Design XML schemas for enterprise data

Use W3C XML Schema features to design data formats for production management

Skill Level: Intermediate

Bilal Siddiqui (xml4java@yahoo.co.uk)
Consultant
Freelance

03 Oct 2006

This tutorial teaches you how to use W3C XML Schema features in different types of enterprise applications. You'll learn when, why, and how to use simple and complex types, regular expressions, unions, lists, and substitution groups while designing data formats for your enterprise applications. You'll also learn how to build multiple file schemas, use external schemas in your XML design, and reuse other schema designers' experience by deploying XML design patterns.

Section 1. Before you start

About this tutorial

This tutorial demonstrates the use of W3C XML Schema features in defining data formats for enterprise applications.

First, you'll learn the major types of enterprise applications and their data-interchange requirements, and why you need to define schema for enterprise XML data. You'll also learn the use of patterns to design high-level business documents and XML structures in enterprise applications. Then, using production data as an example, you will start to design an enterprise XML schema that uses various XML Schema features in enterprise data design. You'll learn:

- How to define and extend complex types
- When and how to define abstract complex types and abstract elements

- How to use regular expressions (string patterns), enumerations, unions, lists, and substitution groups

You'll also learn why and how to develop multiple file schemas, and about using external schemas to design your enterprise data. Finally, you will put the pieces together to design high-level business documents.

Prerequisites

You should be able to write well formed XML 1.0 documents. You should also have a beginner-level understanding of XML schemas. This includes the ability to use simple types in XML schema to build complex types. See Resources for links to material you can read to fulfill these prerequisites.

Should I take this tutorial?

The tutorial will be of value for you if you want to learn when and why to use important features in the XML Schema specification to design production-grade XML schemas for enterprise applications. You can also benefit from this tutorial if you want to learn how to reuse the experience of other schema designers.

Tutorial topics

The remainder of this tutorial is organized in the following sections:

- Explanation of types of enterprise applications and their requirements for data interchange. This section also explains why you need XML schema in your enterprise applications and introduces the use of XML design patterns.
- Demonstration of how to build complex types to represent production resources. This section also demonstrates how to group XML elements for substitution.
- Explanation of using regular expressions, enumerations, unions, lists, and substitution groups.
- Demonstration of developing a schema that spans multiple files. This section also explains how to use external schemas while you design your own schema.
- Demonstration of how to build a schema for high-level business documents comprising basic XML structures. This section also demonstrates the use of abstract elements with substitution groups to build an XML template.
- Wrap-up.

Code samples and installation requirements

A simple Java application named `InstanceVerifier` is included in the source code download for this tutorial (see Download). You'll use `InstanceVerifier` to validate XML instance documents against the XML schema you develop in this tutorial. The source code download also includes several XML instance documents to help you understand schema concepts.

`InstanceVerifier` uses XML schema support that comes with Java Development Kit (JDK) version 1.5. So, you must download and install JDK 1.5 from the Sun Web site to try the schema and instance documents developed in this tutorial (see Resources).

Section 2. XML for enterprise data

Types of enterprise applications

This tutorial demonstrates the design of an XML schema for an enterprise-scale production-management system. "Enterprise-scale" means that the production-management schema can fulfill the major requirements of the following types of enterprise applications:

- **Enterprise Resource Planning (ERP)** applications manage all the resources of an enterprise, including human resources, production equipment, inventory, marketing, and financial resources. (For links to some ERP products, including an open source implementation, see Resources.)

  Different types of ERP modules and applications manage different types of enterprise resources. For example, an accounting module handles a company's financial resources, and a production-management module handles production-related resources.

  Normally it is difficult for any ERP vendor to produce best-of-the-breed solutions for all types of ERP modules. This means that an enterprise normally has ERP modules or applications from a variety of vendors deployed for different purposes across the enterprise.

- **Enterprise Application Integration (EAI)** applications help different ERP modules coordinate and communicate with one another.

Most enterprises develop, implement, and host different modules of their ERP solutions over a period of time. This gives rise to the issue of integrating ERP modules so that different modules can exchange application data with one another.

For example, a production-management module might need to know inventory of raw materials available to carry out a particular production activity, so it must communicate with the enterprise's inventory-management module. Or, if the required inventory is unavailable, the production-management module might interact with a purchase-management module to place orders.

- **Supply Chain Management (SCM)** applications control and monitor the interaction of an enterprise with external actors such as its vendors, customers, and partners. (To learn more about supply chains in the health care industry, see Resources.)

  SCM applications help ERP modules to collaborate with external actors. For example, an inventory-management module uses an SCM application to talk to vendors in order to coordinate material sourcing activity. You can say that organizations that make up a supply chain are linked together through SCM applications.

- **Business Intelligence (BI)** applications provide high-level assistance to managers trying to make business decisions.

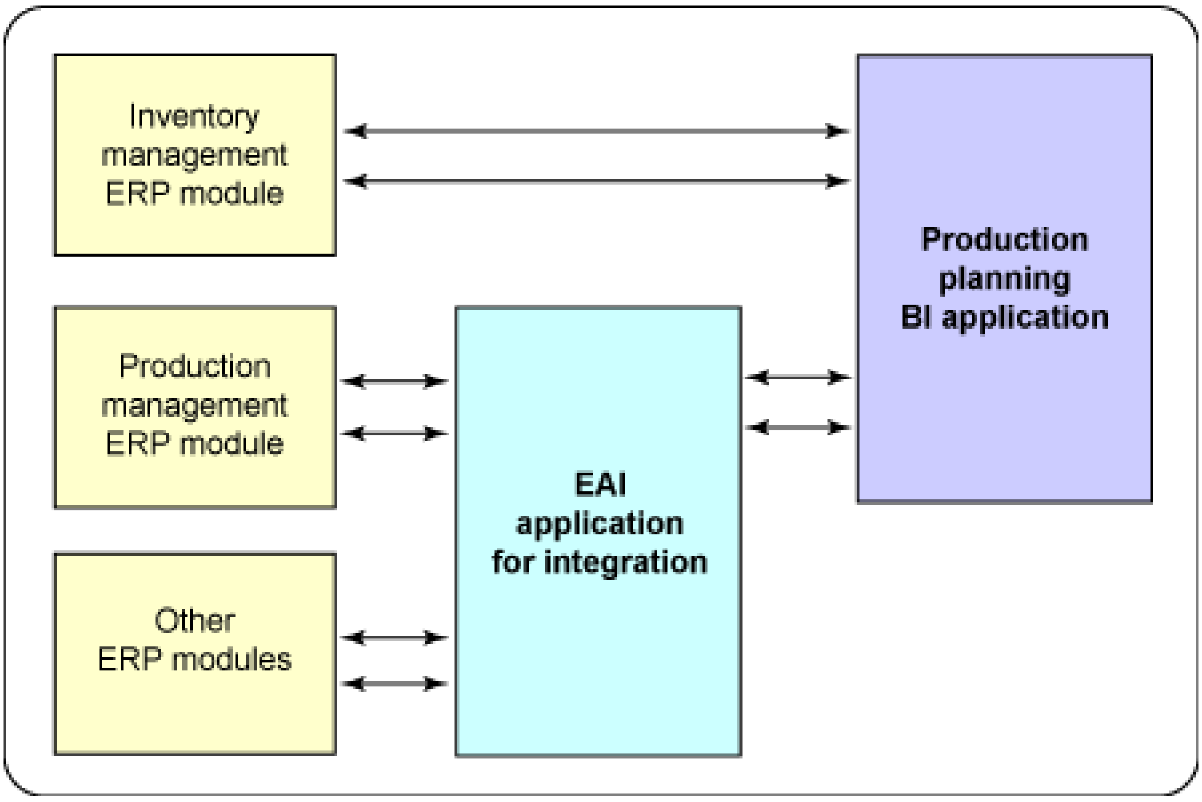
  BI helps track and manage information available within different ERP modules. For example, suppose an enterprise receives an important order that requires earliest execution. The resources (machines, personnel, raw materials, and components) required to execute the order are already reserved for previous orders currently in execution. The production manager wants to squeeze the new order into its existing production schedule. This requires checking which resources need reallocation and how this reallocation will affect execution of existing orders. A production-planning BI application can help in this scenario by providing graphical view of the production hall and even simulate the effects of change in the production plan to fit in a new order.

Data design for an enterprise

Different types of enterprise applications need to communicate and interoperate with one another for their operations. For example, consider the production-planning BI application scenario in Types of enterprise applications.

The BI application might need to coordinate and communicate with different ERP modules, either directly or perhaps through some EAI module, as shown in Figure 1.

Figure 1: Different types of enterprise applications coordinating and communicating with one another


Inventory management ERP module
Production management ERP module
Other ERP modules
EAI application for integration
Production planning BI application

Several techniques enable data interchange and integration across enterprise applications. (See Resources for a link to an IBM white paper on this topic.) The most straightforward and popular integration technique is to design common data formats that different applications can use to talk to, and exchange data with, one another.

XML is a popular language for defining formats for structured data. Applications can wrap complex structures of data in XML format and exchange XML documents to enable interoperability with other applications.

For example, a BI application might need to know a machine's current deployment status -- whether the machine is in operation (and if so, which job is currently assigned to the machine) or is out of order. In this case, you'd need to define several XML structures, such as those that specify machines and machine deployment.

XML instances and schema definitions

Enterprise applications can exchange XML documents to communicate and interoperate with one another. For example, Listing 1 shows an XML document that describes a machine in a production hall. (You will learn more about the XML document in Listing 1 in the next section, under Specifying the deployment details of a resource.)

Listing 1. An XML document describing a production machine

```
<?xml version="1.0" encoding="UTF-8"?>
<pms:Machine
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"

    name="mixingmachine"
    identifier="01"
    type="machine"
    make="AFictitiousMachineCompany"
    model="AFMC2001">

    <pms:Deployment
        xsi:type="pms:MachineCurrentDeploymentType"
        status="IN_PRODUCTION">
            <pms:WorkOrderRef>WorkOrder1</pms:WorkOrderRef>
    </pms:Deployment>

    <pms:OperationProtocol>
        <!--Details of the operation protocol-->
    </pms:OperationProtocol>
</pms:Machine>
```

Normally, XML documents are also referred to as *instances* of XML data.

An application that receives an XML instance parses and processes the instance to extract required information. For example, the `pms:MachineCurrentDeployment` element in Listing 1 has an attribute named `status` whose value (`IN_PRODUCTION`) specifies the machine's current deployment status.

Notice that the XML instance in Listing 1 has a definite structure and format. Later (in Building a vocabulary for production management), you will learn that even an individual attribute value in an XML instance can have a well-defined format.

XML-processing applications need to make sure that data coming from an external interoperating application is structured according to a specified format. If an XML instance doesn't adhere to the format, the processing application might not be able to extract required information from the XML instance.

Several techniques have been developed to specify the format of an XML instance. One of the techniques is standardized by the W3C in the XML Schema specification (see Resources).

According to the XML Schema specification, you need to write schema documents to specify the format for XML instance documents. XML-processing applications validate instance documents against schema documents using tools called *validators*. Various commercial and open source validators are available. For this tutorial, you'll use a free, open source validating XML parser called Apache Xerces (see Resources).

As you develop the production-management schema in this tutorial, you'll learn how to use many XML Schema features. The accompanying source code package includes many instance documents along with a simple Java application named `InstanceVerifier`, which uses Xerces to check whether an instance document is structured according to the production-management schema (see Download).

During the course of this tutorial, you'll also learn that the XML Schema specification has many powerful features that enable XML developers to define validation rules. This relieves you from tedious low-level programming effort you'd otherwise expend to check the format of XML instances.

Work orders for production management

In XML instances and schema definitions, you saw the role that XML can play in integrating different types of enterprise applications. You also saw the main purpose of designing a schema before starting to use XML in your enterprise applications.

Now you'll start to prepare to design an XML schema you'll build for a manufacturing enterprise's production-management system. This schema will serve different enterprise applications that monitor or control various activities in a production hall.

Activities in a production hall are generally controlled through *work orders*. A work order is a business document whose implementation results in production activities. For example, consider the following enterprise scenario:

1. The marketing department receives a purchase order from a customer. The purchase order contains a list of items to be produced.
2. The marketing department sends the purchase order to the production department.
3. The production department evaluates the activities (production, packaging, labeling, and so on) that need to be performed in the production hall in order to fulfill the purchase order's requirements.
4. Each activity required by the purchase order needs a work order. So the production department issues work orders for different activities as required by the purchase order.
5. The production department also needs to plan the execution of work orders. For example, some of the resources (such as machines or assembly lines) in the production hall might already be assigned to other work orders. In that case, the production department tries to manage the use of available resources to fit the required activities into the production hall's current ongoing activities.

Defining the structure of a work order

Enterprise operations depend on the execution of business documents such as purchase orders and work orders.

Just as operations in the production hall are controlled by means of work orders, so the production-management schema needs to define a work order's structure and format. A work order contains the following bits of information:

- Some metadata, such as:
  - The type of work order (production order or maintenance order, for example)
  - An identifier to identify the work order
  - Priority level (for example, whether it needs to be urgently executed or can be executed within normal routine)
  - Date of issuance
- Details of the required work to be executed. For example, if it is a production order, it specifies the products and quantities to be produced.
- Details of the resources allocated to fulfill the work order's requirements. For example, in case of a production order, a quantity of personnel, machines, or assembly lines might be allocated to fulfill the order. This portion of a work order isn't filled at the time the order is issued. Instead, it's filled when resources are allocated during the planning of the work-order's execution.

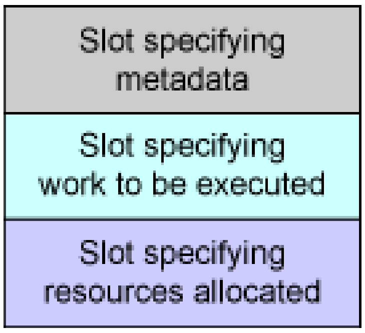
Designing a family of work orders

In addition to the two types of work orders you've seen -- maintenance orders and production orders -- real production-management systems can use other types of work orders as well. For example, a quality-inspection order might be issued to inspect a consignment's quality. However, all the different types of work orders contain the same three bits of information mentioned in Defining the structure of a work order.

This means that when you design the production-management schema, you need to design a family of related business documents.

One possible way of designing a family of documents is to design each document separately. A better technique is to design a template document with so-called slots. Each slot serves a specific purpose, as shown in Figure 2.

Figure 2. A family of business documents as a template with slots


Slot specifying metadata
Slot specifying work to be executed
Slot specifying resources allocated

Once you've designed the template with slots, you'll design families of XML structures to fit into each slot of the template.

For example, consider a family of XML structures meant to fit into the middle slot of Figure 2. If you design an XML structure to fit into the middle slot of a production order, your structure will specify the products and quantities to be produced. If you design the structure to fit into the middle slot of a maintenance order, your structure will specify the machine to be repaired and details of the maintenance required.

The main advantage of this technique is that some of the XML structures will be common across different types of work orders. For example, the metadata slot in production and maintenance orders can be the same. This helps you reuse some of your XML structures.

The XML Schema specification contains some powerful features that you can use to define a document template with slots. In the final section of this tutorial, Designing a work order, you'll use these features to design such a template. For now, I'll show you techniques for designing schemas, called *schema design patterns*.

Patterns in XML schema design

If you solve a particular design problem with a specific background (or context) using a particular technique, you can apply the same solution whenever you come across the same problem in a similar context.

For example, consider the template-with-slots solution that you just saw in Designing a family of work orders. Whenever you need to design a family of related XML documents in which you can identify bits of related of information (that you can reuse across different documents in the family), you can use the template-with-slots technique.

Reusing proven solutions in this manner is called *applying design patterns*. Patterns are powerful tools to enable reuse of experience. You can identify patterns by understanding design problems in their specific contexts. To learn more about patterns in XML design, see Resources.

As you proceed with the design of the production-management schema, you can identify design patterns such as the template-with-slots pattern. This will help you reuse the design concepts presented in this tutorial to solve your own XML design problems.

You can apply some very useful and popular design patterns in Java programming to XML design. I will provide an example later in Working with large and multiple schemas (under Using the Composite pattern to design schema for a production process) to demonstrate how to use Java-based design patterns in XML schema design.

A data model

Designing a family of work orders introduced you to the idea of designing documents templates with slots. Now I'll discuss the strategy for designing individual structures to fit into each slot of the work-order template you saw in Figure 2.

Consider the example of an XML structure that is to fit into the middle slot of a production order. Such an XML structure should specify the products and quantities that need to be produced. Naturally, while designing such a structure, you need to design smaller structures to specify products and quantities.

Now consider an XML structure that fits into the lower slot of a production order. This structure specifies the resources allocated to execute the production order. While designing this structure, you need to design smaller structures to specify:

- Machines (a type of manufacturing resource)
- Assembly lines (another type of manufacturing resource used to assemble components and semifinished products into finished products)
- Raw materials (a consumable resource)
- Personnel (human resources) required to fulfill the production order

Designing smaller structures such as resources, machines, products, and personnel gives rise to the idea of defining a *data model*. The data model consists of basic building blocks. Later you can design business documents based on the data model.

You can find real-world examples of building data models for XML documents. For example, see Resources for a link to the official Web site of Health Level 7 (HL-7), an organization that builds standards for the health care industry. The data model called Reference Information Model (RIM) contains the basic structures for the health care industry. Companies working in the health care industry use RIM to define higher-level business documents and interact with health care authorities.

The rest of this tutorial is about designing a data model for the production-management system and putting the pieces together into larger structures that eventually fit into slots of the document template in Figure 2.

Section 3. Designing a schema for production hall resources

Defining XML syntax for a production resource

A production hall consists of *production resources* that are used to execute *production processes*.

Examples of production resources include machines, assembly lines, and raw materials. Examples of production processes are weighing, mixing, packaging, and labeling.

In this section you'll develop a schema for production resources in a file named resources.xsd. Later, in Working with large and multiple schemas, you'll see how to build a schema for production processes in a file named processes.xsd.

A production hall can contain many types of resources, such as:

- Human resources (machine operators, quality testers, supervisors, and so on)
- Machines (such as weighing machines, mixing machines, tableting machines, and packaging machines)
- Materials (raw materials that are consumed in a manufacturing process)

All these production resources have names. For example, a machine that makes tablets might be called a *tableting machine*, and the person who operates a tableting machine might be called a *tableting machine operator*.

Similarly, each resource has a unique identifier that identifies that resource among all other resources in the enterprise. For example, the tableting machine has a machine identifier assigned to it, and the tableting machine operator has an employee identifier. You can call this a *resource identifier.*

Each resource also has a *type*. For example, the tableting machine is a resource of the *machine* type.

Names, identifiers, and types are *attributes* of a resource. A resource can have other attributes as well.

While writing the schema for the production-management system, you can define a data type named `ResourceType`. Name, identifier, and type fields will become attributes of `ResourceType`. Listing 2 shows how to define `ResourceType` along with its attributes using an XML schema:

Listing 2: Schema definition of ResourceType

```
<xsd:schema
    targetNamespace=
        "http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    elementFormDefault="qualified">

    <xsd:complexType name="ResourceType">
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="identifier" type="xsd:string"/>
        <xsd:attribute name="type" type="xsd:string"/>
        <!--Other attributes of a resource-->
    </xsd:complexType>
</xsd:schema>
```

You can see that the schema definition of `ResourceType` in Listing 2 contains `xsd:schema`, `xsd:complexType`, and `xsd:attribute` elements. The `xsd` prefix is normally used to specify the XML schema definition namespace (`http://www.w3.org/2001/XMLSchema`) defined by the W3C XML Schema specification.

You use these `xsd` elements to define XML attributes and elements. For links to some articles you can read to learn the purpose of these elements, see Resources.

Understanding XML schema elements

`xsd:schema` is the root element in all schema files. The `xsd:schema` element in Listing 2 has two attributes:

- The **`targetNamespace`** attribute defines the namespace URI of your schema. In Listing 2, the value of `targetNamespace` is `http://www.AFictitiousEnterprise.com/ProductionManagement`, which is the namespace URI of the production-management schema that you are building in this tutorial.
- The **`elementFormDefault`** attribute specifies whether or not an instance document is required to use qualified names consisting of both the prefix and the local name (such as `pms:Deployment`) for all elements. In this tutorial you'll use `elementFormDefault="qualified"`, which means all elements in instance documents should use qualified names.

The `xsd:schema` element in Listing 2 has two namespace declarations:

- The `xmlns:xsd` definition is for the XML schema definition namespace
- `xmlns:pms` definition is for the production-management system namespace that you are building in this tutorial.

An `xsd:complexType` element defines the structure of an element that contains attributes and other child elements. Look at the `xsd:complexType` element in Listing 2, which describes `ResourceType` as a complex XML data type. The `name` attribute of the `xsd:complexType` element defines the name of the complex type (`ResourceType` in this case).

A complex data type is composed of simple data types, such as `string`, `integer`, and `float`. For example, `ResourceType` contains attributes of a resource (`name`, `identifier`, `type`, and so on), where each attribute is a `xsd:string` simple type. You will use `ResourceType` (which is a complex type) to build other complex types in the next subsection, Defining different types of resources.

An `xsd:attribute` element specifies an attribute of a complex type. For example, Listing 2 contains one `xsd:attribute` element each for the `name`, `identifier`, and `type` attributes.

Defining different types of resources

Now you have the schema definition for a generic production resource in the form of a complex type named `ResourceType`. The schema definition of `ResourceType` contains the attributes that are common to all types of resources. However, different types of resources can contain some attributes and child elements that are not common to all types of resources.

For example, to specify a machine requires that you specify its `make` and `model`. A human resource does not require these attributes.

This means you must specialize the schema definition of a resource to create separate schema definitions for different type of resources. XML Schema lets you extend your complex types. So now you will extend `ResourceType` to design the schema definition of a machine.

Look at the schema definition for a complex type named `MachineType` in Listing 3:

Listing 3. Schema definition for MachineType

```
<xsd:schema
    targetNamespace=
        "http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    elementFormDefault="qualified">

    <xsd:complexType name="ResourceType">
      <!--Components of ResourceType-->
    </xsd:complexType>

    <xsd:complexType name="MachineType">
        <xsd:complexContent>
            <xsd:extension base="pms:ResourceType">
                <xsd:sequence>
                    <xsd:element name="OperationProtocol">
                       <!--Details of the operation protocol-->
                    </xsd:element>
                </xsd:sequence>
                <xsd:attribute name="make" type="xsd:string"/>
                <xsd:attribute name="model" type="xsd:string"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:element name="Machine" type="pms:MachineType"/>

</xsd:schema>
```

The `MachineType` complex type has a child named `xsd:complexContent`, which in turn has a child named `xsd:extension`. Whenever you wish to extend an existing data type, you can use this combination of `xsd:complexContent` and `xsd:extension` elements.

The `xsd:extension` element in Listing 3 has a `base` attribute whose value is `pms:ResourceType`. This means `MachineType` extends `ResourceType`, which you developed earlier in Listing 2. Therefore, `MachineType` inherits all attributes and child elements of `ResourceType`.

In addition to all the attributes and children of `ResourceType`, the `MachineType` in Listing 3 also contains its own attributes (such as `make` and `model`) and child elements (such as `OperationProtocol`).

Creating an instance according to a complex type

You have seen the `ResourceType` and `MachineType` complex types. Now you'll see how to instantiate XML elements that follow a given complex type.

Look at `xsd:element` in Listing 3 (in the last line, just before the `</xsd:schema>` end tag). Its `name` attribute has a value of `Machine`, and its `type` attribute has a value of `pms:MachineType`. This means that `xsd:element` defines an element named `Machine` that follows the structure defined by `MachineType`.

Listing 4 shows what a valid `Machine` instance looks like. You can see that all attributes and elements defined in `ResourceType` and `MachineType` are included in the `Machine` element in Listing 4:

Listing 4: A valid Machine instance

```
<?xml version="1.0" encoding="UTF-8"?>
<pms:Machine
    xmlns:pms=
        "http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"

    name="mixingmachine"
    identifier="01"
    type="machine"
    make="AFictitiousManufacturingCompany"
    model="AFMC2001">

    <pms:OperationProtocol>
       <!--Details of the operation protocol-->
    <pms:OperationProtocol>
</pms:Machine>
```

Note that the instance document in Listing 4 includes two namespace declarations (`xmln:pms` and `xmlns:xsi`). `xmlns:pms` is the production-management namespace that the instance document is using. The `xmlns:xsi` namespace (whose value is `http://www.w3.org/2001/XMLSchema-instance` in Listing 4) is defined by the XML Schema specification. This namespace contains some schema attributes that are meant to appear in instance documents. For example, Listing 4 contains an `xsi:schemaLocation` attribute, which specifies the location of a schema file.

I have included both valid and invalid instances of the `Machine` element in this tutorial's source code download (see Download). You can use the `InstanceVerifier` application to check the validity of instance documents against the production-management schema.

Similarly, you can define a `PersonType` complex type to represent a human resource, as shown in Listing 5:

Listing 5: Schema definition of a human resource

```
<xsd:schema
    targetNamespace=
        "http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    elementFormDefault="qualified">

    <!--Other schema definitions-->

    <xsd:complexType name="ResourceType">
      <!--Components of ResourceType-->
    </xsd:complexType>

    <xsd:complexType name="PersonType">
        <xsd:complexContent>
           <xsd:extension base="pms:ResourceType">
               <xsd:sequence>
                  <xsd:element name="PersonalDetails">
                  <xsd:complexType>
                      <xsd:sequence>
                         <xsd:element name="Name" type="xsd:string"/>
                         <xsd:element name="Address">
                            <!--Fields of the address-->
                         </xsd:element>
                         <!--Other personal details-->
                      </xsd:sequence>
                  </xsd:complexType>
                  </xsd:element>
               </xsd:sequence>
           </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:element name="Person" type="pms:PersonType"/>

    <!--Other schema definitions-->

</xsd:schema>
```

`PersonType` extends `ResourceType` and also contains the details of employees working in the production department. For example, the `PersonalDetails` child of `PersonType` wraps an employee's personal information, such as name and address.

Listing 5 also contains the definition of an element named `Person`, which uses `PersonType`. You'll use `Person` element in your instance documents, as shown in Listing 6:

Listing 6: An instance of the Person element

```
<?xml version="1.0" encoding="UTF-8"?>
<pms:Person
    xmlns:pms=
        "http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"

    name="Bob"
    type="HumanResource"
    id="H032">

    <pms:PersonalDetails>
        <pms:Name>Bob</pms:Name>
        <pms:Address>
          <!--Fields of the address-->
        </pms:Address>
    </pms:PersonalDetails>

</pms:Person>
```

Making ResourceType abstract

You have seen instances of the `Machine` and `Person` elements in Listings 4 and 6, respectively.

What about creating an instance of the `ResourceType` definition in Listing 2? Perhaps the only purpose of having the `ResourceType` definition is to contain attributes and elements common to all types of resources. An instance of `ResourceType` will therefore not represent any particular resource, and you will never need to instantiate `ResourceType`.

In such situations, you might want to define `ResourceType` as *abstract*, which means the schema does not let you instantiate the complex type.

To declare the `ResourceType` as abstract, you just need an attribute named `abstract` with a value of `true` in the schema definition of `ResourceType`. This is shown in Listing 7:

Listing 7: The abstract ResourceType complex type

```
<!--Other schema definitions-->

<xsd:complexType
    name="ResourceType"
    abstract="true">
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="identifier" type="xsd:string"/>
    <xsd:attribute name="type" type="xsd:string"/>
</xsd:complexType>

<!--Other schema definitions-->
```

Although you have declared `ResourceType` to be abstract and its instance cannot appear in an XML document, XML Schema still lets you declare an element that uses `ResourceType`. For example, Listing 8 defines an element named `Resource`, which uses `ResourceType`:

Listing 8: Defining the Resource element

```
<!--Other schema definitions-->

<xsd:complexType
    name="ResourceType"
    abstract="true">
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="identifier" type="xsd:string"/>
    <xsd:attribute name="type" type="xsd:string"/>
</xsd:complexType>

<xsd:element name="Resource" type="pms:ResourceType"/>

<!--Other schema definitions-->
```

You might wonder when and how you'd use the `Resource` element. Because `ResourceType` cannot be instantiated, the `Resource` element will always represent one of the elements extending `ResourceType` (for example, `Machine` or `Person`).

In this case, a `Resource` instance also needs to specify which element it is representing. The XML Schema specification provides an attribute named `xsi:type`, which appears in an instance document and specifies the complex type that an element represents. For example, look at the `Resource` instance in Listing 9:

Listing 9: A valid Resource instance

```


<?xml version="1.0" encoding="UTF-8"?>
<pms:Resource
    xmlns:pms=
        "http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"
    xsi:type="pms:MachineType"

    name="mixingmachine"
    identifier="01"
    type="machine"
    make="AFictitiousManufacturingCompany"
    model="AFMC2001">

    <pms:OperationProtocol>
       <!--Details of the operation protocol-->
    <pms:OperationProtocol>

</pms:Resource>
```

Later (in Listing 29), you will use an instance of the `Resource` element while defining a production process.

Specifying a resource's deployment details

Take a look at Listing 10, which is a slightly enhanced form of the `ResourceType` definition:

Listing 10: ResourceType with deployment definition

```
<!--Other schema definitions-->

<xsd:complexType name="ResourceType" abstract="true">
    <!--Components of ResourceType-->
    <xsd:sequence>
        <xsd:element name="Deployment" type="pms:DeploymentType"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="DeploymentType" abstract="true">
    <xsd:attribute name="status" type="xsd:string"/>
    <!--Other deployment details-->
</xsd:complexType>

<!--Other schema definitions-->
```

The only enhancement is the inclusion of an element named `Deployment`. The purpose of the `Deployment` child element is to specify the deployment details of the resource (that is, the work order on which the resource is deployed).

The `Deployment` child of `ResourceType` in Listing 10 uses a data type called `DeploymentType` (also defined in Listing 10), which contains an attribute named `status`.

The `status` attribute uses the `string` simple type and specifies the deployment status of the resource (for example, whether the resource is currently deployed on a work order, is idle, or is out of order).

Note that Listing 10 keeps `DeploymentType` abstract. That's because the deployment of different resources might need different structures. For example, the deployment of a machine will be on specific work orders, while the deployment of a machine operator will be on a specific machine. Similarly, a production supervisor might be deployed on a group of machines or perhaps on an assembly line.

This means that while defining the deployment of a particular type of resource, you will extend from `DeploymentType`. For example, Listing 11 shows a `MachineCurrentDeployment` element, which uses a complex type named `MachineCurrentDeploymentType` that extends `DeploymentType`:

Listing 11: The MachineCurrentDeploymentType definition

```
<!--other schema definitions-->

<xsd:complexType name="DeploymentType" abstract="true">
    <xsd:attribute name="status" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="MachineCurrentDeploymentType">
       <xsd:complexContent>
            <xsd:extension base="pms:DeploymentType">
                <xsd:sequence>
                    <xsd:element name="WorkOrderRef" type="xsd:string"/>
                    <!--Other details of machine deployment-->
                </xsd:sequence>
            </xsd:extension>
       </xsd:complexContent>
</xsd:complexType>
```

```
<xsd:element name="MachineCurrentDeployment"
          type="pms:MachineCurrentDeploymentType"/>

<!--other schema definitions-->
```

Note that you have not enhanced the schema definition for `MachineType`. You have instead enhanced its parent complex type (`ResourceType`) to include the `Deployment` element. In addition, you have extended `DeploymentType` to define a specific type of deployment suitable for specifying a machine's current deployment.

By doing so, you let the `Machine` element wrap an instance of the `MachineCurrentDeploymentType`, as shown in the instance document in Listing 12:

Listing 12: An instance of the Machine element containing its deployment description

```
<?xml version="1.0" encoding="UTF-8"?>
<pms:Machine
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"

    name="mixingmachine"
    identifier="01"
    type="machine"
    make="AFictitiousMachineCompany"
    model="AFMC2001">

    <pms:Deployment
        xsi:type="pms:MachineCurrentDeploymentType"
        status="IN_PRODUCTION">
        <pms:WorkOrderRef>WorkOrder1</pms:WorkOrderRef>
    </pms:Deployment>

    <pms:OperationProtocol>
        <!--Details of the operation protocol-->
    </pms:OperationProtocol>
</pms:Machine>
```

In Listing 12, look at the `pms:Deployment` element, which has a `pms:WorkOrderRef` child element. The schema definition of the `pms:Deployment` element in Listing 10 does not contain an element named `pms:WorkOrderRef`. This means the `pms:Deployment` element in Listing 12 is not an instance of the `pms:Deployment` element defined in Listing 10. Rather, it is an instance of the `pms:MachineCurrentDeploymentType` in Listing 11.

This is the reason Listing 12 includes an `xsi:type` attribute, whose value (`pms:MachineCurrentDeploymentType`) specifies the data type of the `pms:Deployment` element.

Just as `pms:MachineCurrentDeploymentType` extends `pms:DeploymentType`, so an instance of `pms:Deployment` element can represent a `pms:MachineCurrentDeploymentType` structure in this manner.

Specifying alternates for deployment

Sometimes, you might need your instance of the `Machine` element to contain some alternate of the `Deployment` element. For example, a production manager might want to check a particular machine's deployment history. The history can be specified in a `History` element containing a number of `Deployment` elements.

Each `Deployment` element specifies a deployment in the history. The definition of such a `History` element is shown in Listing 13:

Listing 13: The definition of the History element

```
<!--Other schema definitions-->

<xsd:complexType name="HistoryType">
     <xsd:sequence>
        <xsd:element
             ref="pms:MachineCurrentDeployment"
             minOccurs="0"
             maxOccurs="unbounded"/>
        </xsd:sequence>
     <xsd:attribute
        name="machineID"
        type="xsd:string"
          use="optional"/>
</xsd:complexType>

<xsd:element name="History" type="pms:HistoryType" />

<!--Other schema definitions-->
```

In this case, you can see that there is no need for the `History` element to directly contain attributes defined in the `Deployment` element. Instead the `History` element wraps a number of `Deployment` instances. This is shown in Listing 14:

Listing 14: An instance of the history element

```
<pms:History
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"
    machineID="01">

    <pms:MachineCurrentDeployment>
        <pms:WorkOrderRef>WorkOrder1</pms:WorkOrderRef>
    </pms:MachineCurrentDeployment>
    <pms:MachineCurrentDeployment>
        <pms:WorkOrderRef>WorkOrder2</pms:WorkOrderRef>
    </pms:MachineCurrentDeployment>
    <pms:MachineCurrentDeployment>
        <pms:WorkOrderRef>WorkOrder3</pms:WorkOrderRef>
    </pms:MachineCurrentDeployment>

</pms:History>
```

You want your `Machine` element to wrap the `History` element, as shown in Listing 15:

Listing 15: A Machine instance shown wrapping a History element

```
<pms:Machine
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"
    name="mixingmachine"
    identifier="01"
    type="machine"
    make="AFictitiousMachineCompany"
    model="AFMC2001">

    <pms:History>
        <pms:MachineCurrentDeployment>
            <pms:WorkOrderRef>WorkOrder1</pms:WorkOrderRef>
        </pms:MachineCurrentDeployment>
        <pms:MachineCurrentDeployment>
            <pms:WorkOrderRef>WorkOrder2</pms:WorkOrderRef>
        </pms:MachineCurrentDeployment>
        <pms:MachineCurrentDeployment>
            <pms:WorkOrderRef>WorkOrder3</pms:WorkOrderRef>
        </pms:MachineCurrentDeployment>
    </pms:History>

    <pms:OperationProtocol>
        <!--Details of the operation protocol-->
    </pms:OperationProtocol>

</pms:Machine>
```

However, Listing 15 has a problem. It contains a `History` element as a direct child of the `Machine` element. The `Machine` and `Resource` schema definitions don't allow this. Instead, the `Resource` definition requires a `Deployment` element. If you try to validate Listing 15 against the schema developed so far, the validation will fail.

So how do you overcome this problem? Somewhere in your schema you need to define that the `History` element can replace the `Deployment` or `MachineCurrentDeployment` element.

With the XML Schema specification, you can specify alternates for an element. For this purpose you will group your elements for substitution, as explained in the next subsection, Grouping elements for substitution.

Grouping elements for substitution

Take a look at Listing 16, which shows a number of schema enhancements required to let an instance of the `History` element occur inside a `Machine` element:

Listing 16: Enhanced form of the History element

```
<!--Other schema definitions-->
<xsd:element name="EmptyDeployment"/>

<xsd:complexType name="ResourceType" abstract="true">
    <xsd:sequence>
        <xsd:element ref="pms:EmptyDeployment"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="identifier" type="xsd:string"/>
    <xsd:attribute name="type" type="xsd:string"/>
</xsd:complexType>

<xsd:element
    name="Deployment"
    type="pms:DeploymentType"
    substitutionGroup="pms:EmptyDeployment"/>

<xsd:element
    name="MachineCurrentDeployment"
    type="pms:MachineCurrentDeploymentType"
    substitutionGroup="pms:EmptyDeployment"/>

<xsd:element name="History" type="pms:HistoryType"
   substitutionGroup="pms:EmptyDeployment"/>
<!--Other schema definitions-->
```

XML Schema lets `xsd:element` declarations contain an attribute named `substitutionGroup`. The `substitutionGroup` attribute specifies that an element carrying the `substitutionGroup` attribute can substitute for the element specified by the `substitutionGroup` value.

In Listing 16, `xsd:element` declarations for the `Deployment`, `MachineCurrentDeployment`, and `History` elements contain the `substitutionGroup` attribute. In all three cases, the value of the `substitutionGroup` attribute is `pms:EmptyDeployment`. This means an instance document can contain a `Deployment`, `MachineCurrentDeployment`, or `History` element instead of `EmptyDeployment`.

The `ResourceType` definition in Listing 16 is also enhanced to use `EmptyDeployment` instead of `Deployment`, so that different elements (such as `Deployment`) can substitute for `EmptyDeployment`.

Now look at the `EmptyDeployment` declaration in Listing 16. It does not contain a `type` attribute, which means you have not defined its data type.

Now try to validate the instance in Listing 15 against the latest schema file of Listing 16 (instead of the previous version of the schema). The validation will succeed.

Whenever you want unrelated elements to substitute for each other, you can create an element without defining its data type (such as `EmptyDeployment` element) and use `substitutionGroup` attribute in all elements that are supposed to substitute for each other.

Section 4. Building a vocabulary for production management

Specifying a pattern for status codes

In the preceding section (under Specifying a resource's deployment details) you saw the `Deployment` element in Listing 10. Now, you'll enhance the `Deployment` element through various XML Schema features (including restrictions, enumerations, unions, and lists) to control precisely the content wrapped in an XML instance document.

Suppose an enterprise uses a string format (for example, 1521-PROD-OUTOFORDER) to represent different types of status codes for use all across the enterprise. For example, recall the `Deployment` element's `status` attribute in Listing 10, which specifies a machine's deployment status. You can specify the deployment status using the enterprise-wide string format.

The data type of the `status` attribute in Listing 10 was `xsd:string`, which means the `status` attribute's value can be any string. You can enhance the `status` attribute to ensure that it is structured according to the enterprise-wide string format.

XML Schema defines a feature called *regular expressions*, which you can use to specify string formats. To specify a particular format, you will use a regular expression while defining your own simple types in the XML schema.

To define a customized simple type, restrict `xsd:string` to follow a particular regular expression. Your simple type will consist of strings, but restricted to follow certain rules.

Listing 17 defines a simple type:

Listing 17: Simple type definition for the enterprise-wide code string format

```
<!--Other schema definitions-->

<xsd:simpleType name="GenericCode">
    <xsd:restriction base="xsd:string">
       <xsd:pattern
           value="[0-9]{4}-([A-Z]|[a-z]){4}-([A-Z]|[a-z]){1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<!--Other schema definitions-->
```

Notice in Listing 17 that you use a `xsd:simpleType` element to define your own simple types. The `xsd:simpleType` element's name attribute in Listing 17 specifies `GenericCode`, which is the name of this simple type.

This `GenericCode` simple type uses an `xsd:restriction` element to define a restriction. The `xsd:restriction` element has a base attribute with a value of `xsd:string`. This means the restriction is based on the `xsd:string` data type. In other words, you can say that the simple type using this restriction creates a subset of `xsd:string`.

The result of this restriction is that the simple type will allow only string type content. If the `xsd:restriction` element were based on the `xsd:int` data type, for example, this simple type definition would have allowed only integer type content.

Now look at the `xsd:pattern` child of the `xsd:restriction` element in Listing 17. Its `value` attribute contains a strange-looking string -- `[0-9]{4}-([A-Z]|[a-z]){4}-([A-Z]|[a-z]){1,12}` -- which is a regular expression. Read on to find out how regular expressions work.

Regular expressions in XML schema

Regular expressions let XML programmers specify patterns in textual data. For example, the regular expression shown in Listing 17 (`[0-9]{4}-([A-Z]|[a-z]){4}-([A-Z]|[a-z]){1,12}`) specifies a pattern with a maximum of 22 characters:

- The first four characters are digits (0 through 9) followed by a hyphen (-).
- The hyphen is followed by four uppercase or lowercase letters followed by another hyphen.
- The pattern ends with a maximum of 12 uppercase or lowercase letters.

For example, the following strings match the regular expression described above:

- 1421-PROD-INPRODUCTION
- 1521-PROD-OUTOFORDER
- 7421-ACCT-PENDING

As you can easily guess, such patterns are helpful in controlling the format of different types of codes that convey useful information. For example, the first two expressions convey the deployment status of machines in the production department. The third expression conveys the status of some pending account.

This also demonstrates the usefulness of XML Schema validation features. By specifying the format (or pattern) of an expression in your XML schema, you free your application programmers from writing low-level type-checking code.

Using patterns to specify deployment status

You have designed a pattern to be used in different codes all across the enterprise. But you have not linked your pattern to any element in your schema. To use the pattern, you will use the `GenericCode` simple type in Listing 17 while defining some element in your schema.

For example, recall the `status` attribute of the `Deployment` element in Listing 10. Its value in Listing 10 was `xsd:string`. Now you can simply change the value of the `status` attribute to `pms:GenericCode`, as shown in Listing 18. Now the restriction and pattern defined in the `GenericCode` is applicable to the `status` attribute value.

Listing 18: Enhanced deployment with pattern-based status code

```
<!--Other schema definitions-->

<xsd:complexType name="DeploymentType" abstract="true">
    <xsd:attribute name="status" type="pms:GenericCode"/>
     <!--Other deployment details-->
</xsd:complexType>


<xsd:simpleType name="GenericCode">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{4}-([A-Z]|[a-z]){4}-([A-Z]|[a-z]){1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<!--Other schema definitions-->
```

Narrowing down a pattern specification

Some readers might feel that it's inappropriate to use the generic code directly as a deployment status code. Deployment status is specific to the production department, while generic codes are to be used across all across the enterprise.

XML Schema offers a mechanism to specify subsets (or more specific forms) of patterns already defined. You can design deployment-status codes as a subset of the generic enterprise-wide codes.

Look at Listing 19 to see how you can extend the `GenericCode` simple type and restrict it to a subset. It shows the definition of a simple type named `ProductionDeploymentCode`:

Listing 19: The definition of deployment codes for the production department

```
<!--Other schema definitions-->

<xsd:simpleType name="GenericCode">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{4}-([A-Z]|[a-z]){4}-([A-Z]|[a-z]){1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="ProductionDeploymentCode">
    <xsd:restriction base="pms:GenericCode">
        <xsd:pattern value="[0-9]{4}-PROD-([A-Z]|[a-z]){1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:element name="Deployment">
    <xsd:complexType>
        <xsd:attribute name="status" type="pms:ProductionDeploymentCode"/>
    </xsd:complexType>
</xsd:element>
<!--Other schema definitions-->
```

Note that the `ProductionDeploymentCode` simple type definition in Listing 19 contains a `xsd:restriction` element, just like the `GenericCode` simple type definition in Listing 17. However, the `GenericCode` simple type was based on `xsd:string`. The `ProductionDeploymentCode` is based on the `GenericCode` simple type. This is why the base attribute of the `xsd:restriction` element in the `ProductionDeploymentCode` definition has a value of `pms:GenericCode`.

Now look at the `xsd:pattern` child of `ProductionDeploymentCode` in Listing 19. Its `value` attribute has a value of `[0-9]{4}-PROD-([A-Z]|[a-z]){1,12}`. You can see that the middle four-character portion of the generic code has been restricted to `PROD`. The other two portions remain the same.

The first two (1421-PROD-INPRODUCTION and 1521-PROD-OUTOFORDER) of the three expressions that you saw in Regular expressions in XML schema fulfill the pattern of Listing 19, while the third expression (7421-ACCT-PENDING) doesn't.

This example provides a simple demonstration of defining a generic pattern and then creating subsets of the generic pattern. Each subset serves a particular purpose.

Creating a union of multiple patterns

Sometimes you might need to work with multiple expressions that serve the same purpose. For example, consider an EAI application in which you are integrating multiple ERP solutions. Each ERP uses different types of codes to specify various bits of business-process information, so you need to use different formats as generic code. Expressions that match any of the formats will be considered valid.

In this case, you can perform the following two steps in order to implement a union of different formats.

First, create simple type definitions to specify regular expressions according to individual code formats. For example, Listing 20 shows two simple type definitions named `FirstGenericCode` and `SecondGenericCode`:

Listing 20: Simple type definitions for two different generic codes

```
<!--Other schema definitions-->

<xsd:simpleType name="FirstGenericCode">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{4}-PROD-[a-z]{1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="SecondGenericCode">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{4}-PRODUCTION-[a-z]{1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<!--Other schema definitions-->
```

Second, create a union of the two simple types. Listing 21 shows the union of the `FirstGenericCode` and `SecondGenericCode` simple types (named `GenericCode`):

Listing 21: A union of two simple types

```
<!--Other schema definitions-->

<xsd:simpleType name="FirstGenericCode">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{4}-PROD-[a-z]{1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="SecondGenericCode">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{4}-PRODUCTION-[a-z]{1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="GenericCode">
    <xsd:union
        memberTypes="pms:FirstGenericCode pms:SecondGenericCode"/>
</xsd:simpleType>

<!--Other schema definitions-->
```

You can see that the `GenericCode` definition in Listing 21 contains an `xsd:union` element, which has an attribute named `memberTypes`. The `memberTypes` attribute contains a space-separated list of simple types.

As you can guess, all the simple types whose names appear in the `memberTypes` attribute value form the union.

Now you know how to create a union of different patterns or formats. You can also create unions of other simple types, as you'll see at the end of the next subsection, Specifying fixed values.

Specifying fixed values

You saw how to combine different patterns using `xsd:union`. However, sometimes you might need to work with fixed expressions that do not follow any pattern. Such fixed expressions can originate from existing ERP applications or from business-process design.

Suppose you have a list of codes that convey different bits of information. You are required to accommodate the same codes in your schema design because users of your schema are familiar with the codes.

In this case you need to design simple types that contain enumerations of hardcoded fixed codes that do not follow any expression format. Listing 22 shows the definition of a simple type named `FixedCodes`, which contains a number of fixed (hardcoded) codes:

Listing 22: A simple type definition containing a number of hardcoded enumerations

```
<!--Other schema definitions-->
```

```
<xsd:simpleType name="FixedCodes">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="DEPLOYED_ON_WORKORDER"/>
        <xsd:enumeration value="OUT_OF_ORDER"/>
        <xsd:enumeration value="OUT_FOR_MAINTENANCE"/>
        <xsd:enumeration value="OUT_FOR_CALIBRATION_TEST"/>
        <!--Other fixed codes-->
    </xsd:restriction>
</xsd:simpleType>
<!--Other schema definitions-->
```

Now you can combine the `FirstGenericCode`, `SecondGenericCode`, and `FixedCodes` simple types together as a union in the `GenericCode` simple type, as shown in Listing 23:

Listing 23: Creating a union of three simple types

```
<!--Other schema definitions-->

<xsd:simpleType name="FirstGenericCode">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{4}-PROD-[a-z]{1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="SecondGenericCode">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{4}-PRODUCTION-[a-z]{1,12}"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="FixedCodes">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="DEPLOYED_ON_WORKORDER"/>
        <xsd:enumeration value="OUT_OF_ORDER"/>
        <xsd:enumeration value="OUT_FOR_MAINTENANCE"/>
        <xsd:enumeration value="OUT_FOR_CALIBRATION_TEST"/>
        <!--Other fixed codes-->
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="GenericCode">
    <xsd:union memberTypes="pms:FirstGenericCode
             pms:SecondGenericCode pms:FixedCodes"/>
</xsd:simpleType>

<!--Other schema definitions-->
```

The `GenericCode` simple type in Listing 23 now allows patterns that match the first and second types of generic code patterns developed in Listings 17 and 20, respectively, as well as the fixed codes in Listing 22.

Working with lists of simple types

You saw how to use regular expressions, unions, and enumerations in designing different codes to be used across your enterprise.

The same idea can be applied in several different aspects of schema design. One common application is to use these concepts in designing unique identifiers to identify different resources and processes in the enterprise.

For example, you might want to design the format for unique identifiers, consisting of a regular expression with multiple portions (for example, PROD-4321-2817). Different portions of the identifier format can serve different purposes. For example, one portion can identify whether it is a resource identifier or a process identifier, another portion can identify its type, and so on.

An example of such multiple-portion identifiers is shown in the simple type named `GenericIdentifier` in Listing 24:

Listing 24: Definition of GenericIdentifier

```
<!--Other schema definitions-->
<xsd:simpleType name="GenericIdentifier">
    <xsd:restriction
         base="xsd:string">
        <xsd:pattern
            value="([A-Z]|[a-z]){4}-[0-9]{4}-[0-9]{4}"/>
    </xsd:restriction>
</xsd:simpleType>
<!--Other schema definitions-->
```

Now suppose a BI application wants to know all the resources currently deployed on a particular work order. This requires the preparation of a list of resource identifiers, where each identifier identifies a single resource deployed on the work order.

XML Schema lets you design lists of simple types. One example is shown in Listing 25, in which a simple type named `ListOfGenericIdentifiers` consists of a list of `GenericIdentifier` simple type values from Listing 24:

Listing 25: Defining a list of GenericIdentifier values

```
<!--Other schema definitions-->
<xsd:simpleType name="GenericIdentifier">
    <xsd:restriction base="xsd:string">
       <xsd:pattern value="([A-Z]|[a-z]){4}-[0-9]{4}-[0-9]{4}"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="ListOfGenericIdentifiers">
    <xsd:list itemType="pms:GenericIdentifier"/>
</xsd:simpleType>

<xsd:element name="ListOfResources"
    type="pms:ListOfGenericIdentifiers"/>
<!--Other schema definitions-->
```

You can see from Listing 25 that defining a list of simple types is very easy. You just need an `xsd:list` element inside an `xsd:simpleType` element. The `xsd:list` element contains an attribute named `itemType`, which identifies the simple type whose values are supposed to comprise the list.

Just as the `itemType` attribute value in Listing 25 is `pms:GenericIdentifier`, so `ListOfGenericIdentifiers` will consist of a list of strings according to the pattern defined by the `GenericIdentifier` simple type.

You might wonder what an instance of a list will look like. Listing 25 includes an `xsd:element` named `ListOfResources` that uses the `ListOfGenericIdentifiers` simple type. Listing 26 shows an instance of the `ListOfResources` element, which simply wraps a list of space-separated generic identifiers:

Listing 26: An instance of the ListOfResources element

```
<?xml version="1.0" encoding="UTF-8"?>
<pms:ListOfResources
    xmlns:pms=
        "http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd">
        PROD-1000-1432 PROD-1200-2432 PROD-1100-3456
</pms:ListOfResources>
```

This concludes the demonstration on using regular expressions, restrictions, enumerations, unions, and lists. The schema developed so far is included in this tutorial's source code download (see Download). The source code download also includes several valid and invalid instance documents to help you understand schema concepts.

Section 5. Working with large and multiple schemas

Managing size of schema files

In the previous couple of sections you used various features of XML Schema to define the format for different types of resources used in a production hall. This section discusses how to represent production processes (such as mixing and packaging) using XML Schema.

However, the file containing the production schema is growing in size and might become difficult to handle. The XML Schema specification lets you divide your schema into multiple files.

Difficulty in handling a schema file is not the only reason to divide your schema into multiple files. Another important reason is that different parts of your XML schema might be controlled and maintained by different people or departments in your enterprise. For example, the schema to represent machines might be maintained by the production department, while the schema for personnel might be controlled by the human resources department, and the schema for defining manufacturing processes by the engineering design department. This means that different departments will have different access rights for different portions of the schema. For example, the production department will have access to view and use the human resources schema, but will not be allowed to modify it. Only the human resources department will have the right to modify the human resources schema. Dividing the schema into multiple files can help in implementing separate access rights for different portions of the schema.

Building a multiple-file schema

You'll design the schema for production processes in a separate schema file named processes.xsd. However, the namespace of the processes schema will remain the same as the namespace of the resources schema (`http://www.AFictitiousEnterprise.com/ProductionManagement`).

Note that the processes schema file needs to use the schema definitions of production resources. That's because manufacturing processes require resources (such as machines and assembly lines).

Therefore, the processes schema will *include* the resources.xsd file that you saw earlier in Designing schema for production hall resources. Look at Listing 27, which shows how to include a schema file into another schema file:

Listing 27: Including resources.xsd into the processes schema

```
<xsd:schema
    targetNamespace=http://www.AFictitiousEnterprise.com/ProductionManagement
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    elementFormDefault="qualified">
    <xsd:include
      schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"/>
    <!--Schema definition of production processes-->
</xsd:schema>
```

Note that Listing 27 has an `xsd:include` element with a `schemaLocation` attribute. The `schemaLocation` attribute specifies the URL of a schema file (`http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd` in this case).

When a schema validator comes across an `xsd:include` element in a schema file, it dereferences the URL, fetches the remote schema file, and includes all schema definitions in the dereferenced schema into the file being processed.

This means that you can say that all schema definitions in the resources schema are included in the processes schema in Listing 27. As a result, you can use the `Machine` element in your definitions in the processes schema.

Note an important point. Recall from Defining XML syntax for a production resource that the target namespace declaration in the resources.xsd file was `http://www.AFictitiousEnterprise.com/ProductionManagement`. The target namespace declaration in Listing 27 is the same. Therefore all definitions in the resources.xsd and processes.xsd schema files belong to the same target namespace, namely `http://www.AFictitiousEnterprise.com/ProductionManagement`.

In fact, the `xsd:include` element only includes schema definitions of one file into another file when the target namespace of the two files is the same. If you try to include a schema file into another file with a different target namespace, your schema validator will not accept the schema for validation.

Describing a production process

Having included the resources.xsd file into the processes.xsd file, you are all set to start using the resources schema to build the processes schema.

You'll represent a production process by an XML element named `Process`.

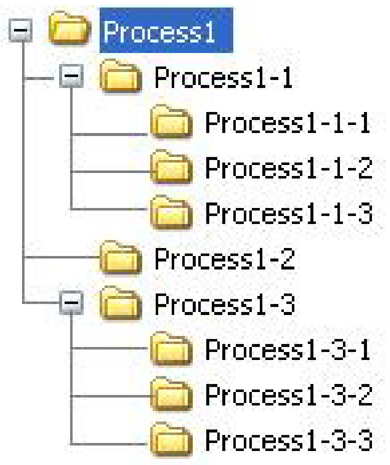
A production process contains other smaller processes, which in turn can contain still smaller processes. For example, consider the process of making a drug in a pharmaceutical manufacturing company. Drug making processes normally consist of a number of subprocesses, such as:

- Chemical processes to make basic pharmaceutical ingredients
- Mixing processes that mix different basic ingredients according to required drug formulation
- Processes that produce a drug in its consumable form (such as a tablet or a capsule)
- Packaging
- Inspection and quality assurance

Now consider the mixing process. Mixing can require a number of smaller processes such as weighing ingredients to be mixed, preparing a batch for mixing, ensuring that ingredients are homogeneously mixed throughout the mixture, and so on.

A process is like a tree of processes, where each process can contain a number of smaller processes, as shown in Figure 3.

Figure 3. A production process as a tree of processes



You can easily see the resemblance of a production process with the familiar hierarchy of folders in a Windows Explorer view.

Whenever you need to design the schema of something that can be represented as a tree of similar structures, you can use a well-known Java design pattern called the Composite pattern. The Composite pattern deals with structures that in turn can contain similar structures, forming a tree of structures. (For useful links to information on several well-known Java patterns, including the Composite pattern, see Resources.)

Next you'll define the schema of a production process using the Composite pattern.

Using the Composite pattern to design a schema for a production process

Listing 28 shows the definition of the `Process` element, which represents a production process:

Listing 28: The definition of Process element

```
<xsd:schema
    targetNamespace="http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    elementFormDefault="qualified">

    <xsd:include
      xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"/>

    <xsd:complexType name="ProcessType">
       <xsd:sequence>
           <xsd:element ref="pms:Resource"
               minOccurs="1"
               maxOccurs="unbounded"/>
           <xsd:element name="Documentation"
               type="xsd:string"/>
           <xsd:element ref="pms:Process"
               minOccurs="0"
               maxOccurs="unbounded"/>
       </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="Process" type="pms:ProcessType" />
</xsd:schema>
```

Note that the `Process` element is defined as a complex type named `ProcessType`, which contains a sequence of three elements:

- The first child of `ProcessType` is an instance of the `Resource` element that you defined earlier in Listing 8. Note the use of `maxOccurs="unbounded"` attribute in Listing 28, which means the number of `Resource` instances in the `Process` element has no upper limit. The `Resource` element inside a `Process` element represents the resource required by the process.

  Recall from Making ResourceType abstract that `ResourceType` is abstract, so it cannot be instantiated. Therefore, `Resource` instances appearing inside a `Process` element will actually represent other elements that extend `ResourceType` (such as `Machine` or `Person`). In a moment you'll see an example of a `Process` instance to show you how a `Resource` instance appears in a `Process` element.

- The second child of `ProcessType` is a `Documentation` element, which provides human- and machine-readable documentation for the process. I will discuss the `Documentation` element in detail later, in Using external namespaces in production management schema.

- The third child of `ProcessType` is the most interesting. It is the same `pms:Process` element that uses the `ProcessType` complex type being defined. This means the schema will allow an instance of the `Process` element to appear within a `Process` element.

  Also note the value of the `minOccurs` attribute in the `xsd:element` declaration for the `pms:Process` element in Listing 28. The value of the `minOccurs` attribute is `0`, which means an instance document might or might not contain a `Process` element inside another `Process` instance.

  You need to allow this `minOccurs="0"` in the schema definition, because the tree in Figure 3 always ends in a leaf process. A leaf process is the one that does not contain further child processes. Such a process will only contain `Resource` and `Documentation` instances. For example, the process of weighing ingredients to be mixed might not contain any child processes. It will only contain resources (such as a weighing machine) and documentation (describing the procedure of weighing). If you don't include the `minOccurs="0"` attribute in the process schema, a `Process` instance will never end, making it impossible to instantiate a valid `Process` element.

Listing 29 shows a valid example instance of the `Process` element:

Listing 29: An instance of the Process element

```
<?xml version="1.0" encoding="UTF-8"?>
<pms:Process
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
      "http://www.AFictitiousEnterprise.com/ProductionManagement/processes.xsd">
    <pms:Resource
        xsi:type="pms:MachineType"
        name="MixingMachine"
        identifier="01"
        type="machine"
        make="AFictitiousManufacturingCompany"
        model="AFMC2001">
        <pms:Deployment
            xsi:type="pms:MachineCurrentDeploymentType"
            status="1421-PROD-INPRODUCTION">
            <pms:WorkOrderRef>WorkOrder1</pms:WorkOrderRef>
        </pms:Deployment>
        <pms:OperationProtocol>
            <!--Details of the operation protocol-->
        </pms:OperationProtocol>
    </pms:Resource>
    <pms:Documentation>Mixing process for granular contents</pms:Documentation>
    <pms:Process>
        <pms:Resource
           xsi:type="pms:MachineType"
           name="WeighingMachine"
           identifier="01"
           type="machine"
           make="AFictitiousManufacturingCompany"
           model="AFMC2005">
           <pms:Deployment
               xsi:type="pms:MachineCurrentDeploymentType"
               status="1421-PROD-INPRODUCTION">
               <pms:WorkOrderRef>WorkOrder1</pms:WorkOrderRef>
           </pms:Deployment>
           <pms:OperationProtocol>
               <!--Details of the operation protocol-->
           </pms:OperationProtocol>
       </pms:Resource>
       <pms:Documentation>Weighing process</pms:Documentation>
    </pms:Process>
</pms:Process>
```

You can also extend the `ProcessType` complex type definition in case some of your processes require additional attributes or child elements. I've already demonstrated how to extend XML schema definitions (in Defining different types of resources), so I'll leave this topic and move on to discussing how to annotate schema definition and XML instances.

Documenting a schema component

XML Schema lets you annotate your schema definition. For example, look at Listing 30, which adds machine- and human-readable annotations to `ProcessType`:

Listing 30: Documented form of ProcessType

```
<xsd:schema
```

```
    targetNamespace=http://www.AFictitiousEnterprise.com/ProductionManagement
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    elementFormDefault="qualified">

    <xsd:include
      xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"/>
    <xsd:complexType name="ProcessType">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
               Process definition describing a production process
            </xsd:documentation>
            <xsd:appInfo>
                ProductionManagementSchema Version 1.7
                LastEdited 20060626
            </xsd:appInfo>
        </xsd:annotation>
        <xsd:sequence>
           <xsd:element ref="pms:Resource"
               minOccurs="1"
               maxOccurs="unbounded"/>
           <xsd:element name="Documentation"
               type="xsd:string"/>
           <xsd:element ref="pms:Process"
               minOccurs="0"
               maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="Process" type="pms:ProcessType" />

    <!--Other schema definitions-->
</xsd:schema>
```

Note the use of the `xsd:annotation` element in Listing 30. This element is used to provide annotations to your schema definitions so that people or machines reading the schema file can understand the purpose of the different element definitions.

An `xsd:annotation` element can contain two child elements: `xsd:documentation` and `xsd:appInfo`.

Note from Listing 30 that an `xsd:documentation` child of the `xsd:annotation` element wraps a human-readable annotation for a schema component. Listing 30 provides a one-line human-readable annotation. It also includes an `xsd:appInfo` element to show that your schema definition can also contain annotations meant for machine consumption, rather than for human consumption.

Many applications of the `xsd:appInfo` element are possible. For example, `xsd:appInfo` can help you in maintaining your schema. Normally you will upgrade only a few of your schema definitions during a maintenance session. This means different components in your schema will have different maintenance histories. You can include the maintenance history of each schema component within an `xsd:appInfo` element wrapped inside an `xsd:annotation` element. Your schema-maintenance or content-management tools will process the `xsd:appInfo` element to learn the history.

Annotating a schema component does not generally affect validation of instance documents.

Using external namespaces in the production-management schema

You have seen how to annotate your schema. Now you'll learn how to incorporate an external (totally independent) schema into your production-management schema to annotate your instance documents.

Listing 31 shows an enhanced form of the `ProcessType` definition:

Listing 31: An enhanced form of ProcessType definition using an external schema to annotate instance documents

```
<xsd:schema
    targetNamespace=http://www.AFictitiousEnterprise.com/ProductionManagement
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    elementFormDefault="qualified"
    xmlns:xhtml=" http://www.w3.org/1999/xhtml">
    <xsd:include
      xsi:schemaLocation=
        "http://www.AFictitiousEnterprise.com/ProductionManagement/resources.xsd"/>
    <xsd:import
    namespace="http://www.w3.org/1999/xhtml"
    schemaLocation="http://www.w3.org/1999/xhtml"/>
    <xsd:complexType name="ProcessType">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
               Process definition describing a production process
        </xsd:documentation>
        <xsd:appInfo>
           ProductionManagementSchema Version 1.7
           LastEdited 20060626
        </xsd:appInfo>
    </xsd:annotation>
    <xsd:sequence>
           <xsd:element ref="pms:Resource"
               minOccurs="1"
               maxOccurs="unbounded"/>
           <xsd:element name="Documentation">
               <xsd:element ref=xhtml:link"/>
           </xsd:element>
           <xsd:element ref="pms:Process"
               minOccurs="0"
               maxOccurs="unbounded"/>
       </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="Process" type="pms:ProcessType" />
<!--Other schema definitions-->
</xsd:schema>
```

Note that Listing 31 contains three enhancements (shown in boldface), as compared to the previous version of the `ProcessType` definition in Listing 30:

- Listing 31 contains an `import` element with two attributes: `namespace` and `schemaLocation`. The `import` element specifies that you want to use an external namespace (that is, other than the target namespace) in your schema file. The `namespace` attribute specifies the namespace URI of the external namespace to be imported. The `schemaLocation` attribute tells where to find the schema file of the external namespace.

Note that value of the `namespace` attribute of the `import` element in Listing 31 is `http://www.w3.org/1999/xhtml`. This specifies the namespace URI of Extensible HyperText Markup Language (XHTML) version 1.0, defined by the W3C. Because you'll use XHTML to annotate the `Process` instance, you import the XHTML schema into the processes schema file.

- Also note that the `xsd:schema` element in Listing 31 has an additional `xmlns:xhtml` declaration, whose value is same as the XHTML namespace URI (`http://www.w3.org/1999/xhtml`). This means you'll use `xhtml` as the namespace prefix for the XHTML namespace.

- Now look at the `Documentation` element in Listing 31, which is a complex type. It has a child `xsd:element` with a `ref` attribute. The `ref` attribute's value is `xhtml:link`. This means the `xhtml:link` element will be a child of the `Documentation` element.

  The `xhtml:link` element contains attributes that define links to external documents. I chose to use `xhtml:link` element because it can refer to external documentation (for example, process documentation residing somewhere in the enterprise's content-management system).

How the `xhtml:link` element works is beyond this tutorial's scope. (For a link to an XHTML tutorial, see Resources.)

An important point here is that the `Documentation` element in the production-management schema is just a placeholder for external namespaces. It wraps the `xhtml:link` XHTML element, which in turn refers to documentation of the production process.

In this and the previous two sections, you used several XML schema concepts to design some basic building blocks of the production-management schema. The next section will put the pieces together in the form of the work order introduced in Figure 2.

Section 6. Designing a work order

Designing an XML-based product catalog

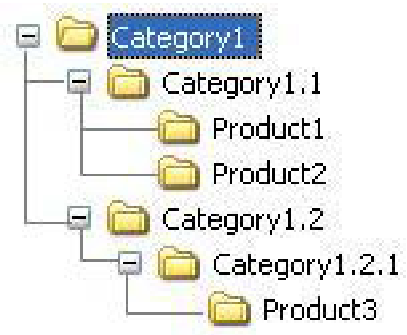
In this section you'll design the XML structures that fit into the slots of the work order that you saw in Figure 2.

But before you can start to design the XML structures, you'll learn about an interesting and useful application of the Composite pattern to design the company's product catalog. Understanding the catalog's design will help you design the work order.

A product catalog contains data about products organized in the form of product categories. Each product category contains products related to the category. Each product category also contains subcategories, which in turn can contain other subcategories. The depth of the category structure has no theoretical limit.

Figure 4 shows a product catalog's tree structure in graphical form.

Figure 4. Tree structure of a product catalog



For example, consider a restaurant menu, which is a type of a product catalog. The menu contains categories like starters, main courses, and desserts. Each category can contain further subcategories and also menu items.

For a link to a tutorial you can take to learn more details of XML-based catalogs, see Resources. The focus here is on using the Composite pattern to design the schema for a product catalog.

Schema for a product catalog

A product catalog has two types of entries: categories and products. `Category` and `Product` elements can represent the two types of entries.

Both types of entries extend from an abstract complex type called `CatalogEntryType`, as shown in Listing 32:

Listing 32: Schema definitions for catalog entries

```
<!--other schema definitions-->
<xsd:complexType name="CatalogEntryType" abstract="true">
    <xsd:sequence>
        <xsd:element name="Identifier" type="xsd:string"/>
        <xsd:element name="Name" type="xsd:string"/>
        <xsd:element name="Reference" type="xsd:string"/>
        <!--Other elements common between ProductType and CategoryType-->
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ProductType">
    <xsd:complexContent>
```

```
        <xsd:extension base="pms:CatalogEntryType">
            <xsd:sequence>
                <xsd:element name="ModelNo" type="xsd:string"/>
                <xsd:element name="ProductDescription"
                    type="xsd:string" />
                <!--Other product details-->
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:element name="Product" type="pms:ProductType"/>

<xsd:complexType name="CategoryType">
    <xsd:complexContent>
        <xsd:extension base="pms:CatalogEntryType">
            <xsd:sequence>
                <xsd:element
                    ref="pms:Category"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
                <xsd:element
                    ref="pms:Product"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
                <!--Other category details-->
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:element name="Category" type="pms:CategoryType"/>

<!--other schema definitions-->
```

The `CatalogEntryType` is abstract and contains child elements that are common between the two types of catalog entries: a unique identifier, name of the entry, a reference (URL) to detailed documentation of the entry, and so on.

The two elements named `Category` and `Product` (defined in Listing 32) extend `CatalogEntryType`. Therefore, both the `Category` and `Product` elements contain child elements of `CatalogEntryType`.

The `Category` element contains zero or more instances of other `Category` elements. It can also contain `Product` elements.

The `Product` element contains details about a product manufactured by the company. It has fields such as the model number and a short product description.

Using the product schema to define product requirements

You have seen the use of several XML Schema features to design a data model for production resources and processes, and how to apply the Composite pattern to design a product catalog for the enterprise.

Now you'll learn how to use the data model and the `Product` element to design an XML structure to fit into slots of the work order in Figure 2.

For example, look at Listing 33, which contains the schema for an XML structure named `ProductRequirements`. `ProductRequirements` uses the schema definition of the `Product` element from Listing 32 to define details of product requirements:

Listing 33: Schema definition for product requirements

```
<!--Other schema definitions-->
<xsd:element name="ProductRequirement">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="pms:Product"/>
            <xsd:element name="Quantity">
                <xsd:complexType>
                    <xsd:attribute name="unit" type="xsd:string" />
                    <xsd:attribute name="value" type="xsd:string" />
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:complexType name="ProductRequirementsType">
    <xsd:sequence>
       <xsd:element
           ref="pms:ProductRequirement"
           maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element
    name="ProductRequirements"
    type="pms:ProductRequirementsType"/>

<!--Other schema definitions-->
```

For your convenience, Listing 34 also provides an instance of the `ProductRequirements` element. This will help you understand the use of the schema definition in Listing 33.

Listing 34: A ProductRequirements instance

```
<pms:ProductRequirements
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
      "http://www.AFictitiousEnterprise.com/ProductionManagement/products.xsd">
    <pms:ProductRequirement>
        <pms:Product>
           <pms:Identifier>
               P001
           </pms:Identifier>
           <pms:Name>
               AFictitiousProductName
           </pms:Name>
           <pms:Reference>
               http://www.AFictitiousEnterprise.com/Products/P001
           </pms:Reference>
           <pms:ModelNo>
               CRN-2150
           </pms:ModelNo>
           <pms:ProductDescription>
               This product is an active ingredient for
               AnotherFictitiousProduct.
           </pms:ProductDescription>
        </pms:Product>
        <pms:Quantity unit="packs" value="150"/>
    </pms:ProductRequirement>
    <!--Other product requirements-->
</pms:ProductRequirements>
```

Note that the `ProductRequirements` element in Listing 34 is just a sequence of `ProductRequirement` elements, where each `ProductRequirement` element in the sequence specifies one requirement. This means the sequence specifies a number of requirements.

Each `ProductRequirement` element contains two child elements: `Product` and `Quantity`. You have already seen the `Product` element in Listing 32. The `Quantity` element specifies the required quantity of a product in a requirement.

The `Quantity` element has two attributes: `value` and `unit`, which together specify the quantity of a product (such as 400kg or 20 packets).

Now refer back to the work order in Figure 2, which contains three slots. The `ProductRequirements` element is suitable to fit into the middle slot of a production order. (Recall from Designing a family of work orders that a production order is a type of work order.)

Next you'll design a work order schema in such a way that other XML structures can fit into its slots.

An extensible work order

Now you'll find out how to fit the `ProductRequirements` element into the work order in Figure 2.

Recall from the discussion accompanying Figure 2 in Designing a family of work orders that the work order's middle slot holds details of the work that the work order requires to be executed. If the work order is a production order, the middle slot specifies the products to be manufactured or produced.

The `ProductRequirements` element of Listing 33 fits well into the middle slot of the work order. But it's not the only element that can go into the middle slot of Figure 2. You might also need to fit other elements -- for example, a `MaintenanceRequirement` element that specifies details of a maintenance job -- into the middle slot.

The enterprise's production-management schema needs to be flexible and extensible. You might need to design new elements in the future that will be required to fit into a slot of the work order. So you need to design the work order in an extensible manner that enables new elements to fit into work-order slots in the future.

The next subsection demonstrates a strategy to design a flexible and extensible work order.

Using substitution groups for extensibility

Listing 35 shows the `WorkOrder` element's schema definition:

Listing 35: Schema definition of a WorkOrder element

```
<!--Other schema definitions-->

<xsd:element name="SchemaSlot" abstract="true"/>
<xsd:element name="WorkDescriptionSlot" abstract="true"/>
<xsd:element name="ResourcesSlot" abstract="true"/>

<xsd:complexType name="WorkOrderType">
    <xsd:sequence>
        <xsd:element ref="pms:SchemaSlot" minOccurs="0"/>
        <xsd:element ref="pms:WorkDescriptionSlot minOccurs="0""/>
        <xsd:element ref="pms:ResourcesSlot minOccurs="0""/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="WorkOrder" type="pms:WorkOrderType" />

<!--Other schema definitions-->
```

Note that the `WorkOrder` element in Listing 35 consists of a `WorkOrderType` complex type that contains three child elements named `SchemaSlot`, `WorkDescriptionSlot`, and `ResourcesSlot`. All three child elements are empty (they have no content) and abstract (they cannot be instantiated).

Now look at Listing 36, which shows the schema definition of an enhanced form of the `ProductRequirements` element that you saw earlier in Listings 33 and 34.

Listing 36: Enhanced form of the ProductRequirements element

```
<!--Other schema definitions-->
<xsd:element
    name="ProductRequirements"
    type="pms:ProductRequirementsType"
    substitutionGroup="pms:WorkDescriptionSlot">
</xsd:element>
<!--Other schema definitions-->
```

Listing 36 contains only one enhancement (compared to Listing 33), which appears in boldface: the addition of a `substitutionGroup` attribute to the `ProductRequirements` element's definition. The value of the `substitutionGroup` attribute in Listing 36 is `pms:WorkDescriptionSlot`.

Recall from Grouping elements for substitution that you can use a `substitutionGroup` attribute in an element definition. Using `substitutionGroup` means the element can be used in place of (instead of) the element specified by value of the `substitutionGroup` attribute. So you can use the `ProductRequirements` element in place of the `WorkDescriptionSlot` element in a work order, as shown in the instance document in Listing 37:

Listing 37. A work order specifying production requirements

```
<?xml version="1.0" encoding="UTF-8"?>
<pms:WorkOrder
    xmlns:pms="http://www.AFictitiousEnterprise.com/ProductionManagement"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
      "http://www.AFictitiousEnterprise.com/ProductionManagement/products.xsd">

    <!--First slot of work order-->

    <pms:ProductRequirements>
        <pms:ProductRequirement>
            <pms:Product>
               <pms:Identifier>
                   P001
               </pms:Identifier>
               <pms:Name>
                   AFictitiousProductName
               </pms:Name>
               <pms:Reference>
                   http://www.AFictitiousEnterprise.com/Products/P001
               </pms:Reference>
               <pms:ModelNo>
                   CRN-2150
               </pms:ModelNo>
               <pms:ProductDescription>
                   This product is an active ingredient for
                   AnotherFictitiousProduct.
               </pms:ProductDescription>
            </pms:Product>
            <pms:Quantity unit="packs" value="150"/>
        </pms:ProductRequirement>
        <pms:ProductRequirement>
          <!--Elements of product requirement-->
        </pms:ProductRequirement>
     <!--Other product requirements-->
    </pms:ProductRequirements>

    <!--Third slot of work order-->

</pms:WorkOrder>
```

Because all three elements in the `WorkOrder` definition of Listing 35 are abstract, using substitution groups is the only way an element can appear in a work order. Whenever you define an element that is meant to fit into one of the work-order slots, you'll include a `substitutionGroup` attribute in the element declaration.

This demonstrates a simple use of substitution groups together with abstract elements to build flexible and extensible schemas.

Section 7. Wrap-up

In this tutorial you've learned data-design requirements for different types of enterprise applications. You've seen that the main advantage of using XML schemas is that it relieves you from writing low-level type- and format-checking code. You've also learned how to apply design patterns while building your schema.

This tutorial has demonstrated incremental development of a production-management schema. During the development, you learned when and how to:

- Design complex types
- Build elements using complex types
- Define and use abstract complex types and abstract elements
- Build customized simple types
- Group elements for substitution
- Use string patterns, enumerations, unions, and lists

While learning features of XML schema, you developed a multiple-file schema along with several instance documents. You can use `InstanceVerifier` (see Download) to check the validity of the instance documents against the schema.

All the Schema features demonstrated in this tutorial have one point in common: using a schema enables strong type checking, which otherwise requires extensive low-level programming effort. XML schema development is an easy and less costly alternative to low-level programming effort as you develop enterprise applications.

XML is filling an increasingly prominent role in information technology (IT). IT professionals can use XML technologies in enterprise applications while maintaining focus on their business requirements. Software engineers, on the other hand, can implement XML processing engines (like the open source validator that you used in this tutorial). XML-based technologies such as XML Schema form an important bridge between IT professionals and software engineers.

Downloads

Description	Name	Size	Download method
Source code	x-schemadata-source.zip	25KB	HTTP

Information about download methods

Resources

Learn

- XML Schema Part 0: Primer Second Edition: Read this W3C primer for a description of the XML Schema facilities.
- XMLPatterns.com: Visit this site to find several patterns you can use in your XML design.
- "Using JSF technology for XForms applications" (Faheem Khan, developerWorks, February 2005): Design an XML-based catalog in this tutorial.
- "Using XML Schema archetypes" (Kevin Williams, developerWorks, June 2001): Learn the benefits of using simple and complex types to design XML schemas.
- "Create flexible and extensible XML schemas" (Ayesha Malik, developerWorks, October 2002): Get a handle on how to design extensible and modular schemas.
- "Reuse it or lose it" (Kevin Williams, developerWorks, March, April, and July 2003): Read a discussion of designing reusable XML structures for enterprise applications in this three-part article series.
- A roadmap to enterprise data integration: In this IBM white paper, take a detailed look at how organizations can plan and build an enterprise infrastructure for supporting data integration applications.
- "Using W3C XML Schema" (Eric van der Vlist, XML.com, October 2001): Check out this introductory article on using various XML schema features.
- Health Level 7: HL7 is an organization that builds standards for across-the-enterprise integration in the health care industry. Check out Reference Information Model (RIM) version 2.13, a popular data model for the health care industry.
- "Java Design Patterns" (Prashant Satarkar, AllAppLabs.com): Learn about Java patterns.
- "XHTML, step by step" (Uche Ogbuji, developerWorks, September 2005): Take this tutorial to learn how XHTML elements work.
- The developerWorks XML zone: Find a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.

Get products and technologies

- Java SE Downloads: Download Java Development Kit (JDK) version 1.5.
- Xerces Java Parser: Download Xerces-J from the Apache Web site.
- Compiere: Get this open source ERP implementation.
- mySAP ERP: Download and try a commercial ERP product.

- IBM trial software: Build your next development project with software available for download directly from developerWorks.

Discuss

- developerWorks blogs: Participate in the developerWorks community.

About the author

Bilal Siddiqui

Bilal Siddiqui is an electronics engineer, an XML consultant, and the founder of XML4Java.com, a company focused on simplifying e-business. After graduating in 1995 with a degree in electronics engineering from the University of Engineering and Technology, Lahore, he began designing software solutions for industrial control systems. Later, he turned to XML and used his experience of programming in C++ to build Web- and WAP-based XML processing tools, server-side parsing solutions, and service applications. He is a technology evangelist and a frequently published technical author.