Approaches to mapping

- XSLT
- RDBMS views
- CORBA IDL
- A declarative approach

'Web Services Made Easier', Sun Microsystems Technical White Paper, http://java.sun.com/xml/webservices.pdf

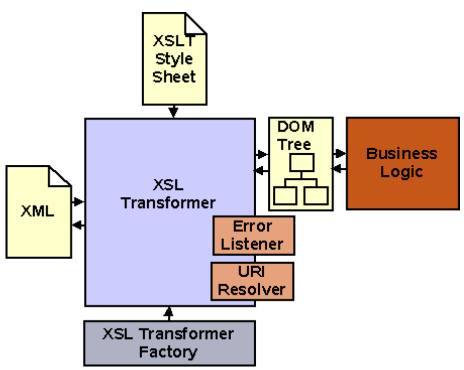
The Java Web Services Tutorial, http://java.sun.com/webservices/tutorial.html

XSL/XSLT

- Extensible Stylesheet Language (XSL) and XSL Transformations (XSLT)
- XSL is a formatting language, for converting XML documents into formatted documents (building upon style sheets)
  - Higher level approach
    - Codes transformations as rules
    - Condition patterns specified using Xpath expressions
    - Little Java coding needed – a scripting approach
- Uni-directional mapping specification

XSLT

- Basic approach, transform from DOM to DOM using XSL stylesheet to specify the transformation
- Resultant DOM represents formatted document which is then walked to produce output
- Some implementations handle SAX inputs directly (so don't need a DOM)



XSL Basic Approach

- XSL uses a rule-based template matching approach
- XSL uses a XML encoding so it has a tagged structure (which makes it difficult to read)
- Example with the coffee price list DTD from the web services paper:

```
<!ELEMENT priceList (coffee)+>
<!ELEMENT coffee (name, price) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

```
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
  <coffee>
    <name>Sumatra</name>
    <price>12.50</price>
  </coffee>
</priceList>
```

XSL Rules

- XSL is a rule-based language. Rules (template rules) have:
  - A match pattern, to match against XML elements specified as an Xpath expression
  - A template which specifies the form of the document to produce if an element matches
  - A template may cause further rules to be applied

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="name">   Matches elements with tag name
    <tr><td>                    Constructs a html table row
      <xsl:apply-templates/>    Apply a stylesheet to bits of name element
                                Result goes in this place
    </td></tr>                  Completes the html table row
  </xsl:template>
```


XSL for Coffee Pricelist

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="priceList">
    <html><head>Coffee Prices</head>
      <body>
        <table>
          <xsl:apply-templates />
        </table>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="name">
    <tr><td>
      <xsl:apply-templates />
    </td></tr>
  </xsl:template>
  <xsl:template match="price">
    <tr><td>
      <xsl:apply-templates />
    </td></tr>
  </xsl:template>
</xsl:stylesheet>
```

Application to an example

```
<priceList>                       <html><head>Coffee Prices</head>
                                    <body>
                                      <table>
  <coffee>
    <name>Mocha Java</name>                 <tr><td>
                                              Mocha Java
                                            </td></tr>
    <price>11.95</price>                    <tr><td>
                                              11.95
                                            </td></tr>
  </coffee>
  <coffee>
    <name>Sumatra</name>                    <tr><td>
                                              Sumatra
                                            </td></tr>
    <price>12.50</price>                    <tr><td>
                                              12.50
                                            </td></tr>
  </coffee>
</priceList>                              </table>
                                    </body>
                                  </html>
```

Xpath and More Complex Matching

- See the handout from Java Web Services Tutorial for a more complete description of Xpath expressions
  - "/" The root element
  - "/priceList/name" name elements of priceList
  - "SECT|PARA|NOTE" Only SECT, PARA, or NOTE elements
  - "LIST/@type" The type attribute of LIST elements
- Using these can pull a XML structure apart and reorder the results to give a very different tree shape as a result

RDBMS views

- Allow database information to be accessed (and sometimes modified) in different forms
- Based on SELECT statement

```
CREATE VIEW titles_view AS
SELECT title, type, price, pubdate FROM titles
```

  - Allows any alternate structure possible through selections, joins, orderings, grouping, and calculations
  - However, to be updatable there are severe restrictions
    - No aggregate functions, grouping, unions, distincts, derived columns (calculations)
    - Insert and update can only reference columns from one table when a join is utilised
    - Delete can only work on views based on one table

RDBMS view example

```
CREATE VIEW publication_view AS
SELECT title, creator AS author, isbn, subject AS classification, description,
  tableOfContents AS contents, cost AS price
  FROM publication
```

```
CREATE VIEW publication_view AS
SELECT title, creator AS author, isbn, subject AS classification, description,
  tableOfContents AS contents, cost/0.5855 AS price
  FROM publication
```


CORBA IDL

- IDL: Interface Description Language
- CORBA IDL is a language-independent interface specification (declarative)
- Consists of modules, interfaces, types (structs, enumerated, ints, reals, strings etc.)
- Also might include exceptions, references to other IDL module specifications
- C++/Java-like syntax, but limited number of types available

IDL Components

- Types
  - Basic types
  - Named types
  - Enumerations
  - Structures
  - Unions
  - Arrays
  - Sequences
  - Recursive structures
- Constants
  - Allow expressions
- Interfaces (are a type)
  - Contain Operations
    - Return result type
    - Operation name
    - Zero or more parameters
      - in, out, inout
- User exceptions
- System exceptions
- Attributes
- Modules
- Forward declarations
- Inheritance

IDL Types Examples

```
typedef long Millimeter;
enum WallTypes { interior, exterior, trombe, underground };
struct WallInfo {
   WallTypes type;
   Millimeter height;
   Millimeter width;
}
union WallAtts switch (WallTypes) {
   case trombe:
      long glazingArea;
   case underground:
      Millimeter soilDepth;
}
typedef WallInfo RectangularRoom[4];
typedef sequence<WallInfo> GeneralRoom;
```

```
struct Node {
   long value;
   sequence<Node> children;
};
```


IDL Interfaces Examples

```
module Building {  // like a Java package
  interface Wall {
    exception Incomplete { string missingAtts };
    // attribute definitions here…
    long wallArea() raises(Incomplete);
    void setHeight(in Millimeter newHeight);
    void setWidth(in Millimeter newWidth);
    …
  }
  interface TrombeWall : Wall {
    void setGlazingArea(in long newArea);
    …
  }
  interface Room {
    boolean fixWalls(inout sequence<Wall> wallPieces);
  }
}
```


XSLT, RDBMS VIEW, IDL

- Allow for the transformation of data in one representation into a new representation
- Limitations on the types of transforms supported
- XSLT and IDL are uni-directional
- RDBMS VIEW is bi-directional in very constrained circumstances
- What can we do which is better than this?

A declarative mapping language

- Motivations for a declarative style
  - Abstract from underlying representations
  - Abstract from implementation language
  - Capture of intent of a mapping
  - Able to generate mapping code
- VML (View Mapping Language)
  - Bi-directional mapping specification
  - http://www.cs.auckland.ac.nz/~trebor/pub/phd/Ch5.pdf

Structure of VML

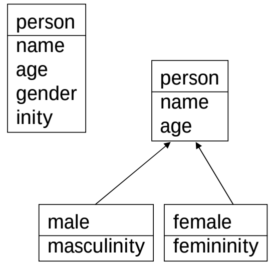
- inter_view
  - Describes the 2 schemas being mapped between
    - Versions being mapped between
    - Type of information transfer required (read-only, read_write, integrated)
    - Whether this is a complete or partial mapping
- inter_class
  - Describes sets of classes that need to combine for a mapping
  - Three parts to each inter_class description
    - Invariants: what must hold true for this mapping to proceed
    - Equivalences: the mappings to perform
    - Initialisers: values to be set when a new object is created

inter_class example

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([person],[male],
        invariants(      gender = 'male'),
        equivalences(    name = name,
                         age = age,
                         inity = masculinity)
).

inter_class([person],[female],
        invariants(      gender = 'female'),
        equivalences(    name = name,
                         age = age,
                         inity = femininity)
).
```



inter_class classes

- Can specify one or more classes from each schema
  - If one class then inter_class is applied to every object of that class (as long as the invariants are satisfied)
  - If more than one class then the cross product of objects is used for the mapping
  - For example:
    - Class a has objects o1 and o2
    - Class b has objects o3, o4, and o5
    - inter_class([a, b], [c], ...) evaluates the mapping for:
      - [o1, o3], [o1, o4], [o1, o5], [o2, o3], [o2, o4], [o2, o5]
    - group() function allows all objects of a class to be grouped
    - E.g., inter_class([a, group(b)], [c], ...) evaluates the mapping for:
      - [o1, [o3, o4, o5]], [o2, [o3, o4, o5]]

invariants

- Define the conditions under which an inter_class is applicable (e.g., gender = ‘male’)
  - Reduce the set of objects which are evaluated
- Each individual invariant may only reference attributes and objects from one of the schemas.
- A constraining condition applied in one direction is a default value in the opposite direction.
  - E.g., when creating a ‘person’ object from one of type ‘male’ in the previous example then the ‘gender’ attribute of the ‘person’ object is set to ‘male’.

initialisers

- Assignment statements for attributes
- Only applicable to newly created objects
  - Can call methods of new objects

```
initialisers(
  idm_space_face.face_property = 'idm_space_face',
  idm_material_face.face_property = 'idm_material_face',
  idm_material_face.material=>type_of_material = 'idm_window_material',
  idm_material_face.material=>type_of_window = 'idm_single',
  idm_material_face.material=>window_subtype = 'clear',
  fe_opening@create(idm_space_face.plane, idm_space_face.plane, 'space', 0, 0,
            idm_space_face.min=>x, 0 - idm_space_face.min=>y,
            idm_space_face.max=>x, 0 - idm_space_face.max=>y,
            idm_material_face.material=>window_subtype)
)
```


equivalences

- Equations, functions, and procedures to perform a mapping
- Ordering of specification is unimportant
- Types of equivalence equations include:
  - Initialisers (e.g., gloss_factor = 90.0)
  - Equality (e.g., name = planeName)
  - Pointer equality (e.g., plane = fe_face_window)
  - Simple equations (e.g., r*sin(theta) = y_coord)
  - Pointer references (e.g., apex1=>x = apex2=>x
  - Functions (e.g., exists(end_point=>z)
  - Aggregate functions (e.g., sum(windows=>(height*width))) = area

equivalences

- Types of equivalence equations include:
  - List and array references (e.g., axes[2] = v_ref)
  - List and array iteration (e.g., classified_by[] = material[].name)
  - Conditional list and array iteration, for example,
    bijection(spaces[]@class('idm_space'), spaces=>list[])
    bijection(spaces[]@class('idm_roof'), roofs=>list[])
  - Functions (e.g., list_splitter(vals, splitvals))
  - Procedures (e.g., map_to_from(procA(), procB()))
  - Method invocation (e.g., plane@view_plane = fe@create_view(name))
  - Type conversion – implicit evaluation or cast explicitly
  - Unit conversion – explicit modelling
  - Temporary/intermediate attributes (e.g., _temp)