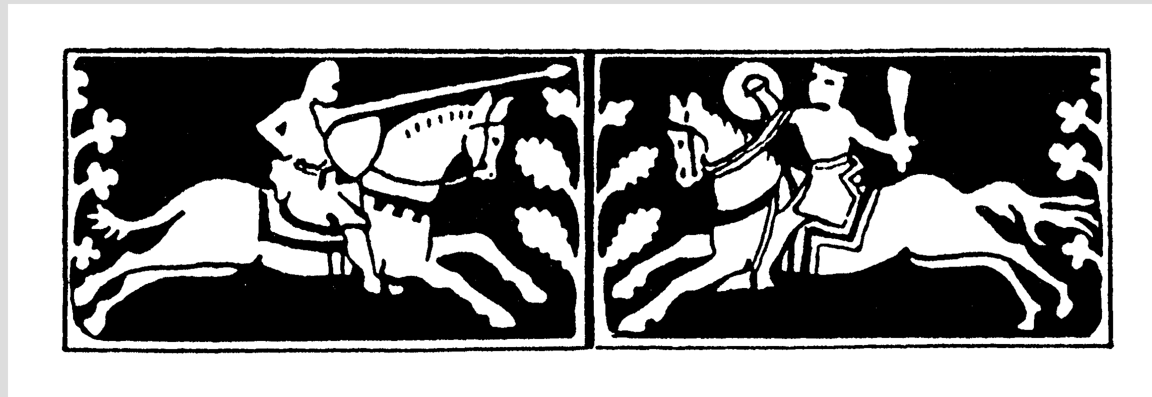
An OpenDocument Developers Kit (ODDK)

What do we need to make
Application Developers Productive with ODF?



Rob Weir
IBM
robert_weir@us.ibm.com
http://www.robweir.com/blog

OpenDocument Day
KDE aKademy 2006
Dublin

The potential

- ODF – a platform and application neutral office file format
- Document data is no longer trapped in proprietary black box binaries

- This can lead to a “golden age” of document processing, both client and server side, with much innovation

- “We have it in our power to create the world over again” -- Thomas Paine

More than just editors

(20 Prototypical App Dev Scenarios)

1. Interactive creation in an a heavy-weight client application
2. Interactive creation in a light-weight web-based application
3. Collaborative (multi-author) editing
4. Automatic creation in response to a database query (report generation)
5. Indexing/scanning of document for search

20 Prototypical App Dev Scenarios

6. Scanning by anti-virus
7. Other types of scanning, perhaps for regulatory compliance, legal or forensic purposes
8. Validation of document, to specifications, house style guidelines, accessibility best practices, etc.
9. Read-only display of document on machine without the full editor (viewer)
10. Conversion of document from one editable format to another

20 Prototypical App Dev Scenarios

11. Conversion of document into a presentation format, such as PDF, PS, print or fax
12. Rendering of document via other modes such as sound or video (DAISY Talking Book)
13. Reduction/simplification of document to render on a sub-desktop device such as cell phone or PDA.
14. Import of data from an office document into a non-office application, i.e., import of spreadsheet data into statistical analysis software.
15. Export of data from a non-office application into an office format, such as an export of a spreadsheet from a personal finance application.

20 Prototypical App Dev Scenarios

16. Application which takes an existing document and outputs a modified version of that presentation, e.g., fills out a template, translates the language, etc.
17. Software which adds or verifies digital signatures on a document in order to control access (DRM)
18. Software which uses documents in part of a workflow, but treats the document as a black box, or perhaps is aware of only basic metadata.
19. Software which treats documents as part of a workflow, but is able to introspect the document and make decisions based on the content.
20. Software which packs/unpacks a document into relational database form.

The Problem

- 706 page ODF Specification
- No objections to it as a specification – it is what it needs to be
- Written from the perspective of word processor implementors
- Too much to ask the average app developer to master

Analogy with XML --
Who actually reads this stuff?

3.3.3 Attribute-Value Normalization

Before the value of an attribute is passed to the application or checked for validity, the XML processor MUST normalize the attribute value by applying the al
other method such that the value passed to the application is the same as that produced by the algorithm.

1. All line breaks MUST have been normalized on input to #xA as described in **2.11 End-of-Line Handling**, so the rest of this algorithm operates on text
2. Begin with a normalized value consisting of the empty string.
3. For each character, entity reference, or character reference in the unnormalized attribute value, beginning with the first and continuing to the last, do th
   - For a character reference, append the referenced character to the normalized value.
   - For an entity reference, recursively apply step 3 of this algorithm to the replacement text of the entity.
   - For a white space character (#x20, #xD, #xA, #x9), append a space character (#x20) to the normalized value.
   - For another character, append the character to the normalized value.

If the attribute type is not CDATA, then the XML processor MUST further process the normalized attribute value by discarding any leading and trailing space
replacing sequences of space (#x20) characters by a single space (#x20) character.

Note that if the unnormalized attribute value contains a character reference to a white space character other than space (#x20), the normalized value contai
(#xD, #xA or #x9). This contrasts with the case where the unnormalized value contains a white space character (not a reference), which is replaced with a s
normalized value and also contrasts with the case where the unnormalized value contains an entity reference whose replacement text contains a white spac
processed, the white space character is replaced with a space character (#x20) in the normalized value.

All attributes for which no declaration has been read SHOULD be treated by a non-validating processor as if declared **CDATA**.

What is really used is SAX

endElement

```
public void endElement(java.lang.String uri,
                       java.lang.String localName,
                       java.lang.String qName)
                throws SAXException
```

Receive notification of the end of an element.

The SAX parser will invoke this method at the end of every element in the XML document; there will be a corresponding `startElement` e

For information on the names, see startElement.

Parameters:

`uri` - the Namespace URI, or the empty string if the element has no Namespace URI or if Namespace processing is not being perfo

`localName` - the local name (without prefix), or the empty string if Namespace processing is not being performed

`qName` - the qualified XML name (with prefix), or the empty string if qualified names are not available

Throws:

`SAXException` - any SAX exception, possibly wrapping another exception

And DOM

```
public Node getParentNode();

public NodeList getChildNodes();

public Node getFirstChild();

public Node getLastChild();

public Node getPreviousSibling();

public Node getNextSibling();

public NamedNodeMap getAttributes();

public Document getOwnerDocument();

public Node insertBefore(Node newChild,
                         Node refChild)
                         throws DOMException;

public Node replaceChild(Node newChild,
                         Node oldChild)
                         throws DOMException;
```

Proposal

We need an ODF API that exposes a higher level abstraction of ODF to application developers, so they can quickly become productive with ODF processing without having to master a 700 page specification

“Create a loan amortization spreadsheet in 30 lines of code”

Desirables

- Open source
- A convergent effort – bring together the projects that are already working in this area
- Wide range of language bindings, Java, Python, Ruby, C++, etc.
- Consider the API itself for standardization

This becomes the preferred way of working with ODF, the layer that the innovation builds upon

Some design ideas

- Useful to think of the toolkit in three classes:
  - The document representation – ODF DOM
    - Represents the state of the document, with get/set methods for manipulation. sheet.setCell("A1","hello")
  - A Parser class that takes an input stream and produces an ODF DOM object from it
  - A Serializer class that takes an ODF DOM object and writes it to an output stream

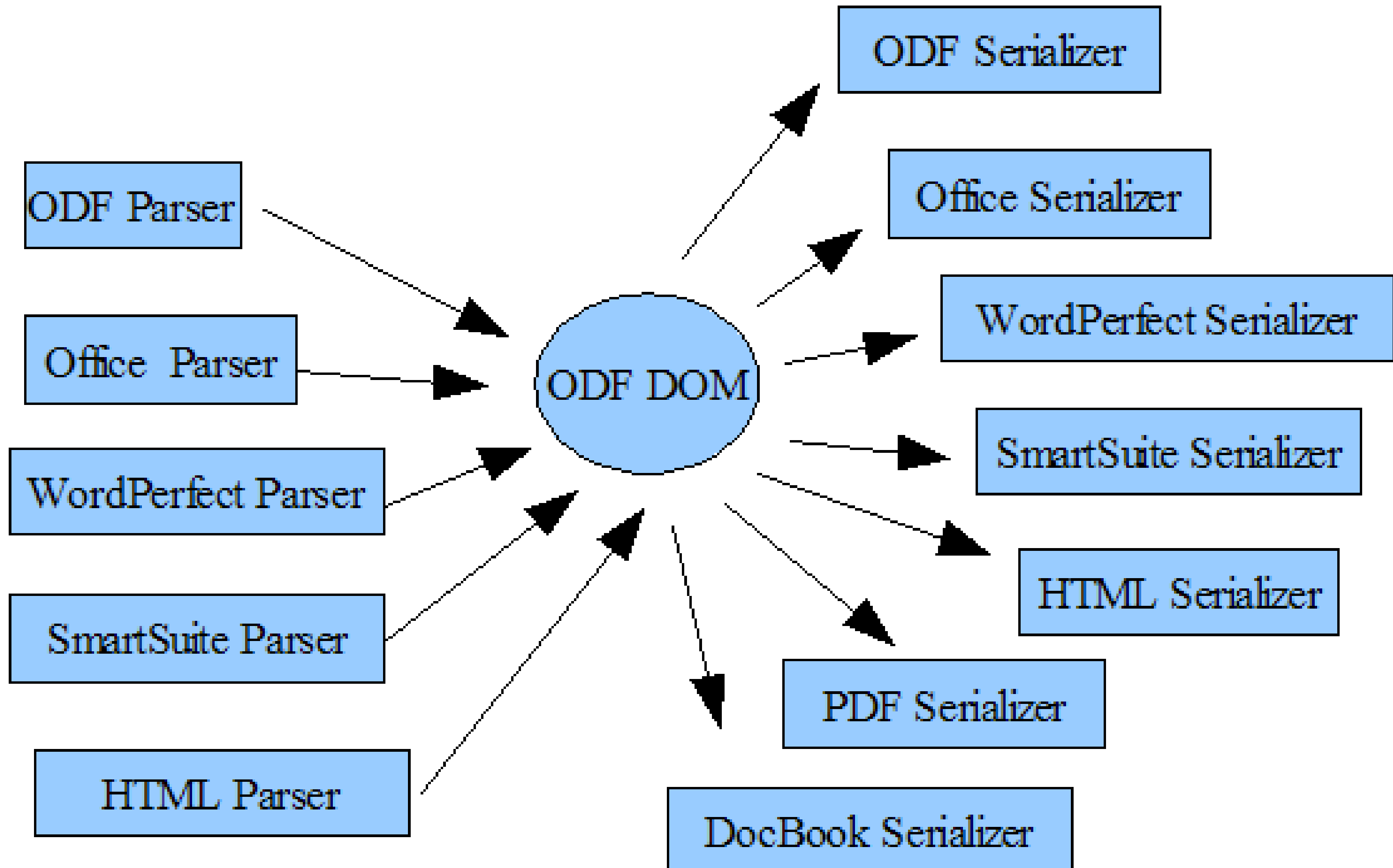
Modes of use

- Report generation
  - Create empty ODF DOM object, query a database, set data into the ODF object, then create Serializer to write it out to ODF document.
- Search engines
  - Create Parser, pass in stream to ODF document, create ODF DOM object, call methods to query document contents
- Mail Merge
  - Create parser, pass in stream, get ODF DOM object, find and replace content in the DOM, and then create a Serializer to write it out again

Key insight

- Factored this way, an additional opportunity emerges:
  - Is ODF the only source/destination format of the Parsers and Serializers? So long as they produce/consume ODF, who cares what the underlying data stream is?
  - Why not have an ExcelParser that reads an Excel document and creates an ODF DOM from it?
  - Why not have an PDFSeralizer that takes an ODF DOM document and renders it as PDF?

Hub and Spokes Model



What you end up with

- A family of Parsers and Serializers which can be treated polymorphically (pluggable), using a common ODF DOM representation
- Could become the preferred way to manipulate all office-like documents, not just ODF
- Makes choice of file format irrelevant from the perspective of the application developer
  - Creating an app that supports ODF & Office is the same cost as creating one that supports only Office
  - Reduces switching costs == greater ODF adoption

Things that we can build on

- OpenOffice.org UNO API's
- Apache POI (http://jakarta.apache.org/poi/)
  - Java code for reading/writing MS Office binary formats
- Apache FOP can render to PDF and SVG
- OpenDocument Fellowship has
  - ODT to HTML
  - DocBook to ODT
- J. David Eisenberg has some code in XSLT, Java and Ruby (http://books.evc-cit.info/odf_utils/)
- Probably many others

The end

A fuller exposition of this topic can be read in my essay “Proposal for an OpenDocument Developers Kit (ODDK)” posted on XML.org at http://opendocument.xml.org/node/154