XML Processing Paradigms

Three different ways of
working with XML

Lars Marius Garshol,

STEP Infotek A/S

Before we begin

- This tutorial consists of two parts:
  - a fixed core (as advertised)
  - a set of extra optional material
- This frees gives us some choice in what we want to cover after the fixed core
- As I talk, please make notes about what you'd like to see in the optional section

Introduction

What this is all about

Why do we process?

- Conversion: move data to a useful format
- Data extraction: pick out data and use them
- Build in-memory structures for use in programs (specialized extraction)
- Semantic validation
- Basically, move information out of the XML serialization syntax and into use

The concept of representation

- Below are many different representations of the number 223:
  - DF (hexadecimal)
  - 337 (octal)
  - 11011111 (binary)
  - two hundred and twenty-three (plain English)
  - `2*100+2*10+3` (expression)
- These are all different ways of saying the same thing

Representations of documents

- Similarly, XML documents have many possible representations:
  - a string of bytes read from a file (or a socket)
  - a string of characters
  - a sequence of parsing events
  - a tree structure
  - Lisp S-expressions
  - records in a relational database
- These are all equivalent, but have different uses

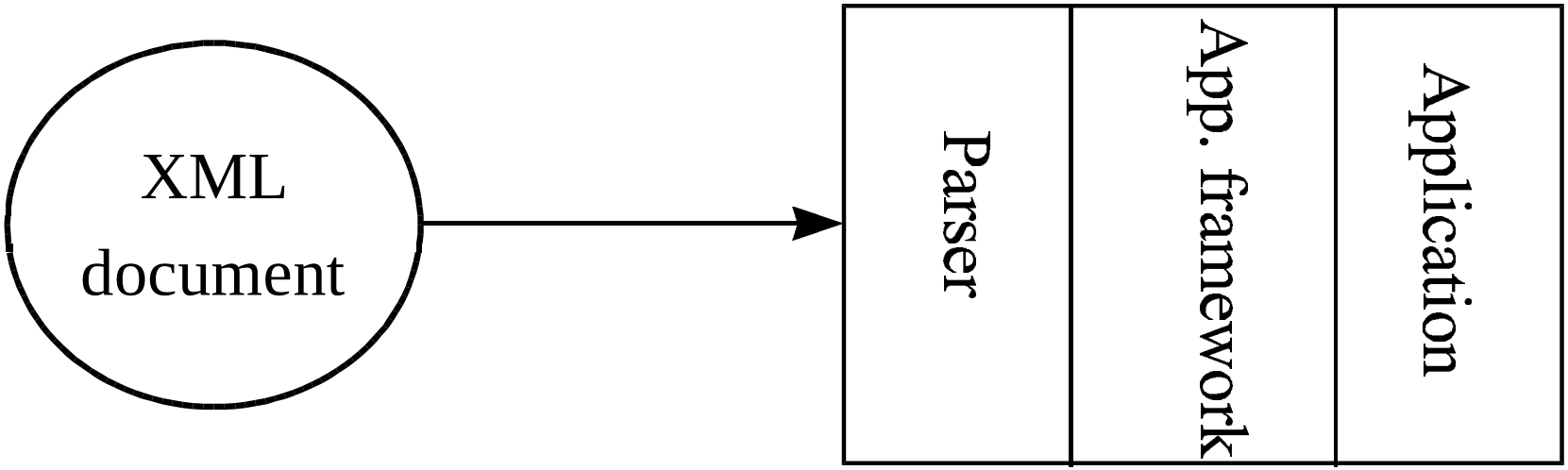
Representations cont'd

- The XML document as a string of bytes is singled out, because XML is mainly intended for exchange across the network
- Similarly, RDBMSs single out the database as a set of tables in a running server as the main representation, because these are mainly intended for interactive use as a single-site storage mechanism

The XML processing model

- Relies heavily on the concept of a parser, something that reads bytes and turns it into elements, character data and attributes
- This is all the parser does: moving up some levels of interpretation from bytes to XML constructs (and processing these)
- On top of the parser other frameworks can be built, but these aren’t parsers or part of it

The processing model, again



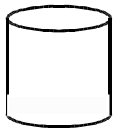
The processing paradigms

- Event-based: attach actions to events like new start tag, new end tag etc
- Tree-based: build a tree and work on it
- Declarative: describe what you want done, and the software does it for you

Required tools

- A parser
- Event-based: nothing more required, though many useful event-based frameworks exist
- Tree-based: a tree builder
- Declarative: a declaration language and processor, usually a tree builder as well

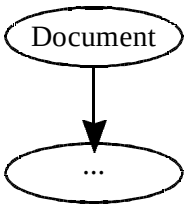
Levels of abstraction



60	63	120	109	108	32	118	101	114	105	...

<	x	m	l		v	e	r	s	i	...

```
startDocument()
xmlDeclaration(...)
startElement(...)
```



Levels of abstraction

- Byte sequence: rock-bottom
- Elements and attributes:
  - Event sequence: better, nesting implicit
  - Tree structure: even better, explicit nesting
- Application-specific
  - Requires custom code, but enables you to forget the XML representation of the information

An example

- XBEL is a simple XML DTD for representing bookmark collections
- To the operating system, an XBEL document is a sequence of bytes with no meaning
- To XML software, it is an XML document, with elements and attributes
- To XBEL software, it is a bookmark collection, with folders, bookmarks and descriptions

Levels of information

- Basic logical:
  - only gives you the logical document
- Full logical:
  - the logical document + the DTD
- Basic lexical:
  - logical + entity boundaries, comments, CDATA sections/PCDATA
- Full lexical:
  - whitespace in tags, character refs, DTD info...

Event-based processing

Turning bytes into events

```
<example>
<line>&quo;Hello,
  world!&quo;</line>
</example>
```

- start document
- start element: example
- start element: line
- text: “Hello, World!”
- end element: line
- end element: example
- end document

Event-based processing

- The most low-level paradigm, which the others can be built on top of
- For simple applications, event-based processing is very natural and easy
- For more complex applications you need to build an apparatus to keep track of state
- Some frameworks do this for you

Event-based processing

- Simple to implement
- Requires few resources
- Processing may be event-based even if the framework gives access to the full tree

Some event-based frameworks

- Most native parser APIs
- SAX
- OmniMark
- Balise
- DSSSL
- SAXON
- MDSAX

Native parser APIs

- The following parsers have event-based native APIs:
  - expat
  - SP
  - sgmlop/xmllib
  - xmlproc
  - Lark
  - XP
  - TclXML
  - XML::Parser
  - Ælfred
  - RXP
  - and many others…

Native parser APIs

- Require you to register handlers for events, either functions (C, tcl) or objects (Java, Python)
- Usually also allow various options to be set
- Some have options to allow non-standard behaviour

The expat API

- Application must register handler functions like:
- `void XML_StartElementHandler (void *userData, const XML_Char *name, const XML_Char **atts)`
- `XML_EndElementHandler(void *userData, const XML_Char *name)`
- `XML_CharacterDataHandler(void *userData, const XML_Char *s, int len)`
- `XML_ProcessingInstructionHandler(void *userData, const XML_Char *target, const XML_Char *data)`

The expat API

- `XML_UnknownEncodingHandler(void *encodingHandlerData, const XML_Char *name, XML_Encoding *info);`
- `XML_DefaultHandler(void *userData, const XML_Char *s, int len)`
- `const XML_LChar XMLPARSEAPI *XML_ErrorString(int code);`
- `int XMLPARSEAPI XML_Parse(XML_Parser parser, const char *s, int len, int isFinal);`

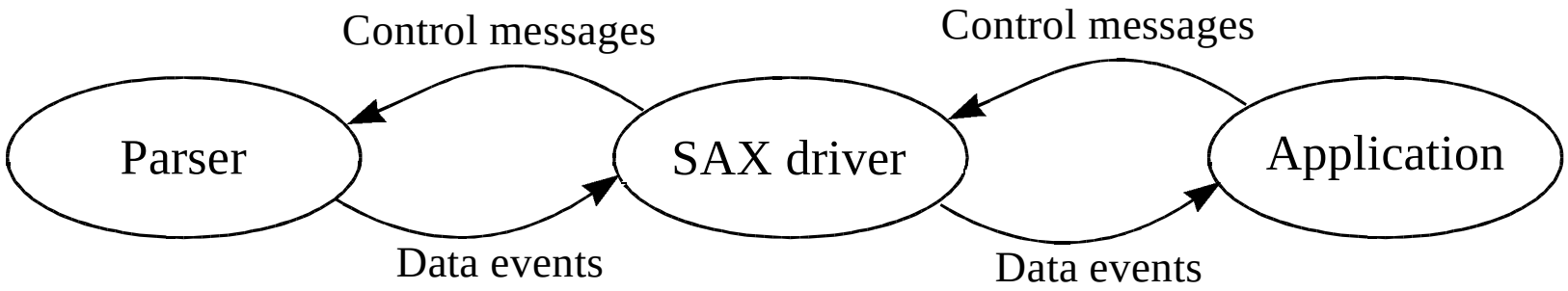
Problems

- All different, so if you want to switch parsers you need to rewrite your application (and learn a new API)
- General applications become parser-bound
- Also, utilities built for one parser only work with that specific parser

SAX: Simple API for XML

- SAX is a standardized API to parsers, developed on the xml-dev mailing list
- Currently supported by nearly all Java parsers and all Python parsers
- Some attempts have been made at translation into Delphi, C/C++ and Perl, but nothing definite and widely supported has yet emerged

SAX: How it works



• The driver implements the SAX parser interface, and at the same time acts as a native application of the parser

SAX: Basic processing

- The SAX driver implements the Parser interface, which has two Parse methods (accepting an InputSource or a URL) and some methods to set various handlers
- The application implements the DocumentHandler interface, which has methods for receiving data events

The Parser interface

- Has these methods:
  - parse(sysid) / parse(InputSource)
  - setDocumentHandler
  - set*Handler
  - setLocale

The DocumentHandler

- Methods:
  - startElement(name, attrs)
  - endElement(name)
  - characters
  - processingInstruction(target,data)
  - startDocument()
  - endDocument()
  - setDocumentLocator(locator)

SAX: A simple example

```
class ExampleApp(saxlib.DocumentHandler):
    def __init__(self):
        self.count=0
    def startElement(self,name,attrs):
        self.count=self.count+1
    def endDocument(self):
        print “There were”,self.count,”elements.”

p=saxexts.make_parser() # Instantiates a parser
p.setDocumentHandler(ExampleApp())
p.parse(“test.xml”)
```

SAX: Error handling

- SAX requires you to register a separate error handler to receive error events
- The same object may play both roles
- Three levels of errors exist:
  - warnings: not true errors
  - errors: validity errors
  - fatal errors: well-formedness errors

The ErrorHandler

- Methods:
  - warning(exception)
  - error(exception)
  - fatalError(exception)
- The exceptions contain the information necessary to find the location of the error

SAX: Working with attributes

- Attribute information is provided by the AttributeList interface
- Provides:
  - attribute values and names
  - complete enumeration
  - attribute type information (if available)

AttributeList

- Methods:
  - getLength()
  - getName(ix)
  - getType(ix) / getType(name)
  - getValue(ix) / getValue(name)
- In Python these can be used as if they were built-in lists or dictionaries

A common technique

- Characters event: add data into an internal buffer
  - event may be split
- Actually handle the contents of the element in the endElement event

Demo

- Show XBEL example
  - go through source
  - run on cos_urls.xml
  - run on pyhoo.xml

Another technique

- To stop parsing (because of errors or whatever):
  - throw a SAXParseException
  - define your own subclass if:
    - you need to provide more information
    - you need to single out your own exceptions

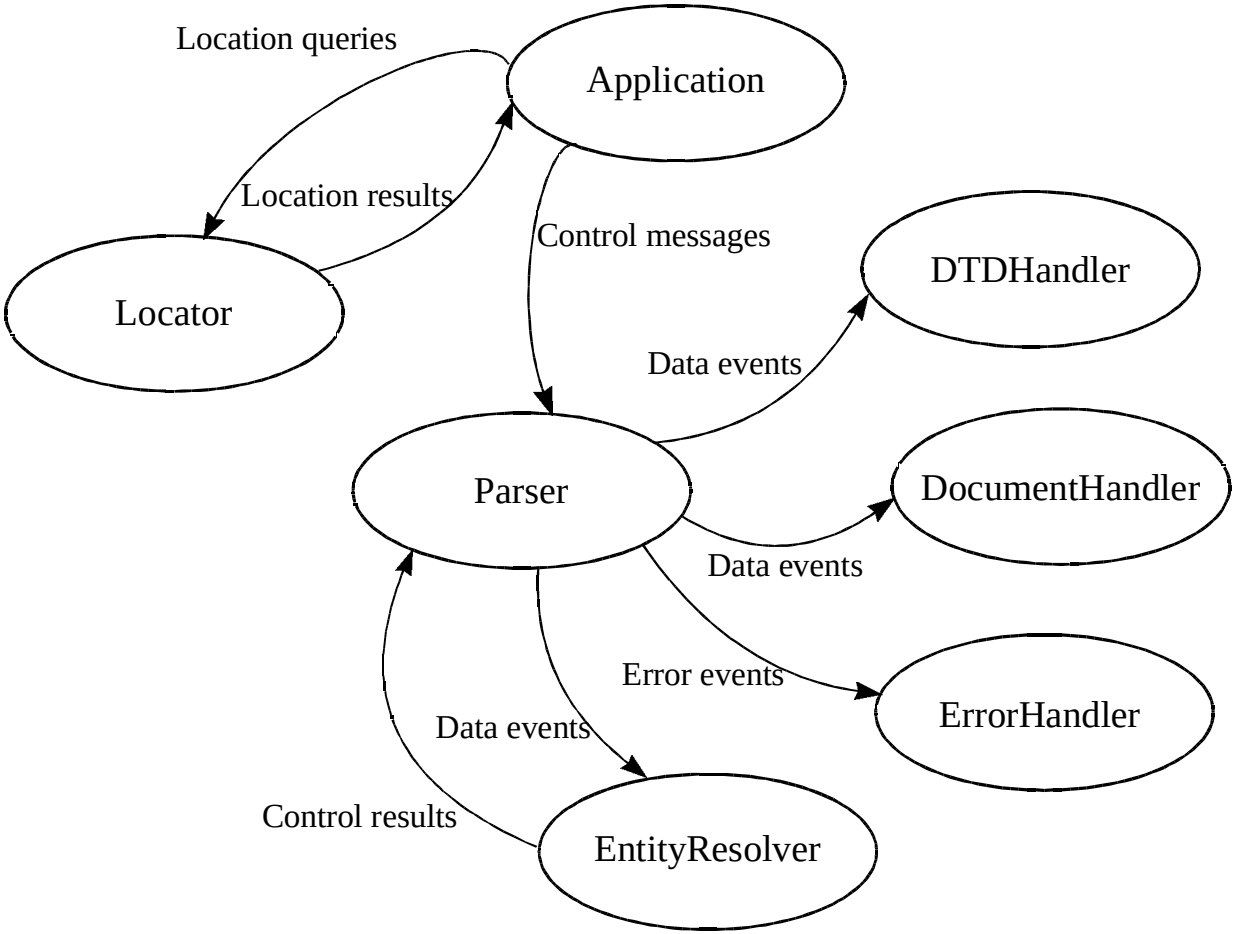
SAX: Esoteric stuff

- InputSources can be used to feed input from sources other than URLs to the parser
- It can also be used to implement your own character encodings
- The EntityResolver handler allows you to interpret system identifiers yourself, and also to resolve public identifiers

SAX: More esoterica

- The DTDHandler lets you receive entity and notation declarations
- The Locator can be used to get information about the current location in the document

SAX: Complete view



Java SAX helper classes

- ParserFactory: Can be used to create an XML parser specified by a parameter or a Java property
- LocatorImpl: Can be used to store copies of location information
- AttributeListImpl: Can be used to store copies of attribute lists

Python SAX helper classes

- ErrorRaiser
- ErrorPrinter
- ParserFactory
  - creates parsers from predefined lists
- Locator
  - like LocatorImpl
- AttributeMap
  - like AttributeListImpl

Python SAX helper classes

- EventBroadCaster
  - forwards events to all handlers in a list
- mllib
  - implements the old-style Python interface

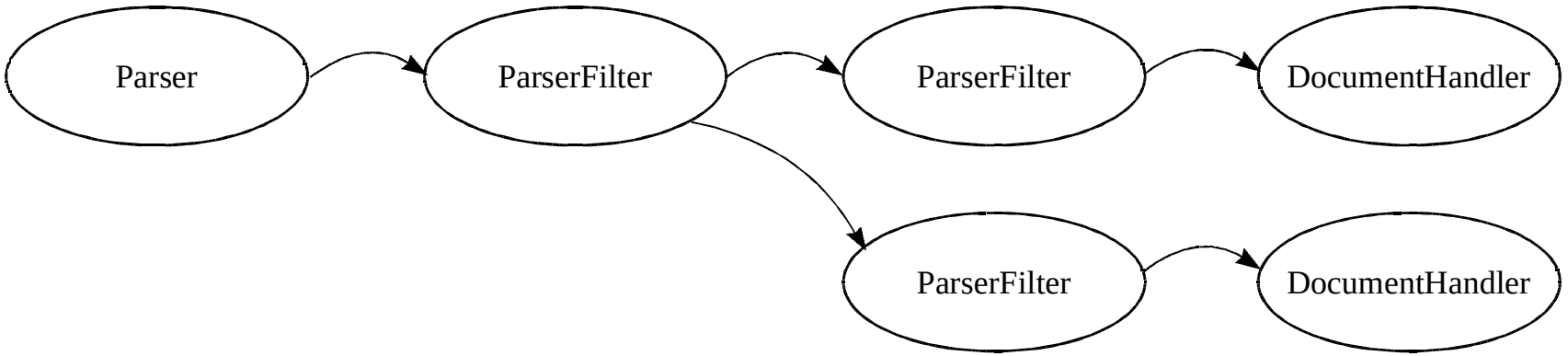
SAX: Event sources

- SAX events need not come from parsers in the traditional sense
- Alternatives:
  - A DOM walker
  - An XSL implementation
  - A program that generates XML

SAX: Parser filters

- Parser filters are objects that receive events from the parser (or another filter) and pass them on to the application (or another filter)
- Possible applications:
  - implement namespaces outside parser
  - implement architectural forms outside parser
  - strip unnecessary whitespace
  - implement attribute inheritance

SAX: Parser filters



Advantages

- Processing components can be developed that:
  - can be mixed (more or less) freely
  - are independent of parsers
  - can be used with XML generators also
- Some filters exist already

Demo

- Go through filters.py source
- Show sax_esis2.py source
- Run on test.xml and show difference
- Play around with various combinations

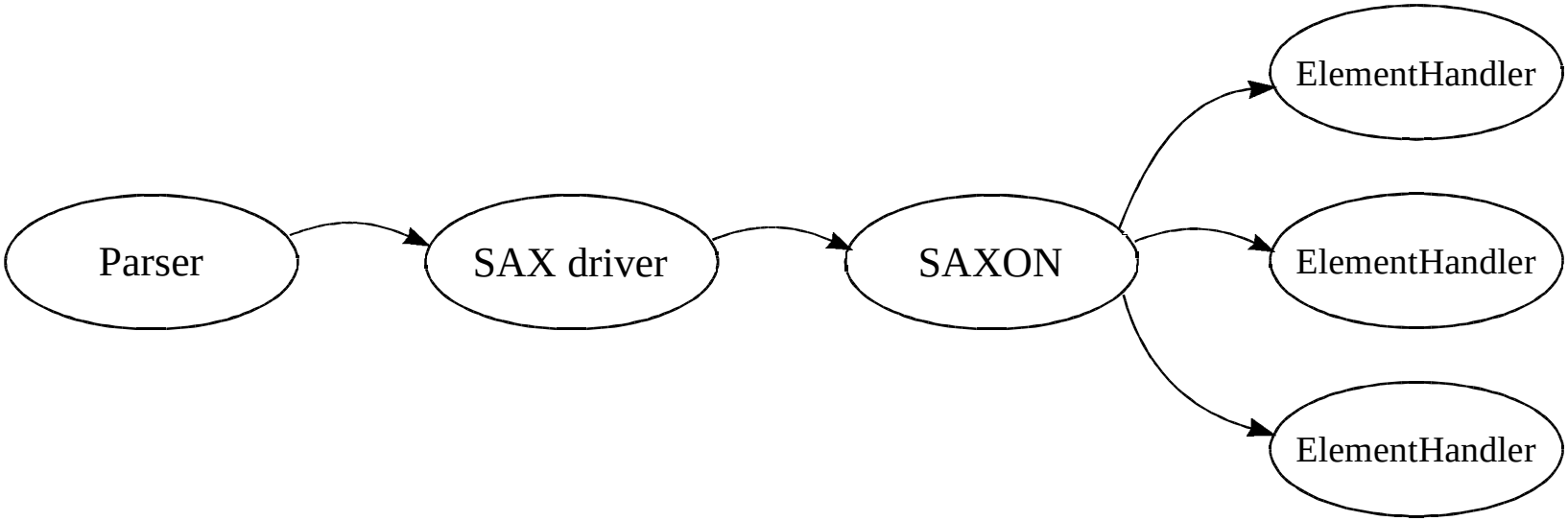
SAX 2

- Is currently being discussed
- Is more open than SAX 1.0, by allowing for:
  - querying of features by ID
  - registering handlers by ID
  - setting parameters, also by ID
- Some standard handlers will be specified in SAX2, probably those for namespaces, lexical information and DTD information
- A set of IDs is also specified

SAXON

- A framework for making XML processing applications, built on SAX and the DOM
- Designed for processing that produces output
- Event-based, but gives you access to the document tree
- Works by defining separate handler objects for each element type
- Comes with a number of useful handlers

SAXON: How it works



SAXON internals

- Two main modes of operation:
  - Distributor: calls handlers in document order
  - Wanderer: ditto by default, but allows handlers to influence the order by controlling processing
- Can use XSL patterns to select handlers and apply processing
- Supports nearly all of XSL

Ready-made handlers

- ElementHandlerBase: Does nothing
- ElementCopier: Just copies the element
- ItemRenderer: Inserts user-defined text before and after content, content is copied
- GroupRenderer: Like ItemRenderer, but acts on a group of consecutive elements
- ItemSorter: Sort consecutive elements

More ready-made handlers

- NumberHandler: Used to number source elements for use by other handlers
- ElementToAttributeConverter: Like it says, but on source elements
- ElementSuppressor: Like it says
- ElementRedirector: Sends output from an element to a specified Writer (which is closed afterwards)

Conclusion

- SAXON makes it easier to develop processing applications by:
  - defining high-level components
  - providing some standard components
  - providing extra XSL-based facilities
- Cost:
  - You have to learn it

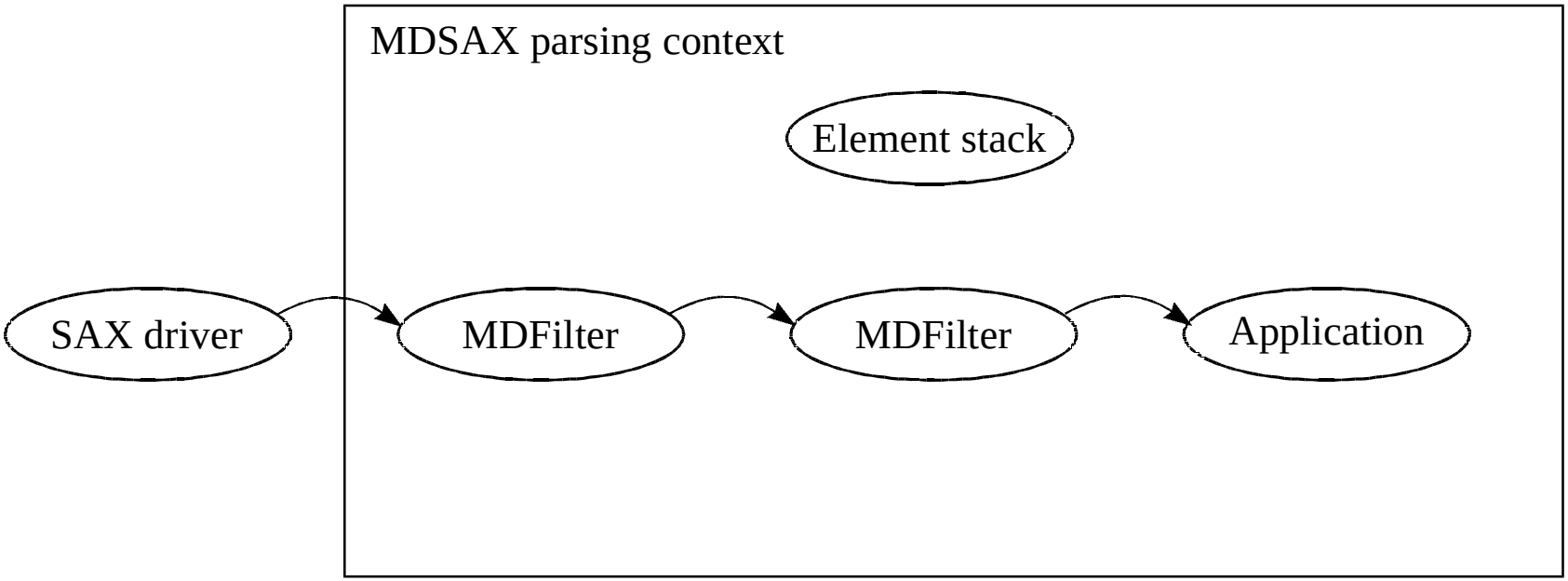
MDSAX

- A framework for building SAX processing applications
- Relies heavily on the concept of parser filters
- Provides a default filter interface, a common environment for filters (for communication and resource sharing) etc

MDSAX services

- Shared element stack between filters
- Queue of operations to perform after the parsing is complete
- Event routing concept (branching the event stream into a tree, keeping element substacks for the branches)
- XML markup language for setting up filter configurations

MDSAX


MDSAX parsing context
Element stack
SAX driver
MDFilter
MDFilter
Application

MDSAX: Standard filters

- MDFlattenFilter: removes the tags of an element, passing on the content
- MDAttlistFilter: validates attributes
- MDNamespaceFilter: performs namespace processing
- MDInheritanceFilter: performs attribute inheritance
- MDXAFFilter: architectural forms

DSSSL

- DSSSL is an ISO-standardized style sheet and transformation language
- It can convert between SGML and XML DTDs as well as to presentation formats
- It is event-based and uses a subset of Scheme for programming
- Allows tree navigation and reprocessing

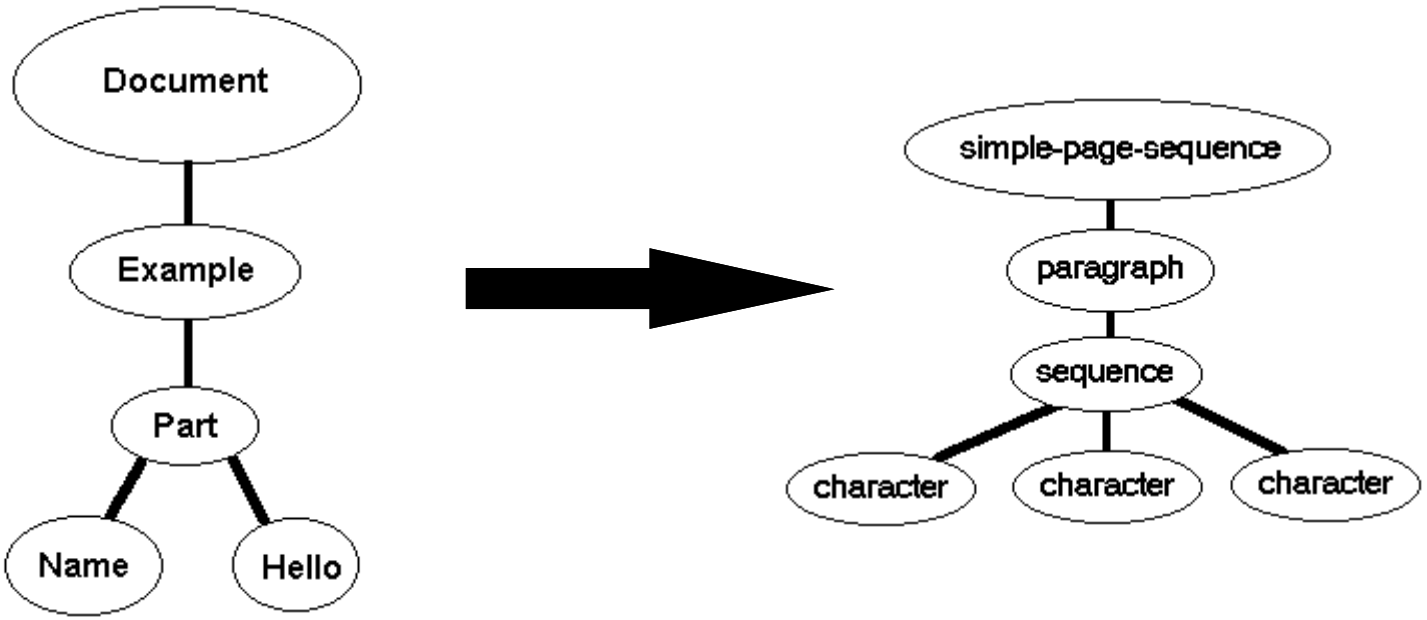
Scheme

- A small programming language in the Lisp family, standardized in R5RS
- Very cleanly designed, with a functional bent, but allows for several different programming styles
- Too limited in standardized tools (not features) for large-scale development, although many implementations provide these things as incompatible extensions
- Much used as an embedded language

DSSSL: Basic workings

- DSSSL stylesheets contain constant definitions, function definitions and rules
- Rules consist of a selector (defines which events it applies to) and an action part
- Typical actions are:
  - create formatting objects
  - create SGML/XML output

DSSSL: Grove to flow objects



DSSSL: A simple example

```
(element document
  (make simple-page-sequence)

(element part
  (make paragraph))

(element emph
  (make sequence
        font-posture: ‘italic))
```

DSSSL: Current status

- The standard was finished in 1996
- Two main implementations exist:
  - Jade: a DSSSL engine by James Clark
  - HyBrick: a browser produced by Fujitsu
- More powerful than XSL
- Fewer implementations, less tutorials
- Less geared toward web use

Building your own structure

- Constantly thinking in terms of elements and attributes has several disadvantages:
  - it’s awkward (sub-optimal level of abstraction)
  - it often means having to repeat work if you use your data for more than one thing
  - it means code depends on the exact shape of your markup, making you vulnerable to changes

DSSSL problems

- Documentation is sparse, especially on tree navigation
- Some tasks are made awkward by the lack of normal assignment
- A selector language like those of XSL and CSS would have been nice
- Not everybody knows Scheme

DSSSL advantages

- Jade is good and blazingly fast
- Full programming, can process substrings
- It’s here now and complete
- Good support for paper-based formats
- Full-featured and very general formatting model

Building your own structure

- A better approach can often be building an application-specific data structure to hold your data
- This is typically something you want to do on top of an event-based interface
- In object-oriented languages the most natural way to do this is to build an object structure

Using the structure

- To generate files:
  - by having ‘dumping’ methods in the classes
  - by using iterators and visitors
- Other ways of navigating the structure are also possible
- In some languages the structure can also be serialized automatically (speed benefits are usually small)

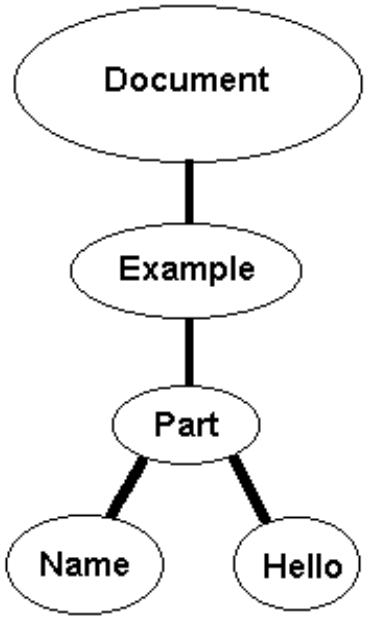
Demo

- Show
  - bookmark.py source
  - demo in interpreter
  - run xbel_parse.py

Conclusion

- Event-based processing is
  - low-overhead
  - low-level
  - often convenient
  - standardized through SAX and DSSSL
  - sometimes awkward
  - useful for building your own data structures

Tree-based processing



Tree-based processing

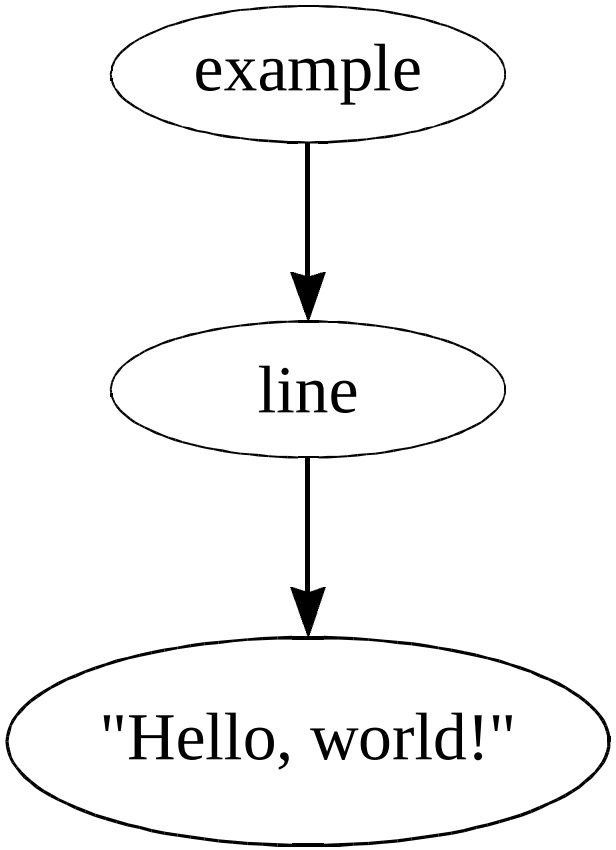
- …where the document is parsed into a tree structure, and processing is done by traversing the tree
- Usually built on top of an event-based layer
- May be unpractical for very large documents, unless the processor is very smart (some are)

Tree-based vs. event-based

- Tree-based:
  - a tree is built first, then your application gets a reference to it and starts working
- Event-based:
  - you specify actions that are executed on specific events
- Bottom line:
  - if main loop in your code and a tree is available, it’s tree-based
  - if main loop in system code, and tree availabe, it’s not

Building a tree from bytes

```
<example>
<line>&quo;Hello,
  world!&quo;</line>
</example>
```



Alternatives

- The DOM (Document Object Model)
- Groves
- Ace
- Balise

DOM

- A language-independent API defined by the W3C for tree-based processing
- Level 1: Deals with all logical aspects of documents, with special handling of HTML
- Level 2: Stylesheets, DTD, filters/iterators, ranges and namespaces (not yet finished)
- Defined in IDL, can be mapped automatically to most languages

Intended uses

- In browsers:
  - dynamic documents (with tweakable styles)
  - information extraction (for use in applets and web scripting)
- In editors:
  - as a data model
- Server-side:
  - for various kinds of processing

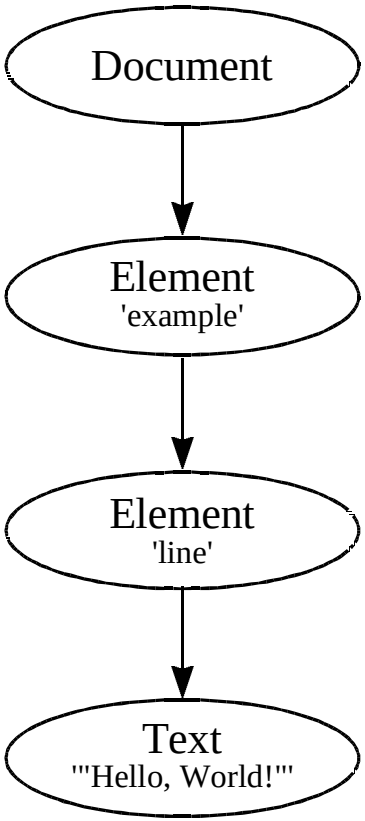
DOM usage

- Creating the tree:
  - with a parser which builds from a document
  - by calling ‘create___’ and ‘insert___’ methods
- Using it:
  - to extract data
  - modify the document
  - locate specific parts (possibly using XPointer, XQL or XSL patterns)

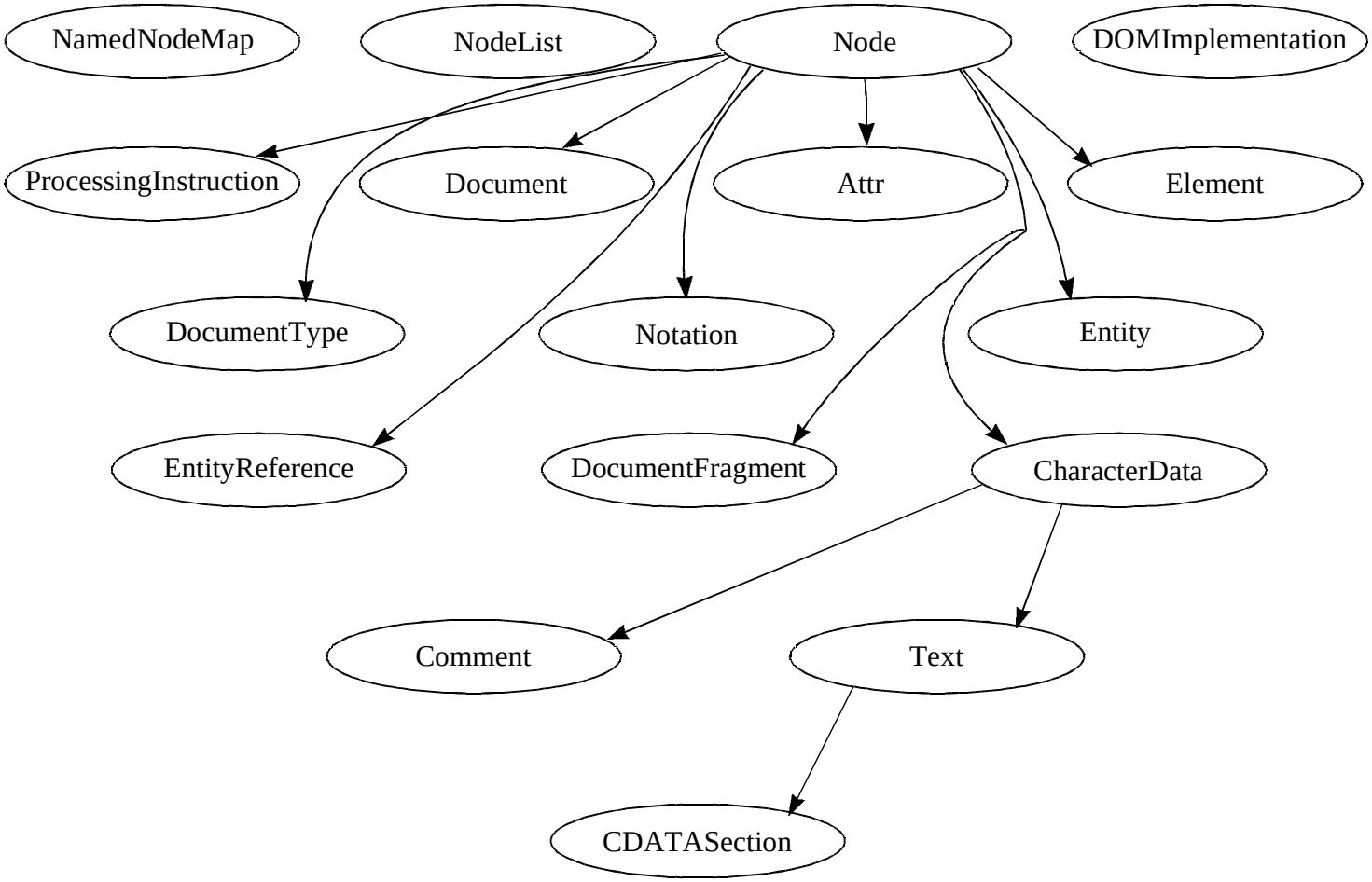
DOM implementations

- Java 9
- Python 2
- Delphi 1
- Perl 1
- Smalltalk 1
- tcl 1
- Common Lisp 1
- C++ 0.2

A DOM document



The DOM classes



Structure

- Really consists of two APIs that duplicate the same functionality:
  - one based on Nodes and NodeLists, where everything is generic
  - one based on the detailed classes, with more specialized attributes and methods
- The latter is defined because it is easier to understand and work with

Document

```
interface Document : Node {
  readonly attribute DocumentType       doctype;
  readonly attribute DOMImplementation implementation;
  readonly attribute Element          documentElement;

  //create___ methods
};
```

Loading a DOM tree

```
from xml.dom import sax_builder
from xml.sax import saxexts

builder=sax_builder.SaxBuilder()
parser=saxexts.make_parser()
parser.setDocumentHandler(builder)
parser.parse(url)
# builder.document now holds the document
```

DOMImplementation

```
interface DOMImplementation {
  boolean hasFeature(in DOMString feature,
                     in DOMString version);
};
```

- Features
  - HTML
  - XML

Node highlights

- Attributes:
  - nodeName
  - nodeValue?
  - nodeType
  - ownerDocument
  - parentNode
- Methods:
  - cloneNode(deep)
  - various tree manipulation methods

Element highlights

- Attributes: tagName, childNodes, attributes
- Methods:
  - getAttribute, setAttribute, removeAttribute
  - insertBefore, replaceChild, removeChild, appendChild, hasChildNodes

Attr

- Represents attributes on elements
- Attributes:
  - name
  - specified (a boolean)
  - value

Text

- Attributes: data, length
- Methods:
  - substringData
  - appendData
  - insertData
  - deleteData
  - replaceData

A useful trick

- If the document contains entity or character references (or comments/PIs) in element content, text nodes may be fragmented
- The DOM offers a convenience method ‘normalize’ on elements, which can normalize the children of the element
- ‘normalize’ is recursive

Some pitfalls

- Not all DOMs will know about entity boundaries, CDATA sections etc
- So normalize will behave differently with different parsers
- The best solution is perhaps to develop your own
- If it’s SAX-based parser filters can be used

A very useful method

- Element.getElementsByTagName
- Returns the nodes in the sub tree with the specified name (preorder)
- * returns all nodes
- Very useful to avoid sequence dependencies

Demo

- Demonstrate dom_load and arch2.xml

Demo

- Demonstrate some examples:
  - dom_create.py
  - dom_xbel.py

DOM Level 2

- Interfaces for stylesheets (CSS only, so far)
- Events: HTML 4.0 ones + mutation events
- Iterators: allow for iteration over subsets of nodes (in depth-first sequence) in the tree
- Filters: can be used to filter iterators
- Ranges: operations on a document range
- Namespaces: no information yet

Further levels

- Functionality for:
  - DTDs and schemas
  - Validation
  - Concurrent access
  - Access control

Groves

- A formalism for defining data models
- Has been used to define data models for SGML and HyTime
- Consists of nodes with associated properties
- Property sets define modules and node classes (of which nodes are instances)
- Node properties are typed and constrained

The uses of groves

- Groves can be used to define data models for practically anything
- These data models easily translate into APIs for working with the data
- The SGML property set can be used for working with XML documents as well

Grove implementations

- GroveMinder
- Jade (for DSSSL tree navigation)
- PyGrove

SGML node classes

• SgmlDocument	The document
• Element	Element instances
• AttributeAssignment	Attribute instances
• CharData	Textual data
• Pi	Processing instrs.
• Comment	Comments
• ElementType	Element type

SgmlDocument

- Some of the properties:
  - GoverningDoctype: the DTD
  - DocumentElement: the root element
  - Elements: list of elements with IDs
- In the two first cases, the value is another node
- In the third it’s a named node list

Element

- Some properties:
  - Gi: Element type name (string)
  - Id: The element ID, if any (string)
  - Attributes: The attributes (named node list)
  - Content: Element content (node list)
  - ElementType: Element type node

An example implementation

- Paul Prescod's PyGrove, which uses SP to build the grove
- Simple API:
  - nodes are Python objects, with properties as attributes
  - classes are Python classes
  - node lists are Python lists
  - named node lists are dictionaries (hashes)

Demo

- Run Pauls PyGrove with the browser, just to show what this looks like

Declarative processing

(do-what-i-mean)

Declarative processing

- You specify what you want, your processor delivers it
- Very high-level, not as flexible as Turing-complete solutions
- Usually less efficient than event-based solutions, also usually tree-based
- Solutions often large or incomplete

Comparison with others

- No programming
- Processing control is done by describing the desired result, not how to get there
- XSL strains the definition somewhat, but is at heart declarative

Alternatives

- XSL (eXtensible Style Language)
- Architectural forms
- PatML
- xtr2any

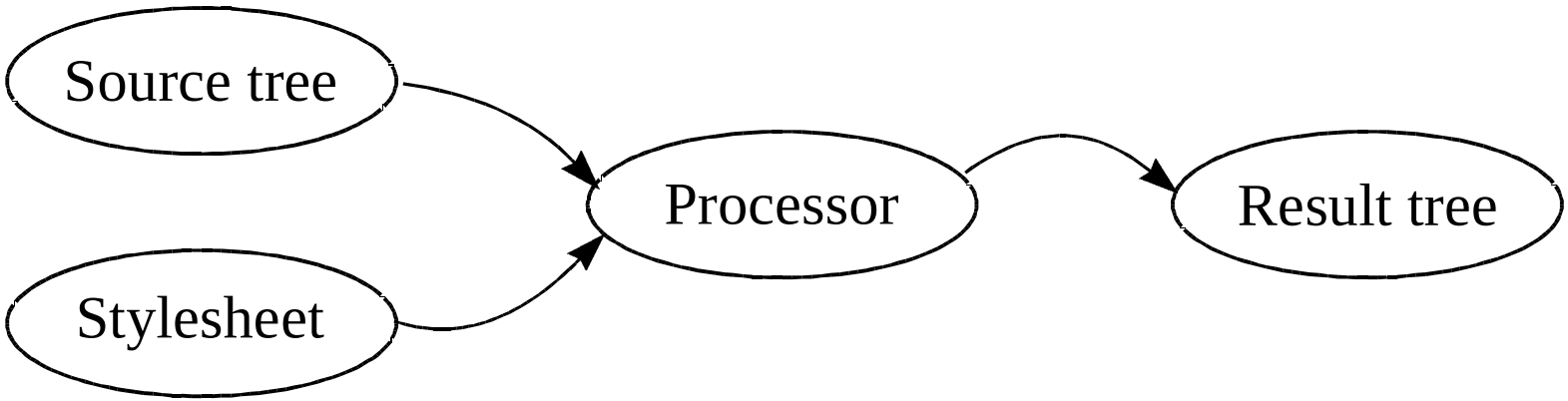
eXtensible Style Language

- The W3C style language for XML
- Uses a mostly XML-based syntax with some ‘extensions’
- Declarative: you specify what you want, not how to get there
- Several Java implementations exist, as does one Python implementation
- Supported by MSIE 5.0

A warning

- Please note that these slides were written when the 19981216 WD was current, and so are no longer in sync with the current working draft...

The XSL model



XSL: How it works

- Two parts:
  - the transformation language:
    - used to transform from XML to some result format
    - uses selectors and actions like DSSSL
    - written in XML
  - the formatting language
    - an XML vocabulary with formatting semantics
    - intended to be used to create screen layout and results in presentational formats

The transformation language

- Consists of template rules (plus plus)
- Each template has a pattern that is matched against the source tree and a template which generates a part of the result tree
- Both XSL and result tree pieces are XML, namespaces are used to tell them apart
- A pattern syntax is embedded in attributes

<xsl:stylesheet>

- The root element of XSL stylesheets
- Specifies the result namespace
- Example:

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/TR/WD-xsl"
   xmlns:html="http://www.w3.org/TR/REC-html40"
   result-ns="html">
   ...
</xsl:stylesheet>
```

Template rules

```
<xsl:template match="…">
  …result elements here…
</xsl:template>
```

- xsl:apply-templates indicates where to insert results from children

XSL: An example rule

```
<xsl:template match="document">
  <fo:basic-page-sequence>
    <xsl:apply-templates/>
  </fo:basic-page-sequence>
</xsl:template>
```

XSL: Equivalent HTML example

```
<xsl:template match="document">
  <html:html>
  <html:title>Demo</html:title>
  <html:body>
    <xsl:apply-templates/>
  </html:body>
  </html:html>
</xsl:template>
```

Patterns

- XSL patterns serve a dual role:
  - they are used for matching, so that templates can select which nodes to work on
  - they are used for selection, relative to a current node
- This last role makes it possible to use patterns for generating values and as tests in conditional statements

A basic pattern tutorial

- ‘`foo`’ matches all elements of the foo type
- ‘`foo | bar`’ matches all foo and bar elements
- ‘`foo/bar`’ matches all bars that have foo parents
- ‘`foo//bar`’ matches all bars that have foo ancestors
- ‘`@baz`’ matches all baz attributes
- It is also possible to match comments, PIs and plain text

Select patterns

- ‘`.`’ selects the current node
- ‘`bar`’ selects all bar children of the current node
- ‘`./bar`’ does the same thing
- ‘`.//bar`’ selects all bar descendants
- ‘`.[@baz]`’ matches the baz attribute of the current node
- It is also possible to select comments, PIs and plain text

Tests

- Patterns can contain tests within []s
- Tests can contain:
  - select patterns (true if they select something)
  - first-of-any(), first-of-type()
  - last-of-any(), last-of-type()
  - not(...test...)
  - and/or
- Test follow an expression and refine it

Demo

- Show a simple demo (make it on the fly!)
- Show xbel.xsl

Conditional inclusion

```
<xsl:template match="p">
  <fo:block>
    <xsl:if test='.[@class="warning"]'>
       Warning:
    </xsl:if>
  </fo:block>

  <xsl:apply-templates/>
</xsl:template>
```

More conditionals

```
<xsl:choose>
   <xsl:when test=‘.[@class=“warning”]’>
      Warning: <xsl:apply-templates/>
  </xsl:when>
  <xsl:when test='.[@class=“Danger”]'>
      DANGER: <xsl:apply-templates/>
  </xsl:when>
  <xsl:otherwise>
      <xsl:apply-templates/>
  </xsl:otherwise>
</xsl:choose>
```

Direct processing

- `for:each` can be used inside a template to repeat parts of it for each of the nodes in a select expression
- The `for:each` contains a template that is instantiated each time it matches
- This allows for easy iteration over list- and table-like structures

for-each example

```
<vendor>
  <name>…</name>
  …

  <product …>
  <product …>
  <product …>
  <product …>
</vendor>
```

for-each example use

```
<xsl:template match="vendor">
  …header stuff…

  <ul>
  <xsl:for-each select="product">
    <li>…name and description…
  </xsl:for-each>
  </ul>
</xsl:template>
```

Generating attributes

- Three ways:
  - string expressions in attribute values of literal result elements
  - using xsl:attribute
  - xsl:attribute-set

Generating attributes 1

- The easiest way of generating an attribute value is often by using string expressions
- These are simply placed inside an attribute value in a template rule and surrounded with {}

Generating attributes 2

- It’s also possible to use xsl:attribute to create attributes, like so:

```
<html:a>
<xsl:attribute name=“href”>…value…
</xsl:attribute>
…link text…
</html:a>
```

Generating attributes 3

- xsl:attribute-set “defines a named set of attributes” which can later be instantiated
- Given

```
<xsl:attribute-set name=“td-attrs”>
  <xsl:attribute name=“align”>left</…>
  <xsl:attribute name=“valign”>top</…>
</xsl:attribute-set>
```

- …

Generating attributes 3b

- …you can do:

```
<html:td>
  <xsl:use name="td-attrs"/>
  …element content…
</html:td>
```

Counters

- `number` can be used, using the `count` and `multi` attributes to control counting
- It can also be done explicitly with
  - `counter/counters`
  - `counter-increment`
  - `counter-reset`
  - `counter-scope`
- Several kinds of numbering are available

Sorting

- `sort` elements can be inserted as children of `apply-templates` to specify what to sort on
- `sort` elements use select patterns to select the values to sort on
- Several kinds of lexicographical sorting are available, as is numerical sorting

Node copy

- `copy` can be used to produce a copy of the node in the source tree that triggered at template instantiation

Generation

- `value-of` lets you insert the value of a string expression in the result tree
- String expressions use
  - select expressions (value of first node selected)
  - name expressions (name of first node selected)
  - constant references
  - macro argument references

Macros

- It’s possible to define template pieces in one place and then refer to them from many different templates
- This is done via:
  - `macro`
  - `invoke-macro`

Processing modes

- Allow parts of the document to be processed more than once
- Useful for different views of the same content
  - Condensed views: tables of contents, indexes
  - Differently sorted views

Using modes

- Templates have a mode attribute which can be used to place a template in a mode
- Apply-templates has a mode attribute which can be used to specify which processing mode to use
- Default rules are used if the mode does not have suitable rules

Demo

- Show xbel2.xsl
- Show rfc.xsl

Mode pitfalls

- If there are intermediate elements between the applying element and the applied element, the mode will be lost
- This happens because the default rules kick in
- Using select or an empty rule can solve the problem

Example

If there are no rules for b, this will go wrong:

```
<a><b><c/></b></a>

<xsl:template match="a">
  <xsl:apply-templates mode="demo"/></…>

<xsl:template match="c" mode="demo">
  …lots of useful stuff…
</xsl:template>
```

Modular stylesheets

- `import` can be used to load in external stylesheets
- `include` can be used to include external files at any point in the style sheet

Idioms

- To ignore an element:
  - make a matching template which is empty
  - you can use | between the element type names
- To get the contents of a sub-element:
  - use xsl:apply-templates with select
  - use xsl:value-of and select the element
  - ditto for attributes

Demo

- Show xbel3.xsl (improved per idioms)

XSL flow objects

- An XML language for describing laid-out documents
- Similar to the flow objects of DSSSL
- Intended to be interpreted directly by a presentational program or converted to presentational formats
- Only one implementation so far: FOP

XSL flow objects

- Support for:
  - paragraphs (blocks)
  - links
  - graphics
  - rules
  - lists
  - page numbering

Architectural forms

- Intended as a way of subtyping element types, but is in fact a declarative processing mechanism
- Uses a set of processing instructions and special attributes to specify what processing is wanted
- Completely declarative and very high-level

Architectural forms

- Map documents from one DTD to another
- Processing instructions declare the forms
- Attributes on elements specify the mapping
- With an AF engine between your parser and your application the mapping becomes transparent

How it works

- During processing a new transient (or virtual) document is created
- Software can now operate on this virtual document as if it were a normal document
- The virtual document (or architectural document) can also be validated

Architectural forms

- Standardized in an appendix to HyTime
- Implemented in:
  - SP, James Clarks SGML parser
  - XAF, a SAX parser filter in Java
  - xmlarch, a SAX parser filter in Python
- Used heavily in HyTime, Topic Navigation Maps and many advanced SGML apps

Example document

```
<?IS10744:arch name="html"?>

<doc>
<head html=“title”>Sample document</head>
<txt html=“p”>
Sample sample sample. Sample. Blah.
</txt>
</doc>
```

Mapped document

```
<html>
<title>Sample document</title>
<p>
Sample sample sample. Sample. Blah.
</p>
</html>
```

Architectural forms

- Functionality:
  - More than one form per document is possible
  - Elements and attributes can be suppressed
  - Attributes can be mapped to content and vice versa
  - The mapped document can be validated in terms of the architectural DTD

Common usage

- To define a common subset DTD of several different variant DTDs
- To identify particular kinds of constructs inside documents, across DTDs, such as:
  - links
  - tables
  - elements with processing semantics
- Usually architectural attributes are #FIXED in the DTD

Demo

- Show arch.xml

Weaknesses

- Declaration syntax a bit awkward:
  - mixed with normal DTD declarations
- No globally unique element identifiers
- Mapping abilities are a bit weak
- Mappings can rarely be created between DTDs that were not designed for it

End of fixed core

- Alternatives:
  - dealing with character encodings
  - a real-world processing application
  - HTML part of the DOM (brief)
  - SAX 2 (brief)
  - an example of DTD processing (very brief)
  - audience suggestions

Character sets

Bonus slides

The basics

- Documents are stored as strings of bits
- Character sets and encodings are used to enable us to store text in terms of bits
- A character set is just that, a set of characters and a code point (number) for each character
- This in itself is not enough

Character sets and encodings

- A character encoding describes how a sequence of character numbers is turned into a string of bits
- For most character encodings this is just done by representing the numbers in the straightforward way
- There are some important exceptions, though

Important character sets

Charset	Chars	Bits	Encoding
US-ASCII	128	7/8	Trivial
EBCDIC (several)	256	8	Trivial
iso-8859-x	191	8	Trivial
ISCII-xx	176	8	Trivial
JIS X-0208-19xx	6879	Variable	Several
Unicode	47400	Variable	Several
ISO 10646	47400	Variable	Several

Unicode/ISO 10646 encodings

Encoding	Features
utf-7	7-bit encoding
utf-8	8-bits, US-ASCII below 128
utf-16	16-bits, non-trivial
UCS-2	16 bits, trivial, lower 65536
UCS-4	32 bits, trivial

XML and character sets

- The standard uses the Unicode characters
- Character references (&#???;) refer to Unicode code points
- Documents can use any encoding, but utf-8 and utf-16 are the defaults
- Other encodings must be declared in the XML (or text) declaration of the entity

XML and transport

- When transferred over the network, the protocol used may override the declaration
- For the MIME content-type text/xml, the default is US-ASCII
- For application/xml it is utf-8/utf-16

Conclusion

- Use whichever encoding you want
- Be sure to declare your encoding to avoid problems with network transfers
- If you want characters not in Unicode you have a problem

An example processing application

Free XML tools

XMLtools

- A list of all the free XML tools I know of
- Started out as a simple hand-maintained list
- Was then expanded to list all the tools for the CD-ROM of ‘The XML Handbook’
- At this point it became an XML application with descriptions and other information

Demo

- Just show the pages
- Show the search interface, but don’t actually do a search

XMLtools architecture

- Maintained as a single 125k XML document
- Published into a set of static web pages using Python scripts built on PyDOM
- Also published into a search index that is accessed through Python CGI scripts

Demo

- Show the XML source

The different processes

- mkindex.py: Creates the search index
- report.py: Creates the main page
- prod_by_*.py: Creates the indexes
- updates.py: Creates the What’s new section

Integration of the processes

- Uses a home-made GUI-based publishing system developed in Java
- This automatically runs the scripts and uploads the output using FTP
- Unfortunate architecture:
  - requires a GUI
  - requires separate processes, not a single one

The processing structure

- A separate module swlib.py uses the DOM to create an application-specific structure
- The various scripts access this using specific interfaces and extract the information they need
- *Very* much easier than working directly on the DOM, because of the multiple-use

The search scripts

- Index maker: Builds Python hashtables and lists and dumps them using the marshal module
- The search scripts then load these data (which is very fast) and search in them (which is pretty fast)

Demo

- Go through source

SAX 2

A quick look at the future

(19.Apr.99)

Basics

- Defined as 100% backwards compatible
- Defined in a separate Java package
- Will also be translated to Python immediately
- Extensible for third parties
- May perhaps not deal with filters

New Parser2 interface

- Extends Parser
- Methods:
  - get(id)
  - set(id,obj)
  - setHandler(id,handler)
  - setFeature(id,state)

IDs

- Use a URI scheme, just like namespaces
- No requirement that the ID point to anything
- Various people wanted something similar to Java package names, but have not won yet

Features

- Validation
- External general entity resolution
- External parameter entity resolution
- Split characters events or not
- Namespace processing on/off
- Provide Locator (or don’t)

Properties

- Namespace name separator
- Element stack (unresolved)
- Literal string associated with current event (to get whitespace in tags etc)
- DOM node for current event (for DOM traversers that fire SAX events)

New AttributeList

- Parsers can now use a subclass of AttributeList
- Provides information about entity references in attribute values
- Unlikely to be needed or wanted by many, but is required for full XML 1.0 compliance

DTD handler

- Both event-based and object-based proposals
- Both seem to include all logical information
- No clear winner as of yet

LexicalHandler

- A separate handler
- Has:
  - a comment event
  - CDATA start/end events
  - entity reference start/end events
  - DTD start/end events
  - the ability to discern internal/external subset

HTML DOM

What it contains

- Basically:
  - HTMLDocument
  - HTMLElement
  - Specializations for elements with more attributes
  - HTMLCollection

HTMLDocument

- Extends Document with:
  - title, referrer, domain, URL, body, images, applets, links, forms, anchors, cookies
  - getElementById(id)
  - getElementsByName(name)

HTMLElement

- Extends Element with string attributes for the HTML global attributes:
  - id
  - title
  - lang
  - dir
  - className

Extended element interfaces

- Contain an attribute for each HTML attribute of the corresponding element type
- Most are strings, but some are boolean or contain direct references to specific elements

dtddoc.py

A DTD documentation generator

dtddoc.py

- Produces an HTML document with an entry for each element type defined in the DTD
- Uses the DTD parser that xmlproc uses to provide validation services
- This parser is 100% general, as is the data structure it normally builds
- dtddoc.py uses both

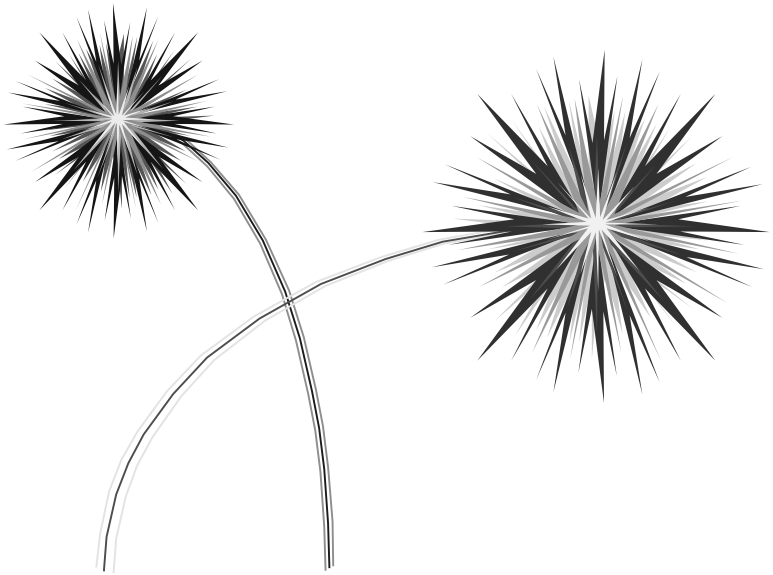
Demo

- Show how it works
- Show the dtddoc.py source
- Show the APIs and implementations

What’s missing?

- An index of elements and attributes
- Notations, entities, parameter entities
- Information about parameter entity structure?
- Textual documentation:
  - some schema languages has this
  - can also be provided with an external document

Conclusion



Declarative processing

- Understandable to non-programmers
- The easiest way to do it, if there is a framework designed for what you want
- Not as flexible as full programming
- Often rather large systems with much to learn

Event-based processing

- The lowest-level solution
- The least resource-intensive solution
- Often the easiest solution for simple things
- Awkward for more complex things
- Can be used to build application-specific data structures

Tree-based processing

- Often awkward for simple processing
- Usually memory-intensive
- Best suited for tasks
  - where parts of the document need to be processed several times
  - there are dependencies between different parts of the document
  - more than one pass is needed

Goodbye!

- That’s it for now.
- The slides from this presentation are also available at:

`http://birk105.studby.uio.no/download/artikler/processing.pdf`

- ZIP file with demo files will appear at:

  `http://birk105.studby.uio.no/download/`