Schema Rules for UBL... and Maybe for You

Eve Maler

XML 2002 Conference

12 December 2002

OASIS

Sun microsystems

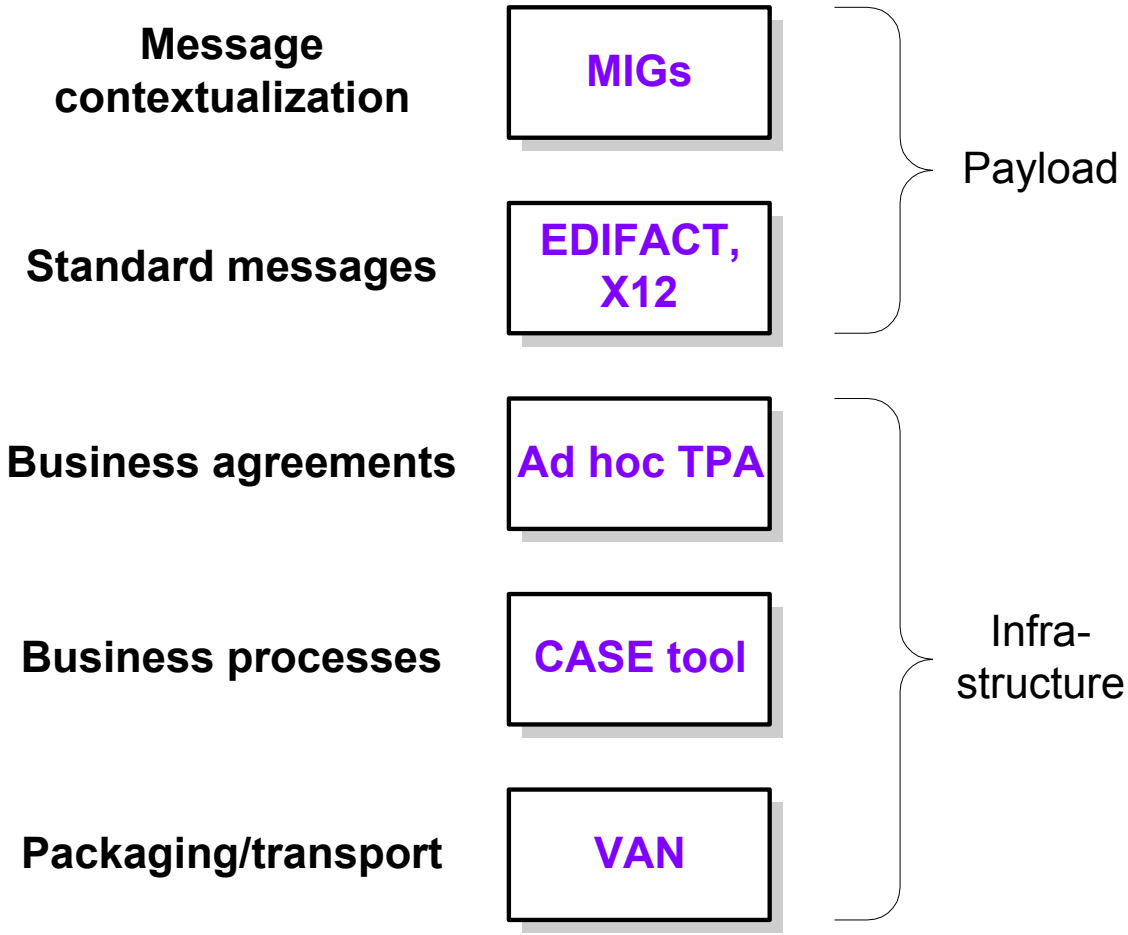
UBL

Lots to cover in this session

- Goals
  - Introduce the Universal Business Language and its unique schema requirements and constraints
  - Describe three major areas of its design, introducing the ebXML Core Components model along the way
  - Help you decide whether you want to apply any of these design rules to your own project, B2B or otherwise
- Assumptions
  - You are familiar with advanced W3C XML Schema concepts
  - But not necessarily an expert in XML B2B in general or ebXML specifically

Overview of UBL and its EDI and ebXML roots

The classic EDI stack



Some EDI pressure points

- It's hard to get in the game
- Private networks are expensive
- You need to do extensive point-to-point negotiation
- The interchange pipe is large, with infinite possible subsets
- You use a "soft" mechanism for adapting to special business contexts

The ebXML initiative

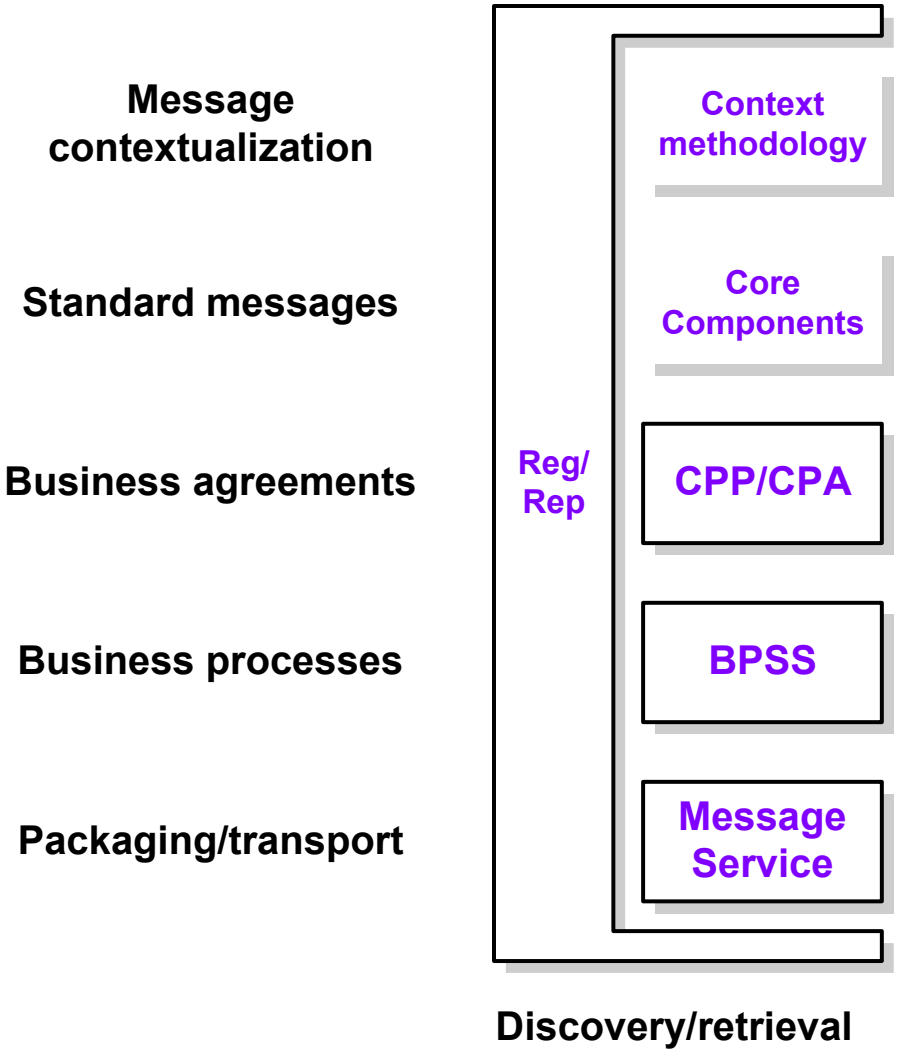
- A joint 18-month effort, concluding in May 2001, of:
  - OASIS (Organization for the Advancement of Structured Information Standards)
  - UN/CEFACT (United Nations Centre for Trade Facilitation and Electronic Business)
- Over 1000 international participants
- The vision: a global electronic marketplace where enterprises of any size, anywhere, can:
  - Find each other electronically
  - Conduct business by exchanging XML messages
- ebXML work continues in OASIS and UN/CEFACT

OASIS

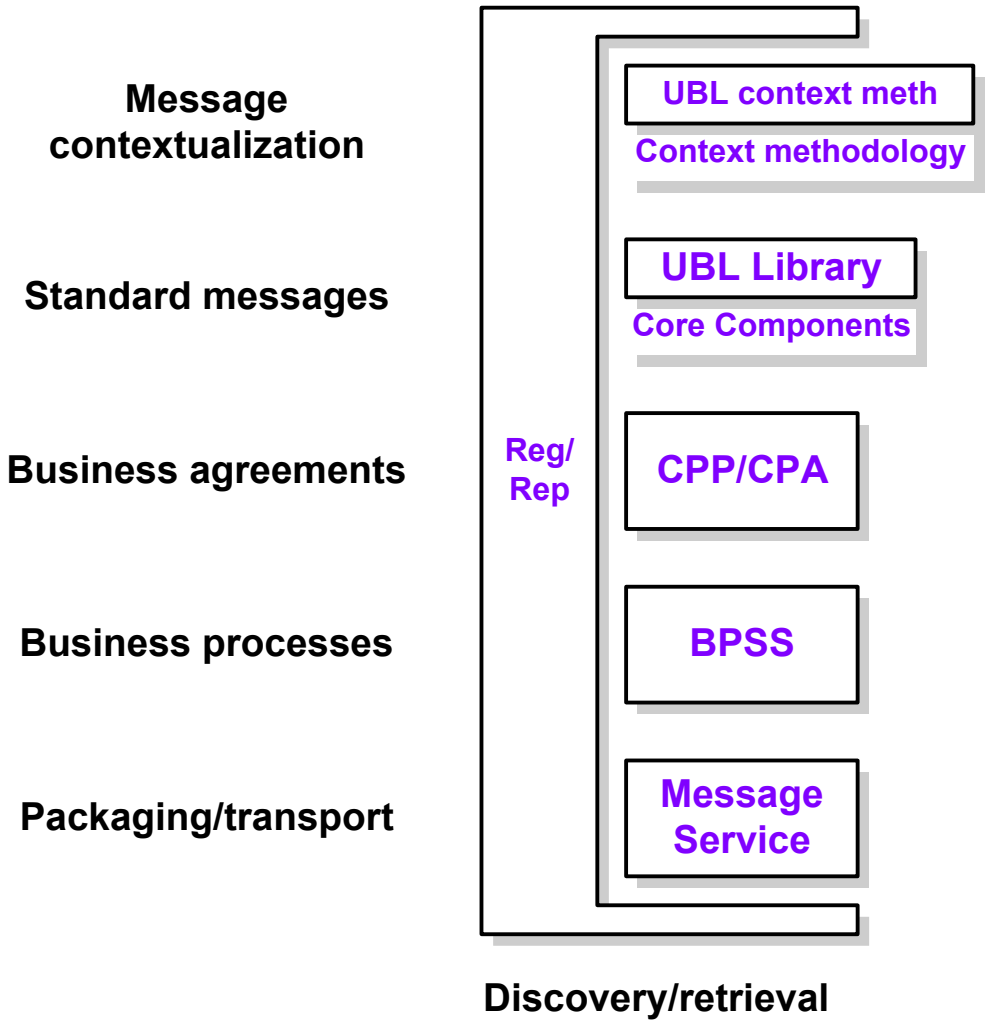
Sun microsystems

UBL

The ebXML stack



UBL proposes to fill out the stack



UBL is…

- An XML-based business language standard being developed at OASIS (though not officially part of ebXML) that…
- …leverages existing EDI and XML B2B concepts and technologies
- …is applicable across all industry sectors and domains of electronic trade
- …is modular, reusable, and extensible
- …is non-proprietary and committed to freedom from royalties
- …is intended to become a legal standard for international trade

The UBL subcommittees that get the work done

- Modeling and content
  - Library Content SC
  - Context Drivers SC
  - (future domain-specific)
- Administrative functions
  - Marketing SC
  - Liaison SC
  - Subcommittee chairs SC
- XML representation and mechanisms
  - Context Methodology SC
  - Tools and Techniques SC
  - **Naming and Design Rules SC**

Requirements on schema design

- Leverage XML technology, but keep it interoperable
- Achieve semantic clarity through a binding to the Core Components model
- Support contextualization (customization) and reuse
- Selectively allow “outsourcing” to other standard schemas

The special requirement for context

- "Standard" business components need to be different in different business contexts
  - Addresses differ in Japan vs. the U.S.
  - Addresses in the auto industry differ from those for other industries
  - Invoice items for shoes need size information; for coffee, grind information
- UBL needs this kind of customization without losing interoperability

A constraint on the design rules themselves

- The UBL Library is being specified in syntax-neutral form using the Core Components model
  - A spreadsheet holds the results
- To convert this *automatically* into schema form requires hard rules, not just guidelines
  - In fact, we do this today with perl scripts
  - W3C XML Schema is our target form of choice

The design rules we'll review today

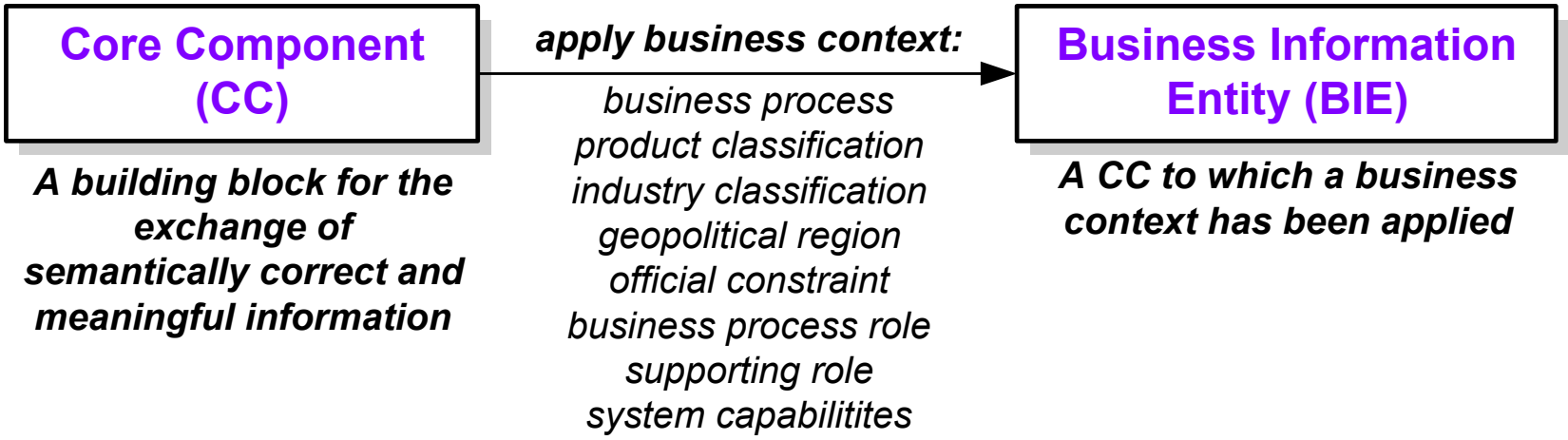
- UBL's mapping to ebXML Core Components, including XML naming rules
- UBL's choice of schema style
- UBL recommendations for the creation of reusable code lists

UBL's mapping to the ebXML Core Components model

Status of the Core Components spec

- The Core Components Technical Specification (CCTS) defines a syntax-neutral *metamodel* for business semantics
  - It is at V1.85 as of 30 September 2002
- Work is ongoing to define an *actual dictionary* in the Core Components Supplementary Documents (CCSD)
  - These are currently non-normative
- UBL is, first and foremost, striving to use the CCTS metamodel accurately
  - And offering feedback for further CCTS/CCSD development

Core components vs. business information entities



- An address might be a generic CC
- A U.S. address has (at least) the geopolitical region set as its business context, making it a BIE
- UBL, by its nature, deals only in BIEs

The Core Components spec follows ISO 11179

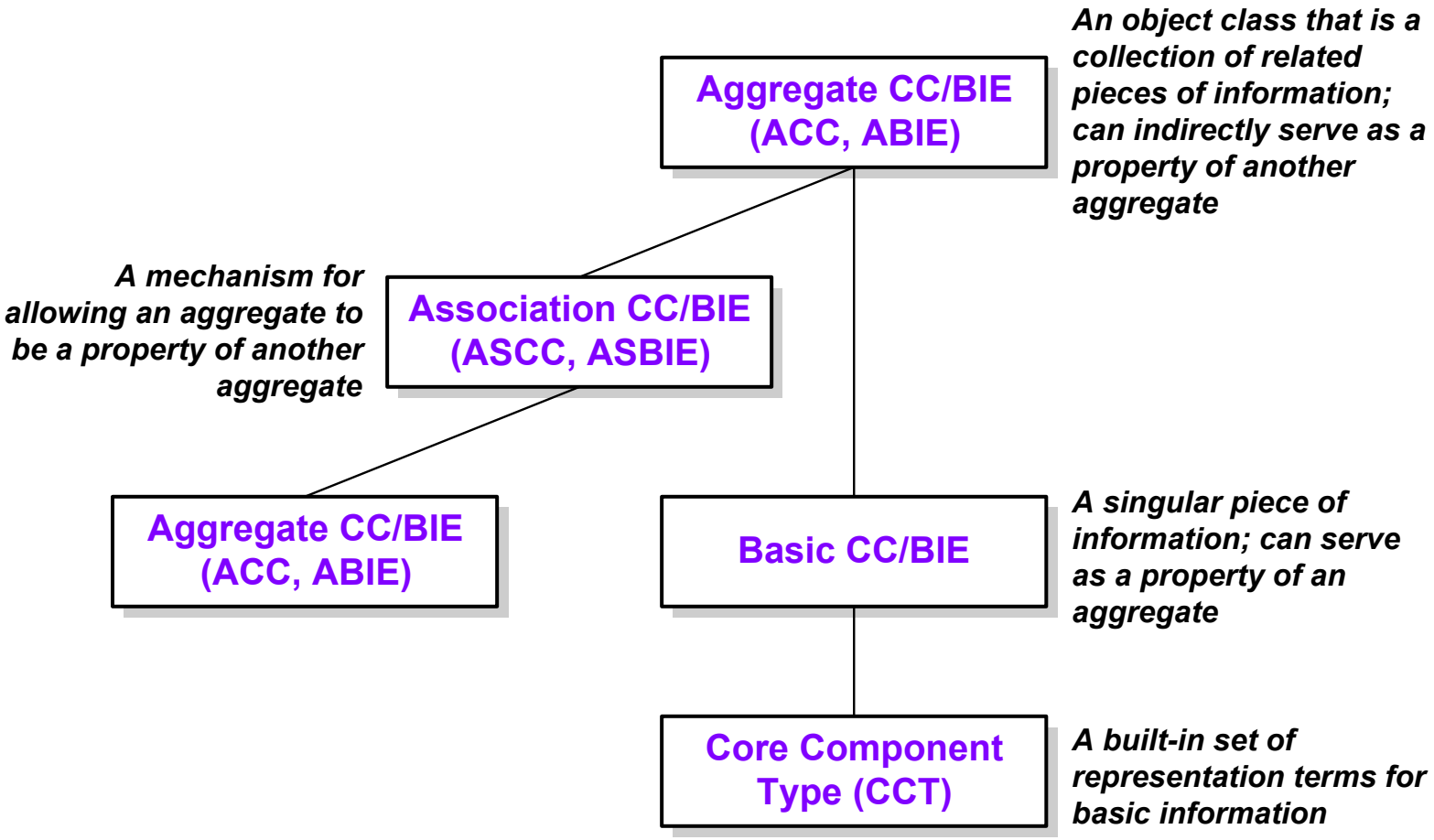
Object class
Property 1: representation 1 Property 2: representation 2 Property 3: representation 3 Property 4: representation 4

Address
Street: text Post code: text Town: text Country: identifier

ISO 11179 governs data dictionaries:
defines the notions of object class, property, and representation term

- This is basic object-oriented “good stuff”

Different kinds of CC and BIE



A tiny sample data dictionary

Person

Name: text
Birth: date
Residence Address: Address
Official Address: Address

Address

Street: text
Post Code: text
Town: text
Country: identifier

Key:
Object class (aggregate BIE) Property (basic BIE)
Property (association BIE) Representation term (CCT)

- This leaves out cardinality considerations for simplicity

The Core Component Types

- The CCTs are built-in ebXML representation terms for indicating constraints on basic information
- The current list of CCTs:
  - Amount
  - Binary Object (plus Graphic, Picture, Sound, and Video)
  - Code
  - Date Time (plus Date and Time)
  - Identifier
  - Indicator
  - Measure
  - Numeric (plus Value, Rate, and Percent)
  - Quantity
  - Text (plus Name)

How dictionary entries are named

- Object classes:
  - *Object Class Term*. “Details”
- Properties:
  - *Object Class Term*. [*Qualifier*] *Property Term*. [*Qualifier*] *Representation Term*
- CCTs:
  - *CCT Name*. “Type”

Person. Details

Person. Name. Text
Person. Birth. Date
Person. Residence Address. Address
Person. Official Address. Address

Address. Details

Address. Street. Text
Address. Post Code. Text
Address. Town. Text
Address. Country. Identifier

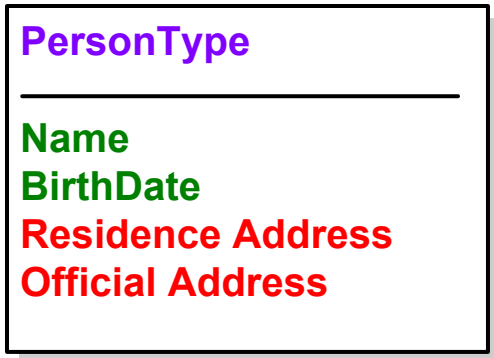
Key:
Object class (aggregate BIE) Property (basic BIE) Property (association BIE)

How this would map to a UBL schema

- Person. Details and Address. Details (and any other object classes) become complex types in the UBL Library
- Person. Name. Text and all the other properties become elements
- Text, date, and other CCTs become complex types in the UBL Library's "built-in" CCT schema module
  - Codes and identifiers are a special case

UBL's XML naming rules

- Remove periods and spaces
- Replace "Details" with "Type"
- On properties (elements), leave out the object class term
  - XPath gives you uniqueness
- Remove redundant words
- Remove "Text" as the default CCT
- Truncate "Identifier" to "ID"



Key:
XSD complex type
XML element bound to a CCT type
XML element bound to a regular complex type

UBL's choice of schema style

XSD offers many options for schema organization

- Elements and types can be managed separately
- Type inheritance and derivation allows for deep type hierarchies
- Elements, datatypes, and attributes can independently be locally or globally scoped
- Namespace support allows for distributed component creation and reuse
  - And importing (outer) schemas can reset some settings

Several options have become well known

- **Russian Doll**, **Salami Slice**, and **Venetian Blind** have been proposed by Roger Costello (xfront.com)
- A fourth obvious option is **Garden of Eden**
- There are many variations we won't go into here
  - There are some weird ones, like making all attributes global

Russian Doll

```
<xs:schema … >
  <xs:element name="Person">
    <xs:complexType> keep nesting ever more deeply…
      <xs:element name="Name" type="NameType" />
      <xs:element name="BirthDate" type="DateType" />
      <xs:element name="ResidenceAddress">
        <xs:complexType>
          <xs:element name="Street" type="TextType" />
          …
        </xs:complexType>
      </xs:element>
      <xs:element name="OfficialAddress">
        <xs:complexType> … </xs:complexType>
      </xs:element>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Salami Slice

```
<xs:schema … >
   <xs:element name="Person"> only elements are at the top level…
     <xs:complexType>
      <xs:element ref="Name" />
      <xs:element ref="BirthDate" />
      <xs:element ref="ResidenceAddress" />
      <xs:element ref="OfficialAddress" />
     </xs:complexType>
   </xs:element>
   <xs:element name="Name" type="TextType" />
   <xs:element name="BirthDate" type="DateType" />
   <xs:element name="ResidenceAddress">
     <xs:complexType> … </xs:complexType>
   </xs:element>
 </xs:schema>
```

Venetian Blind

```
<xs:schema … >  mostly types are at the top level…
   <xs:element name="Person" type="PersonType">
   <xs:complexType name="PersonType">
     <xs:element name="Name" type="NameType" />
     <xs:element name="BirthDate" type="DateType" />
     <xs:element name="ResidenceAddress" type="AddressType"/>
     <xs:element name="OfficialAddress" type="AddressType"/>
   </xs:complexType>
   <xs:complexType name="AddressType">
     <xs:element name="Street" type="TextType" />
     <xs:element name="PostCode" type="TextType" />
     <xs:element name="Town" type="TextType" />
     <xs:element name="CountryID" type="…" />
   </xs:complexType>
 </xs:schema>
```

Garden of Eden

```
<xs:schema
   targetNamespace="http://www.example.com/BIEs"
   … > everything's at the top level…
   <xs:element name="Person" type="PersonType">

   <xs:complexType name="PersonType">
     <xs:element ref="Name" />
     <xs:element ref="BirthDate" />
     <xs:element ref="ResidenceAddress" />
     <xs:element ref="OfficialAddress" />
   </xs:complexType>

   <xs:element name="Name" type="TextType" />

   <xs:complexType name="TextType"> … </xs:complexType>

   …
 </xs:schema>
```

Some potential criteria for choosing a style

- Flexibility:
  - Does the vocabulary need to adapt, chameleon-like, to different namespaces?
- Consistency:
  - Is it okay for the vocabulary to bounce between qualified and unqualified? What happens when importing schemas do overrides?
- Reuse:
  - What constructs might someone else want to reuse wholesale?
- Specialization:
  - What constructs might someone else want to modify?

UBL's specific concerns

- Validators and transformation/query engines need to work
  - Type-awareness in tools isn't always easy to come by
- Both direct reuse and customization need to work
  - No surprises
  - No weird or inconsistent results
  - Simple things should be simple; hard things should be possible
- Semantic clarity needs to be retained at all times
- We ultimately chose **Garden of Eden**

Consequences of this choice

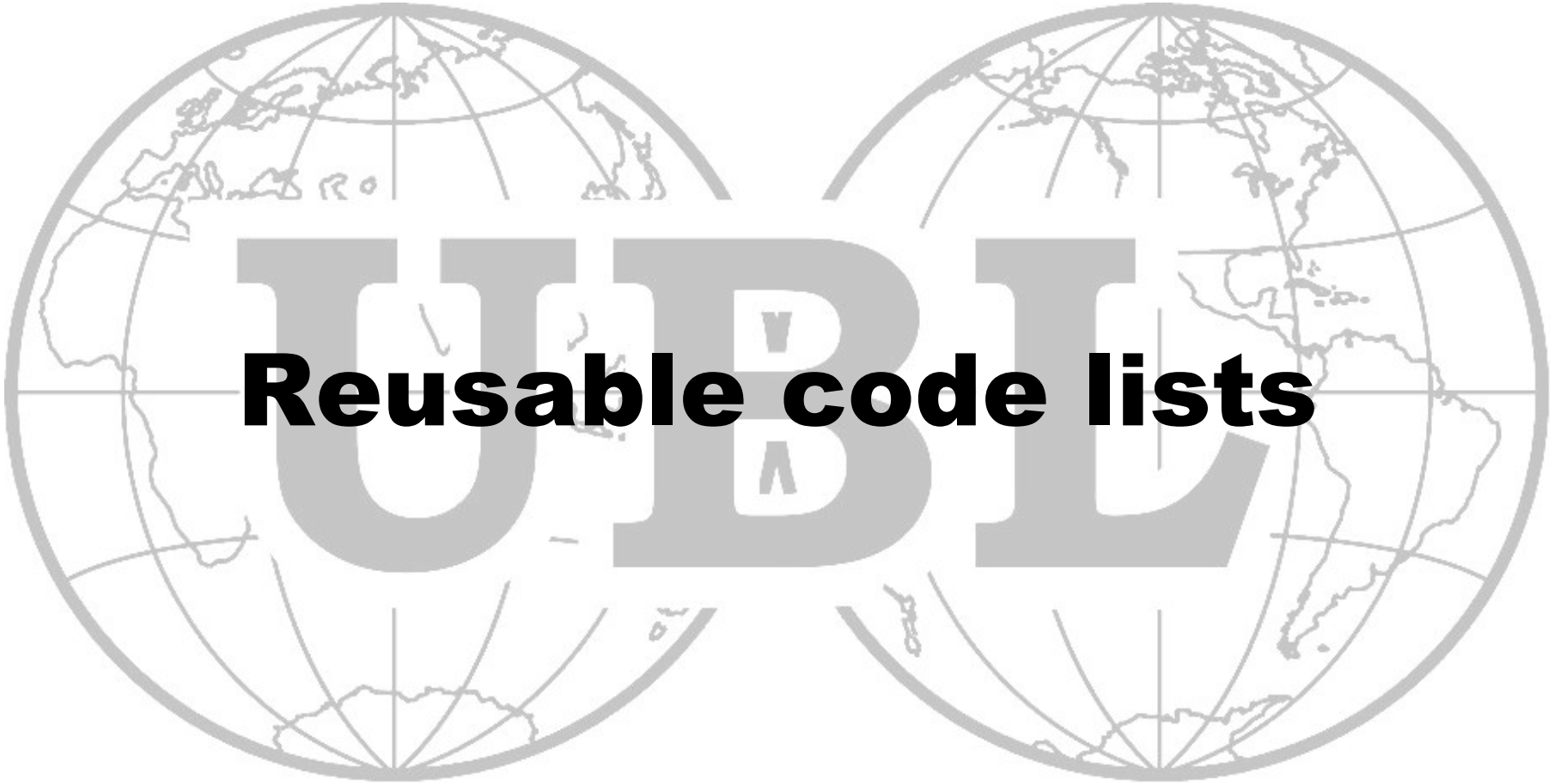
- Every object class/complex type has a corresponding global element declaration for direct reuse
- Properties become references to those declarations
- Properties with the same XML name must be able to share a common object class definition
- This complicates modeling and the algorithm for generating the schema from the syntax-neutral model
  - But it's better to optimize for the users than for ourselves!
- But it has the benefit of rationalizing how we name object classes
- And it gives us some useful new type hierarchy depth

Simple example

```
<xs:complexType name="AddressType">
   gets its semantics from the Address. Details object class
   …
</xs:complexType>
<xs:element name="Address" type="AddressType" />
 same generic Address. Details semantics

<xs:complexType name="PersonType">
   <xs:element ref="Address" />
   gets its semantics from Address as a property of the Person
   …
</xs:complexType>
```

Complex example

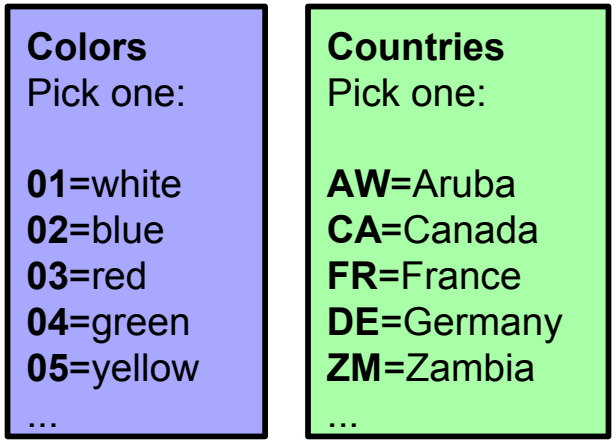
```
<xs:complexType name="AddressType">
   gets its semantics from the Address. Details object class
   …
</xs:complexType>
<xs:element name="Address" type="AddressType" />

<xs:complexType name="ResidenceAddressType">
  <xs:complexContent>
    <xs:extension base="AddressType" />
    gets its semantics from a new ResidenceAddress. Details object class;
    same is true for OfficialAddressType
  </xs:complexContent>
</xs:complexType>

<xs:element name="ResidenceAddress"
  type="ResidenceAddressType" />
  gets referenced in PersonType and maybe other places too, picking up
  property-level additional semantics as it goes
```



Reusable code lists

Code lists in business documents

- A code is a character string that represents a definitive value
- Code lists are valuable as unambiguous taxonomies
- In many cases, code lists are big business

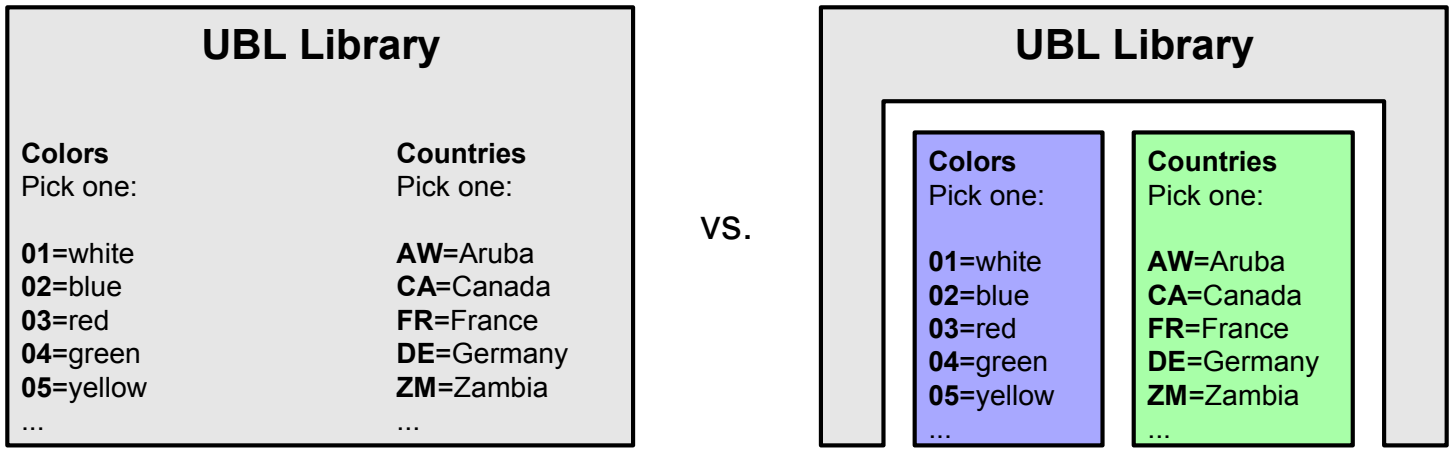


Options for formal representations of code lists

- Often the lists are merely maintained in text documents
- But formal encodings are immensely useful
  - For example, as RDF ontologies or in the ebXML Registry Information Model's **`<ClassificationScheme>`** language
- In addition, UBL and other vocabularies that are "consumers" of code lists need them in XSD form for reasons of validation and semantic clarity

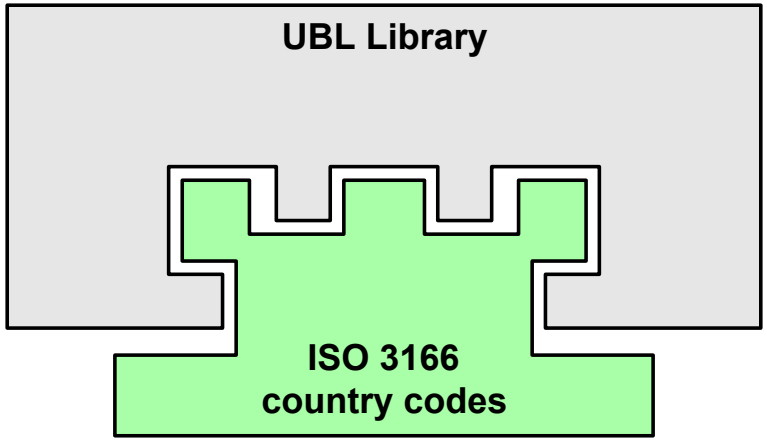
Each consumer schema could create its own version

- But this is costly and prone to error
- Better to help code list producers create their own code list schema modules



The UBL solution: code list schema recommendations

- The code list producer needs to identify the attributes that make the list unique:
  - An XML namespace for its schema
  - A unique agency name, code list name, version, and so on
- …and define a prescribed set of complex and simple XSD types that can be bound in a standard way to a native (e.g., UBL) element



The native element is unique to that code list

```
<CountryID>
  <ISO3166CountryCode attribs…>FR</ISO3166CountryCode>
</CountryID>
```

- The outer element is generic, while the inner element is specific
- The code value itself doesn't have to be a string; it could have nested XML structure
- The simple type governing the value can be "tight" or "loose", depending on what the code list producer wants to maintain over time:
  - Enumerated list
  - Pattern
  - No constraints at all
- The unique attributes can be defaulted, or even fixed

A global marketplace in code lists?

- If these recommendations are followed, we could see…
- …less duplication of work in XML language development
- …wider application platform support for well-known code lists
- …earlier validation of code values
- …standardization of more code lists, and even subsetting and extension

Conclusion

UBL has had to solve some tough schema problems

- Some of its needs are unique, but many might be shared by you
- Our hope is that UBL's schema naming and design rules may be helpful to others
- Please see the paper in the proceedings for further reading
- Please see other talks at this conference for more on other areas of UBL development

Thanks!
Questions?

Eve Maler
eve.maler@sun.com