AreaTreeIntermediateXml NewDesign

This page describes the new proposed design for FOP's intermediate format. The goals can be found on the parent page.

Basic ideas

- Try to put the processing load on the layout side as much as possible. On the rendering side you should only have basic painting functions (basically a simple, XML-based graphical metafile).
  - The format needs to be streamable, i.e. no DOM is necessary, every command can be processed after the other, minimal stack with state information.
  - Difficulty: this works only for FOP's native feature set (basically text and border painting plus coordinate system handling). The processing of images, fonts and other resources are heavily output format specific, so the shift of processing focus to the layout side has its limits (this has to be done in the rendering stage).
  - The intermediate format has to become much easier and faster to parse.
  - Benefit: the renderers may be become leaner and therefore easier to maintain and implement.
  - Difficulty: features needed in the future (mainly: tagged PDF) could become a little more difficult since a graphical metafile will not contain structure information per se. OTOH, the current area tree doesn't have enough structure information, either.
  - Difficulty: until now, supporting extensions was relatively easy as each area tree object could carry extension attributes. With a graphical metafile, individual area tree object cannot directly be identified in the stream anymore.
  - When going in this direction, one question is obvious: Why not take SVG (or Adobe Mars) as the intermediate format? It would most probably be much slower than an optimized, proprietary format. But what if we restricted ourselves to a subset and didn't use Batik for the rendering? Just a thought.

  - It is worth noting that the working draft SVG 1.2 specification would provide pagination for the layout with the <page/> and <pageSet/> elements (see http://www.w3.org/TR/2004/WD-SVG12-20041027/multipage.html) [2].

- It is important to note that the XML representation of the area tree is still very important. The new IF is no replacement for unit testing the layout engine because the area tree contains much more verbose information of the layout result.

Sketching out a new XML format

```
<document xmlns="http://xmlgraphics.apache.org/fop/metafile" xmlns:xlink="http://www.w3.org/1999/xlink">
  <header>
    <x:xmpmeta xmlns:x="adobe:ns:meta/">
      <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
        <rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
          <dc:title>New Intermediate Format Demo Document</dc:title>
        </rdf:Description>
      </rdf:RDF>
    </x:xmpmeta>
  </header>
  <page-sequence>
    <page index="1" name="1" width="595275" height="841889">
      <page-header>
        <ps:ps-setup-code>%FOPTestPSSetupCode: General setup code here!</ps:ps-setup-code>
      </page-header>
      <content>
        <viewport transform="translate(5000, 6000)" width="18000" height="10000">
          <font family="Helvetica" style="normal" weight="400" variant="normal" size="12000"
            color="black"/>
          <text x="1233" y="803" dx="0 0 20 0 0">Hello</text>
          <rect x="1233" y="1200" width="20000" height="20000" fill="yellow" stroke="none"/>
          <viewport transform="translate(1233, 1200)" width="20000" height="20000" clip-rect="0 0 20000 20000">
            <image xlink:href="myimage.svg" x="0" y="0" width="20000" height="20000"/>
          </viewport>
        </viewport>
        [..]
      </content>
    </page>
    <page...
  </page-sequence>
  <trailer>
    <bookmark-tree xmlns="http://xmlgraphics.apache.org/fop/intermediate/document-navigation">
      [PDF bookmarks, see below for details]
    </bookmark-tree>
  </trailer>
</document>
```

- viewport: pushes the graphics state on a stack and applies an optional transformation (SVG-style). This is mainly used for viewport/reference pairs.
- image: in case of an instream-foreign-object, its content would simply be put in the image element's content instead of using xlink:href.

Other needed elements

- color-profile element
- destination (for PDF)
- shape painting facility (for optimized, complex border painting, ex. in tables)

- infrastructure for tagged PDF [1]: Structure tree (in the document header) plus content containers (in the page content), word-break info (could be modelled in a similar way Adobe Mars does it for their XML-based format)

- Accessibility elements: language info and possibly other stuff
- Clearly defined extension points (places where foreign namespaces can be used, at least: document header & page header)
- maybe element-level metadata support

Old design (for reference)

http://people.apache.org/~jeremias/fop/renderer-design-old.png

New design

http://people.apache.org/~jeremias/fop/renderer-design-new.png

Intermediate format design (Java part)

IFDocumentHandler and IFPainter (working titles, better suggestions welcome!) are the central interfaces, like Renderer. There's one implementation for each output format that is useful in the context of the intermediate format (probably includes all current renderers except text. most important are: PostScript , AFP and PCL). Ideally, the interfaces are a direct equivalent to the possible SAX stream for the new IF format, i.e. it is possible to convert between the interfaces and the IF-NG SAX stream with no losses. The IFParser in the graphic above will convert the SAX stream to IFDocumentHandler and IFPainter calls and a special implementation used by IFRenderer would convert the calls to a SAX stream. That way, the IFRenderer could actually render to an IFPainter without the detour over XML.

In contrast to the Renderer interface, the new intermediate format uses two basic interfaces: IFDocumentHandler and IFPainter. IFDocumentHandler is responsible for setting up the output file and handling file/document-level responsibilities. IFPainter handles the actual "painting" of the individual pages. The split has advantages: The various implementation classes are smaller and more readable. Formats like the PCL renderer need to convert certain features to bitmaps (text, borders etc.) as they can't be painted as nicely in the native print language. In this case, FOP uses Java2D to create those bitmaps. Separating document-level code from page-level code makes code-reuse easier.

The interfaces are not fully designed, yet, (pending verification in real life) so the following is just to give an idea what it could look like (all methods will probably throw SAXException):

```
public interface IFDocumentHandler {

    void setUserAgent(FOUserAgent userAgent);
    void setResult(Result result) throws IFException;
    void setFontInfo(FontInfo fontInfo);
    IFDocumentHandlerConfigurator getConfigurator();
    boolean supportsPagesOutOfOrder();
    String getMimeType();

    void startDocument() throws IFException;
    void endDocument() throws IFException;
    void startDocumentHeader() throws IFException;
    void endDocumentHeader() throws IFException;
    void startDocumentTrailer() throws IFException;
    void endDocumentTrailer() throws IFException;
    void startPageSequence(String id) throws IFException;
    void endPageSequence() throws IFException;
    void startPage(int index, String name, Dimension size) throws IFException;
    void endPage() throws IFException;
    void startPageHeader() throws IFException;
    void endPageHeader() throws IFException;
    IFPainter startPageContent() throws IFException;
    void endPageContent() throws IFException;
    void startPageTrailer() throws IFException;
    void endPageTrailer() throws IFException;
    void handleExtensionObject(Object extension) throws IFException;

}
```

```
public interface IFPainter {

    void startViewport(AffineTransform transform, Dimension size, Rectangle clipRect) throws IFException;
    void startViewport(AffineTransform[] transforms, Dimension size, Rectangle clipRect) throws IFException;
    //For transform, Batik's org.apache.batik.parser.TransformListHandler/Parser can be used
    void endViewport() throws IFException;

    void startGroup(AffineTransform[] transforms) throws IFException;
    void startGroup(AffineTransform transform) throws IFException;
    void endGroup() throws IFException;

    void setFont(String family, String style, Integer weight, String variant, Integer size,
            Color color) throws IFException;
    //All of setFont()'s parameters can be null if no state change is necessary
    void drawText(int x, int y, int[] dx, int[] dy, String text) throws IFException;

    void clipRect(Rectangle rect) throws IFException;
    //TODO clipRect() shall be considered temporary until verified with SVG and PCL

    void fillRect(Rectangle rect, Paint fill) throws IFException;
    void drawBorderRect(Rectangle rect,
            BorderProps before, BorderProps after,
            BorderProps start, BorderProps end) throws IFException;
    void drawLine(Point start, Point end, int width, Color color, RuleStyle style)
            throws IFException;
    void drawImage(String uri, Rectangle rect, Map foreignAttributes) throws IFException;
    void drawImage(Document doc, Rectangle rect, Map foreignAttributes)
                throws IFException;

}

public class IFState {

    //all font traits
    //list of transforms since the last state safe (by startBox())
    //maybe the effective clip shape
}

//additional needed classes
public class IFSerializer implements IFPainter {

    public IFSerializer() {
    [..]

    public void setResult(Result result) throws IFException {
    [..]

    //convert IFDocumentHandler/IFPainter calls to XML (IF-NG)
}

public class IFParser {

    public void parse(Source src, IFDocumentHandler documentHandler, FOUserAgent userAgent)
            throws TransformerException {
    [..]
    }

    public ContentHandler getContentHandler(IFDocumentHandler documentHandler, FOUserAgent userAgent) {
    [..]

    //convert SAX stream calls to IFDocumentHandler/IFPainter calls

}
```

Note that IFDocumentHandler and IFPainter should be designed so it is easy to write some kind of filter (like FilteredOutputStream) where implementors can react to certain events like startPageContent() so they can add their own content calls (content enrichment) for things like barcodes, OMR marks, background images etc. This can be done in Java on the interface or in XML using XSLT.

Performance evaluation compared to previous approach

Performance is expected to be higher for the following reasons:

- We have fewer content elements which accounts for a leaner implementation and reduction to essential content. No potentially empty container structures like we have now.
- Currently, the IF is implemented over our area tree which contains a generic structure for traits. Maps present a certain amount over overhead by themselves which this approach avoids. Furthermore, the AreaTreeParser has to inspect various trait sets per area tree object which causes too many Map operations.
- The conversion from ContentHandler to IFPainter only requires one lookup (see AttributesImpl.getValue()) per parameter. The number of parameters is very small. Some parameters are complex Strings which need to be parsed themselves (for example startBox()'s transform parameter). The overhead here should be relatively low since these are not extremely frequent operations. setFont()'s parameters could be changed from String to an enumeration class which would cause a Map lookup, but again setFont() is not a very frequent operation and not every parameter will be set and therefore parsed on every call.

Border Painting

For border painting at least two approaches are possible. Either, the IF is extended by commands to do complex clipping and path painting operations, or we abstract border painting to the same level as in the area tree. The former is problematic since it increases the size of the IF files (and therefore parsing time) and not all output formats have the same capabilities as PDF or PostScript (ex. PCL needs to use bitmaps to paint nicely looking borders). That leaves the latter approach. It probably makes sense to reuse the BorderProps class to encapsulate the properties/traits of the individual borders of a box. So the proposed approach looks like this:

```
  <border-rect x="1233" y="1200" width="20000" height="20000"
    start="(solid,#800080,8000,collapse-inner)"
    end="(solid,#ffff00,6000,collapse-outer)"
    before="(solid,#808080,4000,collapse-outer)"
    after="(solid,#000080,8000,collapse-outer)"/>

  void drawBorderRect(Rectangle rect, BorderProps start, BorderPropds end, BorderProps before, BorderProps
after) throws IFException;
```

As a consequence there won't be an way to combine multiple border segments easily to reduce output file size and painting artifacts (usually not a problem). If that is something desirable, it should probably be done in IFRenderer. But since at the moment there's not enough time to pursue that and we don't really have a problem, this is ignored for now.

Text decoration painting which currently goes through the `drawBorderLine()` method in most renderers will have to be done differently, probably through `drawRect()`.

Extensions

The new intermediate format will need to make extensive use of extensions, mostly output format specific extensions like media selection for PCL or PostScript, or overlay functionality in AFP. The extensions will be produced in a namespace outside the normal intermediate format namespace. Features that are only implemented/supported by a small subset of output formats, ex. bookmarks and named destinations, shall be implemented as an extension in order to avoid cluttering the intermediate format itself for minimal cases.

Example: Bookmarks Extension

```
    [..]
    </page>
  </page-sequence>
  <bookmark-tree xmlns="http://xmlgraphics.apache.org/fop/intermediate/document-navigation">
    <bookmark starting-state="show" title="Chapter 1">
      <goto-xy page-index="0" x="20000" y="20000"/>
    </bookmark>
    <bookmark starting-state="hide" title="Chapter 2">
      <goto-xy page-index="1" x="20000" y="20000"/>
      <bookmark starting-state="show" title="Section 1">
        <goto-xy page-index="1" x="20000" y="51680"/>
      </bookmark>
      <bookmark starting-state="show" title="Section 2">
        <goto-xy page-index="1" x="20000" y="80480"/>
      </bookmark>
    </bookmark>
  </bookmark-tree>
</document>
```

Resource Management (Idea)

Some formats like PostScript and AFP require special processing to optimize resources (images, fonts etc.). The PostScript renderer currently supports an optional two-pass approach where the resources are only added in the second pass to the beginning of the PostScript file, i.e. after you know which resources are needed. The idea now is to enrich the IF renderer with a mechanism to track used resources so a second pass can be avoided when producing the final output format. After all, the IF renderer already processes the full document and knows which resources are necessary.

We can make three categories of renderers:

1. no resource optimization (Java2D-based output formats, PCL, Text, SVG)
2. implicit resource optimization (PDF due to its object structure, Mars)
3. explicit resource optimization (PostScript, AFP)

Please note, that this mechanism is only useful to the third category, so the mechanism will not be enabled unless done so explicitely. For formats like PDF this processing is not necessary since resources are added as they are needed.

We define a listener interface that receives notification of resource usage (**ResourceUsageListener**). The **RenderingResource** interface will be implemented by a handful of classes (at the beginning: FontResource & ImageResource). The object identity of these classes is defined by what the IF supports:

- Font: font-family, -weight, -style, -variant, -size
- Image: URI

We need some infrastructure to keep track of resource usage on page-, page-sequence- and document-level. The most important is page-level. The other levels just summarize the accumulated data. Keeping track of resources down to page-level serves the following purposes:

- Formats like PostScript note which resources are used by page.
- The IF can be split and merged. Resource usage on document-level alone would not be sufficient to list which resources are effectively used in the final print file.

The resource usage information can optionally be integrated into the IF. For this purpose the IF is extended by a structure that is inserted into the page trailer and the document trailer. If desired the IF renderer could also support writing a separate file parallel to the generated IF, if the information needs to be tracked somewhere (could be implemented as a special resource listener).

Subformat:

```
<resource-usage>
  <font family="Arial" weight="normal" style="normal" variant="normal" size="10pt" count="1"/>
  <image uri="http://xmlgraphics.apache.org/fop/images/logo.jpg" count="1"/>
</resource-usage>
```

On a side-note, resource counting will allow to only move resources to the document resources which are needed on more than one page. Resource on the target device can become too high if every resource is moved to the document resources unconditionally.

This whole idea adds some complexity but will make it possible to avoid a two-pass approach for PS and AFP generation which causes a reduction of through-put.

TODO

- [done] Check usage of PPML in this context. (Result: Wouldn't really help in this context but the resemblance of the basic structure to the new IF is remarkable.)
- [OPEN] Don't forget different writing modes
- [done] Decide whether to split IFPainter into IFDocumentHandler and IFPainter to reduce the number of methods per interface. Result: Split performed. Expected improvements realized.
- [OPEN] Deprecate the renderers that have been replaced by respective painter implementations once they are stable.

Comments

[1] JM: I wonder how far we'll need to go for tagged PDF. Do people only need the basic structure of the document (separating headers from flow content, maybe indicating block roles (para, title, footnote, ...) through extension attributes on fo:block? Or does someone need the whole tagged PDF feature set which basically allows you to embed many original semantics from the original FO document to be present in the PDF (spaces, indents, baseline shifts, alignment, etc.)?

[2] AC: Taking a SVG 1.2 subset (Tiny?) format, how slow would Batik be as a final renderer for FOP? JM: as already indicated in the first place: quite slow. The process: build-up of DOM tree, build-up of GVT tree, rendering GVT tree to Graphics2D, conversion of Graphics2D calls to final format. Too many steps in between. I suspect it would even be slower than today's solution. What we need is something that can be streamed and processed on the fly without building up too many intermediate structures in memory. BTW, Tiny is already too powerful for what we need. And I really don't intend to write a second Batik. If we decide to use an SVG subset, it would be an "SVG Nano". 🙂 I'll try to formulate a minimal format first trying to stay as close to SVG as possible. From there, we can check if this can be fully mapped to SVG elements. The risk is not needing all of SVG's features but needing many extensions which could again make the parsing to slow because of the growing complexity.

Results

Now that the implementation is practically complete, performance measurements are possible. Benchmark results have been published on the following page: http://people.apache.org/~jeremias/fop/benchmark-2009-02-13/