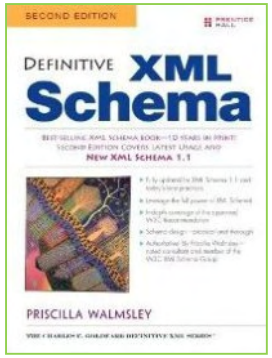
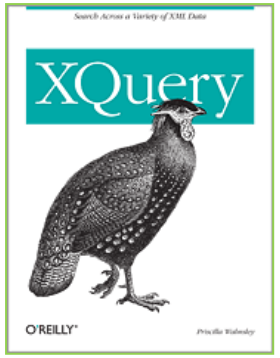
Structuring a Canonical Model

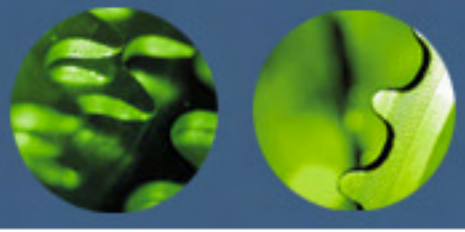
Priscilla Walmsley
Managing Director, Datypic
pwalmsley@datypic.com
http://www.datypic.com

About Datypic

- XML- and SOA- related development, consulting, and training
- Projects large and small, including:
  - Schema development and design reviews
  - SOA design and implementation
  - Training in XML technologies
- Background
  - Government information sharing (NIEM)
  - All things XML (data and documents)
  - Metadata repositories

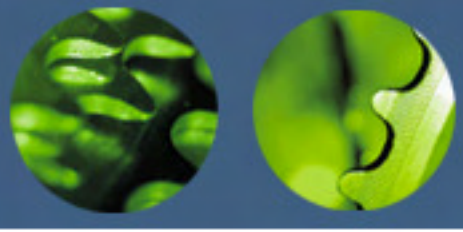






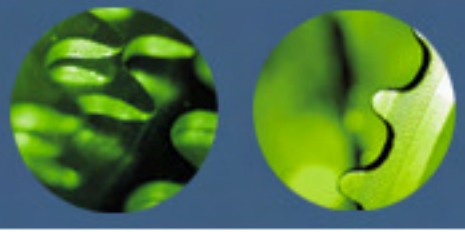
Structuring a Canonical Model

- Components in a "usable" canonical model are:
  - Reusable
  - Optimizable
  - Flexible
  - Extensible
  - Understandable
  - Implementable



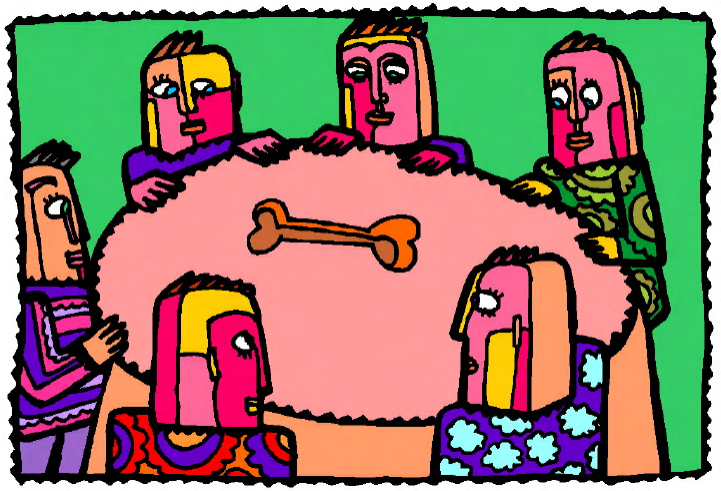
Why is So Little Attention Paid To XML Modeling?

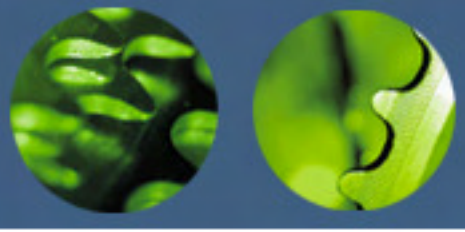
- Wide variety of use cases
  - The answer to most design questions starts with "It depends..."
- Some XML data is temporary
  - so considered not as crucial as permanent data "assets"
- Anyone can create XML
  - not just database administrators or software developers



Reusable

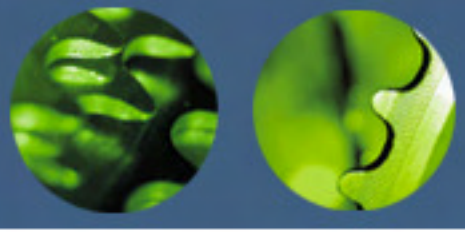
- Reuse of schema components is the whole point, right?
- But only certain schema components are reusable!





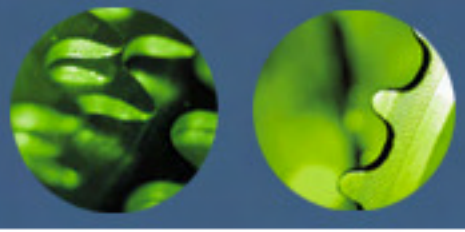
Global vs. Local Definitions

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



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`**

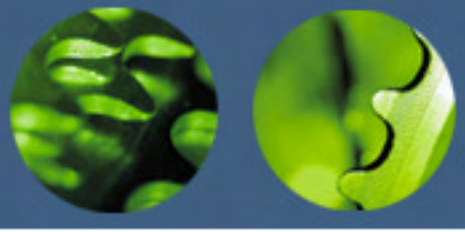


Structural Elements

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

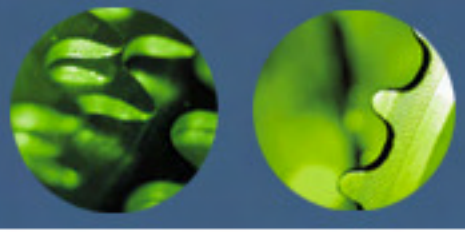
```
<customer>
  <name>PW</name>
  <addr1>1 Main</addr1>
  <city>TC</city>
  <state>MI</state>
</customer>
```

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



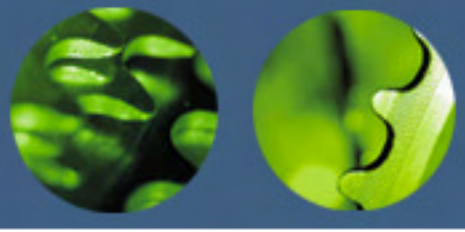
Using Structural Elements

- Advantages
  - Contents are easier to reuse
  - Can provide a stronger content model in the exchange
    - e.g. make the whole address optional or required, and give components individual cardinalities
  - Usually more intuitive to the integration developer
- Disadvantage
  - More verbose



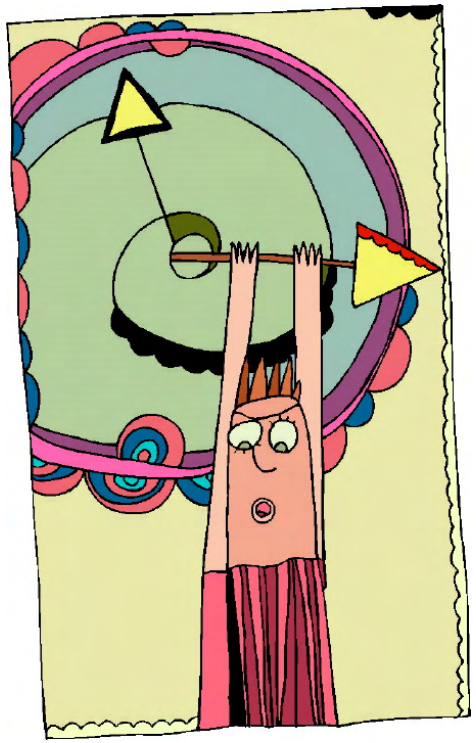
Other Considerations for Reusable Components

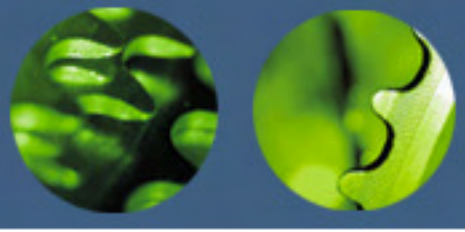
- Think about a broader applicability of your type
  - Other contexts such as geopolitical, industry, etc.
- Use general names for types (if appropriate)
  - AddressType rather than CustomerAddressType
- Consider named model groups and attribute groups for definitions that are commonly used together



Optimizable

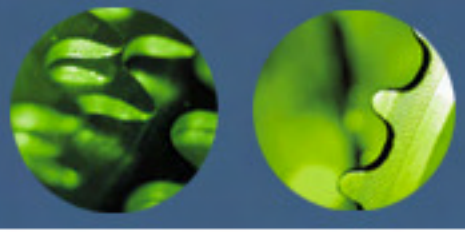
- Messages and their schemas should not contain:
  - Lots of unused elements
    - empty, nilled, absent but in the schema
  - Lots of unnecessary levels of hierarchy
- This makes them:
  - Slow
  - Cumbersome for developers
  - Cumbersome to document





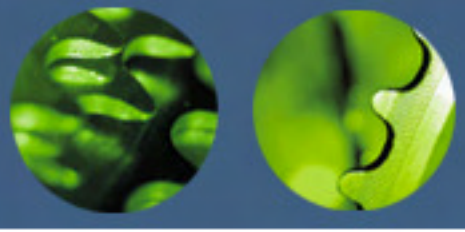
Cardinalities

- Keep them loose!
  - Make everything optional
  - Make most elements repeating
- Even though a Policy must have an associated Expiration Date in the database, it might not be relevant to a particular context
  - Let the integration developer decide what's required in that particular message



Handling Relationships

- A significant difference between "data at rest" and "data in motion"
- One-to-one or one-to-many relationships can be handled by containment
- Many-to-many relationships are more challenging
  - but less common for service models

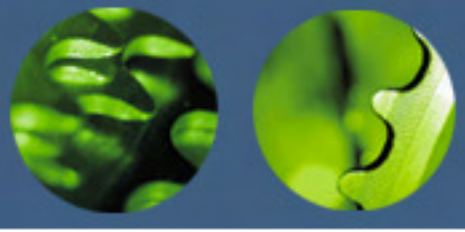


1. Relationships through Containment

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

```
<department>
  <name>Men's</name>
  <products>
     <product>
       <name>Shirt</name>
     </product>
     <product>
       <name>Hat</name>
     </product>
  </products>
</department>
```

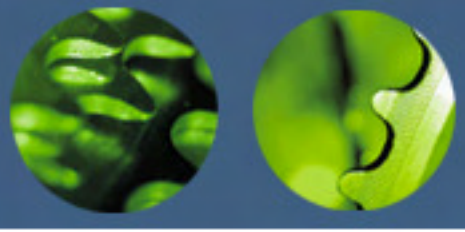
"a department can be associated with one or more products"



2. Relationships through Separate 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

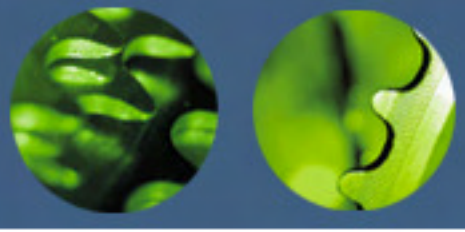
```
<department id="556">
 <name>Men's</name>
</department>
<product id="P400">
 <name>Shirt</name>
</product>
<dept-product>
  <deptref ref="556">
  <productref ref="400">
  <mainDept>true</mainDept>
</dept-product>
```



3. Relationships through Repetition

```
<department>
   <name>Men's</name>
   <products>
      <product>
        <name>Shirt</name>
      </product>
   </products>
</department>
<department>
   <name>Women's</name>
   <products>
      <product>
        <name>Shirt</name>
      </product>
   </products>
</department>
```

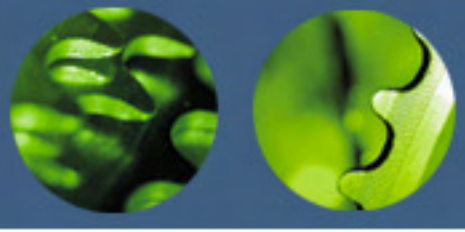
- Not great for optimization
- Acceptable for smaller objects



4. Relationships through References

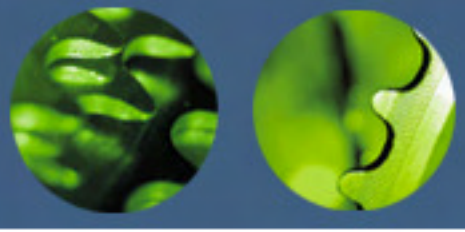
```
<department>
   <name>Men's</name>
   <products>
      <productref ref="P400"/>
   </products>
</department>
<department>
   <name>Women's</name>
   <products>
      <productref ref="P400"/>
   </products>
</department>
<product id="P400">
 <name>Shirt</name>
</product>
```

- Some models allow a choice of reference or containment
- An abstract element can be replaced by either **`product`** or **`productref`**.



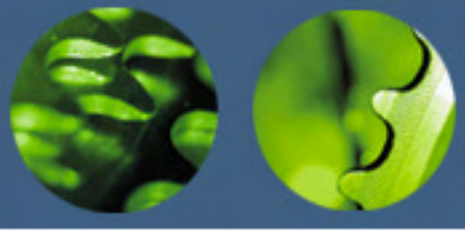
Flexibility

- Models are rarely one-size-fits-all
- Canonical models should be flexible to:
  - meet a variety of implementers' needs
  - adapt to changes over time more easily



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



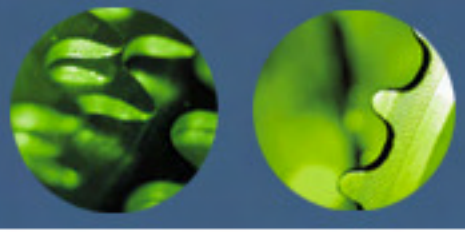
Generic vs. Specific Elements: A Tradeoff

- More generic properties improve flexibility

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

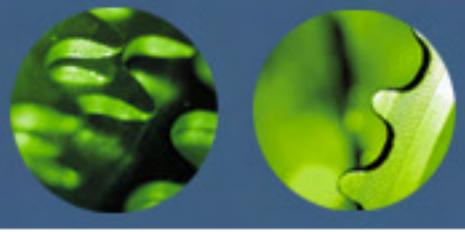
- More specific properties are better defined
  - Data types
  - Cardinalities

```
<length>3</length>
<width>5</width>
<type>ABC</type>
```



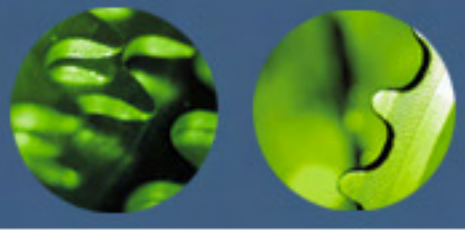
Flexibility of Representation

- A model sometimes requires multiple representations of the same concept for different implementations.
- For example, Person Name could be:
  - Full Name
  - First Name, Middle Name, Last Name
  - More complex name representations
- Choice groups or substitution groups can be used to allow flexibility



Choice Groups vs. Substitution Groups

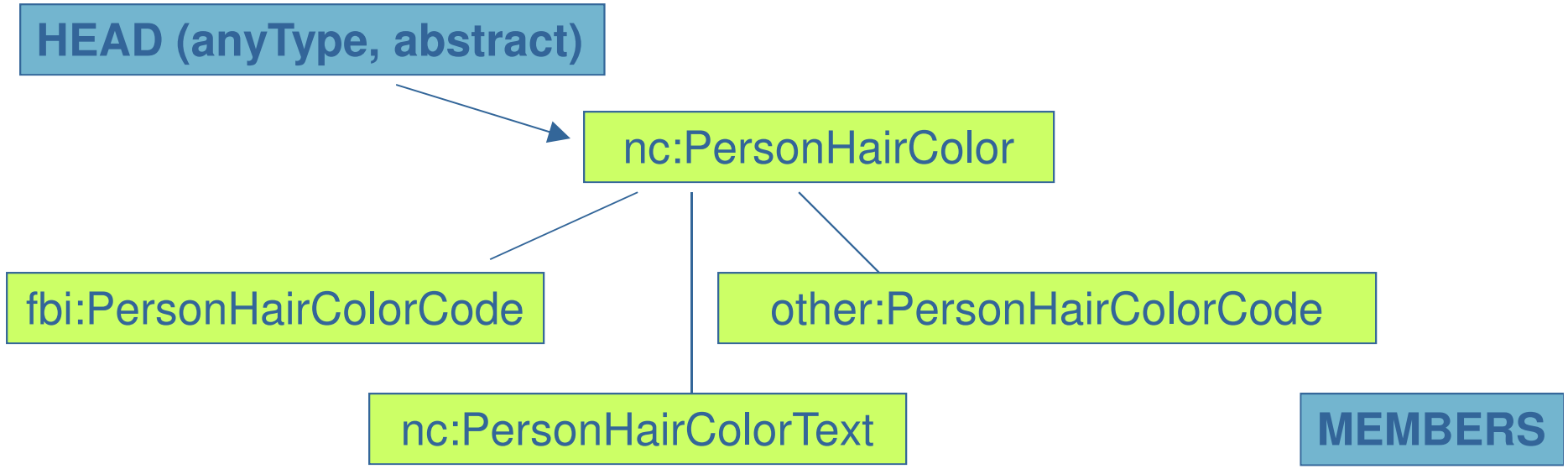
- Use choice groups when:
  - there is a rigid, fixed set of choices
  - the element choices don't have similar types
  - it is desirable to easily see the group in one place
- Use substitution groups when:
  - the set of choices is growing or flexible
  - the element choices can have the same or a derived type
  - the same set of choices is used over and over again, wherever the head element appears

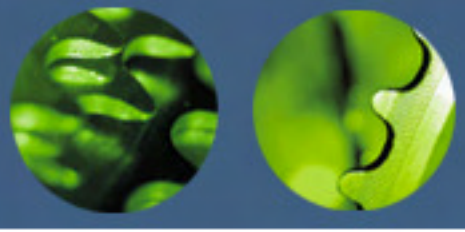


Code Lists

- Code lists also often require flexibility in physical representation
- Canonical models should be flexible enough to allow multiple code lists for a particular property, or an unconstrained text element
- Code lists may or may not be expressed in XSD, depending on:
  - volatility
  - number of valid values

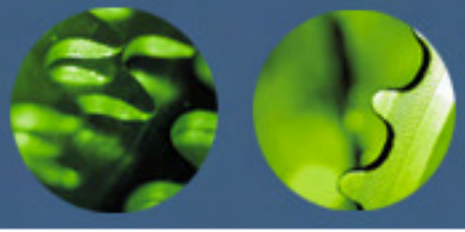
NIEM Code List Substitution Example





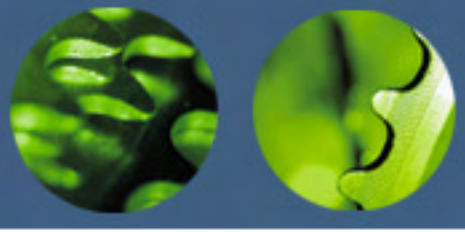
Extensible

- The canonical model will not contain everything required in exchanges
- It should allow for extensions:
  - at the object level
  - at the message level



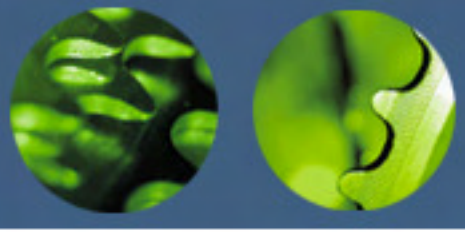
Extensions

- Extension methods
  - Wildcards in original components
  - Complex type extensions
  - Substitution groups
  - Separate extension areas



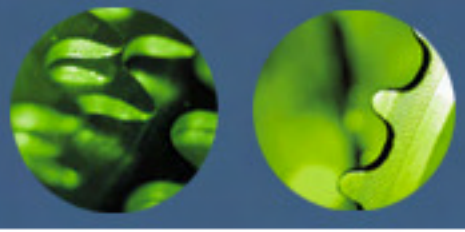
Wildcards as Extension Mechanism

- 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 appear



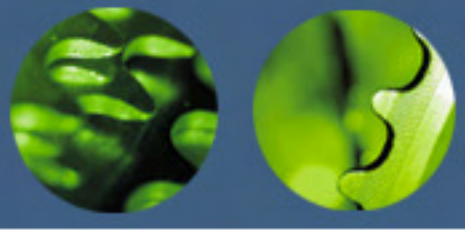
Type Derivation as Extension Mechanism

- Advantages
  - extended types have a relationship with the original types
    - provides type hierarchy information to application
- Disadvantages
  - requires use of **`xsi:type`** or declaration of new elements
- Only possible if:
  - types are named
  - types use **`sequence`** groups, not **`choice`** or **`all`**



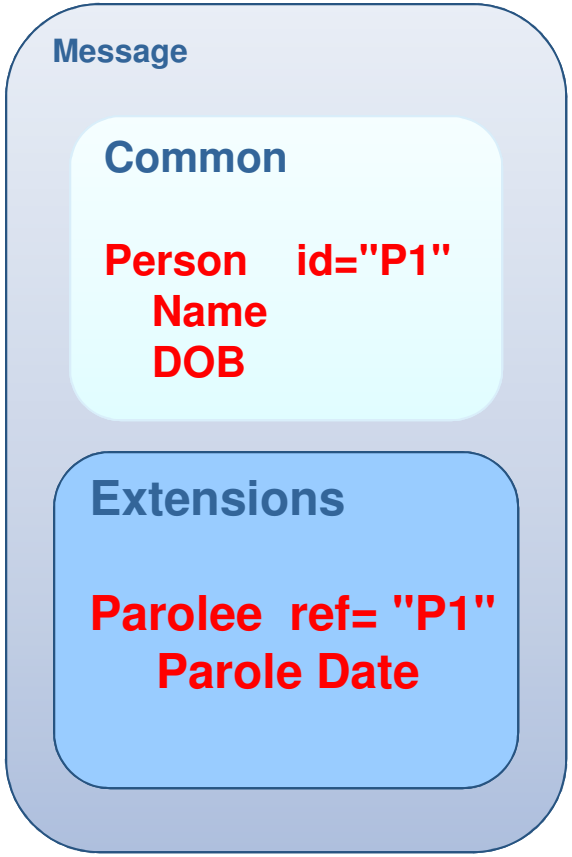
Substitution Groups as Extension Mechanism

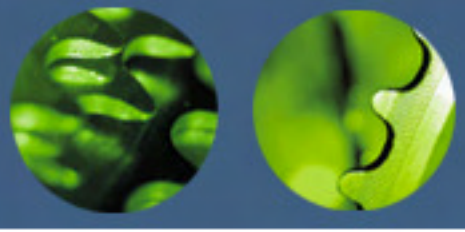
- Advantages
  - the only way to extend choice groups without imposing an order on them
  - more controlled than wildcards
- Disadvantages
  - applications need to be able to handle element names they don't expect based on the original schema
- Only possible if:
  - elements are globally declared
  -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



Extension Area

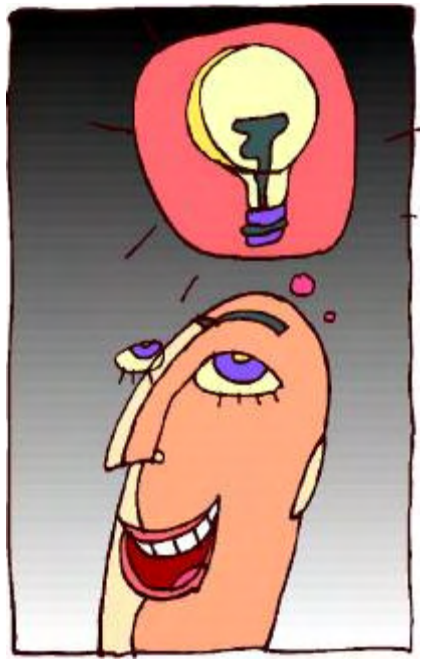
- Example
  - Common area contains completely interoperable objects
  - Extensions are separated with references back to common objects
  - Useful if core interoperability is important

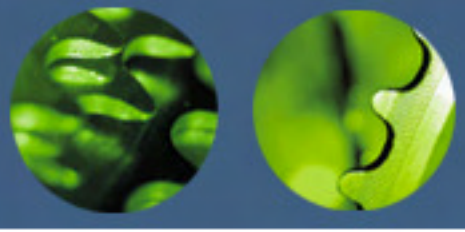




Understandable

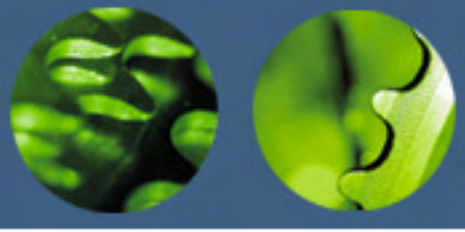
- Models that everyone can understand are:
  - Easier to learn
  - Easier to debug and change
  - More likely to be reused
- Aspects of understandability:
  - Searchability
  - Consistency
  - Simplicity





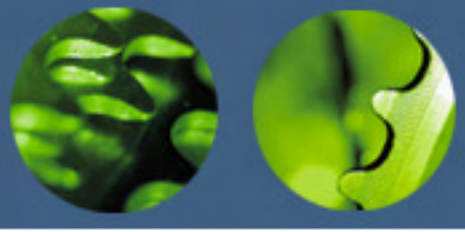
Searchability

- Ability to use tools as a finding aid
- Creation of synonyms
- Annotation of components with keywords
- Complete documentation
- Organization of components for browsing



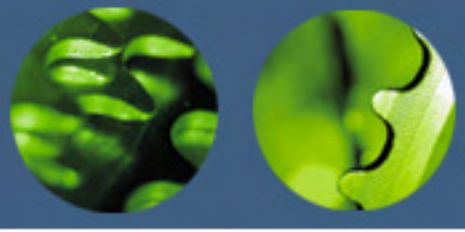
Consistency

- Achieve consistency wherever possible in:
  - Structure
    - Use of structural elements, attributes vs. elements, etc.
  - Naming
    - Separators/capitalization
    - Glossary of standard terms
    - Using name parts (e.g. Object Term, Property Term, Representation Term)
    - Identification of components by type
  - Order of properties



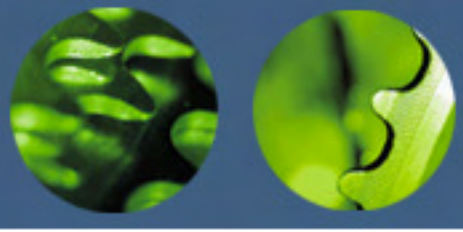
Consistency through Reuse

- Consistency is also achieved by practicing reuse within your canonical model
  - Reusing types
  - Reusing content model fragments through:
    - complex type extension
    - named model groups
    - common structural elements



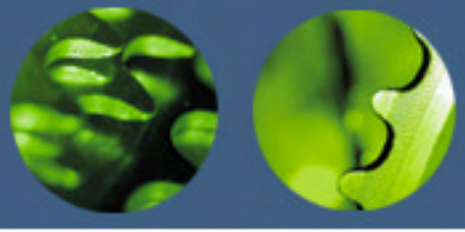
Consistency through Ordering

- Use **`sequence`** groups for properties
- Save **`choice`** groups for true choices
  - Repeating choice groups can't enforce cardinalities
  - Extending choice groups is complicated
- Do not use **`all`** groups due to their limitations
  - Elements can't repeat
  - Types cannot be extended



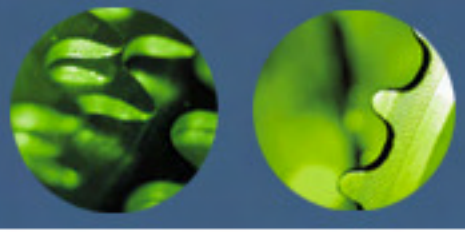
Simplicity

- Limit the number of different XSD features you are using
- Have a simple namespace strategy
  - keep down the number of namespaces
  - always use qualified local elements
  - don't use chameleon components



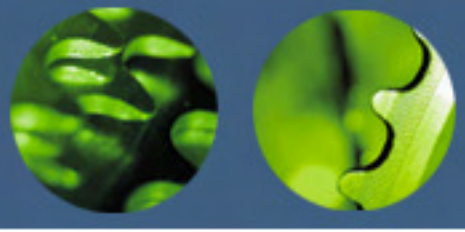
Implementable

- Using the canonical model cannot be seen as interfering with implementation in applications
- Make it easy through:
  - useful tools that take the manual tedium out
  - good examples
  - well-defined (but simple) process



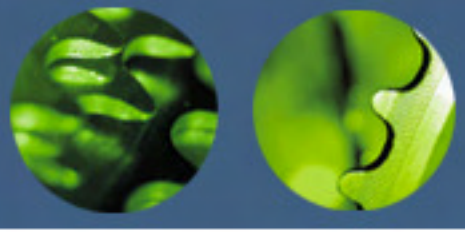
Avoid Non-Mainstream XSD Features

- Some XSD features are not well supported by some toolsets:
  - Mixed content
  - Complicated content models with nested model groups
  - Dynamic type substitution using xsi:type
  - Default and fixed values
  - Redefinition using xsd:redefine



Versioning

- Keep your versioning strategy as simple as possible
  - Version at a coarse level
  - Have a well-defined strategy that is predictable
  - Use namespaces to isolate major versions
- Minimize versioning impact on existing applications
  - e.g., do not require upgrades to latest version if not required



Questions?

- More on XML design in general
  - http://www.datypic.com/services/xmldesign/