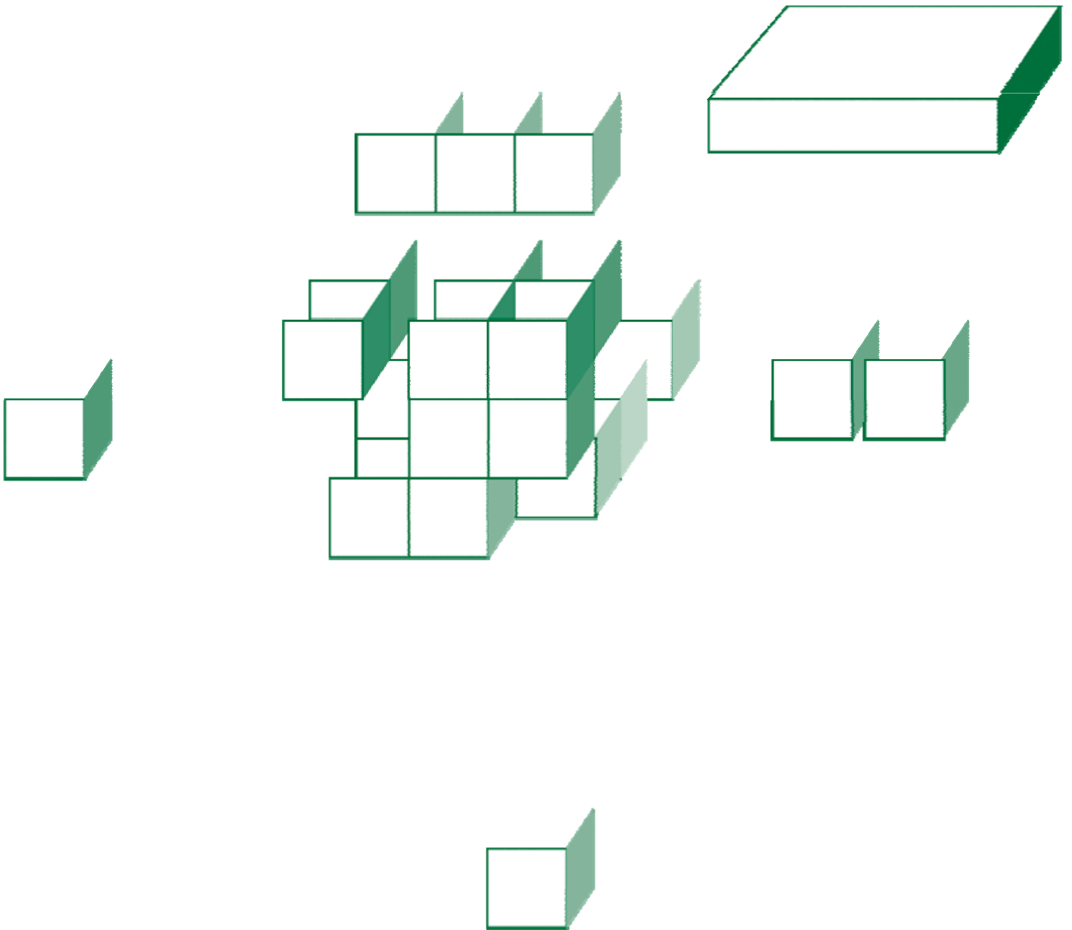
Chapter 2

Reusing Schema Types



OBJECTIVES

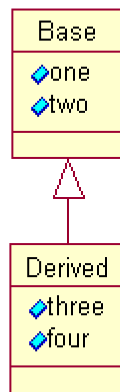
After completing "Reusing Schema Types," you will be able to:

- Create a complex type based on an existing simple type, by extension, so as to create types with simple content as well as attributes.
- Extend a complex type to create a new complex type that adds both attributes and new child elements to its content model.
- Restrict a complex type to create a new complex type that will validate only a subset of the possible values for the original type.
- Use derived-type instances in place of base-type instances in instance documents, and assure that they are validated correctly.
- Implement document designs as expressed in Unified Modeling Language as XML schema.
- Flag abstract types appropriately to forbid instantiation.
- Control type extension and restriction using the `final` attribute.
- Control substitution of derived-type instances using the `block` attribute.

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

UML Specialization

- **We return for a moment to our discussion of the Unified Modeling Language, or UML, to consider another important feature in most type systems: what UML calls type specialization.**
  - A fundamental concept in object-oriented analysis and design, and in type model design in general, is that a type formally identifies a **classification** of possible objects in the problem space; hence the UML term **class**.
  - We say one type **specializes** another when it identifies a useful subset of those possible objects.
  - The specialized type can then be used to express more specific things about those objects than the more general type could, since that original **generalization** must accurately describe objects that don't fall into the specialized subset.

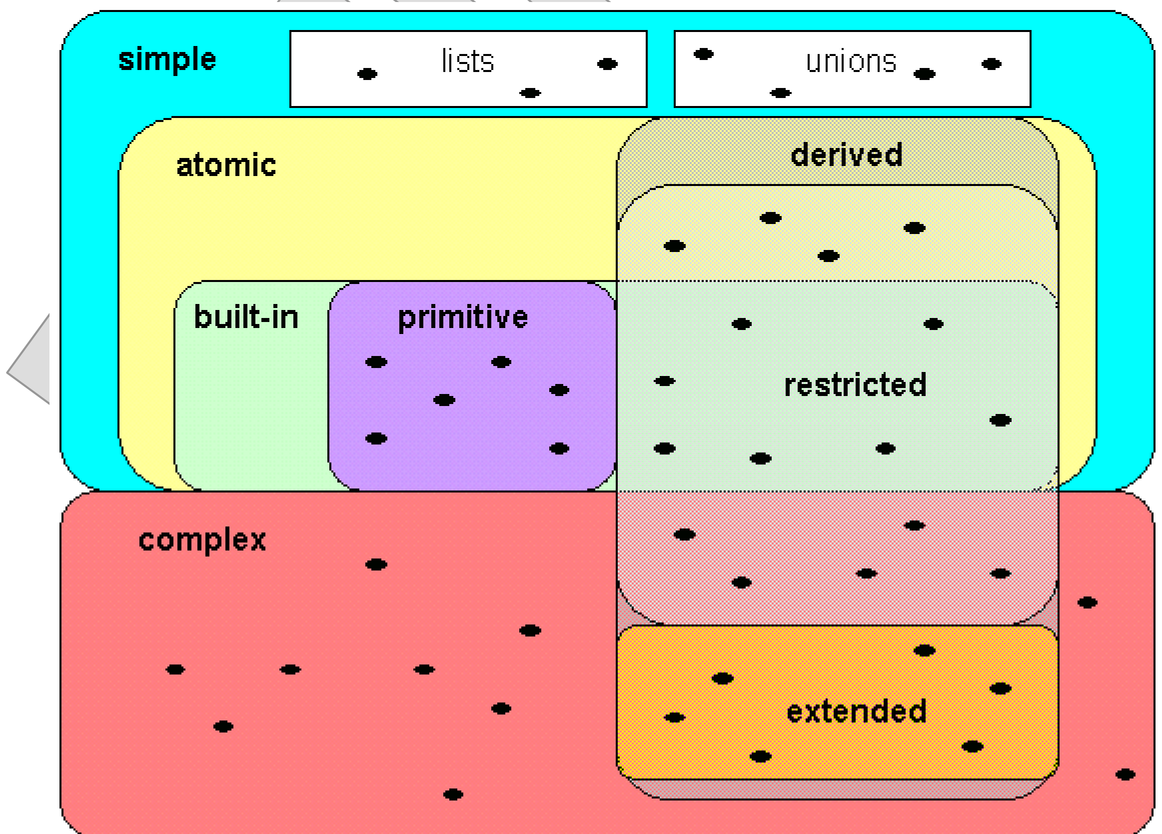


Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Derivation by Extension

- **Various implementation languages and type systems apply a plethora of overlapping terms to describe this one basic concept.**
  - In C++, the generalization is the **base class** and the specialization is a **derived class**. **Superclass** and **subclass** are also common.
  - In Java, the specialization is said to **extend** the superclass.
  - The XML Schema working group has chosen from among the existing terminology and come up with **derivation** to describe all possible type reuse, and **extension** as a way of implementing type specialization.



© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Extending Simple Types

- **Only complex types can extend other types, but simple types can serve as base types for extension.**
  - Within the `xs:complexType` component, the `xs:simpleContent` element allows a simple type to be extended.
  - Inside that, the `xs:extension` element names the base type using the `base` attribute.
  - Additional content can then be specified as usual in a content model:

```
<xs:element name="Result"><xs:complexType>
  <xs:simpleContent>
    <xs:extension base="xs:decimal">
      <xs:attribute name="District"
                    type="xs:positiveInteger" />
      <xs:attribute name="Region"
                    type="xs:string" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType></xs:element>
```

  - This example shows the most common use of this feature: to create a type with its own content, no child elements, but one or more attributes, which of course a simple type cannot have.
  - Valid instances of the example type:

```
<Result District="1" Region="NE">50540</Result>
<Result District="2" Region="SW">9000</Result>
```

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Evaluation Only

Extending Complex Types

- **The same mechanism can be applied to extend a complex base type: use `xs:complexContent` instead of `xs:simpleContent`.**
  - The extending type can then add to the content model: sequences, choices, conjunctions, and attribute definitions can be appended.
  - In a valid instance of the extended type, the base type content must always occur first; in fact it is specified that validation will proceed as if an **`xs:sequence`** had been defined, beginning with all the base-type elements and proceeding to the extending type elements.
- **Note that XML schema extensions can only add to the content model.**
  - There is no mechanism to remove elements or attributes from the base content model.
  - The idea is that if this is a true specialization, everything about the more general type should apply to the specialized type.
  - Later in this chapter, we'll look at complex type **restriction** as (arguably) an exception to this rule.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Using Extended Types

- **An extended type can be used directly as an element type in the document model, either by name or anonymously.**
- **Also, an extended-type instance can be used for an element of any of that type's base types.**
  - That is, the element in question can include the additional attributes and child elements of the extended type, even though its declared type does not support them.
  - This is true for complex type extensions and restrictions.
  - There are some exceptions to this, which we'll encounter later in the chapter – especially, a technique known as **blocking**.
- **For a schema validator to know which content model to use to validate a given element instance, it must be advised of any use of derived types.**
  - The schema-include namespace defines the `xsi:type` attribute for this purpose.
  - This attribute must be present on the element whose content is of a derived type; the value is the derived type name.

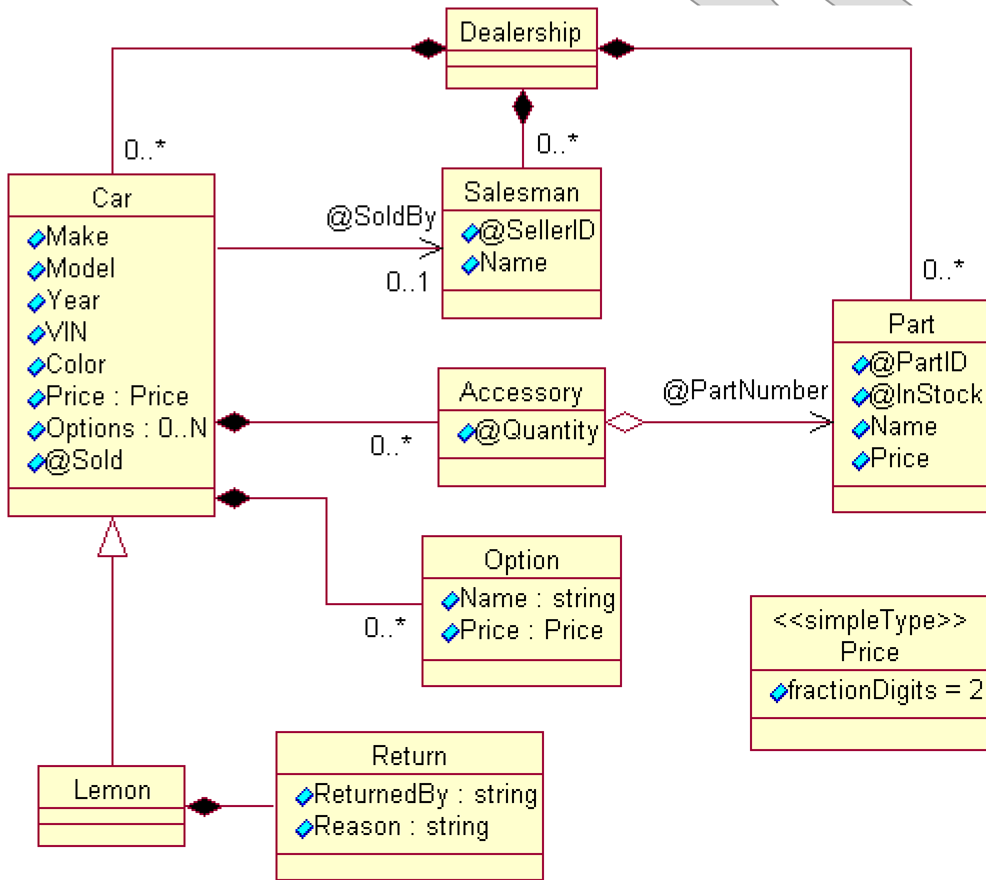
Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

This Car's a Lemon!

DEMO

- To illustrate complex type extension, we're going to expand on our Car Dealership design a little bit.



- The new type **Lemon** specializes **Car**, expressing the concept of a car that has been sold and returned by the buyer on account of some serious flaw.
- It adds a **Return** element to the content model, which in turn introduces the **ReturnedBy** and **Reason** values.

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

This Car's a Lemon!

DEMO

- Starter code is in **Demos\Extension\Step1**.

1. Review the **Cars.xsd** schema, and note the new type **Lemon**. As designed, it extends **Car** and adds the **Result** element and its child content.

```
<xs:complexType name="Lemon">
  <xs:complexContent>
    <xs:extension base="Car"><xs:all>
      <xs:element name="Return"><xs:complexType>
        <xs:all>
          <xs:element name="ReturnedBy"
                      type="xs:string" />
          <xs:element name="Reason"
                      type="xs:string" />
        </xs:all>
      </xs:complexType></xs:element>
    </xs:all></xs:extension>
  </xs:complexContent>
</xs:complexType>
```

2. Now we'll try to update the document **Cars.xml** to flag one of the cars as a lemon. The Ford Pinto might be a good choice:

```
<Car>
  <Make>Ford</Make>
  <Model>Pinto</Model>
  <Year>1974</Year>
  <VIN>AR7993</VIN>
  <Color>Dust</Color>
  <Price>0.99</Price>
  <Return>
    <ReturnedBy>Chandra MacArthur</ReturnedBy>
    <Reason>Explodes on rear-end collision</Reason>
  </Return>
</Car>
```

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

This Car's a Lemon!

DEMO

3. Try validating the document.

```
cvc-complex-type.2.4.a: Invalid content starting
with element 'Return'. The content must match
'(("":Make),("":Model),("":Year),("":VIN),("":Color
),("":Price),("":Option){0-
UNBOUNDED},("":Accessory){0-UNBOUNDED})'.
```

4. The new content does not pass, because it is not in the content model for the expected type, which is of course **Car**.

5. Clarify that this **Car** element instance is of type **Lemon**:

```
<Car xsi:type="Lemon">
  <Make>Ford</Make>
  <Model>Pinto</Model>
  ...
```

6. Now the document is found to be valid.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Limits of XML Extension

- Much is often made of the distinction in various programming languages and other type models between **single-** and **multiple-inheritance** models.
- XML's complex type extension system is a single-inheritance model, meaning that each extending type can have at most one base type.
- However, it is possible to extend one type to the next as many times as desired: each new type will append its element content to the conceptual `xs:sequence` that governs the derived instance's element order.
- Also, multiple types may extend the same base type.
- The best way to reuse multiple content models in one new type is simply to include the complex types or existing element definitions as part of a sequence, conjunction, or choice component.
  - This is what would be the distinction between inheritance and delegation in an OO programming language.
  - For XML, the distinction is less significant, since there are no direct behavioral implications for pure data.
  - The consequence of the choice of composition instead of extension is that instances of the resulting type will have to name the content for the reused type in a child element, which in turn will hold what would otherwise be direct children of the resulting type.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Derivation by Restriction

- **Recall that simple types can be reused with the help of the `xs:restriction` component.**
  - The aim of type **restriction** is to preserve the basic representation and interpretation of a value while setting new (tighter) limits on the **value space**.
  - For simple types this is accomplished by the use of **constraining facets** such as `totalDigits`, `pattern`, and `enumeration`.
- **Complex types can also derive by restriction.**
  - The base type must also be complex.
  - The aim once again is to restrict the value space, although this is achieved differently for complex types.
- **Complex type restrictions come in two main forms:**
  - Adding default or fixed values for attributes, or specifying attribute type where it had been unspecified
  - Setting new occurrence constraints for child elements

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Implementing Restriction

- **The definition for a restricted complex type is regrettably verbose and – worse – redundant.**
- **It is necessary to restate the entire content model for the new restricted type; changes can then be made component by component, each change overriding the base content model.**
  - This of course makes for multiple maintenance points, and thus for brittle type definitions.
  - When something in the base type changes, the change must be reflected in each restriction-derived type.
  - If a restricted type has several base types, each of which extends the other, it must be kept in sync with all of them.
  - The worst part is that, where such changes are being maintained manually, using copy-and-paste, it's easy to mistakenly remove a restriction when propagating changes.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

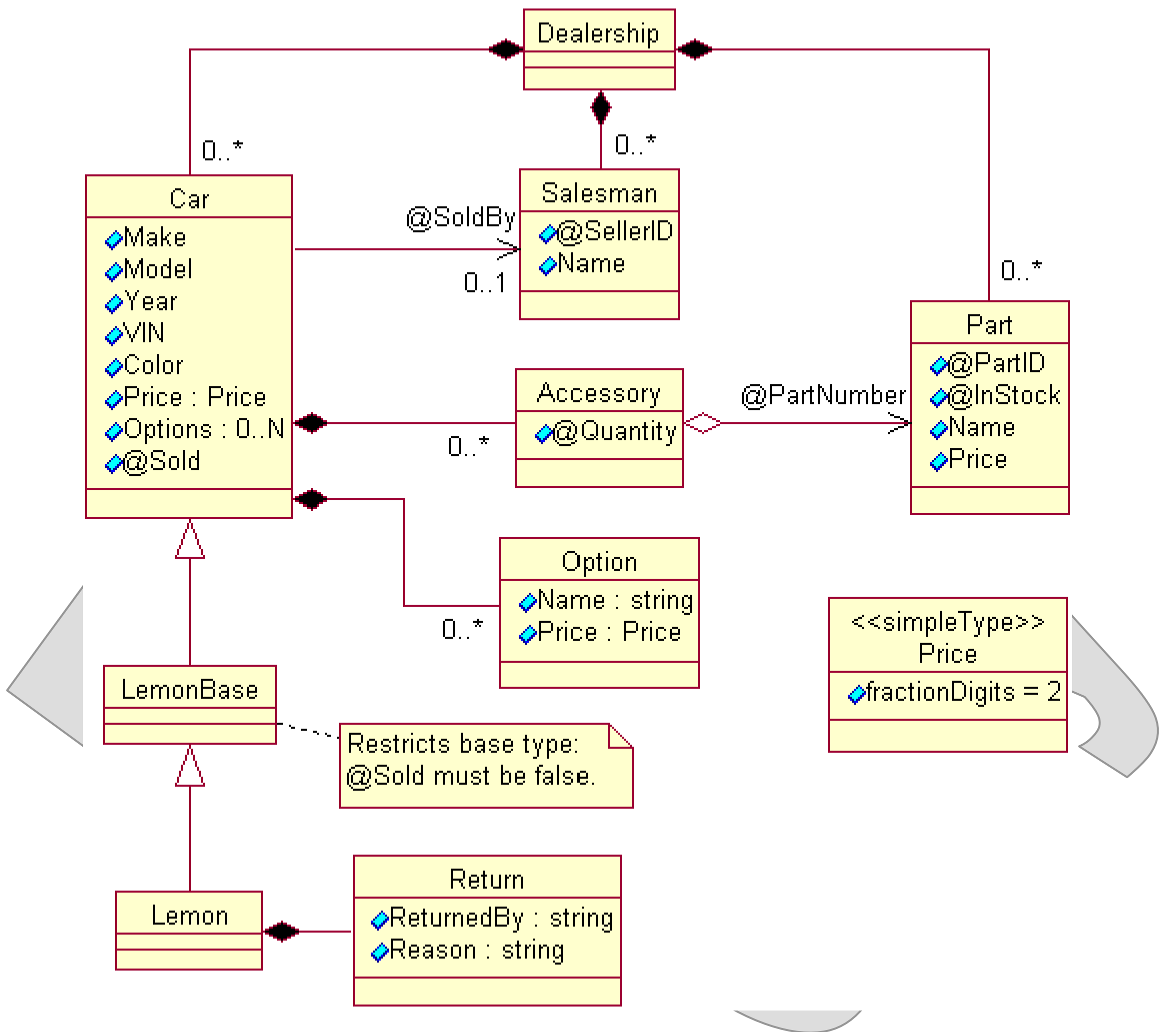
Limits of Complex-Type Restriction

- **Earlier, restriction was mentioned as a partial exception to the rule that complex-type reuse in XML does not support content model deletions or changes.**
- **There are ways in which a restriction may seem to remove something from the content model:**
  - An element that had been constrained as optional (minimum/maximum occurrence = 0/1) can be omitted entirely in a restricted type.
- **There are strict limits on what restrictions can be placed.**
  - Generally, restriction cannot make any instance valid that would not have been valid under the base type.
  - For instance, while omitting an element that had been required (min/max = 1/1) may feel like a restriction – a simplification of the content model, even – it is not. Such a restriction would make valid an instance that could not be validated using the base type.
  - The schema specification, Part 0, puts it this way: "an application prepared for the values of the base type would not be surprised by the values of the restricted type."
  - The precise constraints on type restriction are myriad: see the schema specification, Part 1, sections 3.4.6 and 3.9.6.

Evaluation Only

Lemons Can't be Sold

- **Examples\Cars\Step5** adds an intermediate base type to the system, called **LemonBase**.
  - This type restricts **Car** to say that the optional attribute **Sold** must be “false”; if it’s been flagged as a lemon, it must be back on the lot, and we can’t sell it again.



  - Note the implication for UML design: while in the abstract one subtype might satisfactorily express both extension and restriction of a supertype, in XML practice the derivation will require two subtypes.

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Lemons Can't be Sold

EXAMPLE

- The revised schema defines **LemonBase** to do the restriction, before **Lemon** proper extends it:

```
<xs:complexType name="LemonBase" abstract="true">
  <xs:complexContent>
    <xs:restriction base="Car">
      <xs:sequence>
        <xs:element name="Make" type="xs:string"/>
        <xs:element name="Model" type="xs:string"/>
        <xs:element name="Year"><xs:simpleType>
          <xs:restriction base="xs:integer">
            <xs:totalDigits value="4" />
            <xs:minInclusive value="1900" />
            <xs:maxInclusive value="2002" />
          </xs:restriction>
        </xs:simpleType></xs:element>
        <!-- Lots more here! -->
      </xs:sequence>
      <xs:attribute name="Sold" type="xs:boolean"
                    fixed="false" />
      <xs:attribute name="SoldBy" type="xs:Name" />
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="Lemon">
  <xs:complexContent>
    <xs:extension base="LemonBase"><xs:all>
```

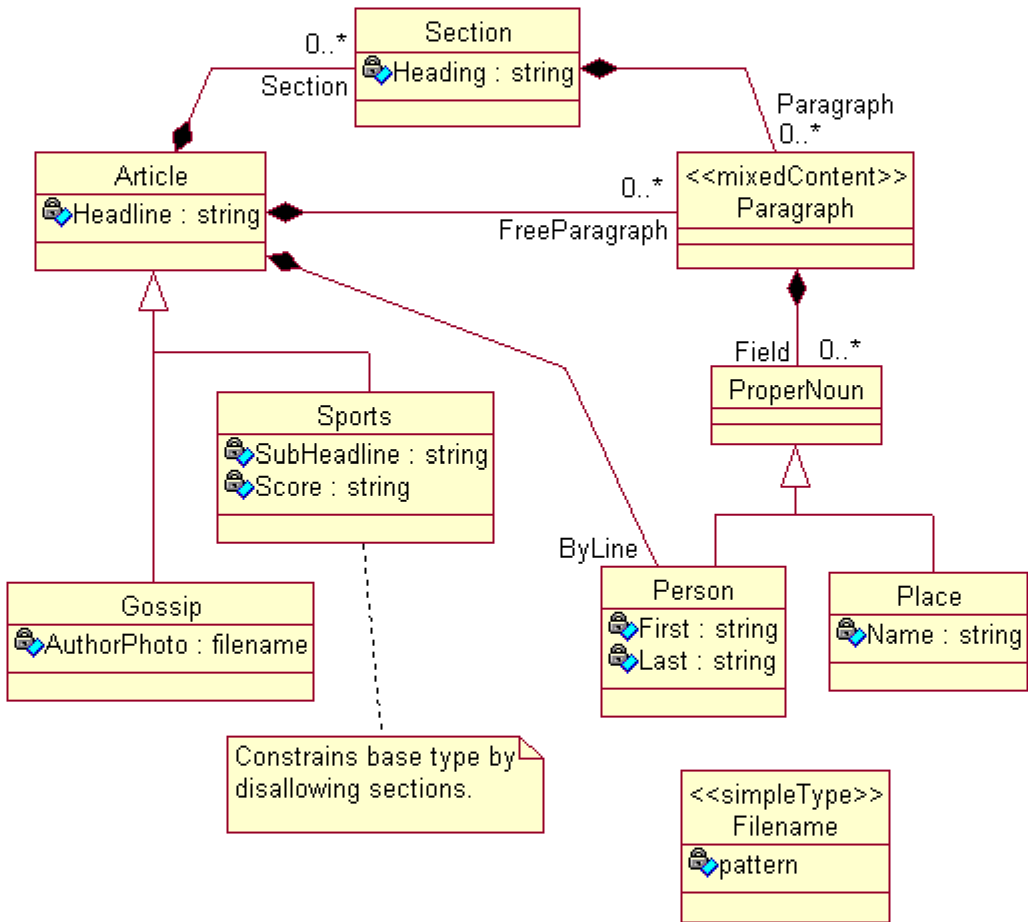
- Note how much XML must be written to effect what seems like a very simple change.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

A Newspaper Layout

- In preparation for the next lab, we now introduce a new case study: an XML-based newspaper.



© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Type Derivation for the Newspaper

- Note that there are two complex-type derivation systems in play.
- **Paragraph**s are designed to hold mixed content, which will mostly be text, but the **ProperNoun** type is a placeholder for various types of content that the application designers wish to formalize when it appears in a paragraph.
  - So far, known subtypes include **Person** and **Place**.
  - Authors' use of this element in paragraph content will make it possible to run precise queries later, without resorting to keyword searches.
- The **Article** sets the basic structure of a printed article for the paper.
  - Subtypes **Gossip** and **Sports** each add content that is relevant to their purpose, and **Sports** also restricts the article structure, insisting that the article not be broken into sections, but be one flat stream of free paragraphs.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

News You Can Use

LAB 2

In this lab you will complete the implementation of the Newspaper document design. The starter schema implements the simple types and all complex types that don't derive any others. You will implement the derived types and validate a few prepared instance documents.

Detailed instructions are contained in the Lab 2 write-up at the end of the chapter.

Suggested time: 45 minutes.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Abstract Types

- **Another common OO concept is that of an abstract type: that is, one which cannot be instantiated.**
  - Abstract types can be used to identify common content for several derived types, without allowing any element instances to validate against the base type.
  - Returning to the notion of UML **classifications** of possible objects, an abstract type is a good choice when all classified objects fall into one of the subsets encapsulated in some known derived type.
  - Another way to say this is that abstract types work well to define common content for a general class of objects when no possible object could be defined with just the base content; some derived type must "complete the concept".
  - The antonym is **concrete type**, although XML schema doesn't carry this terminology forward into the specification.
- **Define a type as abstract by setting its `abstract` attribute to "true". We've seen a few examples of abstract types already:**
  - **LemonBase** is abstract because it is only meant as the first step in realizing a derived type in the UML design: one should always instantiate either a **Car** or a **Lemon**.
  - As of **Examples\News\Step4**, **ProperNoun** is made abstract, because it has no content and is meant only as a placeholder in the **Paragraph** content model. This allows the design to be extended over time, as new uses are found for more formalized paragraph content.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Element Substitution

- **Another important OO concept is that of polymorphism: this is the ability to provide an instance of a derived type where an instance of the base type had been expected.**
  - Polymorphism is most interesting in programming languages, where objects have behavior that can vary unbeknownst to client code.
  - In XML, the concept really boils down to the ability to seamlessly substitute one element for another in a valid instance document.
- **We have a kind of polymorphism with the features we've studied already, since a schema validator can validate an element instance against a derived type, given the `xsi:type` hint.**
- **The substitution group schema component provides a similar capability, at the level of elements instead of types:**
  - A group of elements can be defined that can be quietly substituted for one special **head element**. No `xsi:type` attribute or other type declaration per instance is necessary.
  - The group members must each be of the same type as the head element, or of a type derived from that element's type.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Car Substitution

EXAMPLE

- **Examples\Cars\Substitution** is a slightly modified copy of the **Step5** code that we discussed earlier as an example of complex-type restriction.
  - In this revision of the schema, a new **Lemon** element is identified as a possible substitute for the **Car** element.

```
<xs:element name="Lemon"
            type="Lemon"
            substitutionGroup="Car" />
```

  - Thus, the much-maligned Ford Pinto can be expressed as a **Lemon** instance directly – that is, by element name rather than via the use of `xsi:type`:

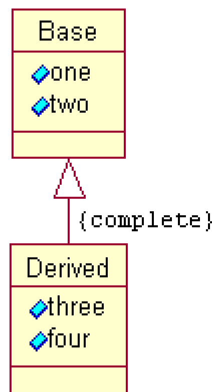
```
<Lemon>
  <Make>Ford</Make>
  <Model>Pinto</Model>
  <Year>1974</Year>
  <VIN>AR7993</VIN>
  <Color>Dust</Color>
  <Price>0.99</Price>
  <Return>
    <ReturnedBy>Chandra MacArthur</ReturnedBy>
    <Reason>Explodes on rear-end collision</Reason>
  </Return>
</Lemon>
```

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

“Complete” Specialization

- UML allows for certain constraints on specialization relationships.
- One of these is the **complete** constraint, which says that a given specialization is the final statement of the relevant design concept, and therefore should not itself be specialized any further.



- This is a relatively new idea in OO theory, and is supported to varying degrees in popular programming languages.
  - Java offers the **final** keyword to modify a class definition; such a class cannot be extended.
  - C# does the same thing for **sealed** classes.
- XML schema allows completion to be asserted, via the `final` attribute on the `xs:complexType` component.
  - This attribute can have a list of the values “extension” or “restriction”, or the value “#all”, thus indicating which sorts of derivation are prohibited.

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Evaluation Only

Making Car Complete

EXAMPLE

- In **Examples\Cars\Final**, another branch revision of the **Step5** example experiments with the `final` attribute, implementing the hypothetical constraint on **Car** shown in the UML fragment on the previous page.
  - In **NoRestriction.xsd**, the **Car** type prohibits restriction, which invalidates the schema because **LemonBase** attempts to derive it by restriction.

```
<xs:complexType name="Car" final="restriction">
```

  - In **NoExtension.xsd**, **LemonBase** prohibits extension, which similarly invalidates the schema, since **Lemon** tries to extend **LemonBase**.

```
<xs:complexType name="LemonBase" abstract="true"
  final="extension">
```

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Limits of the final Attribute

- **Oddly, the constraint on derivation imposed by the `final` attribute is only observed for <u>immediate</u> derivation.**
  - For `final="#all"`, this is moot: there can be no immediate derivation whatsoever.
  - However, if `final` is used to prohibit restriction only, or extension only, a legal derivation may exist.
  - This derived type does not carry the `final` value forward for validation purposes.
- **The upshot of this is that it is possible to wiggle out of the constraint entirely, by first defining a subtype that derives legally, and then deriving from this type in whatever way was originally prohibited.**
- **For instance, if Car in the previous example had the `final` value "extension", the schema would be valid!**
  - **LemonBase** restricts **Car**, which is clearly legal.
  - More surprising is that **Lemon** is allowed to extend **LemonBase**.
  - An example of this is in **Examples\Cars\Final\Sneaky.xsd**.
  - This seems to be an oversight in the 1.0 recommendation – see Part 1, section 3.4.6, final constraint.
- **If subtyping is to be strictly disallowed, the "#all" value for `final` is the only practical option.**

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Derivation Without Polymorphism

- **Sometimes type derivation is purely a convenience for implementation purposes, gathering some common content definitions for several derived types.**
- **It may be that at a design level, polymorphism of any sort is actually undesirable.**
- **In this case, XML schema provides the `block` attribute to forbid use of a derived type to validate an instance that was expected to be of a base type.**
  - This attribute can have the same values as `final`.
  - When this attribute is defined for a complex type, derivations of the indicated sort are still allowed, but no such derived types may be instantiated in place of the base.
  - This applies to `xsi:type` declarations as well as to element substitution.

Evaluation Only

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

SUMMARY

- **Whatever name it goes by, specialization is an important feature of any complete type model.**
- **XML complex-type derivation is a great tool for reuse of document-design work.**
  - Types can be decomposed, based on careful conceptualization of the design problem, into more general and specific parts.
  - These can then be defined separately and related via restriction or extension.
  - This makes for easier schema authoring, and especially for easier maintenance as the application and schema evolve.
- **Some of the derivation features in the current schema recommendation are not really ready for prime time.**
  - The complex-type restriction mechanism is onerous and prone to maintenance errors.
  - Enforcement of the `final` attribute is not propagated from derived type to derived type, as one would hope.
- **Still, the basic idea of type extension is enough to support some very sophisticated type designs.**
- **As we'll see in the next chapter, the ability to leverage existing schema and namespaces will amplify the power of these type-reuse features, making it possible to borrow types from other, existing schema.**

© 2001-2004 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.