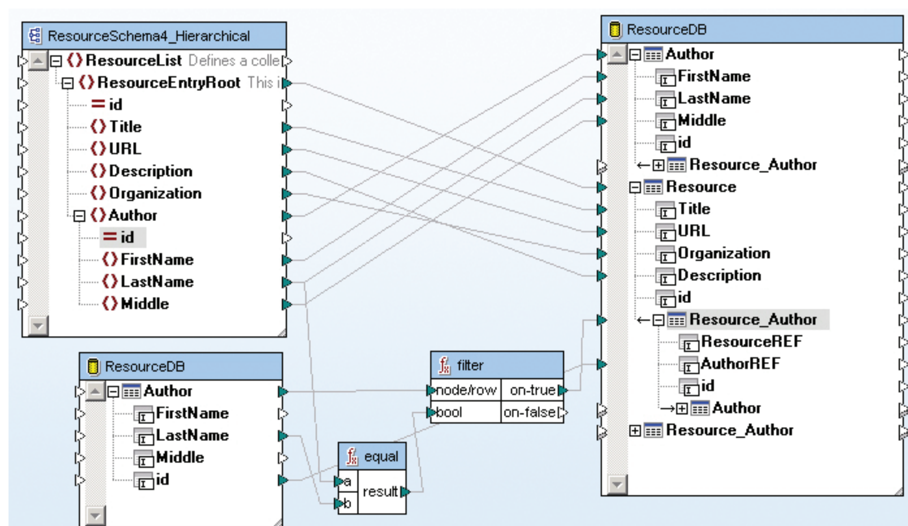

*Figure 13. Mapping the **Resource XML** to the database*

In this example we have mapped the data entry fields from the hierarchical XML onto the relational tables defined by our aggregate schema. In order to preserve the relationships implicit in the hierarchical structure of the XML, we map the *id* fields from *Resource* instances and their corresponding *Authors* onto the references defined on the *Resource_Author* join table. Notice that not just one but two connections to the database appear in the mapping. This is because I've altered the database to use autogenerated primary keys for all table entities. Consequently, in order to map the *Author* primary key field onto the

© Copyright 2007 Altova® GmbH. All rights reserved.

join table as a foreign key, we need to retrieve the value from the database after it's been generated. We can do this by including a second connection to the database as a data source in our mapping.

This completes the mapping from the XML obtained from the entry page to the RDBMS on the back-end. In walking through this example, we've seen how the XSD data model provides necessary and sufficient information to achieve a considerable degree of autogenerated code on multiple tiers of a distributed system, which both cuts development time and allows non-technical users access to points within the work-flow.

4. Charting the Model with UML

The preceding examples have shown the power of leveraging XSD as a data modeling language, and how visual development systems such as that available in Altova XMLSpy, MapForce, and StyleVision can dramatically facilitate the formal specification of entities comprising virtually any domain. That said, project requirements may demand additional clarification and/or documentation. The good news is that your XSD model is fully compatible with – and in fact complements – a Unified Modeling Language (UML) analysis.

To illustrate, let's look at another view of our current example using Altova UModel, a graphical tool that facilitates and enhances the architectural analysis of complex information processing systems. Using UModel, we can easily obtain multiple diagrams of our XSD-defined data model simply by importing the schema documents. Figure 14 shows a UML diagram rendered from our *Resource Data Entry* structure.

© Copyright 2007 Altova® GmbH. All rights reserved.

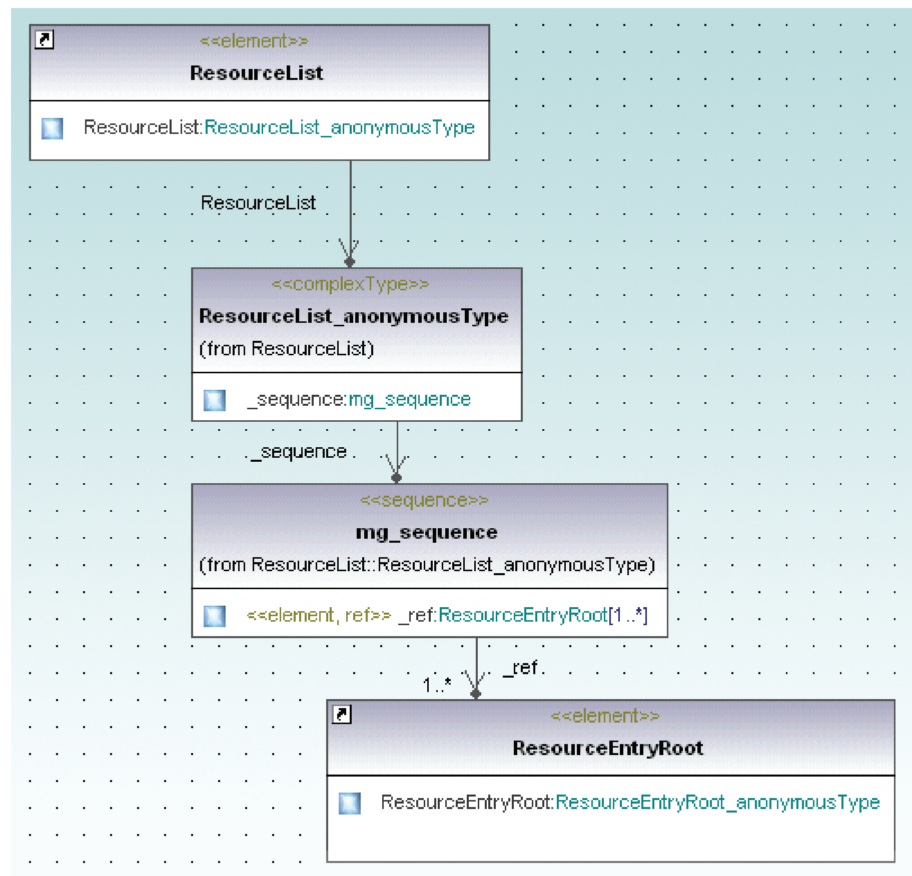

Figure 14. A UML diagram of the ***Resource Data Entry*** *structure*

This UML diagram highlights the composite nature of the schema. Here we see that the *ResourceList* element contains an anonymous type definition (it is defined as an XSD *complexType*), which, in turn, comprises a sequence of one-to-many *ResourceEntryRoot* elements. The *ResourceEntryRoot* itself is a composite the structure of which is revealed in the *ResourceEntryRoot* diagram (Figure 15).

© Copyright 2007 Altova® GmbH. All rights reserved.

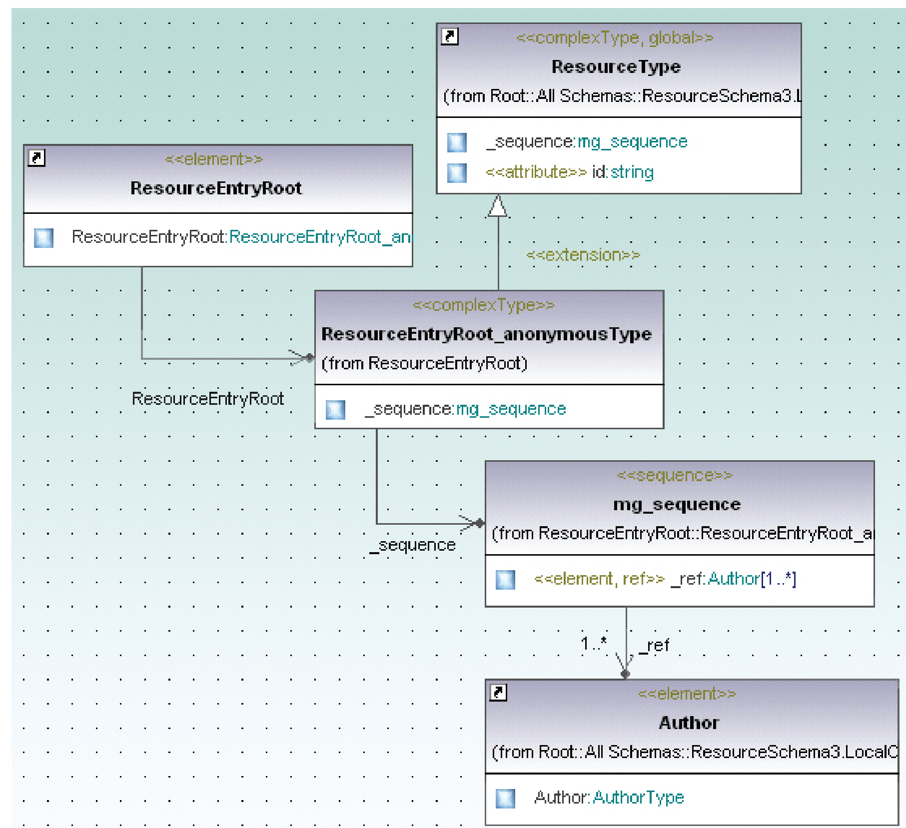

*Figure 15. The **ResourceEntryRoot** structure as shown in a UML diagram*

These UML diagrams highlight relationships that might not be immediately obvious from cursory analysis of the XSD. For example, here we see that the *ResourceEntryRoot* is also an anonymous type. Beyond that, the UML diagram goes on to show that the *ResourceEntryRoot* type definition is also an extension of the global *ResourceType* (Recall that we noted this object oriented relationship earlier when we derived the type). The extension (an added sequence of one-to-many *Authors*) is also readily apparent in the UML representation.

© Copyright 2007 Altova® GmbH. All rights reserved.

5. Conclusion

Having walked through these exercises in data modeling and application development using XSD, we've examined the potential role that this system might play in guiding and facilitating complex information processing system development. The approach illustrated here may be regarded as a “top-down” or “design-first” approach to distributed application development wherein we specify our model components using XSD and use the model to drive development across the various tiers of a distributed application.

The examples we've seen here serve to reveal and highlight the many benefits associated with this approach to development and implied separation of concerns. In particular we've seen that an XSD-centric approach to data modeling fully supports data-centric application development, object oriented modeling, and object-relational mapping. Most importantly, XSD does all this in a completely platform agnostic manner. In addition to these benefits, we've seen that XSD provides an excellent system for information asset management. With document-focused development and the judicious application of well-established development patterns, XSD meets all the requirements necessary for cataloguing and managing an organization's information resources. This, coupled with the industry-wide adoption and support XSD has gained in the years since its inception, enable virtually universal interoperability. The list of benefits goes on, but, at this point it should be easy to see that an initial investment in an XSD-centric approach to development can have huge payoffs in the end.

It should be apparent from this discussion that far from being static and cascade-like, XSD development is highly dynamic and amenable to changing and evolving requirements. While modularity cannot be guaranteed by any system of development per se, we've seen that many of the features built in to XSD foster modular component development. Furthermore, the XSD-centric approach illustrated here renders it far easier to propagate changes throughout a system should conditions dictate that the model must change, than an approach lacking a formal model specification.

© Copyright 2007 Altova® GmbH. All rights reserved.

The approach to development illustrated here is consistent with a paradigm shift that has emerged in enterprise application development in recent years: that of operating on XML-defined data models with business logic encapsulated in modular, object oriented processing components. We've seen this in the evolution of the J2EE specification (see the EJB 3.0 specification, 2006) where model components are defined as “entity beans”, the properties of which are specified in XML deployment descriptors (enabling the auto generation of the model logic at deploy time) and in the emergence of the Service Oriented Architecture (SOA) where XSD is used explicitly to specify data structures suitable for exchange among processing components defined using course-grained APIs (see the Web Services Interoperability Basic Profile, 2004). The .NET architecture is completely retooled for an XML-centric approach to development along these lines. In the future, the role of the XSD-specified data model will only become more critical as XML-based data-processing specifications such as XPath 2.0 and XQuery achieve widespread adoption. Though we've really just barely scratched the surface here in terms of the potential for XSD development, from this vantage point, it appears that this technology stands poised to drive development efforts for years to come.

Please note: the Altova tools mentioned in this whitepaper are all available individually or as part of the Altova MissionKit product bundle. Free 30-day trials of all products mentioned in this paper are available for download from the Altova Web site: **www.altova.com/download**

© Copyright 2007 Altova® GmbH. All rights reserved.

References

1. Booch, Grady; Rumbaugh, James; and Jacobson, Ivar.
The Unified Modeling Language User Guide.
Addison Wesley Professional, 2005.

2. Sun Microsystems. *The Java Language Specification.*
http://java.sun.com/docs/books/jls/third_edition/html/classes.html#8.9
1996-2005 (Last visited 2007).

3. Sun Microsystems. *The EJB 3.0 Specification* (JSR-000220).
http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html
2006.

4. Web Services Interoperability Organization. *Basic Profile Version 1.0.*
http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html 2004
(Last visited 2007).

5. World Wide Web Consortium. *The XSD Specification.*
http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/
2004 (Last visited 2007).

6. World Wide Web Consortium. *The XHTML Standard.*
http://www.w3.org/TR/xhtml1/
2002 (Last visited 2007).

7. Altova, XML, data management, UML, and Web services tools:
http://www.altova.com

The information contained in this document represents the current view of Altova with respect to the subject matter herein contained as of the date of the publication. Altova makes no commitment to keep the information contained herein up to date and the information contained in this document is subject to change without notice. As Altova GMBH must respond to the changing market conditions, Altova GMBH cannot guarantee the accuracy of any information presented after the date of publication. The document is presented for informational purposes only.

ALTOVA PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OF IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Altova®, XMLSpy®, MapForce®, StyleVision®, UModel®, DatabaseSpy™, DiffDog®, SchemaAgent®, SemanticWorks®, ACXE, AltovaXML™, and Authentic® are trademarks and/or registered trademarks of Altova GmbH in the United States of America, the European Union, and numerous other countries. Other brands may be trademarks or registered trademarks of others.

© Copyright 2007 Altova® GmbH. All rights reserved.