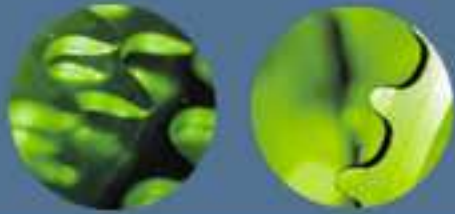




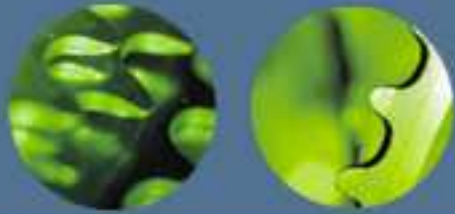
XML Design Considerations *Using W3C XML Schema*

Priscilla Walmsley
Managing Director, Datypic
pwalmsley@datypic.com
<http://www.datypic.com>



About this course...

- These slides are for a one-day tutorial offered by Priscilla Walmsley at various industry conferences.
- If you are interested in having Priscilla present this tutorial, or a longer course, to your group, please contact her at pwalmsley@datypic.com.
- The slides were last updated on June 21, 2006.



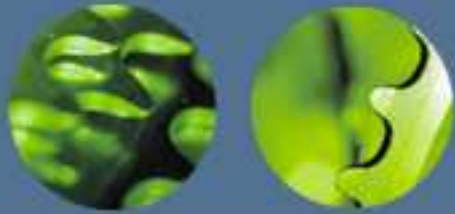
Agenda

- **Morning**

- Introduction to XML Design
- Structuring Complex Elements
- Considerations for Simple Content
- Structuring Schemas
- Names and Namespaces

- **Afternoon**

- Flexibility
- Reuse
- Extensibility
- Versioning
- Schema Documentation
- XML Design in Multi-User Environments



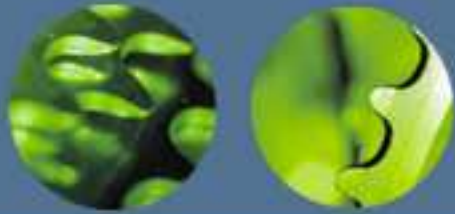
Assumptions

- You already know something about W3C XML Schema
- You plan to use/are using W3C XML Schema
- You are designing schemas for a data-oriented (highly structured) application
 - as opposed to a document-oriented (narrative) one
- You have some control over the structure of your documents



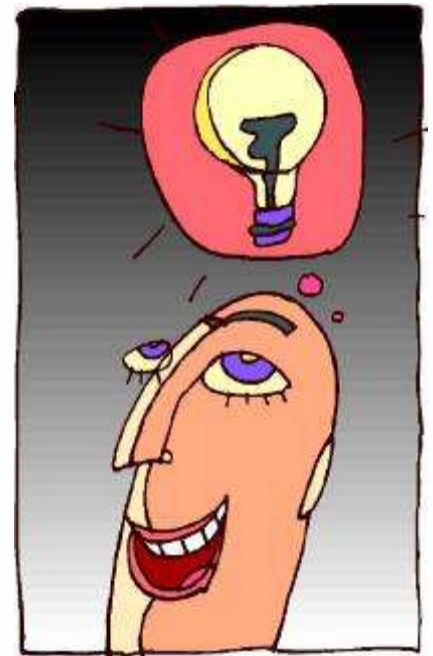
Introduction to XML Design





XML Design Goal 1: Clarity

- Documents and schemas that everyone can understand are:
 - easier to learn
 - easier to debug and change
 - more likely to be reused

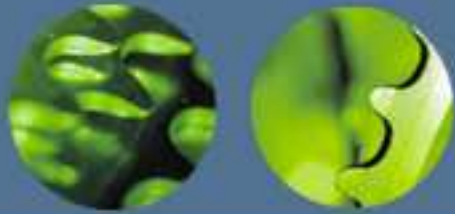




XML Design Goal 2: Accuracy & Precision

- Precise types can catch data quality problems
 - sharing data with outside parties makes catching data errors even more crucial
- Correctly structured data tends to be more useful and easier to process
- Accurate data structures are more intuitive and tend to be used for a longer time





XML Design Goal 3: Reusability

- Reuse of schema components:
 - saves time and money
 - results in a better design
(two heads are better than one)
 - helps to avoid the mess of redundant but incompatible schemas





XML Design Goal 4: Flexibility & Extensibility

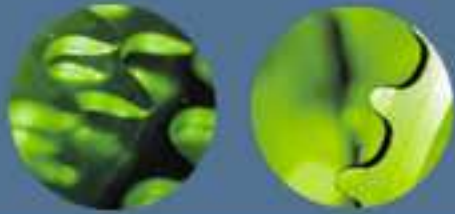
- XML documents and schemas should be:
 - flexible to meet a variety of users' needs
 - extensible for future use
 - set up to handle changes gracefully





Why is So Little Attention Paid To XML Design?

- Wide variety of use cases
 - the answer to most design questions starts with "It depends..."
- Some XML data is temporary
 - so considered not as crucial as data in databases
- Anyone can create XML
 - not just database administrators or software developers



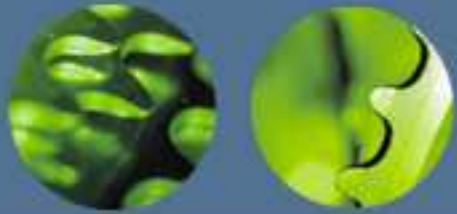
Influences on XML Design

- Use case
- Consumers
 - humans or machines?
- Schema language and its implementation
 - sad but true
 - should avoid designing documents that can't be validated using your schema language(s)
- Processing technologies
 - certain structures are better for XSLT 1.0, XSLT 2.0, XQuery, DOM, SAX, data binding software, etc.
 - ease of programming and maintenance
 - performance



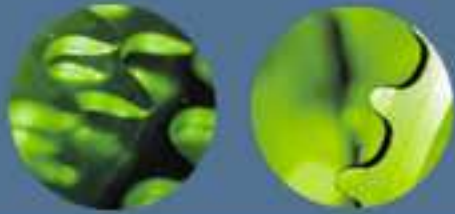
Using Schemas





Schemas are For...

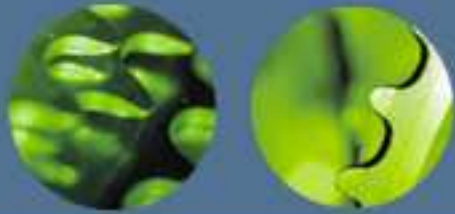
- Validation
 - structure, order, types
- Instance Augmentation
 - default values, white space processing
- System Documentation
- Application Information



Validation

- Can determine whether an instance conforms to a schema
 - Valid names
 - Structure/order of elements
 - Required/optional values
 - Valid values
 - Types
- Has plenty of limitations
- Not necessarily required; a parser option

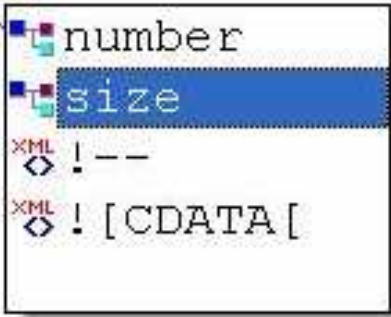


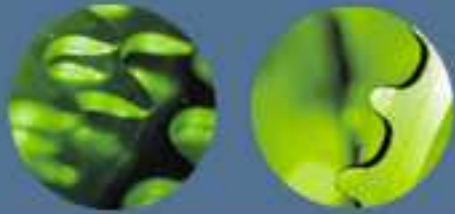


Assisted Editing

- Schemas allow document authors to see what is valid while editing

```
1 <?xml version="1.0"?>
2 <product effDate="2001-04-02"
3     <number>557</number>
4     <s
5 </pr number
6
```

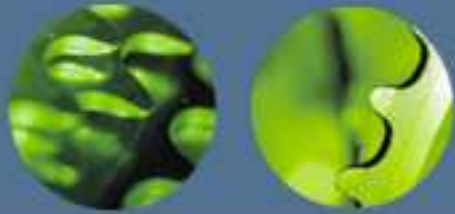




System Documentation

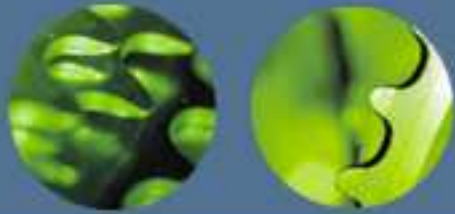
- Serves as a guide to the data elements and structure of a document
- You can specify additional annotations with any XML content
- Serves as a contract with trading partners





Web Services Contracts

```
<wsdl:definitions name="EndorsementSearch" ...>
  <wsdl:types>
    <xsd:schema targetNamespace="http://datypic.com"
      xmlns:xsd="http://www.w3.org/1999/XMLSchema">
      <xsd:element name="GetStockPrice">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="symbol" type="xsd:string"/>
            <xsd:element name="value" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    ...
  </wsdl:types>...
</wsdl:definitions>
```



Assisting in Creation of User Forms

Order

Number:

Date:

Customer Information

Customer Name:

Address 1:

Address 2:

City:

State:

Zip:

Comment:

Items

Prod #	Description	Quant	Size	Price	Cost
<input type="text"/>	<input type="text"/>	0	Medium	0.00	<input type="text"/>

Repeating Table

Data Source

Layout

Controls

Data Source

Views

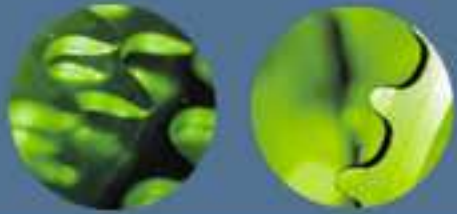
Work with the data source:

- order
 - :number*
 - date*
 - comment
 - customer
 - shipTo
 - addr
 - addr2
 - city
 - state
 - postal_code
 - items

Show details

Add...

Help with the Data Source



Catching Errors in Queries and Stylesheets

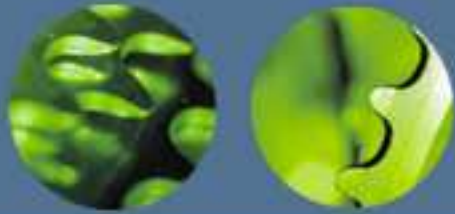
```
import schema
  default element namespace
    "http://datypic.com/prod"
  at "http://datypic.com/prod.xsd";

for $prod in doc("catalog.xml")/product
order by $prod/product/number
return $prod/name + 1
```

misspelling

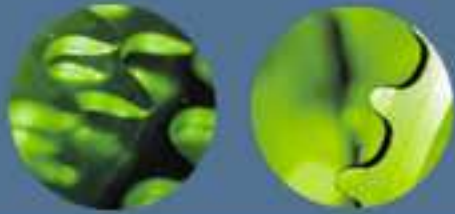
invalid path;
product will never
have product child

type error: name is declared
to be of type xs:string, so
cannot be used in an add
operation



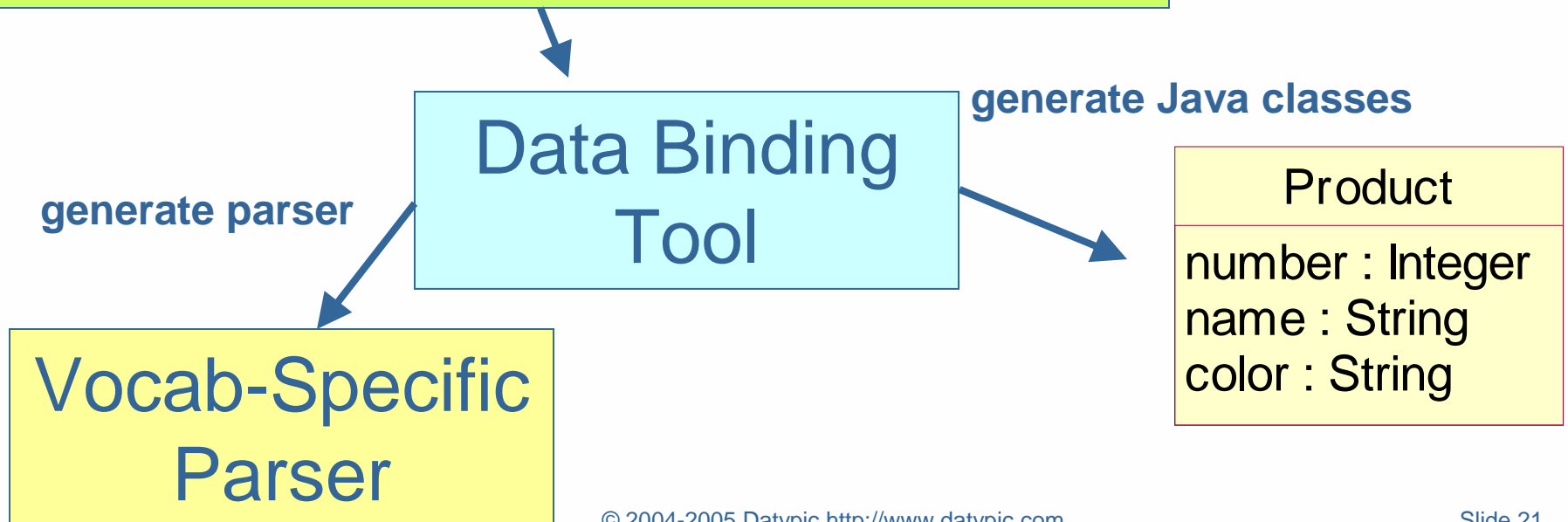
Special Processing Based on Type

```
<xsl:stylesheet .....  
<xsl:import-schema ...  
  
<xsl:template match="element(*,USAddressType)">  
  .... <xsl:value-of select="city"/>  
      <xsl:value-of select="zipCode"/>  
</xsl:template>  
  
<xsl:template match="element(*,UKAddressType)">  
  .... <xsl:value-of select="postCode"/>  
      <xsl:value-of select="city"/>  
</xsl:template>
```



Data Binding

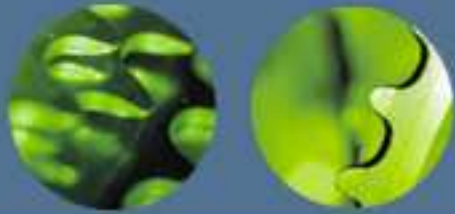
```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="product" type="ProductType"/>
  <xs:complexType name="ProductType">
    <xs:sequence>
      <xs:element name="number" type="xs:integer"/>
      <xs:element name="size" type="SizeType"/>
    </xs:sequence>
  </xs:complexType>
  ....
```





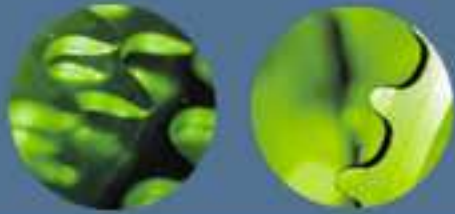
Structuring Complex Elements





Structuring Complex Elements

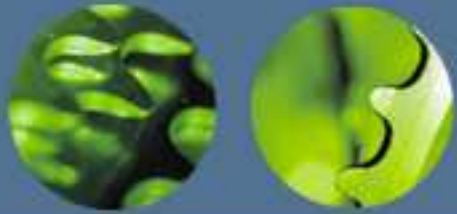
- Agenda
 - Child elements vs. attributes
 - Composing content models
 - Name/value pairs
 - Relationships



Child Elements vs. Attributes

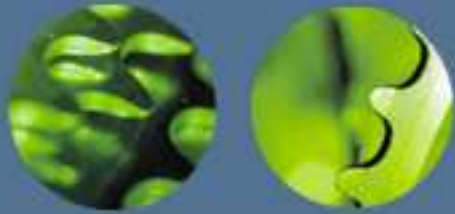
```
<product>  
  <number>557</number>  
  <name>Short-Sleeved Linen Blouse</name>  
  <size>10</size>  
</product>
```

```
<product number="557"  
  name="Short-Sleeved Linen Blouse"  
  size="10" />
```



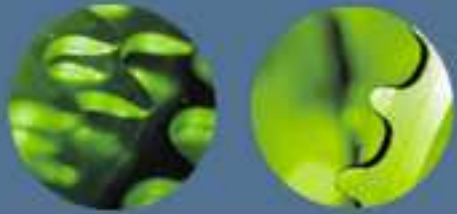
Advantages of Attributes

- Less verbose
- More readable for short values
- Easier to process using SAX or DOM
- Can be added to the instance by specifying default values; elements cannot



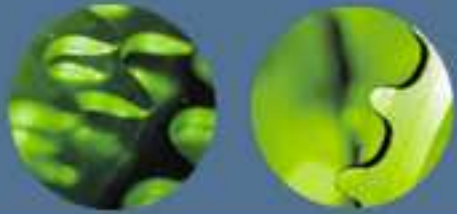
Advantages of Child Elements

- More extensible because attributes can later be added to them without affecting a processing application
 - e.g., if you realized that you needed to keep track of the currency of a price, you can later declare a `currency` attribute of price.
- Can contain other elements
 - e.g., to express a textual description using XHTML tags
- Can be repeated
 - an element may only appear once now, but later you may wish to extend it to appear multiple times
 - e.g., if you decide later that a product can have multiple colors, you can allow a `color` child to appear more than once.



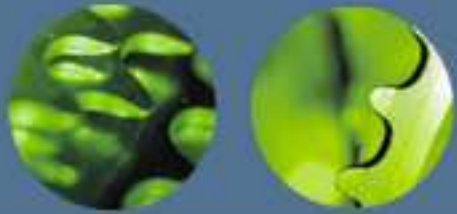
Advantages of Child Elements (cont.)

- Allow more control over the rules of their appearance
 - e.g., you can say that a product can either have a `number` or a `productCode` child
- Can be used in substitution groups
- Can be given nil values
- Can use type substitution to substitute derived types in the instance
- Order is significant, while the order of attributes is not
- Can be more readable for lengthy values



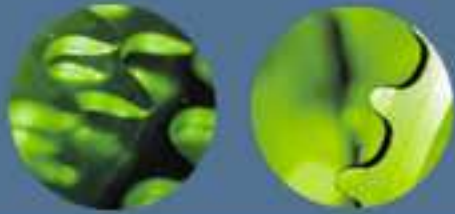
Recommendation

- Use attributes for meta data
 - e.g. units, language, or time-dependence of an element value
- Use attributes for ID and IDREF values
- Use elements for everything else
- Making consistent choices is the most important thing



Content Models

- Three choices for model groups
 - **xs:sequence**
 - elements must appear in the specified order
 - **xs:choice**
 - a (possibly repeating) choice between elements
 - **xs:all**
 - elements must appear zero or one times, in any order

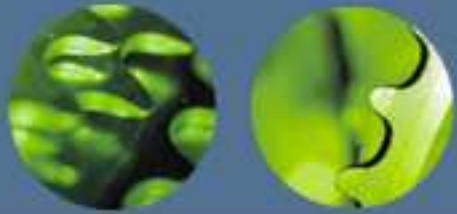


all Groups

```
<xs:complexType name="ProductType">
  <xs:all>
    <xs:element name="name" />
    <xs:element name="number" />
    <xs:element name="size" minOccurs="0" />
  </xs:all>
</xs:complexType>
```

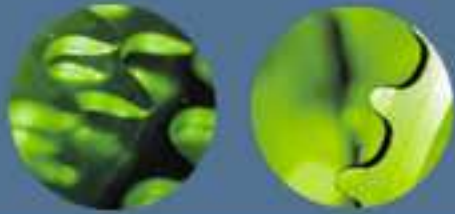
Any order is allowed; name and number must appear, size is optional

```
<product>
  <name>$
  <number>
  <size>
</product>
<product>
  <size>
  <name>$
  <number>
</product>
<product>
  <size>
  <number>557</number>
  <name>Shirt</name>
</product>
```



Limitations of `all` Groups

- Elements can be optional or appear just once
 - can't say "element X and five of element Y, in any order"
- Cannot appear within, or contain, other model groups
 - can't have a choice between two `all` content models
- Prevent extension of the type
 - neither the base type nor the extended type can have an `all` group as its content model

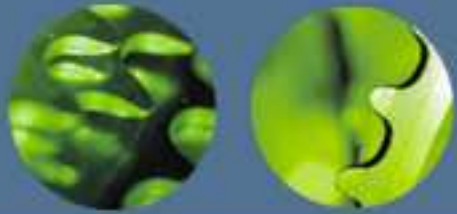


Repeating Choice Groups

```
<xs:complexType name="ProductType">  
  <xs:choice minOccurs="0" maxOccurs="unbounded">  
    <xs:element name="number" />  
    <xs:element name="name" />  
    <xs:element name="size" />  
  </xs:choice>  
</xs:complexType>
```

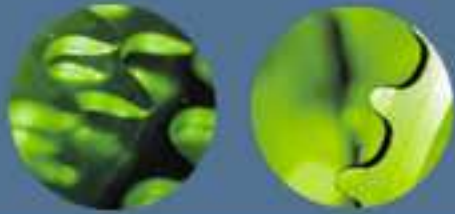
An unlimited number of number, name, and size elements are allowed, in any order

```
<product>  
  <size>12</size>  
  <number>557</number>  
  <name>Shirt</name>  
  <number>557</number>  
</product>  
<product>  
  <number>557</number>  
  <number>557</number>  
</product>  
<product />  
</product>
```



Limitations of Repeating choice Groups

- Can't control the min/maxOccurs of individual elements
 - again, can't say "element X and five of element Y, in any order"
 - unless you enumerate all of the combinations
- Not very extensible
 - except when using substitution groups

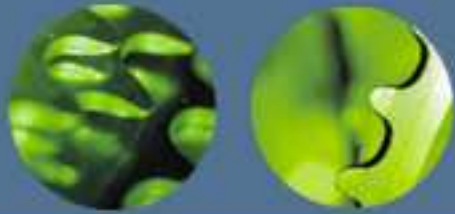


Sequence Groups

```
<xs:complexType name="ProductType">  
  <xs:sequence>  
    <xs:element name="number" />  
    <xs:element name="name" maxOccurs="unbounded" />  
    <xs:element name="color" maxOccurs="3" />  
  </xs:sequence>  
</xs:complexType>
```

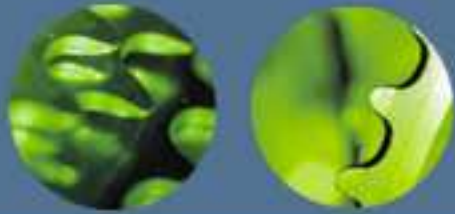
```
<product>  
  <number>557</number>  
  <name>Shirt</name>  
  <color>Blue</color>  
  <color>White</color>  
</product>
```

Elements appear in the order they are declared



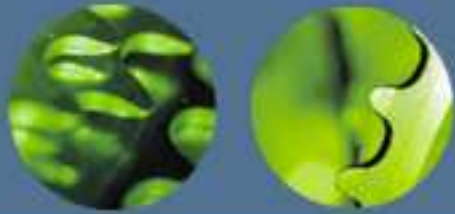
Using sequence Groups

- Advantages
 - can enforce min/maxOccurs on individual elements
 - authors tend to remember an ordered list better than a jumbled choice of children
 - maps more cleanly for data binding software
 - can be easier to process in SAX
- Recommendation
 - if you don't care about order, then enforce order



Suggested Order of Children

- General to specific
 - put common (shared) children first
 - be consistent in the order (and cardinality) of common children
 - named model groups can help
 - next, put those specific to that type
- Required elements before optional ones
- Use a logical and intuitive order
 - don't be tempted to alphabetize
 - put related elements together

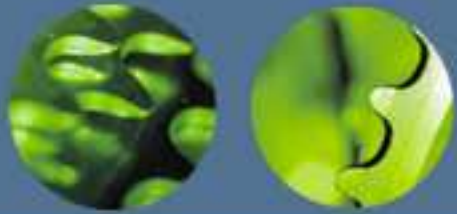


Name/Value Pairs (Properties)

- Simple (often flexible) lists of properties with names and values
- Two approaches:

```
<length>3</length>  
<width>5</width>  
<numberOfSlots>3</numberOfSlots>
```

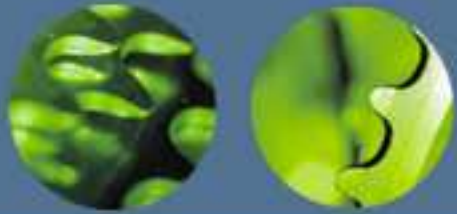
```
<property name="length">3</property>  
<property name="width">5</property>  
<property name="numberOfSlots">3</property>
```



Specific Element Names

- Advantages
 - data types (e.g. length is a positive integer)
 - occurrence constraints (e.g. width is required)
- Disadvantages
 - inflexibility
 - schema must be changed to add new property types

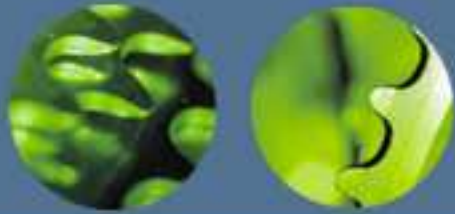
```
<length>3</length>  
<width>5</width>  
<numberOfSlots>3</numberOfSlots>
```



Generic Element Names

- Advantages
 - flexibility
 - no need to change schema if new property added
 - processing easier if all treated the same
 - e.g. write stylesheet to show all properties in a table
- Disadvantages
 - can't enforce data types or occurrence constraints

```
<property name="length">3</property>  
<property name="width">5</property>  
<property name="numberOfSlots">3</property>
```

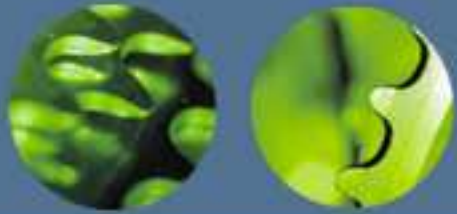


Structural Elements

- Elements whose sole purpose is to group together other elements
 - e.g., using an **address** element to contain address-related elements instead of a flat structure

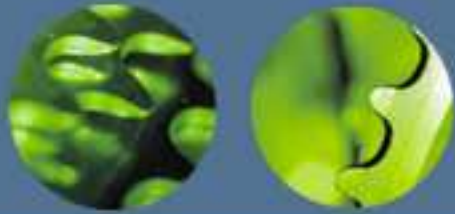
```
<customer>
  <name>PW</name>
  <addr1>1 Main</addr1>
  <city>TC</city>
  <state>MI</state>
</customer>
```

```
<customer>
  <name>PW</name>
  <address>
    <line1>1 Main</line1>
    <city>TC</city>
    <state>MI</state>
  </address>
</customer>
```



Using Structural Elements

- Advantages
 - Can be a way to get around weaknesses in content model semantics
 - e.g. have child that uses a repeating choice, when the parent content model is a sequence
 - Usually more intuitive to the human reader
 - Less confusing if hand-authored (fewer choices)
 - Easier to process using XSLT
 - especially if optional or repeating
- Disadvantage
 - more verbose

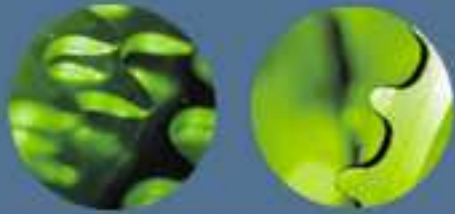


Collections

- Structural elements used to create lists of identical elements

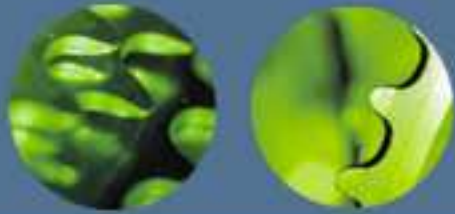
```
<product>  
  <name>Shirt</name>  
  <availSize>10</availSize>  
  <availSize>12</availSize>  
  <availSize>14</availSize>  
</product>
```

```
<product>  
  <name>Shirt</name>  
  <availSizes>  
    <size>10</size>  
    <size>12</size>  
    <size>14</size>  
  </availSizes>  
</product>
```



Relationships

- Many methods of representing relationships. Two popular ones:
 - containment relationships
 - relationship elements
- Methods of validating referential integrity:
 - attribute values with ID/IDREF types
 - element or attribute values with identity constraints
 - application-enforced rules

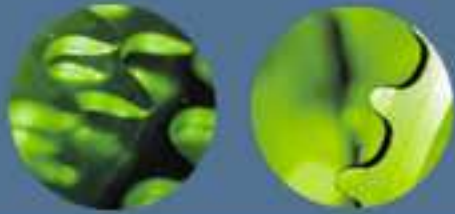


Containment Relationships

- Good for 1-to-1, or 1-to-many relationships
 - where "child" entity is not reused anywhere

```
<department>
  <number>556</number>
  <products>
    <product>
      <name>Shirt</name>
    </product>
    <product>
      <name>Hat</name>
    </product>
  </products>
</department>
```

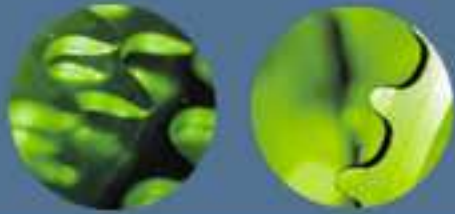
"a department
can be associated
with one or more
products"



Relationship Elements

- Good for many-to-many relationships
 - or where one entity is used in many different relationships
- Especially when the relationship has properties of its own

```
<department num="556">  
  <name>Men's</name>  
</department>  
<product num="466">  
  <name>Shirt</name>  
</product>  
<dept-product>  
  <dept>556</dept>  
  <product>466</product>  
  <mainDept>true</mainDept>  
</dept-product>
```



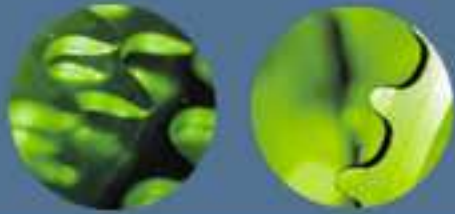
Using ID/IDREF to Indicate a Relationship

- Attributes of type `xs:ID`/`xs:IDREF` are used

```
<department id="D556">  
  <name>Men ' s</name>  
</department>  
<product deptref="D556">  
  <name>Shirt</name>  
</product>
```

department is uniquely identifiable by its `id` attribute

product identifies its department through its `deptref` value



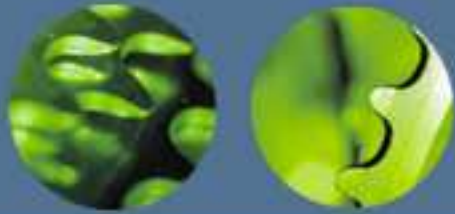
Using key/keyref to Indicate a Relationship

- Identity constraints are used to validate relationship

```
<department num="556">  
  <name>Men ' s</name>  
</department>  
<product>  
  <dept>556</dept>  
  <name>Shirt</name>  
</product>
```

department is uniquely identifiable by its `num` attribute

product identifies its department through its `dept` child



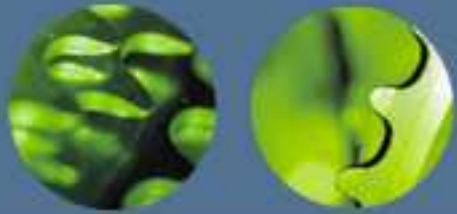
Limitations of ID/IDREF vs. key/keyref

- Limitations of ID/IDREF
 - scoped to the entire document only
 - based on one value, as opposed to multi-field keys
 - require ID or IDREF to be the type of the attribute, precluding data validation of that attribute
 - require that the values be valid XML names
 - recommended only for attributes, not elements
 - based on string equality, as opposed to value equality
- Advantages of ID/IDREF
 - support in XML technologies (XPath/XQuery)



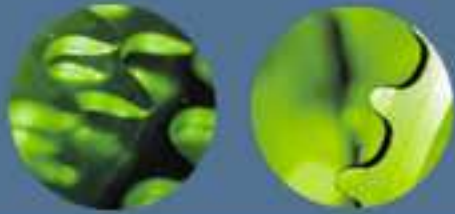
Considerations for Simple Content





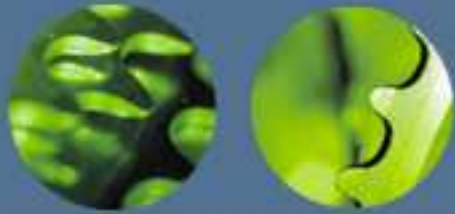
Considerations for Simple Content

- Agenda
 - Granularity
 - Lists
 - Whitespace and strings
 - Representing missing values
 - Code lists



Use the Built-In Simple Types

- Even if they aren't exactly what you want
 - Dates: only one format YYYY-MM-DD
 - Decimals: only one decimal separator
- Reformat to display if necessary
- If you define a new type you lose:
 - validation of the values
 - sorting capabilities
 - type-related functions in XSLT, XQuery



Granularity

- How much to break down data values?

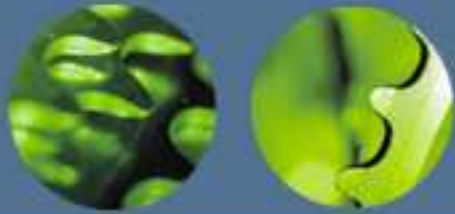
```
<updateDateTime>2005-05-24T09:32:04-05:00</updateDateTime>
```

```
<updateDateTime>  
  <date>2005-05-24</date>  
  <time>09:32:04</time>  
  <zone>-05:00</zone>  
</updateDateTime>
```

```
<updateDateTime>  
  <date>  
    <year>2005</year>  
    <month>05</month>...
```

```
<name>Priscilla  
Walmsley</name>
```

```
<name>  
  <first>Priscilla</first>  
  <last>Walmsley</last>  
</name>
```



Granularity Considerations

- Will it be accessed separately?
- Will it be sorted?
- Is it easy to get the parts if I store it as one value?
- Avoid combining values with different types into one element

– can't validate each part

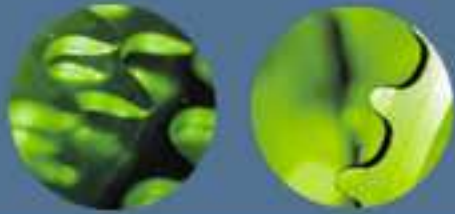
- except with cumbersome pattern

– have to parse to get each part

– don't have type information for each part

~~`<length>32cm</length>`~~

`<length unit="cm">32</length>`



Lists of Similar Simple Values

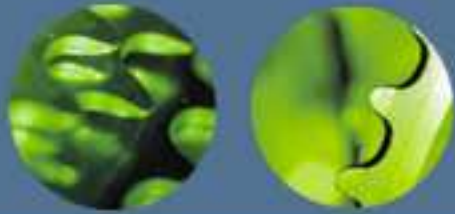
- Two approaches:

```
<availableSizes>  
  <size>10</size>  
  <size>12</size>  
  <size>14</size>  
</availableSizes>
```

using elements for
separate values

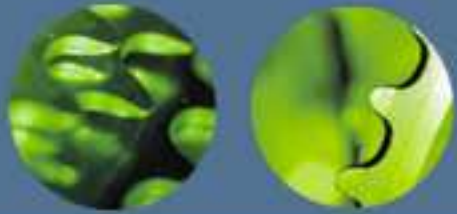
```
<availableSizes>10 12 14</availableSizes>
```

using whitespace-
separated lists



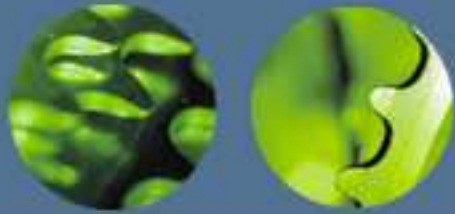
List Type Example

```
<xs:simpleType name="AvailableSizesType">
  <xs:list>
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="2"/>
        <xs:maxInclusive value="18"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:list>
</xs:simpleType>
```



When To Use Lists?

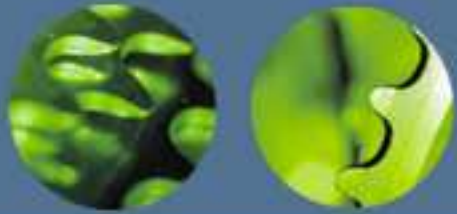
- Disadvantages
 - cannot represent values that contain spaces
 - less extensible
 - limited support in some processing technologies
 - e.g. XSLT/XPath 1.0, DOM
 - have to parse values yourself
- Advantages
 - less verbose
 - supported in XSLT 2.0 and XQuery



Whitespace Processing Examples

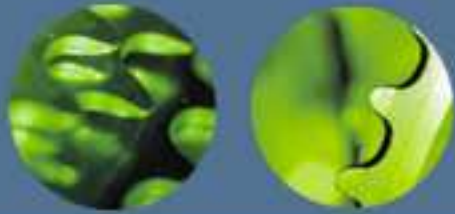
```
<name>  Name
on  2  lines  </name>
```

If the type is:	The whitespace facet is:	The final value passed to the processor is:
<code>xs:string</code>	<code>preserve</code>	<code> Name on 2 lines</code>
<code>xs:normalizedString</code>	<code>replace</code>	<code> Name on 2 lines</code>
<code>xs:token</code>	<code>collapse</code>	<code>Name on 2 lines</code>



string, normalizedString, or token?

- **xs:string**
 - when formatting (tabs, line breaks, indentation) is significant
 - for long strings, consider using a mixed content type instead
- **xs:normalizedString**
 - formatting is not significant, but consecutive whitespace is, perhaps because information is positional
- **xs:token**
 - good for most short atomic strings, especially ones constrained by enumerations or patterns.



Representing Missing Values

- Three methods (at least)
 - Empty elements
 - Missing elements
 - Nilled elements
- No semantics are assigned to any of these methods by XML Schema



Missing Values

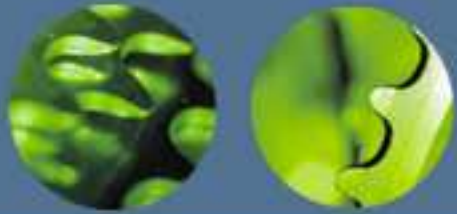
```
<order>
  <custName>
    <first>John</first>
    <middle/>
    <last>Smith</last>
  </custName>
  <items>
    <shirt>
      <giftWrap/>
      <number>557</number>
      <size></size>
    </shirt>
    <umbrella>
      <number>443</number>
      <size></size>
    </umbrella>
  </items>
</order>
```

empty middle element

empty giftWrap element

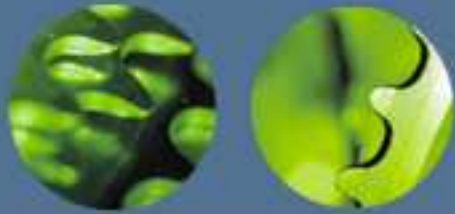
empty size elements

absent giftWrap element



Reasons for "Missing" Elements

- Not applicable
 - umbrellas do not come in different sizes
- We do not know whether the information is applicable
 - we do not know whether the customer has a middle name
- It is not relevant to this particular application of the data
 - the billing application does not care about product sizes
- It is the default, so it is not specified
 - the customer's title should default to "Mr."
- It actually is present and the value is an empty string
 - the gift wrap value for the shirt is empty, meaning "none"
- It is erroneously missing because of a user error or technical bug
 - we should have a size for the shirt

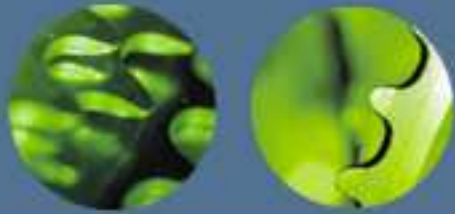


Nilled Element

- Use `xsi:nil="true"` on the element.
 - no content is allowed, but other attributes are

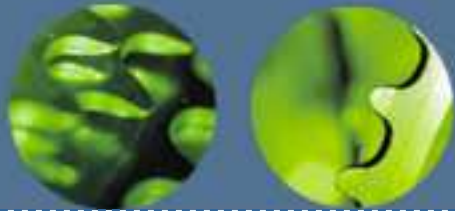
```
<xs:element name="size" type="xs:integer"  
  nillable="true" />
```

```
<size xsi:nil="true" />  
<size xsi:nil="true"></size>  
<size xsi:nil="true" system="US-DRESS" />  
<size xsi:nil="false">10</size>  
<size xsi:nil="true">10</size> <!--INVALID! -->
```



Benefits of Nillable Elements

- Do not weaken the type by allowing empty content and/or making attributes optional
- Allows you to make a deliberate statement that the information does not exist
 - a clearer message than simply omitting the component declaration
- Application may be relying on that element or attribute being there, for example as a placeholder
 - provides a way for it to exist without imparting any additional information
- Can easily turn off default value processing
 - default values will not be added if it is marked as nil



Missing Values Revisited

```
<order xmlns:xsi=...>
  <custName>
    <title/>
    <first>John</first>
    <middle xsi:nil="true"/>
    <last>Smith</last>
  </custName>
  <items>
    <shirt>
      <number>557</number>
      <size></size>
    </shirt>
    <umbrella>
      <number>443</number>
    </umbrella>
  </items>
</order>
```

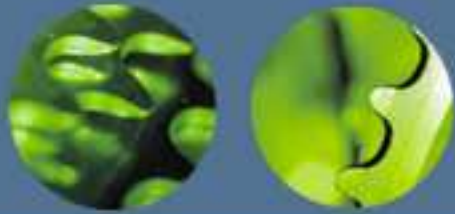
a default value will be filled in

middle is an unknown value

absent giftWrap element

invalid empty size element

absent size element

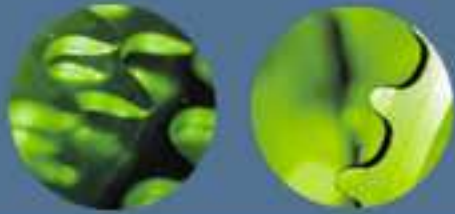


Empty Values and Non-String Types

- Non-string-based simple types do not allow empty values
- Consider making the element/attribute optional

```
<xs:element name="size" type="xs:integer"/>
```

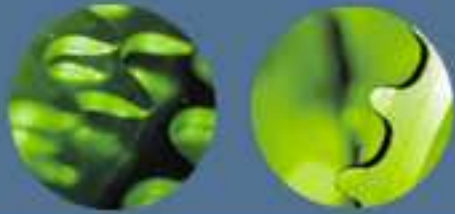
```
<size/>  
<size></size>
```



Allowing Empty Values for Non-String Types

```
<xs:simpleType name="DressSizeType">
  <xs:union memberTypes="xs:integer">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="" />
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

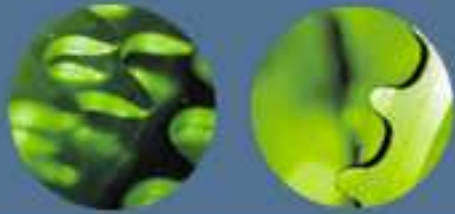
```
<dressSize>12</dressSize>
<dressSize />
<dressSize></dressSize>
<dressSize>     </dressSize>
```



Code Lists

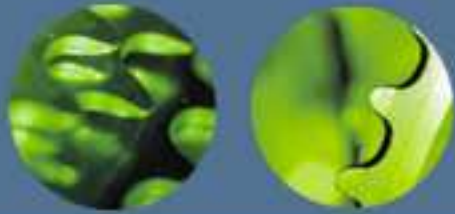
- Lists of valid values
 - generally represented by enumerations
 - usually tokens, often abbreviations

```
<xs:simpleType name="USState">
  <xs:restriction base="xs:token">
    <xs:enumeration value="AK" oth:desc="Alaska" />
    <xs:enumeration value="AL" oth:desc="Alabama" />
    <!-- ... -->
  </xs:restriction>
</xs:simpleType>
```



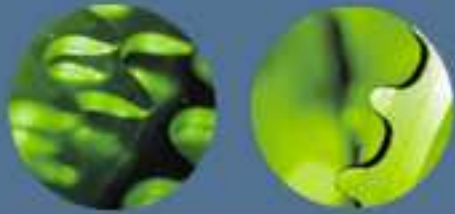
Code List Challenge 1: Constant Change

- Code lists change all the time
 - sometimes beyond your control
 - e.g. ISO languages, currencies, countries
- Recommendation: keep volatile code lists in separate schema documents
 - allows them to be versioned separately
 - other schema documents can include the most recent version



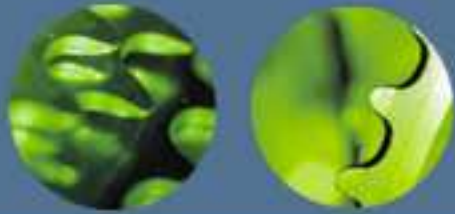
Code List Challenge 2: Lengthy Code Lists

- Some code lists can have hundreds of values
- Lengthy code lists can:
 - slow down validation
 - clutter up schemas
 - be more difficult to manage as they change
- Recommendation: do not use code lists of more than 15 or 20 values
 - instead, document them in a separate place
 - consider using a pattern to weed out obviously wrong values



Code List Challenge 3: Lack of Extensibility

- Simple types cannot be extended
 - this does not allow you to add custom values
- Recommendation: if customization is important:
 - use a simple token, or a union of a token plus a list of recommended values
 - other schema documents can restrict the type to include the values they want



Extensible Code Lists

```
<xs:simpleType name="TransactType">  
  <xs:union memberTypes="xs:token">  
    <xs:simpleType>  
      <xs:restriction base="xs:token">  
        <xs:enumeration value="UPDATE"/>  
        <xs:enumeration value="DELETE"/>  
      </xs:restriction>  
    </xs:simpleType>  
  </xs:union>  
</xs:simpleType>
```

allows any token

lists suggested values

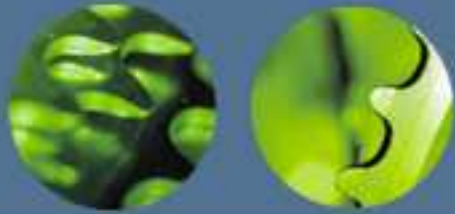
```
<xs:simpleType name="MyTransactType">  
  <xs:restriction base="TransactType">  
    <xs:enumeration value="UPDATE"/>  
    <xs:enumeration value="CREATE"/>  
  </xs:restriction>  
</xs:simpleType>
```

can include new values



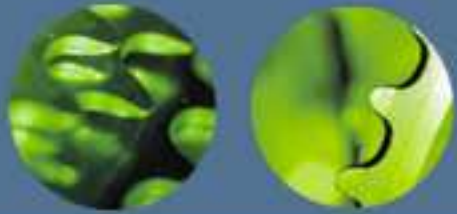
Namespaces and Names





Namespaces and Names

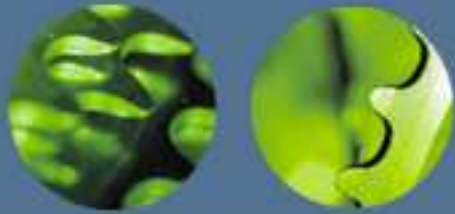
- Agenda
 - General namespace recommendations
 - Namespaces in schema documents
 - Qualified and unqualified forms
 - Structuring namespaces
 - Naming standards



The Purpose of Namespaces

- Disambiguate names used in many contexts
- Identify the source of the definition/standard to which it belongs

```
<prod:product xmlns:prod="http://datypic.com/prod">  
  <prod:number>...</prod:number>  
  <prod:size>...</prod:size>  
</prod:product>
```



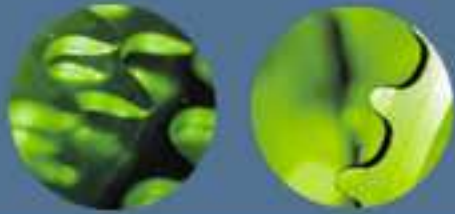
Namespace Names

- Namespace names are URIs
 - this does not necessarily mean they can be dereferenced
- URIs include
 - URLs, e.g. `http://datypic.com/prod`
 - URNs, e.g. `urn:prod:datypic`
- In reality, many text strings are valid relative URI references, so anything goes
- Relative URI references are deprecated
 - `prod` does not really disambiguate



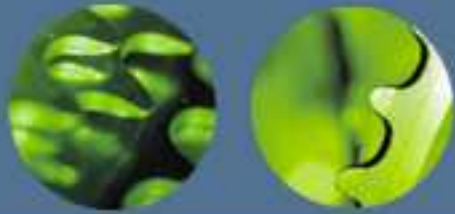
Namespace Declarations and Prefixes in Instances

- Namespace declarations
 - put all namespaces in the document on the root element
 - use default namespace declarations only if one namespace is clearly dominant
- Prefixes
 - keep them reasonably short for clarity
 - max 4 digits
 - following conventions is better, e.g.
 - **xs** or **xsd** for schemas (either one is fine)
 - **xs1** for XSLT stylesheets
 - **wsdl** for Web Services Description Language



Namespace Declarations in Schema Documents

- Schema documents use regular XML namespace declarations
 - must declare the XML Schema Namespace for it to be recognized as a schema
 - must declare the target namespace to reference components in it
- 3 approaches to using namespace declarations in a schema document:
 - Default the target namespace
 - Default the XML Schema namespace
 - Default neither namespace



Approach 1: Default the Target Namespace

- Most common approach
- Easy to see what is built into XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://datypic.com/prod"
           xmlns="http://datypic.com/prod">
```

```
<xs:element name="product" type="ProductType"/>
```

```
<xs:element name="size" type="xs:integer"/>
```

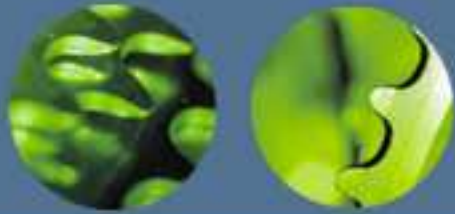
```
<xs:complexType name="ProductType">
```

```
<!-- ... -->
```

```
</xs:complexType>
```

```
</xs:schema>
```

References to other components in target namespace are unprefixed



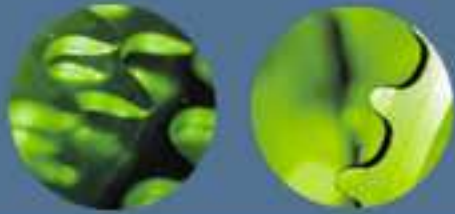
Approach 2: Default the Schema Namespace

- Do not use if there's no target namespace
 - no way to reference those components

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://datypic.com/prod"
        xmlns:prod="http://datypic.com/prod">

  <element name="product" type="prod:ProductType"/>
  <element name="size" type="integer"/>
  <complexType name="ProductType">
    <!-- ... -->
  </complexType>
</schema>
```

Schema element names
and built-in type names are
unprefixed



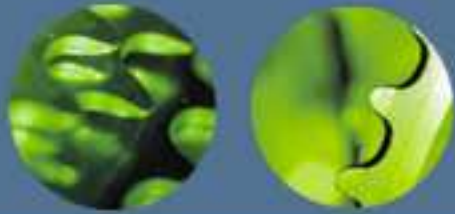
Approach 3: Default Neither Namespace

- Clearest approach, especially when multiple namespaces are imported

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://datypic.com/prod"
           xmlns:prod="http://datypic.com/prod">

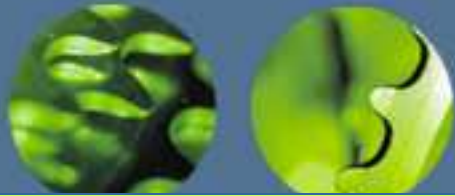
  <xs:element name="product" type="prod:ProductType"/>
  <xs:element name="size" type="xs:integer"/>
  <xs:complexType name="ProductType">
    <!-- ... -->
  </xs:complexType>
</xs:schema>
```

Both references and schema elements are prefixed (but not contents of name attribute)



Qualified vs. Unqualified Forms

- Form determines whether locally declared elements are qualified (in a namespace) in the instance document
- Specified using either
 - `elementFormDefault` attribute on the `schema` element (preferred)
 - `form` attribute on the `element` element
- Unqualified is the default for elements

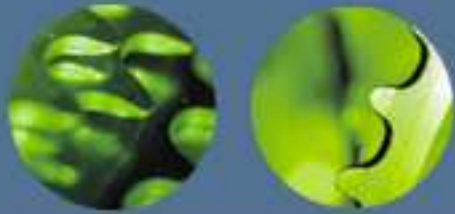


Unqualified Local Elements

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://datypic.com/prod">
  <xs:element name="product"> ← globally declared
    <xs:complexType>
      <xs:sequence>
        <xs:element name="number" /> ← locally declared
        <xs:element name="size" /> ← locally declared
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

qualified

```
<prod:product xmlns:prod="http://datypic.com/prod">
  <number>...</number> ← unqualified
  <size>...</size> ← unqualified
</prod:product>
```



Qualified Local Elements

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://datypic.com/prod"
  elementFormDefault="qualified">
  <xs:element name="product">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="number" />
        <xs:element name="size" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

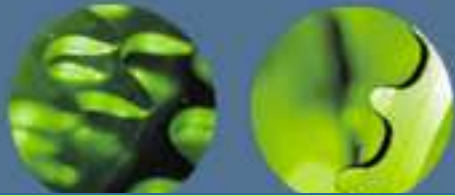
form is qualified

globally declared

locally declared

```
<prod:product xmlns:prod="http://datypic.com/prod">
  <prod:number>...</prod:number>
  <prod:size>...</prod:size>
</prod:product>
```

all elements qualified



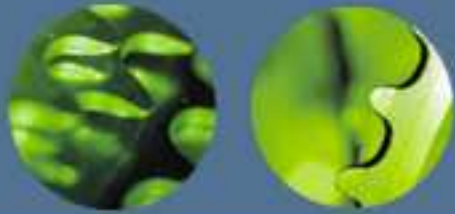
Form and Default Namespace Declarations

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://datypic.com/prod">
  <xs:element name="product">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="number"/>
        <xs:element name="size"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

WARNING: Unqualified local elements do not mix with default namespace declarations

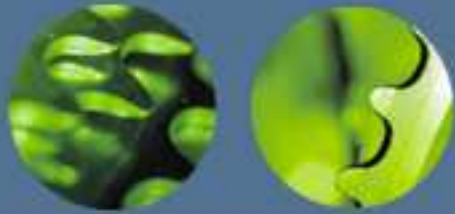
```
<product xmlns="http://datypic.com/prod">
  <number>...</number>
  <size>...</size>
</product>
```

all elements qualified



Advantages of Qualified Forms

- Very obvious which namespace an element is in; more clear
- Allow use of default namespace declarations
- Allow mixing of global and local element declarations without confusing instance author



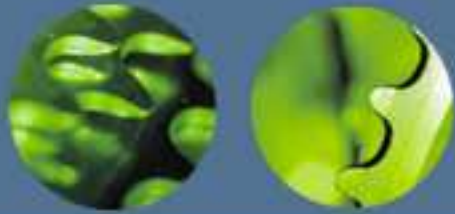
Advantages of Unqualified Forms

- When many namespaces are involved, can simplify instance documents

```
<ord:order xmlns:ord="http://datypic.com/ord">  
  <number>123ABBCC123</number>  
  <items>  
    <product>  
      <number>557</number>  
    </product>  
  </items>  
</ord:order>
```

No need to declare prod namespace

- Allow type restriction in other namespaces

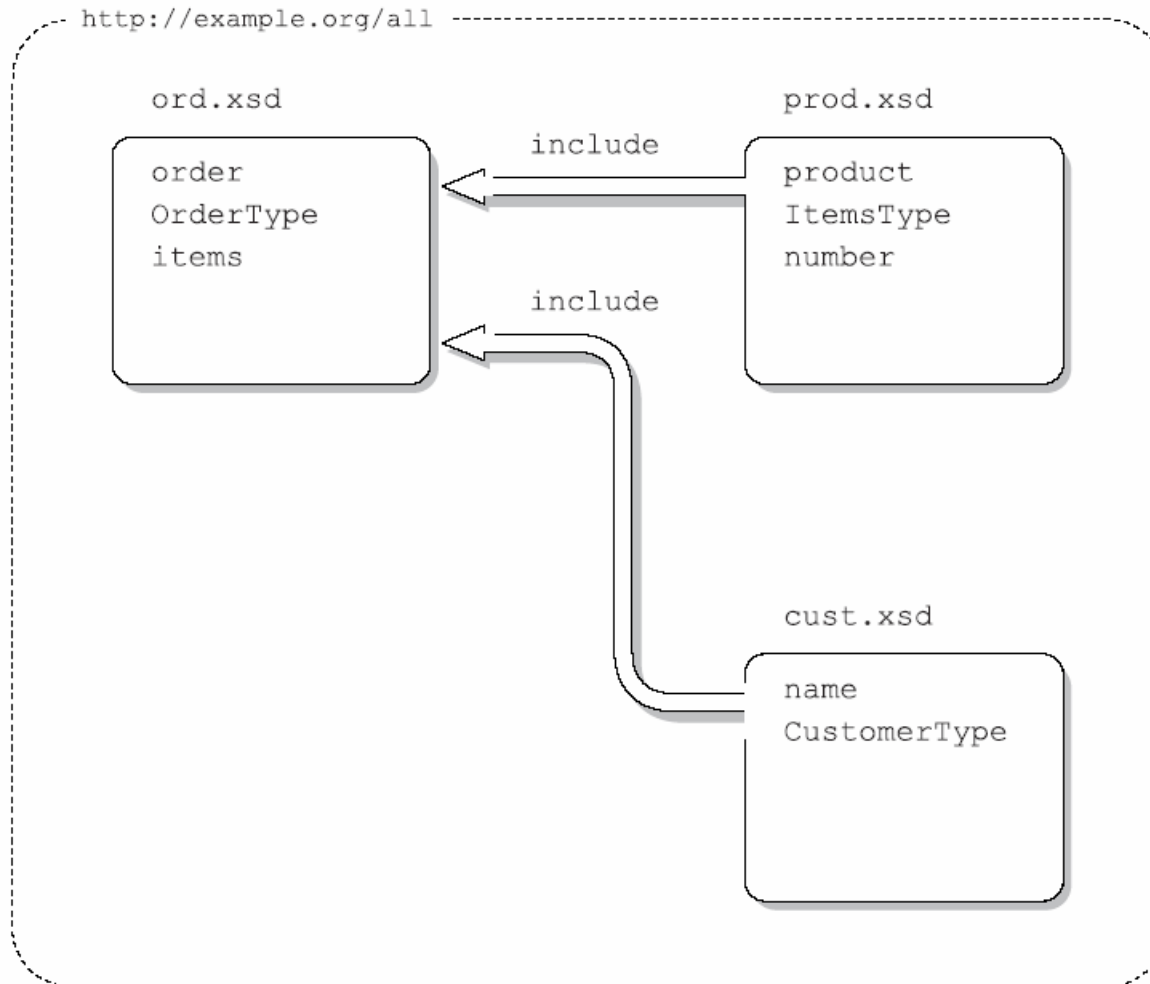


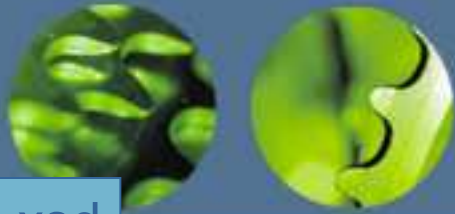
Structuring Namespaces

- Three approaches:
 - Same Namespace
 - all schema documents use the same namespace
 - Different Namespaces
 - many different namespaces are used, up to one per schema document
 - Chameleon Namespace
 - shared components have no target namespace; take on the namespace of the including schema document



Single Namespace

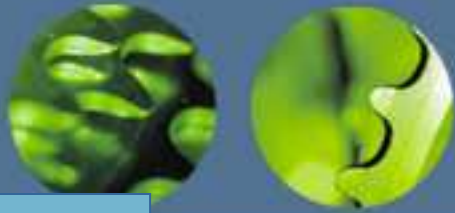




Single Namespace Approach: Schema

ord.xsd

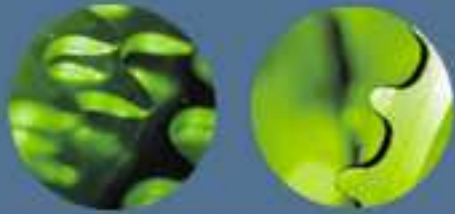
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://example.org/all"
           targetNamespace="http://example.org/all"
           elementFormDefault="qualified">
  <xs:include schemaLocation="prod.xsd"/>
  <xs:include schemaLocation="cust.xsd"/>
  <xs:element name="order" type="OrderType"/>
  <xs:complexType name="OrderType">
    <xs:sequence>
      <xs:element name="customer" type="CustomerType"/>
      <xs:element name="items" type="ItemsType"/>
    </xs:sequence>
  </xs:complexType>
  <!--...-->
</xs:schema>
```



Single Namespace Approach: Schema

prod.xsd

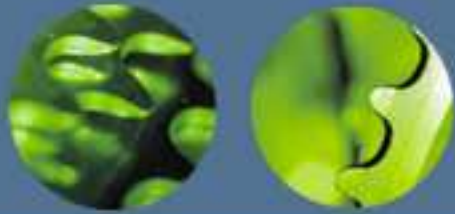
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://example.org/all"
           targetNamespace="http://example.org/all"
           elementFormDefault="qualified">
  <xs:complexType name="ItemsType">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="product" type="ProductType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ProductType">
    <xs:sequence>
      <xs:element name="number" type="ProdNumType"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```



Single Namespace Approach: Schema

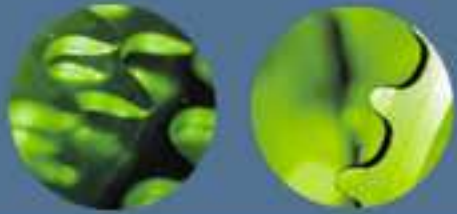
cust.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://example.org/all"
           targetNamespace="http://example.org/all"
           elementFormDefault="qualified">
  <xs:complexType name="CustomerType">
    <xs:sequence>
      <xs:element name="name" type="CustNameType" />
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="CustNameType">
    <xs:restriction base="xs:string" />
  </xs:simpleType>
</xs:schema>
```



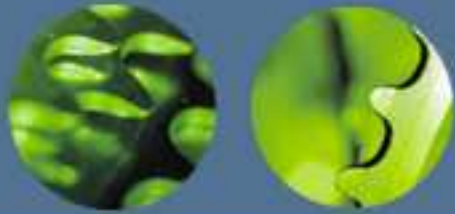
Single Namespace Approach: Instance

```
<order xmlns="http://example.org/all"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://example.org/all ord.xsd">  
<customer>  
  <name>Priscilla Walmsley</name>  
</customer>  
<items>  
  <product>  
    <number>557</number>  
  </product>  
</items>  
</order>
```

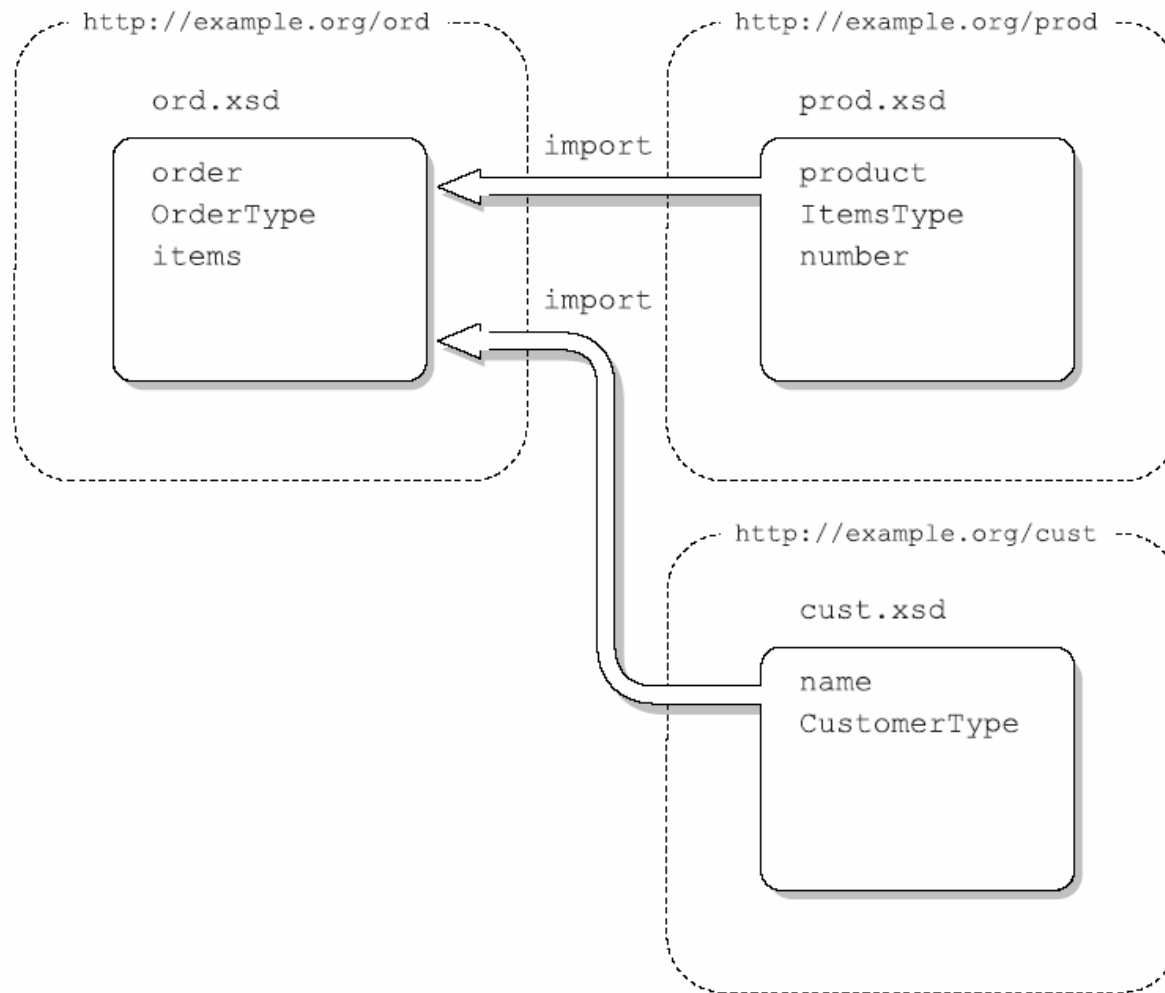


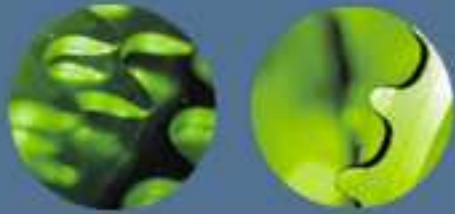
Single Namespace Approach: Analysis

- Advantages
 - uncomplicated
 - instance not cluttered by prefixes
- Disadvantages
 - name collisions possible
 - not as easy to assemble different schema documents dynamically
 - if using namespaces for versioning, harder to version components separately



Different Namespaces





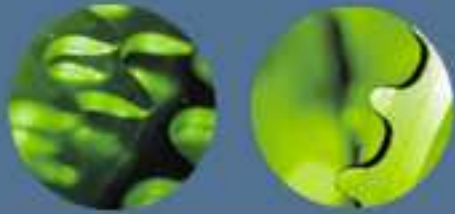
Different Namespaces Approach: Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:prod="http://example.org/prod"
            xmlns:cust="http://example.org/cust"
            xmlns="http://example.org/ord"
            targetNamespace="http://example.org/ord"
            elementFormDefault="qualified">
  <xs:import schemaLocation="prod.xsd"
            namespace="http://example.org/prod"/>
  <xs:import schemaLocation="cust.xsd"
            namespace="http://example.org/cust"/>
  <xs:element name="order" type="OrderType"/>
  <xs:complexType name="OrderType">
    <xs:sequence>
      <xs:element name="customer" type="cust:CustomerType"/>
      <xs:element name="items" type="prod:ItemsType"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

ord.xsd

all 3
namespaces
declared in
schema

use import
instead of
include



Different Namespaces Approach: Schemas

prod.xsd

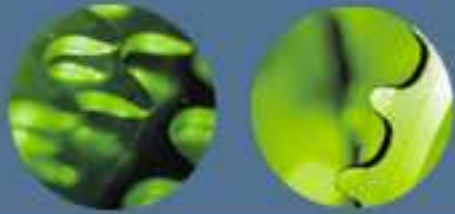
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns="http://example.org/prod"
            targetNamespace="http://example.org/prod"
            elementFormDefault="qualified">

    <!-- ... -->
</xs:schema>
```

cust.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns="http://example.org/cust"
            targetNamespace="http://example.org/cust"
            elementFormDefault="qualified">

    <!-- ... -->
</xs:schema>
```



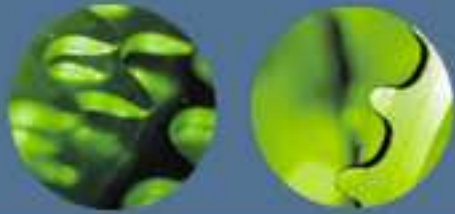
Different Namespaces Approach: Instance

all 3 namespaces declared in instance

```
<order xmlns="http://example.org/ord"
  xmlns:prod="http://example.org/prod"
  xmlns:cust="http://example.org/cust"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://example.org/ord ord.xsd">
  <customer>
    <cust:name>Priscilla Walmsley</cust:name>
  </customer>
  <items>
    <prod:product>
      <prod:number>557</prod:number>
    </prod:product>
  </items>
</order>
```

schemaLoc. only refers to "main" schema

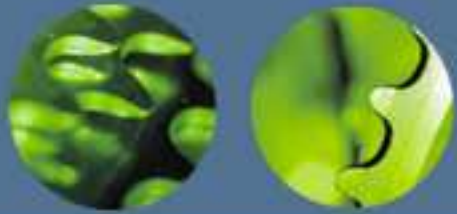
elements declared in imported schemas prefixed



Different Namespaces Approach: Instance 2

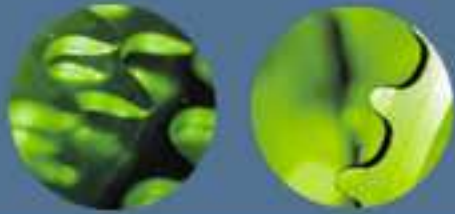
```
<order xmlns="http://example.org/ord"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://example.org/ord ord.xsd">
  <customer>
    <name xmlns="http://example.org/cust">
      Priscilla Walmsley</name>
    </customer>
    <items>
      <product xmlns="http://example.org/prod">
        <number>557</number>
      </product>
    </items>
  </order>
```

default namespace declarations can appear at different levels to avoid prefixes



Different Namespaces Approach: Analysis

- Advantages
 - source and context of an element are clear
 - divides responsibility among groups
 - namespaces can be maintained (and versioned) separately
 - less concern about name collisions
- Disadvantages
 - instances are more complex
 - schema imports slightly more complex than includes
 - cannot use redefines



Different Namespaces with Unqualified Locals

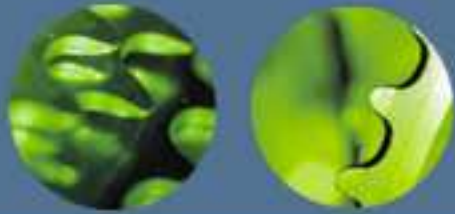
- If elementFormDefault="unqualified":

```
<ord:order xmlns="http://example.org/ord"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://example.org/ord ord.xsd">
  <customer>
    <name>Priscilla Walmsley</name>
  </customer>
  <items>
    <product>
      <number>557</number>
    </product>
  </items>
</ord:order>
```

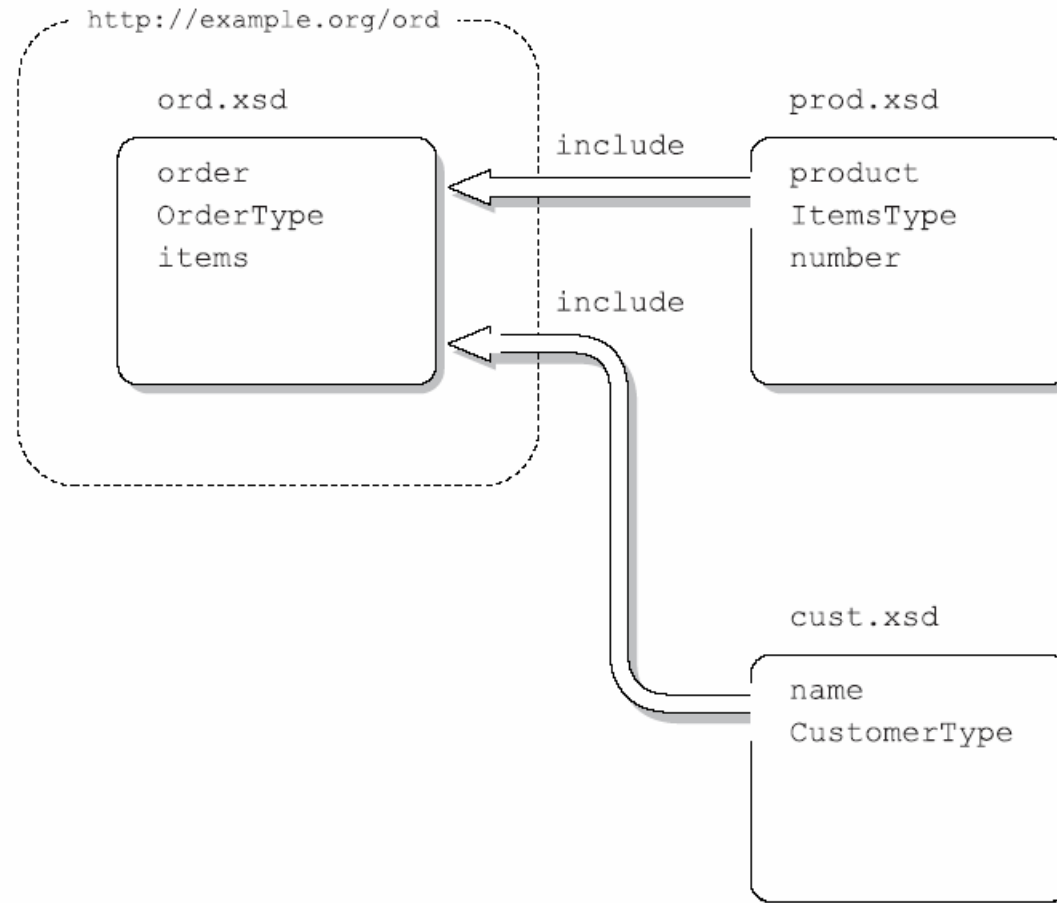
only the ord namespace is declared

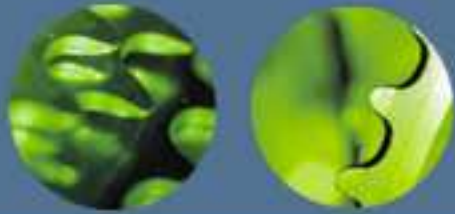
only the order element is prefixed

Why? Because we imported types, not elements.



Chameleon Namespace





Chameleon Namespace Approach: Schemas

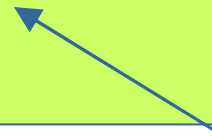
ord.xsd



Same as `ord.xsd` schema used in Same Namespace Approach

prod.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <!-- ... -->
</xs:schema>
```

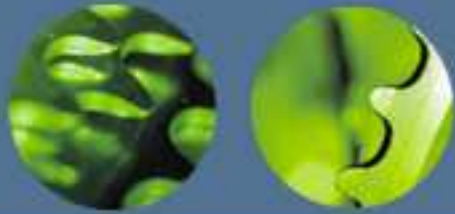


cust.xsd

No target namespaces

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <!-- ... -->
</xs:schema>
```

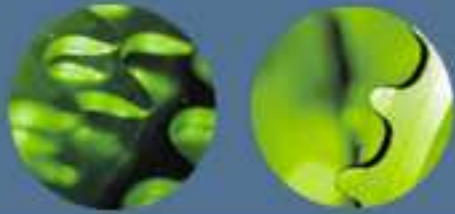




Chameleon Namespace Approach: Instance

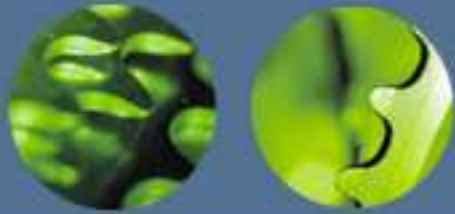
```
<order xmlns="http://example.org/all"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://example.org/all ord.xsd">
  <customer>
    <name>Priscilla Walmsley</name>
  </customer>
  <items>
    <product>
      <number>557</number>
    </product>
  </items>
</order>
```

The instance is exactly the same as in the same namespace approach.



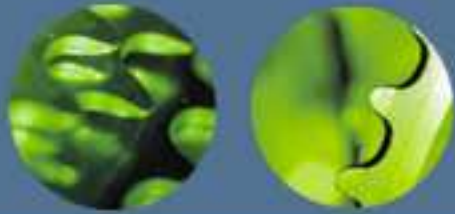
Chameleon Namespace Approach: Analysis

- Advantages
 - flexibility
 - components can be included in multiple namespaces, as desired
 - build a library of reusable components in many contexts
- Disadvantages
 - risk of name collisions
 - Among no-namespace components, as well as schemas that include them
 - components lack recognizable identity
 - Specific semantics, documentation, and application code



Choosing Names

- Use names that:
 - match your users' experience
 - should be easy to understand and remember
 - are not cryptic
 - e.g. PTNM
 - use only well-known abbreviations
 - space is not the priority it used to be
 - are not unwieldy
 - `CarrierDetailsSpecialHandlingOrHazardousMaterialsOrBoth`



Name Consistency

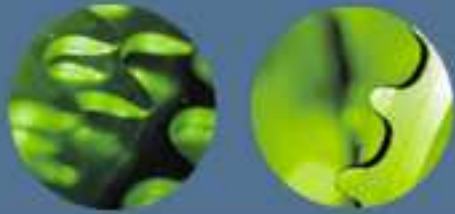
- Capitalization
- Separators
 - camel case, or
 - use of special characters -, _, .
- Develop a glossary of standard terms
 - name, abbreviation, description, synonyms
 - use them consistently

PartNumber
part-number
part_number
part.number
partNumber



Sample Glossary

Term	Abbreviation	Synonyms
color	color	
currency	curr	
customer	cust	client, purchaser, account holder
date	date	
description	desc	
effective	eff	begin, start
identifier	id	

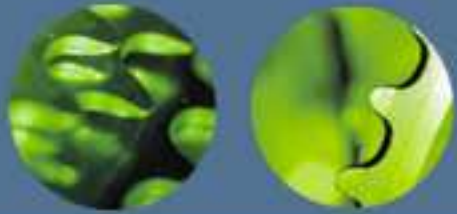


Repeating Parent Names

- Prefixing child elements with their parent's name (or an abbreviation for it)

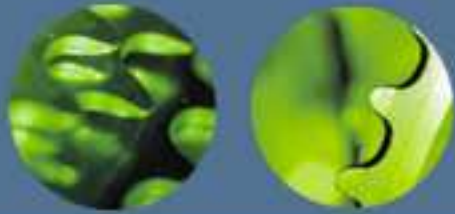
```
<product>  
  <prodNumber>557</prodNumber>  
  <prodName>Short-Sleeved Linen Blouse</prodName>  
  <prodSize sizeSystem="US-DRESS">10</prodSize>  
</product>
```

```
<product>  
  <number>557</number>  
  <name>Short-Sleeved Linen Blouse</name>  
  <size system="US-DRESS">10</size>  
</product>
```



Repeating Parent Names

- Advantages
 - full meaning of the element is more clear
 - easier to process using technologies like:
 - SAX: no need to keep track of parent
 - DOM: `getElementsByTagName` is more useful
- Disadvantages
 - more verbose, generally redundant
 - hides the fact that two elements might represent the same thing



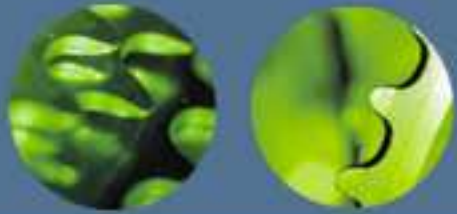
Names of Types and Groups

- Types and groups and elements can all have the same names
 - but this is confusing
- Recommendations:
 - add "Type" or "Group" to the end of the name to identify its component type clearly
 - start type and group names with an upper case letter, like Java class names



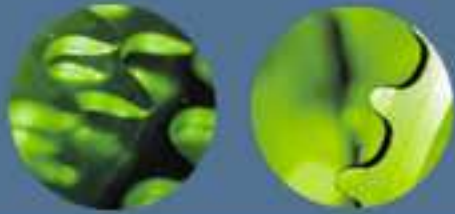
Structuring Schemas





Structuring Schemas

- Agenda
 - Global and local declarations
 - Named and anonymous types
 - Schema structure patterns
 - Schema modularity



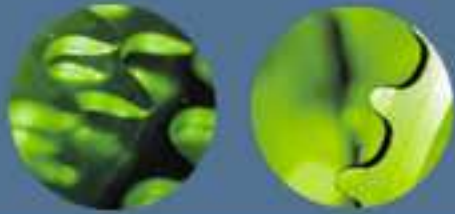
Global Element Declarations

- Direct children of `xs:schema` element
- Referenced from complex types

```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://datypic.com/prod"
  targetNamespace="http://datypic.com/prod">
  <xs:element name="name" type="xs:string"/>
  <xs:element name="size" type="xs:integer"/>
  <xs:complexType name="ProductType">
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="size" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

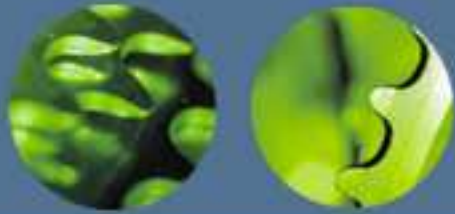
declarations

references



Global Element Declarations

- Must have unique names within entire schema
 - not just that schema document, but all included schema documents
- Can be reused in many complex types
- Are always qualified in instance documents
 - with the target namespace of the schema

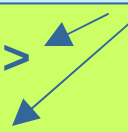


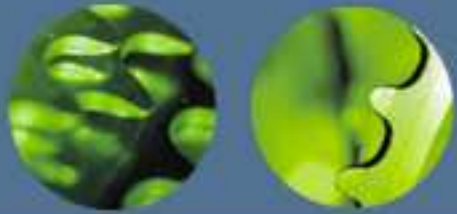
Local Element Declarations

- Completely contained in a complex type

```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://datypic.com/prod"
  targetNamespace="http://datypic.com/prod">
  <xs:complexType name="ProductType">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="size" type="xs:integer"
        minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

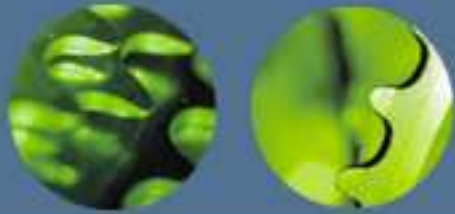
declarations





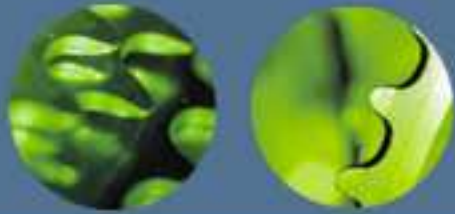
Local Element Declarations

- Are not required to have unique names within the schema
- May be qualified or unqualified in instance documents
 - depends on the element form
- Cannot be used to validate the root element



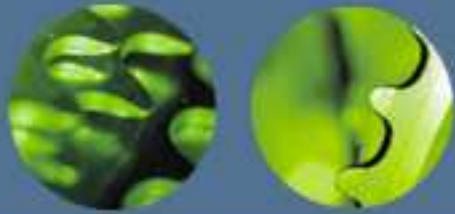
Use Global Element Declarations If:

- The element could ever be the root element
 - locally declared elements cannot be validated independent of their parents.
- The element could be used to match a wildcard
- You want to use the exact same element declaration in more than one complex type
 - including things like default value, identity constraints, etc.
- You want to use the element in a substitution group
 - local element declarations cannot participate in substitution groups.



Use Local Element Declarations If:

- You want to use unqualified element names
 - make all of the element declarations local except for the declaration of the root element
 - mixing global and local declarations will make instances confusing
- You want to declare several elements with the same name but different types or other properties
 - For example: two element declarations for `size`
 - one that is a child of `shoe` has the type `ShoeSizeType`
 - one that is a child of `hat` has the type `HatSizeType`

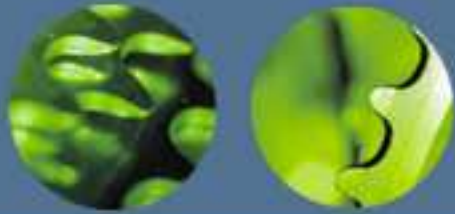


Global vs. Local Attribute Declarations

- Global attributes are *always* qualified
 - usually not what you want, because you will have to prefix every attribute
 - adds clutter without adding information

```
<prod:product prod:effDate="2004-08-14"  
              prod:number="556"  
              prod:size="12" />
```

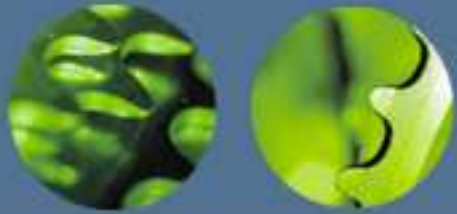
- Qualified attributes are appropriate if they will be used in different unrelated vocabularies
 - e.g. `xml:lang`, `xsi:type`



Global vs. Local Attribute Declarations

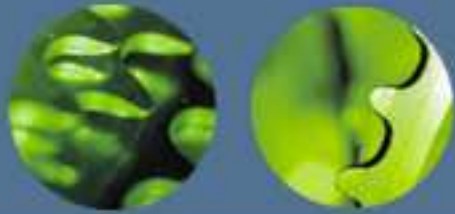
- Recommendation:
 - almost always use local attribute declarations
 - always use `attributeFormDefault="unqualified"` (the default)
- To achieve attribute reuse, either:
 - use local declarations and just reuse the type
 - put declarations in attribute groups and reuse the groups

```
<xs:attributeGroup name="reusedAttr">  
  <xs:attribute name="reusedAttr" type="xs:string"/>  
</xs:attributeGroup>
```



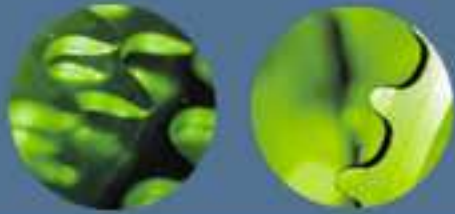
Named vs. Anonymous Types

- Named types
 - uniquely named within a schema
 - reusable by many elements and/or attributes
 - able to be restricted
- Anonymous types
 - local to a particular element or attribute declaration
 - cannot be reused or restricted



Advantages of Named Types

- Can be defined once and used many times
 - For example, a reusable type named `ProductCodeType` that represents the standard product codes in your organization
 - This has all the advantages of reuse (described later!)
- Can be restricted and extended, redefined, used in a list or union type
 - Enhances reusability, extensibility and ability to change
- Can make the schema more readable, when the type definitions are complex



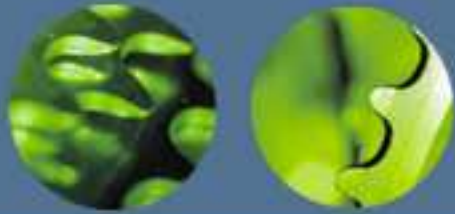
Advantages of Anonymous Types

- Not many!
- If the definition will never be reused, can make maintenance slightly easier
 - because you don't need to worry about impacts on other components
- Can be more readable when they are relatively simple
 - definition of the type right there with the element or attribute declaration



Schema Structure Patterns

		Element Declarations	
		Local	Global
Type Definitions	Anon/ Local	Russian Doll	Salami Slice
	Named/ Global	Venetian Blind	Garden of Eden

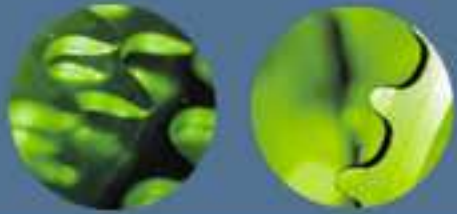


Salami Slice

```
<xs:element name="product">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="size" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="name" type="xs:string"/>
<xs:element name="size">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

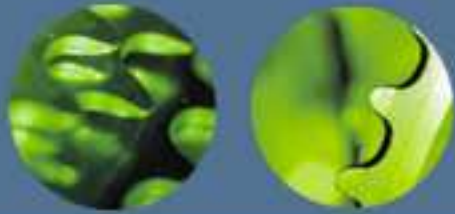
global element
declarations

anonymous type
declarations



Salami Slice

- Element declarations are global
- Types are anonymous (local)
- Similar to DTDs
- Characteristics
 - cannot use unqualified locals
 - every element must be uniquely named
 - cannot have a type that is dependent on context
 - no ability to reuse, extend, restrict or redefine types
 - any element can be the root

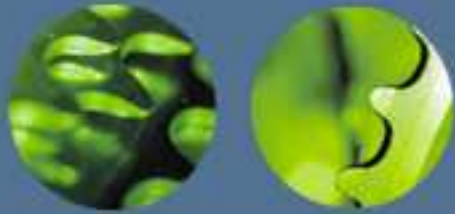


Russian Doll

```
<xs:element name="product">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="size" minOccurs="0">
        <xs:simpleType>
          <xs:restriction base="xs:integer">
            <xs:minInclusive value="1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

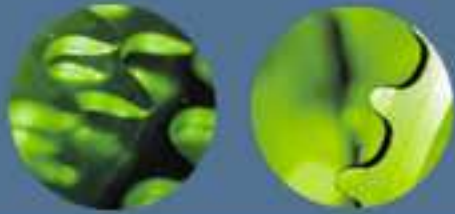
local element
declarations

anonymous type
declarations



Russian Doll

- Element declarations are local (except for root element)
- Types are anonymous (local)
- Characteristics
 - no reuse at all
 - very difficult to read
 - no ability to reuse, extend, restrict or redefine types

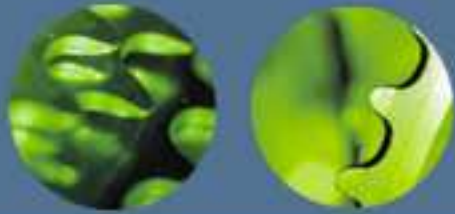


Garden of Eden

```
<xs:element name="product" type="ProductType"/>
<xs:element name="name" type="xs:string"/>
<xs:element name="size" type="SizeType"/>
<xs:complexType name="ProductType">
  <xs:sequence>
    <xs:element ref="name"/>
    <xs:element ref="size" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="SizeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
  </xs:restriction>
</xs:simpleType>
```

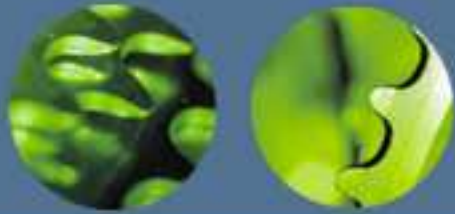
global element
declarations

named type
declarations



Garden of Eden

- Element declarations are global
- Types are named (global)
- Characteristics
 - lots of opportunity for reuse
 - can reuse both types and elements
 - difficult to read; disjointed
 - every element must be uniquely named
 - no unqualified elements
 - any element can be the root

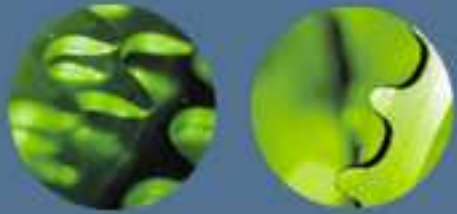


Venetian Blind

```
<xs:element name="product" type="ProductType"/>
<xs:complexType name="ProductType">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="size" type="SizeType"
      minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="SizeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
  </xs:restriction>
</xs:simpleType>
```

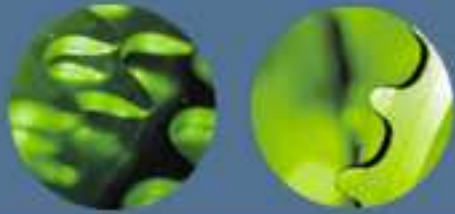
local element
declarations

named type
declarations



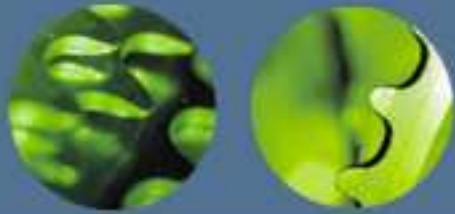
Venetian Blind

- Element declarations are local
- Types are named (global)
- Characteristics
 - the best of both worlds
 - types are reusable
 - which is all you need!
 - if using unqualified locals, great for reusable libraries of types



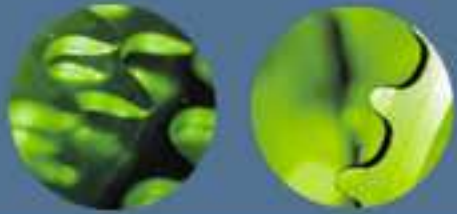
Schema Modularity

- Splitting schemas into multiple schema documents
- Why?
 - enable reuse and therefore consistency and interoperability
 - make maintenance easier
 - allow subsets to be versioned separately
 - provide control over read and write access
 - combine multiple namespaces
 - a schema document may only have one target namespace
 - allow processors and/or users to use only a subset of a schema



Lines Along Which to Divide Schemas

- Subject area
 - e.g. customers, products, accounting
- General/specific
 - e.g. core components vs. specific document types
- Person/group responsible
- Separate versioning schedule or volatility level
 - e.g. code lists may change more often



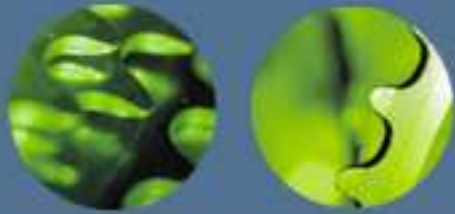
How Many Schema Documents?

- Too few schema documents
 - inhibit reuse
 - may slow down validation
 - force many components to be versioned together
- Too many schema documents
 - may speed up validation
 - if only the necessary components are included
 - may slow down validation
 - if all schema documents are included
 - can be confusing to maintain and manage



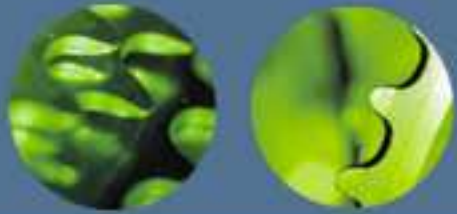
Flexibility and Substitutability





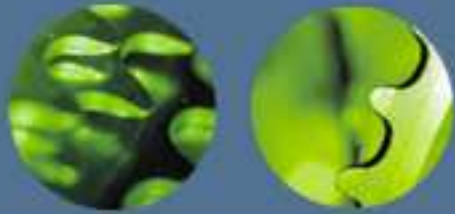
How Much Flexibility?

- Document structures that are less flexible are:
 - faster to learn and easier to remember
 - easier to write code to process
- Document structures that are more flexible are:
 - often more representative of the real world
 - easier to reuse and adapt for other purposes
 - including future versions



Several Methods for Providing Flexibility

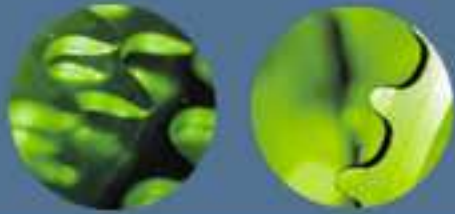
- Choice groups
- Substitution groups
- Type substitution
- Wildcards



Choice Groups

- More than one path in a content model

```
<xs:complexType name="ParaType">  
  <xs:choice minOccurs="0" maxOccurs="unbounded">  
    <xs:element name="italics"/>  
    <xs:element name="bold"/>  
    <xs:element name="break"/>  
    <xs:element name="pagebreak"/>  
  </xs:choice>  
</xs:complexType>
```

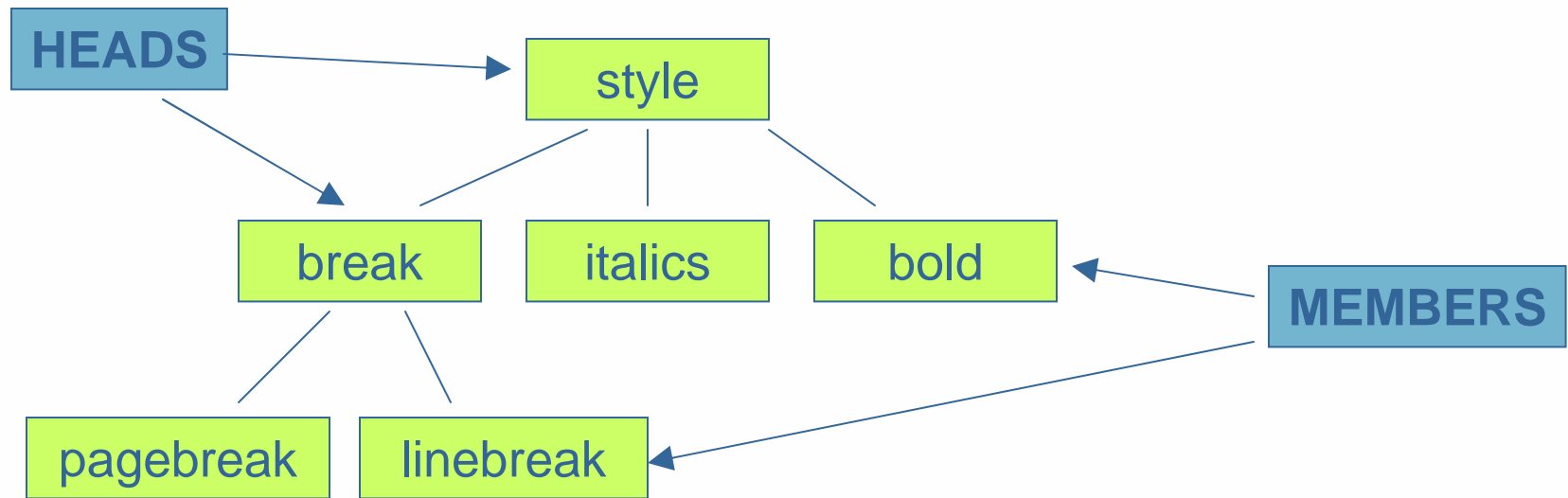


Substitution Groups

- Declaring groups of elements as substitutable for another element
 - easy replacement of elements in the same "class"
 - provide extensibility for choice groups

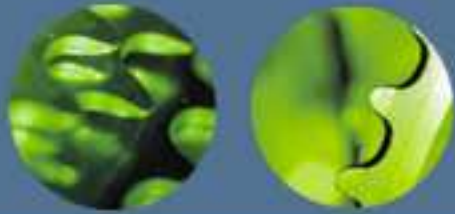


The Substitution Group Hierarchy



wherever **style** is valid, **break**, **italics**, **bold**, **pagebreak** and **linebreak** are also valid

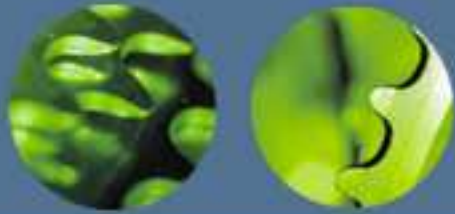
wherever **break** is valid, **pagebreak** and **linebreak** are also valid



Declaring a Substitution Group

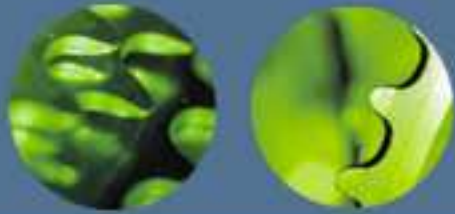
- Head element declared normally
- Member elements refer to head using `substitutionGroup` attribute
- Member types must be derived from head type

```
<xs:element name="style" type="StyleType"/>  
  
<xs:element name="break" type="BreakType"  
  substitutionGroup="style"/>  
  
<xs:element name="italics" type="ItalicsType"  
  substitutionGroup="style"/>  
  
<xs:element name="bold" type="BoldType"  
  substitutionGroup="style"/>
```



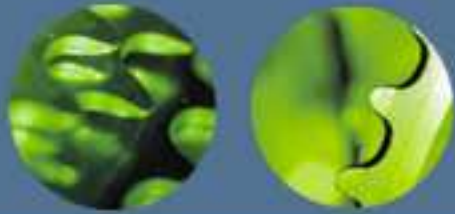
Choice Groups vs. Substitution Groups

- Use choice groups when:
 - there is a fixed set of choices
 - the element choices don't have similar types
 - you are using unqualified locals and don't want to break the standard with global declarations
 - it is very important to easily see what you're getting
- Use substitution groups when:
 - the set of choices is growing or flexible
 - the element choices can have the same or a derived type
 - the same set of choices is used over and over again, wherever the head element appears



Type Substitution

- Can substitute derived types for base types anywhere in an instance document
- Use the **xsi:type** attribute
 - specify the derived type name (qualified) as the value
- Element names must still be as stated in the content model



Type Substitution Example

```
<a>
  <street>123 Main</street>
  <city>New York</city>
</a>
```

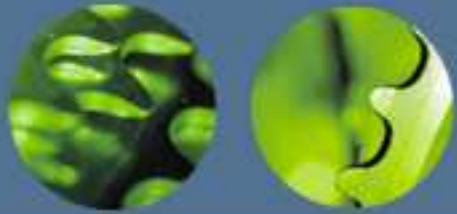
element a is of type AddrType, which allows (street, city)

USAddrType extends AddrType, adds state, zip

UKAddrType extends AddrType, adds postcd

```
<a xsi:type="USAddrType">
  <street>123 Main</street>
  <city>New York</city>
  <state>NY</state>
  <zip>10025</zip>
</a>
```

```
<a xsi:type="UKAddrType">
  <street>123 Main</street>
  <city>London</city>
  <postcd>SW14 8HA</postcd>
</a>
```



Content Models that Depend on Attributes

- Typical requirement:
 - if a product is perishable, make expiration date required, otherwise not
- One approach
 - declare a `product` element with:
 - a `perishable` attribute (boolean), and
 - an `expDate` child (optional)
 - XML Schema will not enforce that if `perishable="true"`, `expDate` is required.

```
<product perishable="true">  
  <expDate>2004-08-12</expDate>  
</product>
```

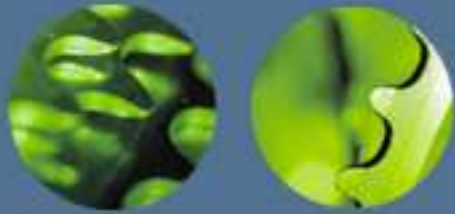


Content Models that Depend on Attributes

- Type substitution approach
 - declare a `product` element of type `ProductType` with no `expDate` child.
 - define a type called `perishable` that extends `ProductType` to add a required `expDate` child.

```
<product>  
  ...  
</product>
```

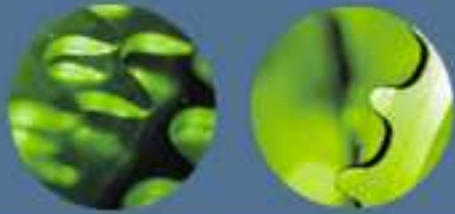
```
<product xsi:type="perishable">  
  ...  
  <expDate>2004-08-12</expDate>  
</product>
```



Wildcards

- Deliberately allowing for flexibility using **any** and **anyAttribute**
- Can be placed in a specific location in the content model
- Less control/more flexibility than choice groups or even substitution groups
 - no ability to control choices based on name or type, just namespace name and how many

```
<xs:any minOccurs="0" maxOccurs="unbounded"  
        namespace="##other" processContents="lax" />
```



Flexibility: Summary

- Choice groups
 - predefined set of allowed elements
- Substitution groups
 - extensible set of allowed elements
- Type substitution
 - element names are fixed, types can be extended
- Wildcards
 - allowed set constrained only by namespace, not name or type



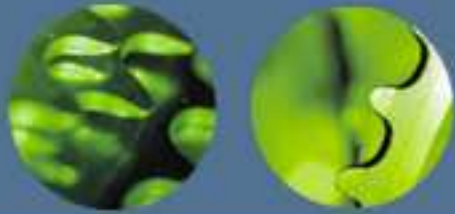
Reuse





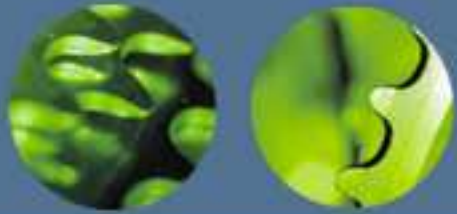
Reuse

- Reusing schema components in multiple:
 - definitions
 - declarations
 - schema documents
- Two kinds of reuse:
 - within your own schema documents
 - using components from other organizations
 - vendor standards organizations
 - trading partners



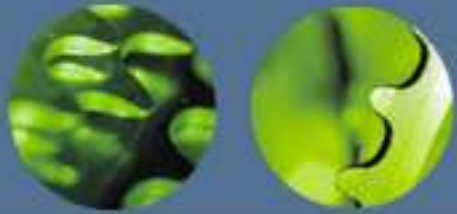
Reuse Benefits

- Reduces development time
 - developers are not busy reinventing the same schemas
 - processing code can be reused too
- Reduces maintenance time
 - global changes need only be made once
 - in both schemas and processing code
- Results in better-designed schemas with fewer errors
 - two heads are better than one
- Reduces the learning curve on schemas
 - consistency is easier to learn and understand
 - reused components only have to be learned once
 - schemas are less verbose and complex



Reusable Components

- Named types
 - both simple and complex
- Global element declarations
- Global attribute declarations
- Named model groups
- Attribute groups
- Entire schema documents



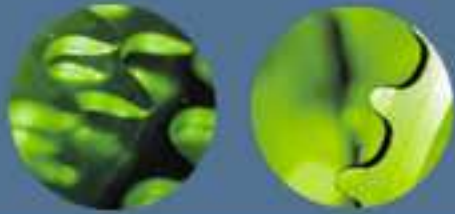
Reusing Elements and Types

- The element/type separation is one of the beauties of XML Schema
- You can have:
 - elements with different names and the same type
 - `shippingAddress` and `billingAddress` both have type `AddressType`
 - elements with the same name and different types
 - `number` child of `order` has a different pattern than `number` child of `product`



Three Methods of Reusing Shared Content

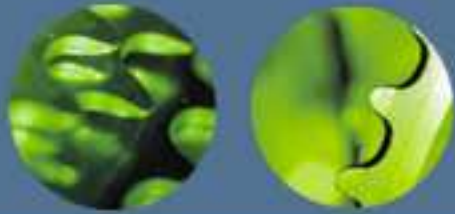
- Named model groups
- Type derivation
- Child elements



Named Model Group Example

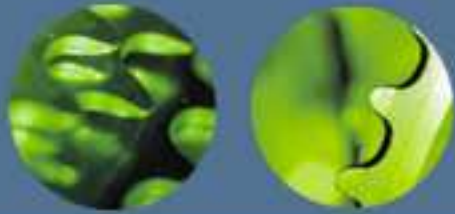
```
<xs:group name="DescGroup">
  <xs:sequence>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="comment" type="xs:string"/>
  </xs:sequence>
</xs:group>
```

```
<xs:complexType name="InvoiceType">
  <xs:complexType name="PurchaseOrderType">
    <xs:sequence>
      <xs:group ref="DescGroup" minOccurs="0"/>
      <xs:element ref="items"/>
      <!--...-->
    </xs:sequence>
  </xs:complexType>
```



Named Model Groups

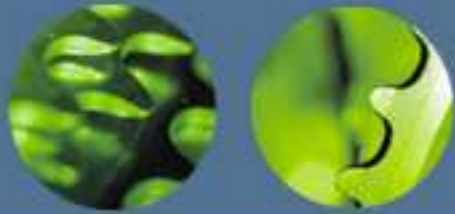
- Content model fragment defined once, used many times
- Advantages
 - shared content does not appear at the beginning of the content model
 - allows for redefinition more flexibly than types
 - allows you to reuse from multiple places
- Use named model groups when:
 - types are dissimilar concepts that happen to share a few elements/attributes
 - flexibility is important



Type Derivation as Reuse Mechanism

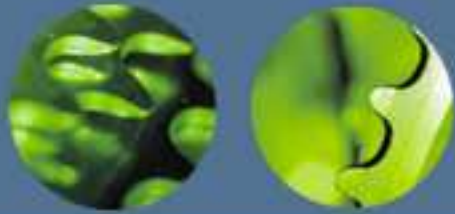
```
<xs:complexType name="DescribedType">
  <xs:sequence>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="comment" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="InvoiceType">
  <xs:complexType name="PurchaseOrderType">
    <xs:complexContent>
      <xs:extension base="DescribedType">
        <xs:sequence>
          <!-- ... -->
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```



Type Derivation as Reuse Mechanism

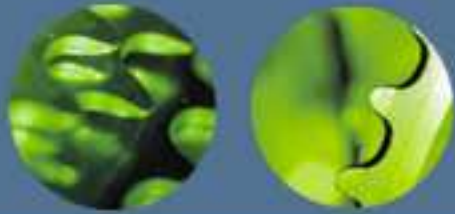
- Content model is inherited from base type
- Advantages
 - creates a type hierarchy that can be useful to applications
- Disadvantages
 - far less flexible in the ways you can reuse
- Use type derivation when:
 - shared content appears at the beginning
 - entire content models are very similar with just a few differences



Child Element as Reuse Mechanism

```
<xs:complexType name="DescribedType">  
  <xs:sequence>  
    <xs:element name="description" type="xs:string"/>  
    <xs:element name="comment" type="xs:string"/>  
  </xs:sequence>  
</xs:complexType>
```

```
<xs:complexType name="InvoiceType">  
  <  
    <xs:complexType name="PurchaseOrderType">  
      <xs:sequence>  
        <xs:element name="descInfo" type="DescribedType"/>  
      <  
        <!-- ... -->  
      </xs:sequence>  
    </xs:complexType>
```



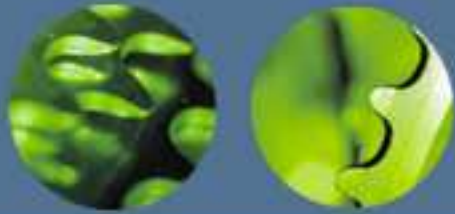
Child Element as Reuse Mechanism

- All shared content is included in a child container element
- Advantages
 - more obvious that something is being shared
- Disadvantages
 - changes the structure of the instance, possibly complicating it
- Use a child element when:
 - when it makes logical sense to have a container element there



Common Components

- For simple things like monetary amounts, measurements
 - but usually not so simple they are just a simple type
- Building blocks that lack a specific context
 - reusable across a set of schemas in unrelated contexts
- Generally defined as types rather than elements
 - so they can be used with many different element names

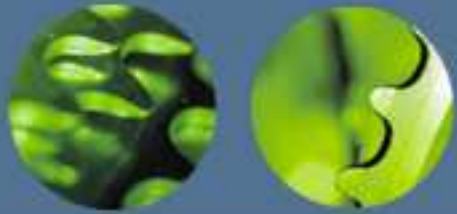


Common Component Example

```
<xs:complexType name="MeasurementType">
  <xs:simpleContent>
    <xs:extension base="xs:decimal">
      <xs:attribute name="unit" use="required"
        type="UnitListType" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

```
<xs:element name="height" type="MeasurementType" />
<xs:element name="quantity" type="MeasurementType" />
```

```
<height unit="meters">12.56</height>
<quantity unit="EA">15</quantity>
```



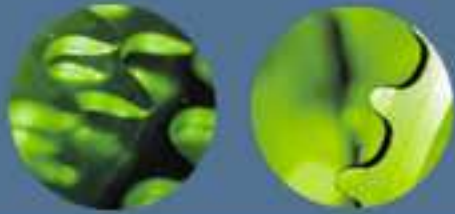
Creating Reusable Components

- Think about a broader applicability of your type
 - other contexts such as geopolitical, industry, etc.
- Use general names for types (if appropriate)
 - AddressType rather than CustomerAddressType
- Use named types and named model groups
- Break your schemas into smaller documents
 - others who will reuse your components will not have to include or import them all



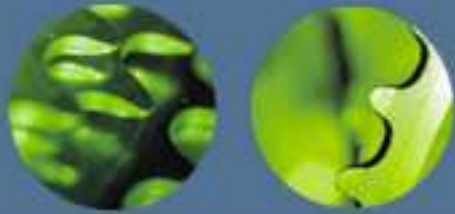
Extensibility





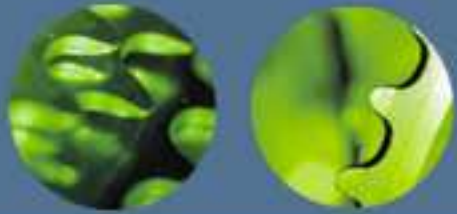
Extending Schemas

- Reusing existing schema components (usually a defined standard), while:
 - adding additional content to them, or
 - making them more restrictive
- Reasons
 - adding more specific information for your:
 - context (geopolitical, industry, etc.)
 - application (special instructions to processor)
 - creating a lightweight version of a standard



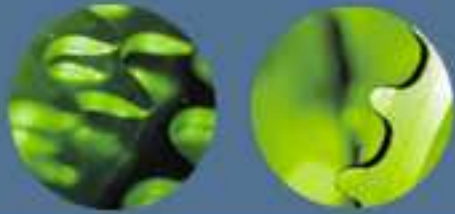
Benefits of Extending Schemas

- It is tempting to creating a completely new schema that copies the parts you want
 - easy and more flexible
- Why bother trying to extend existing components?
 - ensures compatibility with the standard you are extending
 - an obvious record of the differences
 - as the standard changes, your extensions will change along with it
 - no need to maintain definitions in two places



Methods of Extending Schemas

- Wildcards
- Type derivation
- Substitution groups
- Type redefinition
- Group redefinition

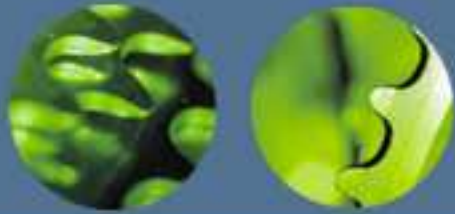


Wildcard Example: Original Type

```
<xs:complexType name="ProductType">
  <xs:sequence>
    <xs:element name="number" type="ProdNumType"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="size" type="SizeType"
      minOccurs="0"/>
    <xs:any minOccurs="0" maxOccurs="unbounded"
      namespace="##other" processContents="lax"/>
  </xs:sequence>
  <xs:anyAttribute namespace="##other"
    processContents="skip"/>
</xs:complexType>
```

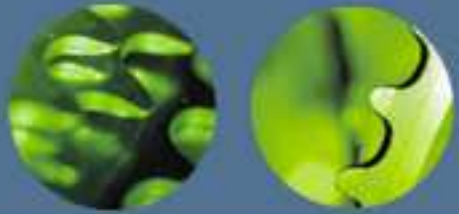
element wildcard

attribute wildcard



Wildcard Example: Extended Instance

```
<order xmlns="http://datypic.com/ord"
      xmlns:spc="http://datypic.com/spc">
  <product spc:points="100">
    <number>557</number>
    <name>Short-Sleeved Linen Blouse</name>
    <size>10</size>
    <spc:giftWrap>ADULT BDAY</spc:giftWrap>
  </product>
</order>
```

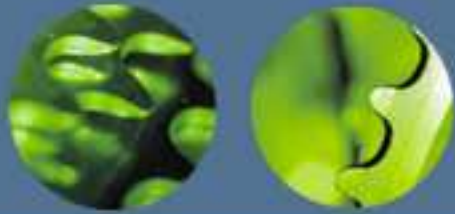


Controlling Wildcard Validation

- `processContents` attribute indicates how strictly to validate
 - `skip`: do not validate
 - `lax`: validate if you can find a declaration
 - `strict`: validate

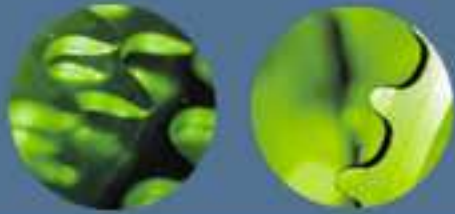
use global declarations to validate

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns="http://datypic.com/spc"
            targetNamespace="http://datypic.com/spc">
  <xs:element name="giftWrap" type="GiftWrapType"/>
  <xs:attribute name="points" type="xs:integer"/>...
```



Controlling Wildcard Namespaces

- **namespace** attribute specifies what namespace(s) the elements may belong to
 - **##all** (the default)
 - any namespace, or no namespace
 - **##other**
 - any namespace other than the target namespace
 - whitespace-separated list of namespaces that may include:
 - **##local** for no namespace
 - **##targetNamespace** for target namespace
 - specific namespaces



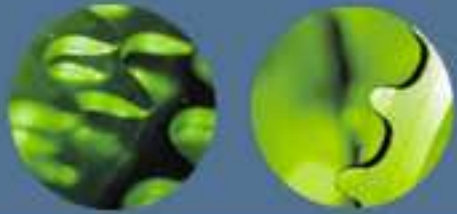
Non-Determinism and Wildcards

- Wildcards can create non-deterministic situations

```
<xs:complexType name="ProductType">
  <xs:sequence>
    <xs:element name="number" />
    <xs:element name="size" minOccurs="0" />
    <xs:any />
  </xs:sequence>
</xs:complexType>
```

If the processor encounters `size`, does it match the `size` declaration or the wildcard??

- Can be remedied by:
 - not putting wildcards after optional or repeating elements
 - using `namespace="##other"`

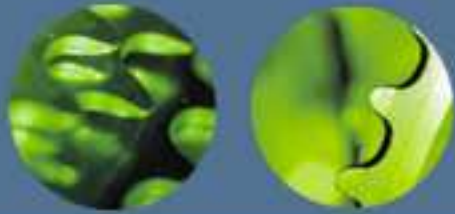


Using an Extension "Bucket"

- Wildcards can be used along with an **extensions** element

```
<product>  
  <number>557</number>  
  <size>10</size>  
  <extensions>  
    <spc:giftWrap>ADULT BDAY</spc:giftWrap>  
  </extensions>  
</product>
```

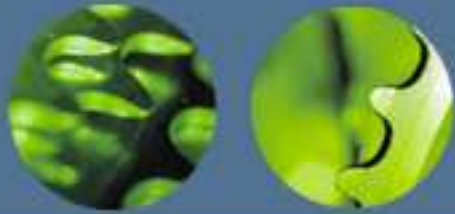
- Advantages
 - eliminates non-determinism problem
 - makes it obvious what is an extension



Using an Extension "Bucket": Schema

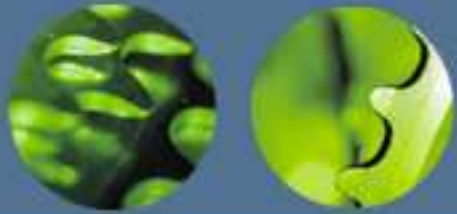
```
<xs:complexType name="ProductType">
  <xs:sequence>
    <xs:element name="number" />
    <xs:element name="size" minOccurs="0" />
    <xs:element name="extensions" type="ExtType"
      minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="ExtType">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded"
      namespace="##any" processContents="lax" />
  </xs:sequence>
</xs:complexType>
```



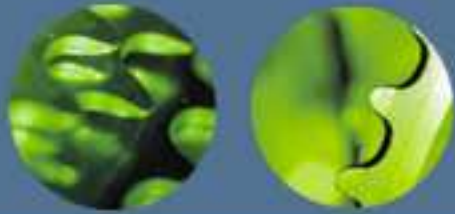
Wildcards as Extension Mechanism

- Advantages
 - straightforward
 - location of extensions controlled by original schema author
 - allows extended instances to be valid according to original schema
 - if `processContents` is not strict, or replacement elements are declared in original schema
- Disadvantages
 - not much control over extensions
- Only possible if:
 - wildcards are used in the original schema



Type Derivation

- Can apply to either extension or restriction
 - extension to add new elements or attributes
 - restriction to allow a subset of the original standard
- Two methods:
 - define only derived types, use original elements with `xsi:type` attribute
 - define derived types as well as declare new elements



Type Derivation

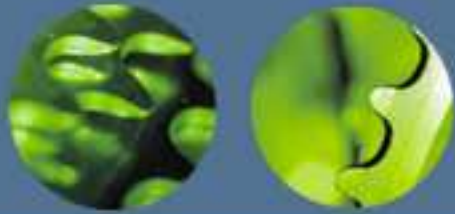
Example: Original Type

must be named

no block or final
attributes to
prohibit derivation

```
<xs:complexType name="ProductType">
  <xs:sequence>
    <xs:element name="number" type="ProdNumType"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="size" type="SizeType"
      minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

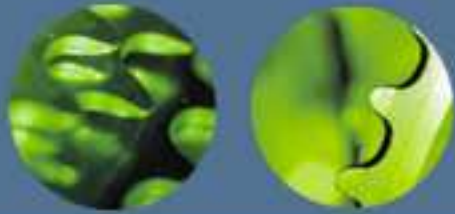
should use a
sequence group
for extensibility



Type Derivation

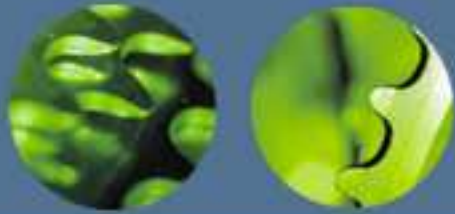
Example: Extended Type

```
<xs:complexType name="ExtendedProductType">
  <xs:complexContent>
    <xs:extension base="ProductType">
      <xs:sequence>
        <xs:element ref="spc:giftWrap" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute ref="spc:points"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```



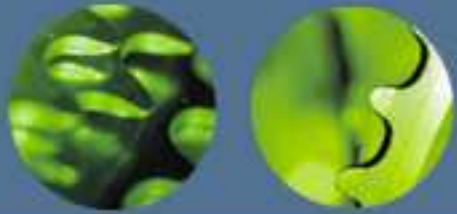
Type Derivation Example: Extended Instance

```
<order xmlns="http://datypic.com/ord"
  xmlns:spc="http://datypic.com/spc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <product spc:points="100"
    xsi:type="ExtendedProductType">
    <number>557</number>
    <name>Short-Sleeved Linen Blouse</name>
    <size>10</size>
    <spc:giftWrap>ADULT BDAY</spc:giftWrap>
  </product>
</order>
```



Type Derivation as Extension Mechanism

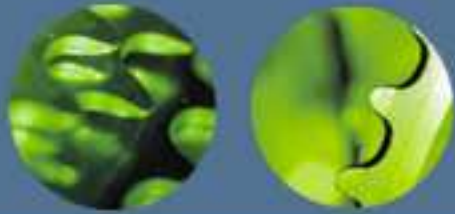
- Advantages
 - extended types have a relationship with the original types
 - provides type hierarchy information to application
- Disadvantages
 - complex rules govern what is a valid extension/restriction
 - requires use of `xsi:type` or declaration of new elements
- Only possible if:
 - types are named
 - types use **sequence** groups, not **choice** or **all**
 - if restricting, either:
 - the same target namespace is used, or
 - unqualified local elements are used



Limitations of Extending choice Groups

```
<xs:complexType name="CharacteristicType">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="size"/>
    <xs:element name="color"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="ExtCharacteristicType">
  <xs:complexContent>
    <xs:extension base="CharacteristicType">
      <xs:choice maxOccurs="unbounded">
        <xs:element name="weight"/>
        <xs:element name="height"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```



Effective Content Model of Previous Example

```
<xs:complexType name="ShirtType">
  <xs:sequence>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="size"/>
      <xs:element name="color"/>
    </xs:choice>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="weight"/>
      <xs:element name="height"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```

weight and height must appear
after size and color

```
<shirt>
  <size>12</size>
  <color>Red</color>
  <weight>12</weight>
  <height>Shirt</height>
</shirt>
```



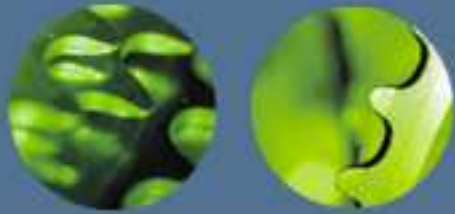
Substitution Groups

```
<xs:complexType name="ShirtType">  
  <xs:choice maxOccurs="unbounded">  
    <xs:element ref="size"/>  
    <xs:element ref="color"/>  
    <xs:element ref="ext"/>  
  </xs:choice>  
</xs:complexType>  
<xs:element name="ext" abstract="true"/>
```

```
<xs:element name="weight" substitutionGroup="ext" />  
<xs:element name="height" substitutionGroup="ext" />
```

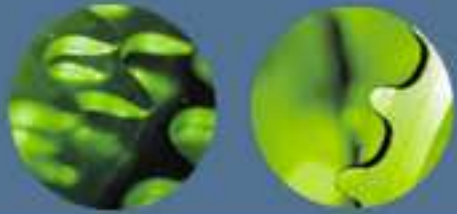
```
<shirt>  
  <height>Shirt</height>  
  <size>12</size>  
  <weight>12</weight>  
</shirt>
```

children can appear in
any order



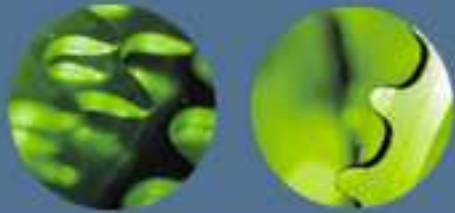
Substitution Groups as Extension Mechanism

- Advantages
 - the only way to extend choice groups without imposing an order on them
 - more controlled than wildcards
 - crosses namespaces easily
- Disadvantages
 - applications need to be able to handle element names they don't expect
- Only possible if:
 - elements are globally declared
 - new elements' types are derived from originals



Type Redefinition

- Including another schema document while changing the definition of some types
- Unlike derivation, does not require the use of `xsi:type` in instances
- New component must be an extension or restriction of the original component
- Can only be done within the same namespace
 - but extended elements/attributes are not required to be in that namespace



Type Redefinition Example: Original Type

- Same as type derivation example

must be named

```
<xs:complexType name="ProductType">
  <xs:sequence>
    <xs:element name="number" type="ProdNumType"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="size" type="SizeType"
      minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

should use a
sequence group
for extensibility



Type Redefinition

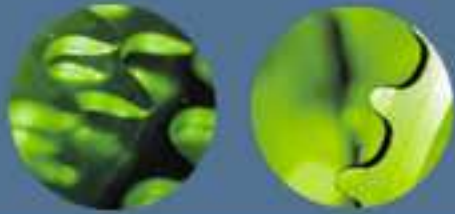
Example: Extended Type

```
<xs:schema xmlns:xs= ...
  <xs:import namespace="http://datypic.com/spc"/>
  <xs:redefine schemaLocation="original.xsd">
    <xs:complexType name="ProductType"> ← same name
      <xs:complexContent>
        <xs:extension base="ProductType"> ← for base and
          <xs:sequence>
            <xs:element ref="spc:giftWrap" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute ref="spc:points"/>
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:redefine>
  </xs:schema>
```



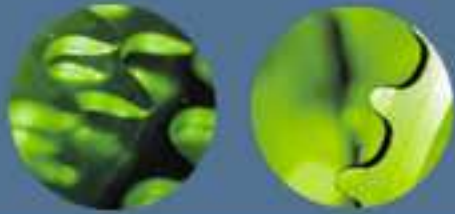
Type Redef. Example: Extended Instance

```
<order xmlns="http://datypic.com/ord"
        xmlns:spc="http://datypic.com/spc">
  <product spc:points="100">
    <number>557</number>
    <name>Short-Sleeved Linen Blouse</name>
    <size>10</size>
    <spc:giftWrap>ADULT BDAY</spc:giftWrap>
  </product>
</order>
```



Group Redefinition

- Named model groups and attribute groups can also be redefined
- New definition must be a subset or superset of original definition



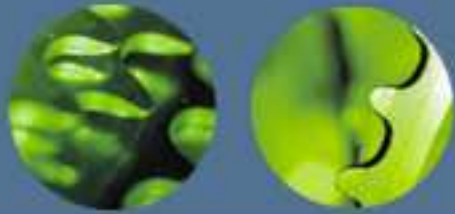
Group Redefinition

Example: Original Type

```
<xs:complexType name="ProductType">
  <xs:group ref="ProductPropertyGroup" />
  <xs:attributeGroup ref="ExtensionGroup" />
</xs:complexType>

<xs:group name="ProductPropertyGroup">
  <xs:sequence>
    <xs:element name="number" type="ProdNumType" />
    <xs:element name="name" type="xs:string" />
    <xs:element name="size" type="SizeType"
      minOccurs="0" />
  </xs:sequence>
</xs:group>

<xs:attributeGroup name="ExtensionGroup" />
```



Group Redefinition

Example: Extended Type

```
<xs:schema ...
  <xs:import namespace="http://datypic.com/spc"/>
  <xs:redefine schemaLocation="original.xsd">
    <xs:group name="ProductPropertyGroup">
      <xs:sequence>
        <xs:element ref="spc:giftWrap"/>
        <xs:group ref="ProductPropertyGroup"/>
      </xs:sequence>
    </xs:group>
    <xs:attributeGroup name="ExtensionGroup">
      <xs:attributeGroup ref="ExtensionGroup"/>
      <xs:attribute ref="spc:points"/>
    </xs:attributeGroup>
  </xs:redefine>
</xs:schema>
```

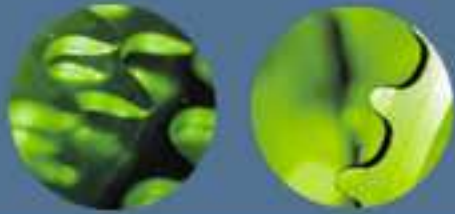
new groups
reference the
original group
with the same
name



Group Redef. Example: Extended Instance

```
<order xmlns="http://datypic.com/ord"
      xmlns:spc="http://datypic.com/spc">
  <product spc:points="100">
    <spc:giftWrap>ADULT BDAY</spc:giftWrap>
    <number>557</number>
    <name>Short-Sleeved Linen Blouse</name>
    <size>10</size>
  </product>
</order>
```

extended
giftWrap
element can
appear first



Redefinition as Extension Mechanism

- Advantages
 - derive new types without requiring use of `xsi:type`
- Disadvantages
 - not very cleanly specified
 - so not consistently implemented in processors
 - risk of rendering types in the redefined schema document invalid
 - types derived from the ones that you are redefining
- Only possible if:
 - types/groups are named
 - you are in the same target namespace



Redefinition Using Adapter Schema

```
<xs:schema targetNamespace="http://datypic.com" ...  
  <xs:import namespace="http://notmine.com"  
    schemaLocation="notmine2.xsd">
```

created by
extender

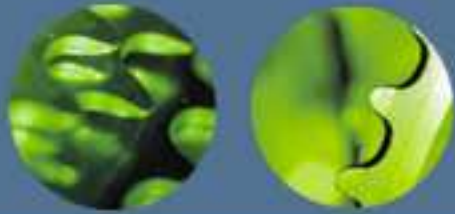
imports

```
<xs:schema targetNamespace="http://notmine.com" ...  
  <xs:redefine schemaLocation="notmine1.xsd">  
    <xs:complexType name="ProductType">  
      <xs:complexContent>  
        <xs:extension base="ProductType">  
          ...
```

redefines

```
<xs:schema targetNamespace="http://notmine.com"  
  <xs:complexType name="ProductType">  
    ...
```

original
schema



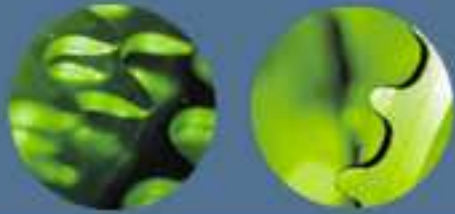
Methods of Extending Schemas Compared

	Wild cards	Type deriv	Subst group	Type redef	Group redef
Extended instance valid against original definition?	yes	no	no	no	no
Requires use of <code>xsi:type</code> in instance?	no	yes	no	no	no
Can define extended types in a different target namespace?	yes	yes	yes	no	no
Must element extensions appear at the end?	no	yes	no	yes	no



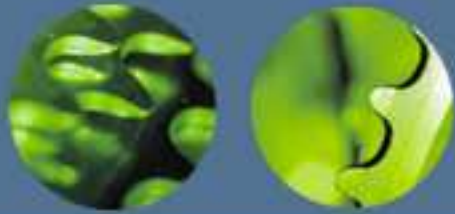
Versioning





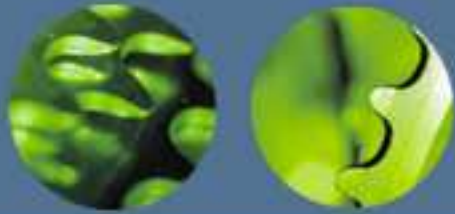
Versioning

- The only constant is change
- New versions usually involve defining entirely new components
 - rather than literally extending, restricting or redefining an existing schema
- Compatibility (especially backward compatibility) important



Schema Compatibility

- Backward compatibility
 - 2.0 instances conform to 3.0 schema
 - something to aim for in minor versions
- Forward compatibility
 - 3.0 instances conform to 2.0 schema
 - less easy unless 2.0 schemas are very flexible



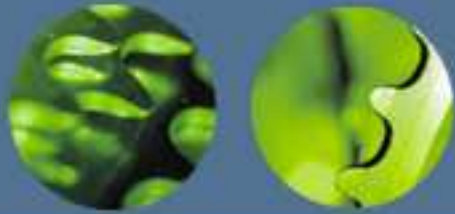
Backward Compatible Changes to Schemas

- Adding optional elements/attributes
- Making required elements/attributes optional
- Making occurrence constraints less restrictive
 - e.g. allow many where only one was allowed
- Turning element declarations into choice groups
 - where color was allowed, allow color or size or weight
- Making simple types less restrictive
 - making ranges or patterns less restrictive
 - adding enumerated values



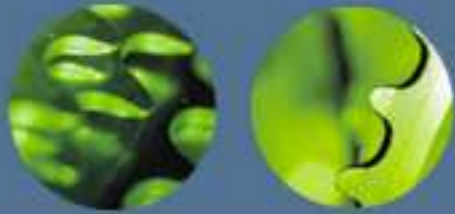
Backward *Incompatible* Changes to Schemas

- Adding required elements/attributes
- Removing elements/attributes
- Making occurrence constraints more restrictive
 - making an optional element required
- Changing the order
- Adding structural elements



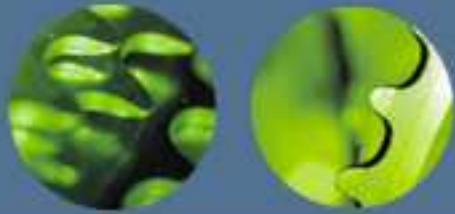
Application Compatibility

- Separate from schema compatibility
- Applications able to process version 2.0 instances should ideally be able to process version 3.0 instances
 - and vice versa
 - e.g. Microsoft Word model
 - Word 6.0 can read 5.0 documents perfectly
 - Word 5.0 can read 6.0 docs but may lose some information



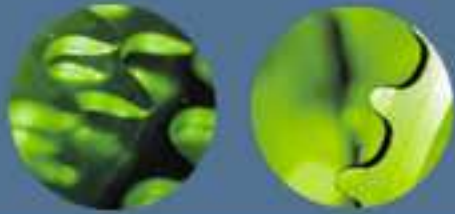
Granularity of Versions

- Can version components at different levels of granularity:
 - individual types or declarations
 - schema documents
 - entire schema groups
- The more granular, the more difficult to manage
- Recommendation: use the schema document level
 - changes to individual components usually necessitate a new version of the schema document anyway



Indicating Version Number

- Four methods (at least)
 - in the schema document
 - in the schema location
 - in the instance document
 - in the namespace name



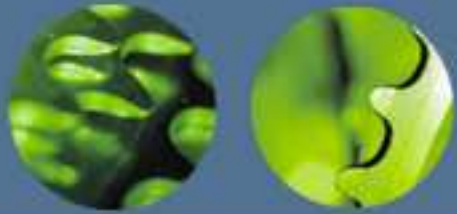
Indicating Version in the Schema Document

- `version` attribute of `xs:schema`
 - for documentation only; parsers/validators not required to process in any way

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:dtyc="http://datypic.com"
           version="2.0.1">
  <xs:complexType name="ProductType" dtyc:version="2.0">
    <!-- ... -->
```

standard version
attribute at schema level

custom version
attribute at type level

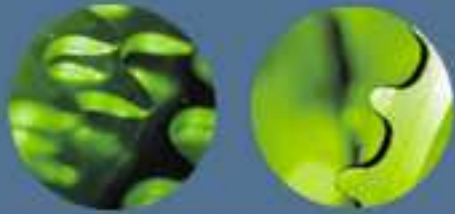


Indicating Version in the Schema Location

- The file name or URL contains version information
- Warning: schemaLocation is just a hint; processors may ignore
- Ties including schemas into specific versions

```
<order xmlns="http://datypic.com/ord"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://datypic.com/ord  
                      http://datypic.com/ord2_0_1.xsd">
```

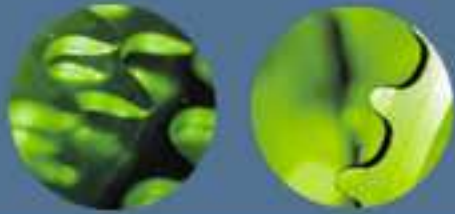
...



Indicating Version in the Instance

- Use a `version` attribute on the root or other element
- No special semantics
 - processing application will have to read and handle accordingly
- Approach used for XSLT

```
<order version="2.0.1" xmlns="http://datypic.com/ord"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://datypic.com/ord ord.xsd">  
  ...
```

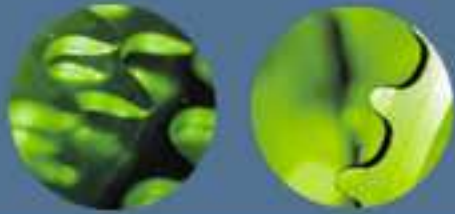


Indicating Version in the Namespace Name

- Make a version number part of the namespace name
- Warning: breaks backward compatibility
 - for both schema validation and application processing

```
<order xmlns="http://datypic.com/ord/2.0.1"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://datypic.com/ord/2.0.1  
    http://datypic.com/2.0.1/ord.xsd">
```

...



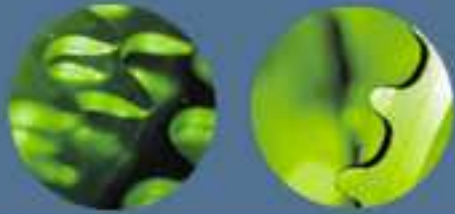
Complete Versioning Strategies

- Usually involve more than one of the four methods mentioned
- Often involve separate version-handling tools
 - schema management software products
 - source code management software
 - custom-built utilities



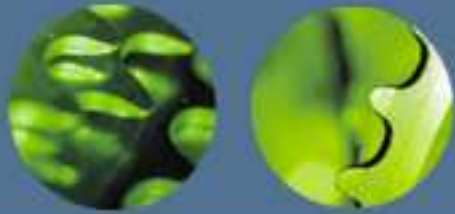
Schema Documentation





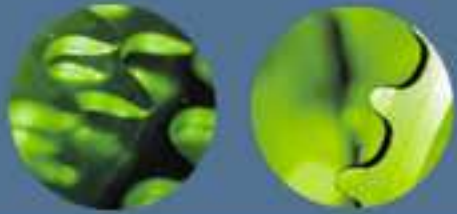
Annotations

- Structured XML documentation about schema components
- All schema components can be annotated
 - anywhere as a child of `schema`
 - as the first child of any schema component



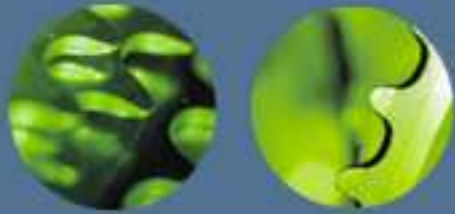
Document Components Using Annotations

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation>This schema is for...
    </xs:documentation>
  </xs:annotation>
  <xs:element name="product" type="ProductType">
    <xs:annotation>
      <xs:appinfo source="http://datypic.com/prodmast">
        <app:dbmapping>PRODUCT_MASTER</app:dbmapping>
      </xs:appinfo>
    </xs:annotation>
  </xs:element>
</xs:schema>
```



Uses for Application Information

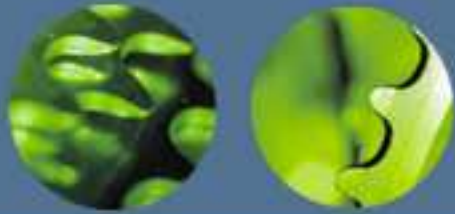
- Extra validation rules
 - e.g. Schematron constraints
- Mappings to other technologies
 - e.g. relational databases, EDI documents
- Mappings to forms or other user input mechanisms
 - e.g. XHTML form control to use to display this type



Non-Native Attributes

- Any schema element can have attributes in any other namespace
- These non-native attributes are not validated; they do not need to be declared
- Less verbose than annotations

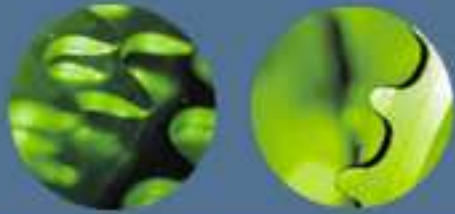
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:doc="http://datypic.com/doc">
  <xs:element name="product" type="ProductType"
             doc:description="This element represents a product."/>
</xs:schema>
```



Comments

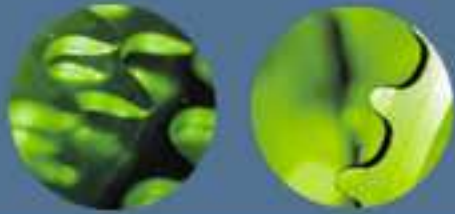
- As XML documents, schemas can also contain regular XML comments
- Not recommended because they are
 - less structured
 - not necessarily passed to the application

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">  
  <!-- This element represent a product -->  
  <xs:element name="product" type="ProductType"/>  
</xs:schema>
```



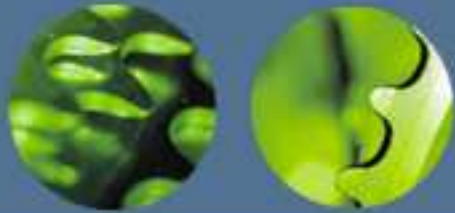
Documenting Namespaces

- Can put the schema at the namespace URL, but not recommended because:
 - many schemas may describe that namespace
 - a variety of documents in other formats may also describe that namespace
 - DTDs, human-readable documentation, schemas written in other schema languages, and stylesheets.
 - schema documents are not particularly human-readable, even by humans who write them!
- Instead, use RDDL
 - extension of HTML with machine readable attributes



RDDL Example

```
<div class="resource" id="xmlschema">
  <rddl:resource xlink:title="Products schema"
    xlink:role="http://www.w3.org/2001/XMLSchema"
    xlink:arcrole=
      "http://www.rddl.org/purposes#schema-validation"
    xlink:href="prod.xsd">
    <h3>XML Schema</h3>
    <p>An <a href="prod.xsd">XML Schema</a> for the
      Product Catalog.</p>
  </rddl:resource>
</div>
```



RDDL Example

Product Catalog - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address <http://www.pivotal.com/proc/> Go

Related Resources for the Product Catalog Namespace

[DTD](#)
A [DTD](#) for the Product Catalog.

[XML Schema](#)
An [XML Schema](#) for the Product Catalog.

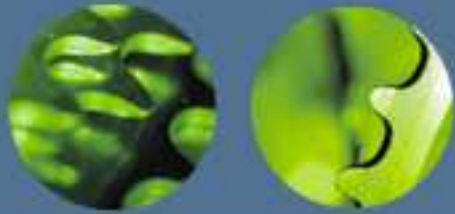
[Application Documentation](#)
[Application documentation](#) for the Product Catalog application.

Done My Computer



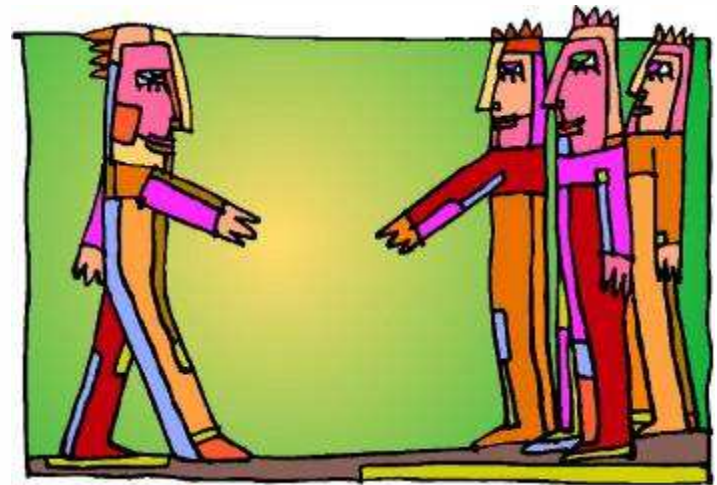
XML Design in Multi-User Environments





Encouraging Good Schema Design

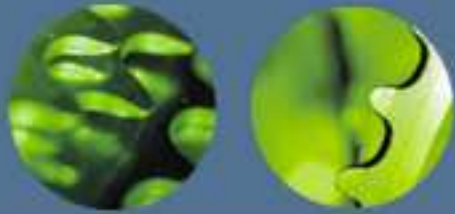
- Clearly documenting your organization's schema best practices
- Providing help and coaching to schema authors
- Conducting schema design reviews
- Providing examples of good schemas





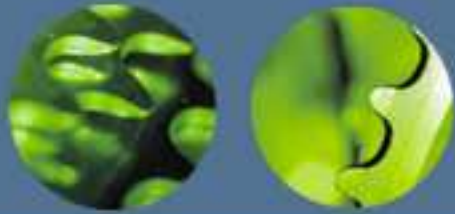
NDR (Naming and Design Rules) Documents

- A document that lays out a consistent approach to design for a group of related schemas
- Defines rules for:
 - naming standards
 - what features of XML Schema may be used
 - approach to versioning
 - approach to extensions
 - etc.



Schema Management

- In complex environments, use software tools to keep track of:
 - versions of schema documents and components
 - relationships between schema documents
 - subsets of the schema documents relevant to particular sets of users
 - who is updating what
- Different approaches
 - source code management tools
 - custom code to assemble/track schemas
 - collaborative design tools like CortexML



Resources

- **See**
 - <http://www.datypic.com/services/xmldesign>
- **For:**
 - these slides in PDF format
 - links to other resources relating to XML Design



Questions?

