The story of Gerald Cinamon's germandesigners.net

Transforming a complex MS Word manuscript for the web

Matt Patterson
Saxonica Limited
<matt@wekstatt.io>

Abstract

The renowned graphic designer and author Gerald Cinamon [1] wrote a comprehensive biographical dictionary about graphic designers working in Germany during the Nazi regime, 'German graphic designers during the Hitler period', a project well summarised in its author's introduction:

> *Design historians in America and Britain have tended to ignore the talents or lives of those who remained in Germany during the Nazi regime as being unworthy of attention. But the talents were there, and the lives went on. Simply because these designers lived in Nazi Germany is no reason to ignore their work. Any history of twentieth-century graphic design – and Germany's particularly – must take note of them and their work.*
>
> *Generally, the lives and work of most émigré designers (usually Jewish) have been covered in books and design journals. Here is featured, where known, what happened to the designers that remained during the period 1933 to 1945 – the Hitler period.*
>
> *—Gerald Cinamon, from the introduction to the German Designers site [3]*

The book was written as a manuscript in Microsoft Word, containing over 900 entries across several separate files.

When turning the manuscript into a viable print-published book seemed unlikely, I was approached about whether it would be possible to publish it as a website.

This is the story of how we were able to combine the author's Word files, metadata annotations in Word, and XSLT to create well-structured, richly annotated markup that could, in turn, be transformed into a work of Hypertext.

We'll cover:

- *How the source material was prepared in Word itself, and what techniques we used to annotate and enrich the text.*

- *How the Word files were processed into usable, structured, XML.*
- *How that XML was turned into a website.*

Keywords: XML, XSLT, HTML, JSON, Design history, MS Word

1. A brief history of the project

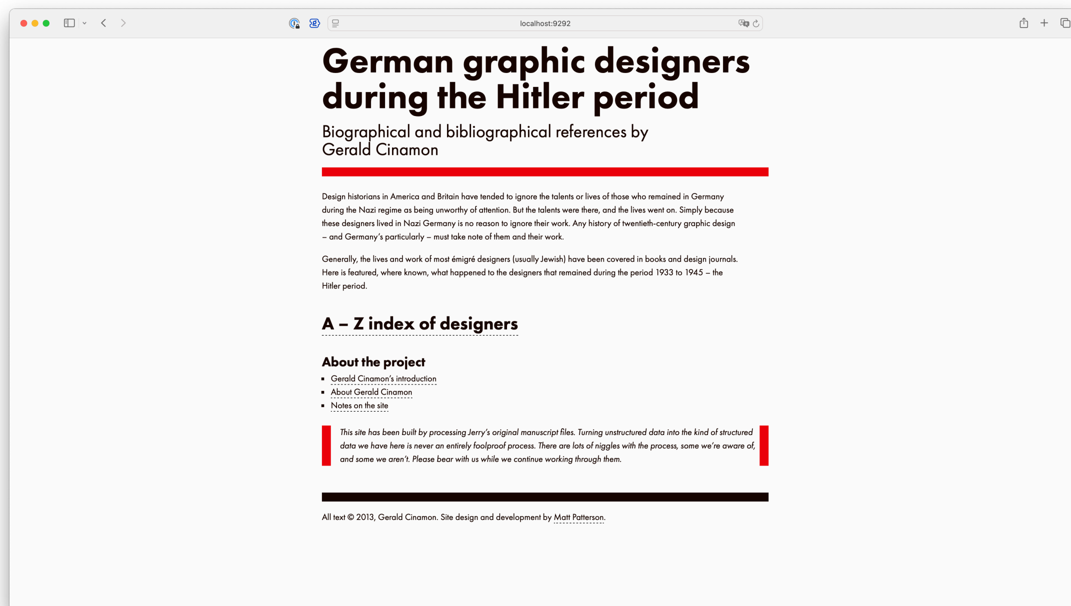


Figure 1. The home page

The project began back in 2011 with a call from the writer and academic Alex Butterworth about an unpublished biographical dictionary written by Gerald Cinamon. Gerald Cinamon, 1930-2024, is a significant figure in British Graphic Design and Typography – he was Penguin's Chief Designer, for example – who became a historian of the field.

I was intrigued, not least because Jerry was also the external examiner for my BA in Typography & Graphic Communication at Reading...

As a dictionary of biography, the finished book would be set of entries, one for each subject (person or organisation) being included. That format obviously lends itself to hypertext very well, and after much consideration and exploration we settled on a fairly straight hypertext adaptation approach, with one page per subject, and a central alphabetical index.

The book, still in author's manuscript form as a set of Microsoft Word documents, presented a number of challenges to convert into a coherent web site. Those challenges, and how we solved them, is the subject of this paper.

The original version of the site was a Ruby on Rails app, using XSLT 2 to perform basic extraction, but all the post processing in Ruby, with everything ending up in a relational database. The site went live to coincide with a major exhibition of Jerry's work in 2013 at the ICA in London [2]. Since then, the site has sat essentially unchanged until this year. Motivated by the need to move away from a largely unnecessary Web app server infrastructure, and the desire to make progress on features we'd had to abandon for the original site, we undertook to rewrite the machinery of the site from scratch, while preserving the URL structure and HTML output exactly. The preparation work done on the manuscript itself required no changes, but we are choosing to ignore the previous Rails site in this paper, for reasons of time and space.

This was a reasonably sized project, with a reasonably sized data set. The challenges are similar to those faced by people with bigger, and smaller, data sets, and more, or less, complex presentation requirements. It's a good size: big enough to have challenges, but small enough to be for the approaches we took to those challenges to be (hopefully) useful and understandable examples.

This paper will cover the preparation of the manuscripts, their extraction and processing, and the generation of a site from the processed results. Again, for reasons of time and space, we don't cover absolutely everything There will be things we elide, and very little discussion of the build system that glues the various parts of the process together. That said, all the examples are real, and all the code should run.

2. Preparing the source material in Word

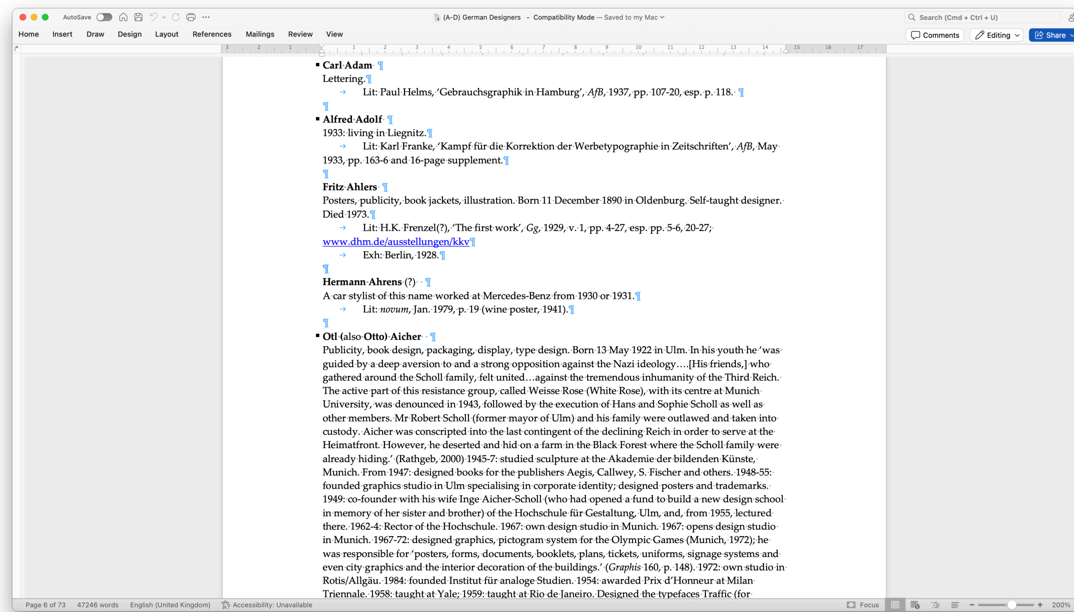
Consisting of over 700 entries, some only a sentence, some several pages, the manuscript was split between a number of Word documents, largely to stave off crashes. Entries contained a variable-length biography, and could contain some or all of a bibliography of works referencing the subject, a bibliography of works by the subject, and a list of exhibitions about, or featuring the work of, the subject.

You can see a page from the original manuscript on Figure 2.

While Word documents can contain structure, it's inferred from paragraph styles, not made explicit through markup structure like nesting. The main questions for processing the Word documents are therefore:

- What structure is present within the file?
- How can it be extracted?
- What kinds of processing would this allow without editing the manuscript?

We are concerned, here, with the logical hierarchies of content within the manuscript, as they are expressed visually. That is, how are the parts of the document differentiated from each other, and how do those parts relate to each other as a hierarchy?



Carl Adam
Lettering.
Lit: Paul Helms, 'Gebrauchsgraphik in Hamburg', *AfB*, 1937, pp. 107-20, esp. p. 118.

Alfred Adolf
1933: living in Liegnitz.
Lit: Karl Franke, 'Kampf für die Korrektion der Werbetypographie in Zeitschriften', *AfB*, May 1933, pp. 163-6 and 16-page supplement.

Fritz Ahlers
Posters, publicity, book jackets, illustration. Born 11 December 1890 in Oldenburg. Self-taught designer. Died 1973.
Lit: H.K. Frenzel(?), 'The first work', *Gg*, 1929, v. 1, pp. 4-27, esp. pp. 5-6, 20-27; www.dhm.de/ausstellungen/kkv
Exh: Berlin, 1928.

Hermann Ahrens (?)
A car stylist of this name worked at Mercedes-Benz from 1930 or 1931.
Lit: *novum*, Jan. 1979, p. 19 (wine poster, 1941).

Otl (also **Otto**) **Aicher**
Publicity, book design, packaging, display, type design. Born 13 May 1922 in Ulm. In his youth he 'was guided by a deep aversion to and a strong opposition against the Nazi ideology….[His friends,] who gathered around the Scholl family, felt united…against the tremendous inhumanity of the Third Reich. The active part of this resistance group, called Weisse Rose (White Rose), with its centre at Munich University, was denounced in 1943, followed by the execution of Hans and Sophie Scholl as well as other members. Mr Robert Scholl (former mayor of Ulm) and his family were outlawed and taken into custody. Aicher was conscripted into the last contingent of the declining Reich in order to serve at the Heimatfront. However, he deserted and hid on a farm in the Black Forest where the Scholl family were already hiding.' (Rathgeb, 2000) 1945-7: studied sculpture at the Akademie der bildenden Künste, Munich. From 1947: designed books for the publishers Aegis, Callwey, S. Fischer and others. 1948-55: founded graphics studio in Ulm specialising in corporate identity; designed posters and trademarks. 1949: co-founder with his wife Inge Aicher-Scholl (who had opened a fund to build a new design school in memory of her sister and brother) of the Hochschule für Gestaltung, Ulm, and, from 1955, lectured there. 1962-4: Rector of the Hochschule. 1967: own design studio in Munich. 1967: opens design studio in Munich. 1967-72: designed graphics, pictogram system for the Olympic Games (Munich, 1972); he was responsible for 'posters, forms, documents, booklets, plans, tickets, uniforms, signage systems and even city graphics and the interior decoration of the buildings.' (*Graphis* 160, p. 148). 1972: own studio in Rotis/Allgäu. 1984: founded Institut für analoge Studien. 1954: awarded Prix d'Honneur at Milan Triennale. 1958: taught at Yale; 1959: taught at Rio de Janeiro. Designed the typefaces Traffic (for

Figure 2. A page from the original manuscript

As we will discuss in the next section, the markup underlying a Word document is a flat sequence of paragraphs. It's possible to attach named paragraph styles to paragraphs, in which case the paragraph takes on the same visual appearance as all the other paragraphs with that style. With this approach it's possible to make some aspects of the underlying logical structure of the document explicit: this paragraph is body text, that one is a heading, and so on. It's also possible to directly apply visual formatting to the text without using styles (all paragraphs will have the 'Normal' style, but vary in visual formatting). Worse, you can directly apply visual formatting to text, overriding the appearance of, but not removing the association with, other styles. A paragraph may be marked as body text, but styled as a heading (or vice versa).

The editorial lever we have to make hierarchy explicit (as explicit as possible, anyway) in a Word document is the consistent application of paragraph and character styles. If that is done, it's possible to map those styles to another document model, or to imply grouping and nesting reliably. While Word styles are fundamentally limited by the Word document model (no more than one paragraph style per paragraph, and only one character style per character), they are the tool we have.

The manuscript, while visually consistent, with a clear visual hierarchy that maps to the author's underlying logical hierarchy, did that through direct formatting of the text. To transform the manuscript into another format required that we

develop, and consistently apply, a set of styles throughout the entire document set in order to make them transformable.

The work of devising a set of styles that balanced the competing needs of richness to allow complex transformations, and simplicity to allow consistent and reliable application of the styles to the documents by an editor made up the bulk of the project.

What we came up with was a set of simple paragraph styles for the components of an entry – name, the biography, the various sorts of bibliographic details, and a richer set of character styles that could be used to annotate the entry. There were styles to capture the various parts of bibliographic entries, as well as the general - a date, the name of another designer, and the specific: dates and places of birth and death.

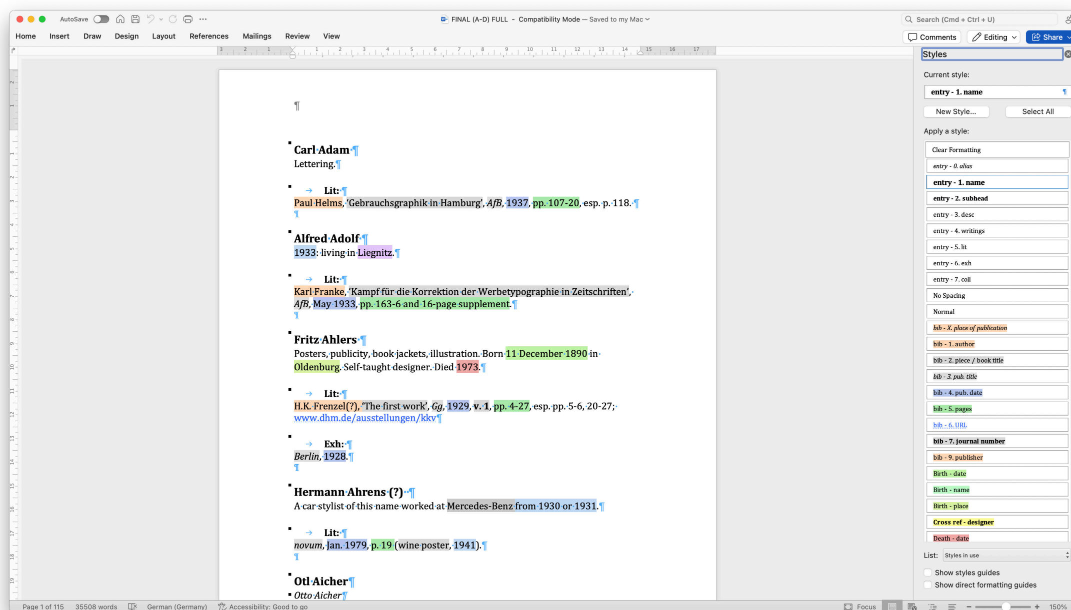


Figure 3. Short styled entries

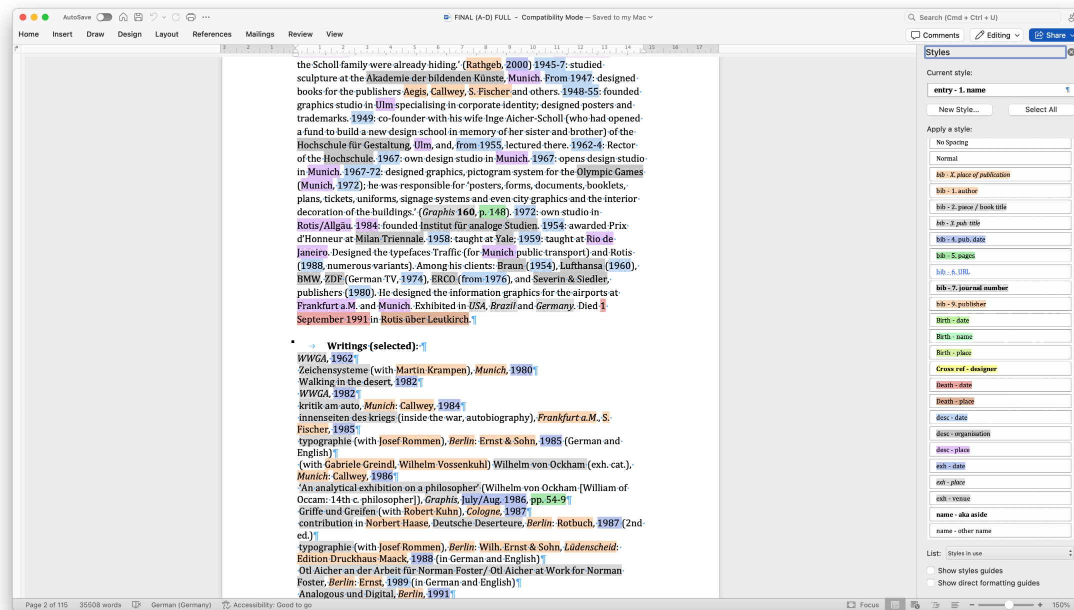


Figure 4. An excerpt of a longer entry

Once we had a set of styles that worked, we recruited someone to go through the whole document set and apply them. To aid that, the styles had a comprehensive set of keyboard shortcuts set up through macros to make it possible to do all the styling, without having to take your hands off the keyboard, an enormous productivity booster.

3. Turning one Word file into 100 dictionary entries

The primary aim of processing the Word documents is to extract the entries within them as separate documents. To do that we need to consider the structure of the WordprocessingML flavour of Microsoft's Office Open XML format, the affordances it offers us, and the restrictions it imposes. (We'll also look at some of the common pitfalls and gotchas I encountered.)

The format is standardised as ECMA-376 [7] & ISO/IEC 29500 [6], and there's a lot of documentation available. One thing is clear, though. It's an extremely complex format. This screen grab of the start of a WordprocessingML document should give you a flavour:

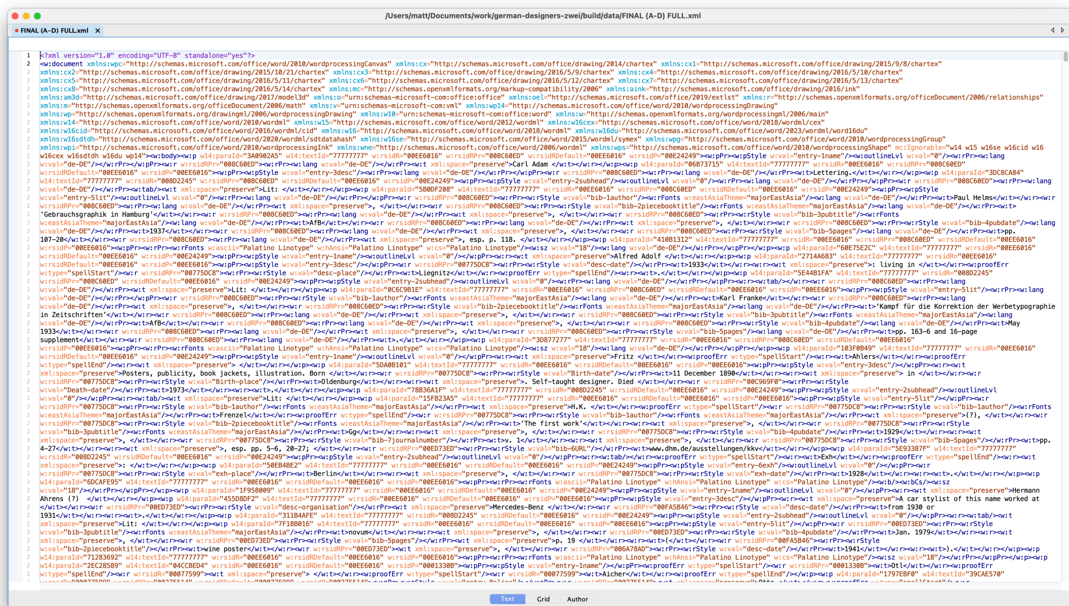


Figure 5. A WordprocessingML document

Critical things to note: First, that's an unholy number of namespace declarations. Second, namespace-qualified attribute names are used throughout, and third, the content is all one line. The line breaks in the screenshot above are only there because of oXygen's soft wrap. Let's look in more detail at the structure.

3.1. The structure of the WordprocessingML ML document

`.docx` files are actually just Zip archives with a different file extension. If we unzip a document at the command line you can see the result:

```
$ unzip ../a-word-file.docx
Archive:  ../a-word-file.docx
  inflating: [Content_Types].xml
  inflating: _rels/.rels
  inflating: word/_rels/document.xml.rels
  inflating: word/document.xml
  inflating: word/footnotes.xml
  inflating: word/endnotes.xml
  inflating: word/header1.xml
  inflating: word/footer1.xml
  inflating: word/theme/theme1.xml
  inflating: word/_rels/settings.xml.rels
  inflating: word/settings.xml
  inflating: word/fontTable.xml
```

```
  inflating: docProps/core.xml
  inflating: word/styles.xml
  inflating: word/webSettings.xml
  inflating: docProps/app.xml
```

Note that the Zip archive contains no root folder name as the archive, so unzip into a folder you have created for the task. Also note that, at least sometimes, macOS's built-in archive handling will refuse to unzip a `.docx` whose extension you have changed to `.zip`.

Depending on the complexity of your needs, you may need to look at several of these files, but if you're lucky, or otherwise careful you'll be able to do as I did, and ignore everything except the `word/document.xml` file. This file contains the main text of the document. If we strip the file back to its basic outline it looks like this:

```
<w:document>
  <w:body>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
  </w:body>
</w:document>
```

The document is a flat sequence of `<w:p>` paragraph elements. Each paragraph, in turn, consists of some metadata followed by the text:

```
<w:p>
  <w:pPr>
    <w:pStyle w:val="para-style"/>
  </w:pPr>
  <w:r>
    <w:t>The actual paragraph text!</w:t>
  </w:r>
</w:p>
```

There's often more to the `<w:pPr>` metadata, but for our purposes the `<w:pStyle>` element is all we need. Before we celebrate too early though, the text may well be a sequence of runs of text, with metadata of their own:

```
<w:p>
  <w:r>
    <w:t xml:space="preserve">The actual </w:t>
  </w:r>
```

```
  <w:r>
    <w:rPr>
      <w:rStyle w:val="character-style"/>
    </w:Pr>
    <w:t>paragraph text!</w:t>
  </w:r>
</w:p>
```

This gives us two things: a handle on character-level markup, and the `xml:space` warning klaxon. Effectively, all whitespace in a document is significant, and careless processing, even serialising with indenting switched on, can wreak havoc. The text in a paragraph is not just broken into runs by changes in character styling, either. They can also be interrupted by control codes, like markers for the start or end of (alleged or real) spelling errors:

```
<w:p>
  <w:r>
    <w:t xml:space="preserve">The actual </w:t>
  </w:r>
  <w:proofErr w:type="spellStart"/>
  <w:r>
    <w:t>paragraph text!</w:t>
  </w:r>
</w:p>
```

There are many kinds of these empty tag markers that can be inserted by Word into your text flow, and there will be cases where text runs are split within a word to allow the placement of one of these. Evan Lenz has extensive coverage of all the things that can occur within WordprocessingML in his article about it [4]. If you want to find out more, this is an excellent resource.

3.2. The affordances of the WordprocessingML document

The document is flat sequence of paragraphs, which contain flat sequences of text runs. It's simple. The main affordance we are provided with is the `<w:pStyle>` and `<w:rStyle>` metadata tags which identify the style that is applied to a paragraph or text run. Using the presence of identifiers for styles we can view the styled manuscript like this:

```
<p entry-title>
<p>
<p>
<p>
<p entry-title>
<p>
<p bibliography-entry>
<p bibliography-entry>
```

```
<p bibliography-entry>
<p>
```

We can view the document as a series of entries delimited by the presence of a paragraph with the entry-title style, and, within those entries, sections like the bibliography represented by a sequence of adjacent paragraphs with a known style name:

```
<entry>
    <p entry-title>
    <p>
    <p>
    <p>
</entry>
<entry>
    <p entry-title>
    <p>
    <bibliography>
        <p bibliography-entry>
        <p bibliography-entry>
        <p bibliography-entry>
    </bibliography>
    <p>
</entry>
```

Within a paragraph, the principle is very similar, with the main wrinkle being the presence of marker elements breaking up runs of text:

```
<p>
    <r>They worked with the artist </r>
    <r crossref>Käthe </r>
    <spelling-marker/>
    <r crossref>Kollwitz</r>
    <r> in Berlin.</r>
</p>
```

If we squint, this resolves to:

```
<p>They worked with the artist <crossref>Käthe Kollwitz</crossref> in
Berlin.</p>
```

While it doesn't have the simplicity of gathering up directly adjacent elements, we can use similar techniques.

3.2.1. A brief digression into of style names in the UI, and IDs in the markup

The style ID in the `w:pStyle/@w:val` / `w:rStyle/@w:val` is just that – an ID. It's not the style name. Word has transformed the name you see in the UI, like "`entry`

`1. - name"`, to a token, `"entry1-name"`. The `.docx` also contains a `styles.xml` file that contains all styles, their UI names and their IDs:

```
<w:style w:type="paragraph" w:styleId="entry-1name">
    <w:name w:val="entry - 1. name"/>
    …
</w:style>
```

Within `styles.xml`, `<w:style w:styleId="…">` is the key the corresponds to `<w:pStyle w:val="…">` used in the document. You may find that you need to use this file to figure out which ID maps to a particular name style.

LibreOffice can also read and write `.docx` files, and, historically, LibreOffice / OpenOffice would use random opaque style IDs, and you really had to look up the ID in styles.xml to be sure of what style you were getting.

3.3. Splitting entries and grouping paragraph-level content

XSLT's `<xsl:for-each-group group-starting-with="…">` is exactly the tool for us. Using this, it's easy to chunk our document into entries and wrap them in a tag for later processing:

```
<xsl:for-each-group select="$body/*" group-starting-with="w:p[w:pPr/
w:pStyle/@w:val='entry-1name']">
    <entry>
        <xsl:sequence select="current-group()"/>
    </entry>
</xsl:for-each-group>
```

The expression given to `group-starting-with` is evaluated against the input sequence and if it matches the body of the `<xsl:for-each-group>` is executed with the `current-group()` function returning all the items that have been evaluated since the previous match.

The XPath selector we need is quite long, but it's also very simple. What it relies on, above all else, is consistent and correct styling of the document in Word. If one entry misses its subject name paragraph being styled correctly, you get two entries being merged, and assumptions that later processing steps make about the structure of the `<entry>` will be wrong, and if the input contained several non-entry items before the first paragraph styled `"entry1-name"` those would be returned as their own group. As well as this, because `group-starting-with` breaks at the beginning of a group, if the final group of items contains extra stuff that doesn't belong in an `<entry>`, the final `<entry>` would contain unexpected items.

In the case of this project, we worked hard to ensure that those sorts of problems were corrected in the source.

I'm able to use the same approach to subdivide the entry into chunks that align with entry sections because of the presence of subheadings between sec-

tions. The main difference is that significantly different behaviour is needed for the first chunk, which contains the name and biography of the entry subject, and doesn't contain a section subheading. I wrap each in a temporary `<chunk>` element and rely on being able to discriminate the first from following chunks to drive later processing:

```
<xsl:template match="entry">
    <xsl:copy>
        <xsl:for-each-group select="*" group-starting-with="w:p[w:pPr/
w:pStyle/@w:val='entry-2subhead']">
            <xsl:variable name="chunk">
                <chunk>
                    <xsl:sequence select="current-group()"/>
                </chunk>
            </xsl:variable>

            <xsl:apply-templates select="$chunk"/>
        </xsl:for-each-group>
    </xsl:copy>
</xsl:template>
```

I can use the presence of the subject name para, or the absence of a subhead, to discriminate the first chunk from the rest, like this:

```
<xsl:template match="chunk[w:p/w:pPr/w:pStyle/@w:val='entry-1name']"
priority="10">
    <xsl:apply-templates mode="subject" select="."/>
    <xsl:apply-templates mode="biography" select="."/>
</xsl:template>

<!-- the rest of the entry -->
<xsl:template match="chunk">
    <xsl:apply-templates mode="process"/>
</xsl:template>
```

For the subject and biography parts of the first chunk I need to do two different things. For the subject, I need to provide the `<name>`, and generate a sortable version, `<sort-name>`, from it. And I need to do that for any aliases the subject had too. Because we have a fixed output order in mind, and it's unambiguous what's a name or alias and what isn't, we can apply-templates with a mode and not worry about document order, filtering in the templates:

```
<xsl:mode name="process" on-no-match="shallow-skip"/>

<xsl:template match="chunk" mode="subject">
    <subject>
        <xsl:apply-templates select="*[w:pPr/w:pStyle/
@w:val='entry-1name']" mode="process"/>
```

```
        <xsl:where-populated>
            <aliases>
                <xsl:apply-templates select="*[w:pPr/w:pStyle/
@w:val='entry-0alias']" mode="process"/>
            </aliases>
        </xsl:where-populated>
    </subject>
</xsl:template>
```

Note the use of `on-no-match="shallow-skip"` on the `process` mode. Doing that lets us match whatever we're interested in without worrying about extra text-nodes littering our output, which is, as already mentioned, extremely sensitive to whitespace being added or removed. Handling the biography content is simply a case of outputting the right element and then ignoring the names when applying templates:

```
<xsl:template match="chunk" mode="biography">
    <biography>
        <xsl:apply-templates select="*[not(
                w:pPr/w:pStyle/@w:val = ('entry-1name', 'entry-0alias')
            )]" mode="process"/>
    </biography>
</xsl:template>
```

The biography section can contain multiple styles of paragraph: standard text, and lists, so we need to be able to group the flat list paras and wrap them into a list-container-with-list-items style lists. Grouping this kind of paragraph-level content relies on adjacency rather than a starting marker. `<xsl:for-each-group group-adjacent="…">` provides the means to achieve this. However, it's actually easier (in this case) to do that grouping at the outputting-HTML stage, so we'll return that to that for paragraphs later in the paper. We do need to figure out which paragraphs will eventually become list items, though. To do that we're going to use a trick where we store the mapping between input style name and desired element in a sequence of elements in a variable. We can use standard XPath pattern matching to get what we need, with minimal fuss:

```
<xsl:variable name="list-section-elements" as="element()+">
    <writings style-name="entry-4writings"/>
    <lit style-name="entry-5lit"/>
    <exhibitions style-name="entry-6exh"/>
    <collections style-name="entry-7coll"/>
</xsl:variable>

<xsl:template match="w:p[w:pPr/w:pStyle/@w:val = $list-section-elements/
@style-name]"
    mode="process" priority="10">
```

```
    <li><xsl:apply-templates select="." mode="para-guts"/></li>
</xsl:template>
```

If the `<w:p>` we're processing has a style name that matches the `style-name` attribute of one of our `$list-section-elements`, it's supposed to be a list item. We'll come back to processing the guts of paragraphs in a minute. Now we've handled matching paras in the first chunk, we can turn to the other chunks, which are effectively all containers for lists. All the bibliographic entries should occur together in the source, as should all the exhibitions, and so on. We can reuse our `$list-section-elements` tactic above, together with an XSLT 3 addition to `<xsl:copy>`:

```
<xsl:template match="chunk">
    <xsl:variable name="style-name" select="w:p[1]/w:pPr/w:pStyle/
@w:val"/>
    <xsl:copy select="$list-section-elements[@style-name = $style-name]">
        <xsl:apply-templates mode="process" select="$chunk"/>
    </xsl:copy>
</xsl:template>
```

The approach here is to select the element from `$list-section-elements` that matches the style name used by the paragraphs in the chunk, and copy it to wrap the chunk elements, which we can then process. `select="…"` on `<xsl:copy>` is an XSLT 3 feature, which makes this very simple, because we don't need to extract the element name of the wrapper element to feed to `<xsl:element>`.

Unfortunately, real life isn't as simple as that. This is a long manuscript and if there are errors, or some edge cases where the content model legitimately differs, we don't want to wind up with `$style-name` containing a multi-item sequence: offering multiple elements to `<xsl:copy select="…">` will cause a dynamic error and halt processing.

To handle this case, we can get all the paragraph style names in the chunk, throw out ones not related to the section type (probably the legitimate edge cases), and then match that against our section elements. If we get more than one section element back, we can send an `<xsl:message>` and then just pick the first section element:

```
<xsl:template match="chunk">
    <xsl:param name="name" tunnel="yes"/>

    <xsl:variable name="style-name"
        select="distinct-values(w:p/w:pPr/w:pStyle/@w:val)
            [. = $list-section-elements/@style-name]"/>
    <xsl:variable name="chunk" select="."/>

    <xsl:if test="count($style-name) gt 1">
        <xsl:message expand-text="yes">Entry {$name} has multiple
```

```
section-type paras
            ({string-join($style-name, ', ')}) in a single chunk</
xsl:message>
    </xsl:if>
    <xsl:copy select="$list-section-elements[@style-name = $style-
name[1]]">
        <xsl:apply-templates mode="process" select="$chunk"/>
    </xsl:copy>
</xsl:template>
```

To facilitate sending a useful message, the original template that matches the `<entry>` generated by the splitting function adds the subject's name as a tunnelling param, which can then be used when needed, as we did above. It adds a couple of lines to the template:

```
<xsl:template match="entry">
    <xsl:variable name="name" select="string(w:p[w:pPr/w:pStyle/
@w:val='entry-1name'])"/>
    <xsl:copy>
        <xsl:for-each-group select="*" group-starting-with="w:p[w:pPr/
w:pStyle/@w:val='entry-2subhead']">
            <xsl:variable name="chunk">
                <chunk>
                    <xsl:sequence select="current-group()"/>
                </chunk>
            </xsl:variable>

            <xsl:apply-templates select="$chunk">
                <xsl:with-param name="name" tunnel="yes" select="$name"/>
            </xsl:apply-templates>
        </xsl:for-each-group>
    </xsl:copy>
</xsl:template>
```

All the section chunks are just lists with no other content, and we already have a template that will turn all the relevant `<w:p>`'s into `<li>`'s, all we really need to do now is handle the contents of a paragraph (or list item). This requires us to deal with removing Word markers, concatenating runs of the same-styled text node, and generating the right output without altering the para text's white-space. It was surprisingly straightforward:

```
<xsl:template match="w:p" mode="process">
    <p><xsl:apply-templates select="." mode="para-guts"/></p>
</xsl:template>

<xsl:template match="w:p" mode="para-guts">
    <xsl:if test="w:pPr/w:pStyle/@w:val">
```

```
        <xsl:attribute name="class" select="w:pPr/w:pStyle/@w:val"/>
    </xsl:if>
    <xsl:for-each-group select="w:r" group-adjacent="if (self::w:r[w:rPr/
w:rStyle]) then self::w:r/w:rPr/w:rStyle/@w:val else '1'">
        <xsl:sequence select="if (current-grouping-key() = '1') then
gd:unstyled-runs(current-group()) else gd:styled-runs(current-group())"/>
    </xsl:for-each-group>
</xsl:template>

<xsl:function name="gd:unstyled-runs" as="text()?">
    <xsl:param name="runs" as="element()*"/>

    <xsl:value-of select="$runs/w:t" separator=""/>
</xsl:function>

<xsl:function name="gd:styled-runs">
    <xsl:param name="runs" as="element()*"/>

    <span class="{$runs[1]/w:rPr/w:rStyle/@w:val}">
        <xsl:value-of select="$runs/w:t" separator=""/>
    </span>
</xsl:function>
```

The first template is just the should-be-a-para equivalent of the should-be-a-list-item template we saw earlier. The mode="para-guts" template does the heavy lifting. Because WordprocessingML text elements inside a paragraph are as flat as the paragraphs are within the document, we can take advantage of the fact that all the Word non-text elements we want to throw away are only ever siblings of `<w:r>` elements. Through the magic of selecting just the `<w:r>` elements when we start our processing, we eliminate all the things we're uninterested in. Next we need to regroup all the runs of identically-styled text that were broken apart by the Word non-text elements we have just ignored out of existence. To do this, we turn to `<xsl:for-each-group group-adjacent="…">`. Unlike `group-starting-with`, the return value of the `group-adjacent` expression is used to generate a grouping key, and adjacent items in the input with the same grouping key are returned in the same group. The value returned by the expression must be a single item, not a multi-item sequence, and cannot be the empty sequence either, which is why we generate the value `"1"` as a placeholder for those text runs that do not have a character style applied. Once we have a group, it's a simple matter of either concatenating all the text nodes in the group into one, or doing that and also wrapping the result in a `<span class="{$style-name}">` element. The new-in-XSLT 3 `separator` attribute on `<xsl:value-of>` allows us to ensure that no extra whitespace is being added.

3.4. URL-slugs, the index, and writing all the entries to disk

Now we have a method for generating individual XML documents we need to think about how we output them to disk, and how we will generate the alphabetical index we need. Clearly, <xsl:result-document> is the mechanism we need to actually output the documents, but what should the filenames be? We're intending this to be a website, so the obvious thought is that the filenames should be valid and convenient parts of a URL path. We can define a transformation from the unicode-letters-and-spaces world of the subject name to the ASCII-only standard URL path, so that's not an especially hard problem. We could also, in theory, generate the index by crawling the generated entry document files. There is, however, a problem with both of these approaches, at least in the case of this project. The obvious title of these documents is the name of the subject, and the obvious URL slug is that name transformed into a valid path segment. However, the subject of these documents are people, and different people sometimes have the same name, which would generate the same URL slug.

That, in turn, raises the question of how you know if there's a duplicate name, and what to do about it. We are going to use a map/reduce approach using `<xsl:iterate>` to tackle all these problems. Firstly, knowing if there's a duplicate URL slug requires that we keep track of the URL slugs we've generated. Keeping track requires some sort of index, which we can later reuse as the source for the alphabetic index.

`<xsl:iterate>` was introduced for streaming processing, but provides the right kind of hooks to do the map/reduce job we need here. We can pass an initial value via a parameter to it, and at the end of each iteration set that parameter for the next iteration. If we pass in an empty map to the first iteration, and add each entry using its URL slug as the key as we go, we can use that map to check if a proposed new URL slug already exists, and modify it appropriately. At the end of the process we'll have an index containing every entry in the file:

```
<xsl:output method="json" indent="yes"/>
<xsl:output name="entry-xml" method="xml" indent="yes" suppress-
indentation="p li"/>

<xsl:function name="gd:as-entries" as="element(entry)*">
    <xsl:param name="body" as="element()"/>

    <xsl:for-each-group select="$body/*"
        group-starting-with="w:p[w:pPr/w:pStyle/@w:val='entry-1name']">
        <entry>
          <xsl:sequence select="current-group()"/>
        </entry>
    </xsl:for-each-group>
</xsl:function>
```

```
<xsl:template match="w:body">
    <xsl:iterate select="gd:as-entries(.)">
      <xsl:param name="index" select="map { }" as="map(xs:string,
map(*))"/>
      <xsl:on-completion select="$index"/>

      <xsl:variable name="entry" as="element(entry)">
          <xsl:apply-templates select="."/>
      </xsl:variable>

      <xsl:variable name="entry-with-slug" as="element(entry)">
          <xsl:apply-templates select="$entry" mode="add-url-slug">
            <xsl:with-param name="index" select="$index"/>
          </xsl:apply-templates>
      </xsl:variable>

      <xsl:variable name="entry-slug" select="$entry-with-slug/@url-
slug" as="xs:string"/>

      <xsl:result-document format="entry-xml" href="{resolve-
uri(concat($entry-slug, '.xml'), $output-path)}">
          <xsl:sequence select="$entry-with-slug"/>
      </xsl:result-document>

      <xsl:next-iteration>
          <xsl:with-param name="index" select="map:put($index, $entry-
slug, gd:index-entry($entry))"/>
      </xsl:next-iteration>
    </xsl:iterate>
</xsl:template>
```

We've already seen the body of the `gd:as-entries` function above. Declaring it as a function rather than a template makes it very easy to plug into the `select="…"` of the `<xsl:iterate>`. The first `<xsl:param>` sets up the index map, which begins as an empty map. Using `<xsl:on-completion>` means that the $index param will be returned at the end of the iteration, with the value that was calculated for it in the `<xsl:next-iteration>` block. The `<xsl:next-iteration>` block in question simple adds the index entry generated for the current entry to the existing `$index` map and passes that as the new value of `$index` for the next entry in the iteration.

In between we process the entry, using the approaches discussed in the previous section, and then we process that to generate a URL slug for the entry. Post-processing the entry means we've access to the finished entry, where we've already pulled out the main subject name and all the aliases. Once we have the

entry with a URL slug, we use `<xsl:result-document>` to write the complete entry to a file in the directory `$output-path` (a global stylesheet param), appending `.xml` to the URL slug for the filename. When the iteration is finished, the complete index map is serialized as JSON and written to disk by the process that started the transform.

The real complexity here is in the URL slug generation:

```
<xsl:mode name="add-url-slug" on-no-match="deep-copy"/>

<xsl:template match="entry" mode="add-url-slug">
    <xsl:param name="index" as="map(*)"/>

    <xsl:copy>
        <xsl:attribute name="url-slug"
            select="gd:entry-slug($index, subject/name)"/>
        <xsl:apply-templates mode="#current"/>
    </xsl:copy>
</xsl:template>

<xsl:function name="gd:entry-slug" as="xs:string">
    <xsl:param name="index" as="map(xs:string, map(*))"/>
    <xsl:param name="name-node" as="node()"/>

    <xsl:variable name="slug"
        select="translate(lower-case($name-node)), ' ', '_')"/>

    <xsl:sequence select="gd:unused-entry-slug($slug, 1, $index)"/>
</xsl:function>

<xsl:function name="gd:unused-entry-slug" as="xs:string">
    <xsl:param name="slug" as="xs:string"/>
    <xsl:param name="count" as="xs:integer"/>
    <xsl:param name="index" as="map(xs:string, map(*))"/>

    <xsl:variable name="test-slug" as="xs:string"
        select="if ($count > 1) then concat($slug, '-', $count) else
$slug"/>

    <xsl:sequence select="if (not(map:contains($index, $test-slug)))
        then $test-slug
        else gd:unused-entry-slug($slug, $count + 1, $index)"/>
</xsl:function>
```

The template just calls the function to generate the URL slug, then adding a `url-slug` attribute to the `<entry>` with the generated value. The current state of the index is passed as a param to the template, and on to the `gd:entry-slug()`

function, which performs the transform needed to create a URL slug from the subject's name (a process which is massively simplified in this example, since it's unrelated to the problem at hand). With the base URL slug generated, the `gd:unused-entry-slug()` checks the index to see if the URL slug already exists, and if it doesn't then the base slug is returned. If it does already exist, the count is incremented and the function calls itself with the base slug and the count. When the count is greater than 1, it's appended to the base slug before being checked against the index. This continues until an unused URL slug is found.

The last step of the index creation process is the creation of a new index entry, which is done by the `gd:index-entry()` function:

```
<xsl:function name="gd:subject-name" as="map(xs:string, xs:string)">
    <xsl:param name="subject" as="element()"/>

    <xsl:sequence select="map { 'name': string($subject/name), 'sort-
name': string($subject/sort-name) }"/>
</xsl:function>

<xsl:function name="gd:index-entry" as="map(xs:string, item())">
    <xsl:param name="entry" as="element(entry)"/>

    <xsl:variable name="aliases" as="map(xs:string, xs:string)*"
        select="for $alias in $entry/subject/aliases/alias return
gd:subject-name($alias)"/>

    <xsl:map>
        <xsl:sequence select="gd:subject-name($entry/subject)"/>
        <xsl:sequence select="if (empty($aliases)) then () else map
{ 'aliases': array { $aliases } }"/>
    </xsl:map>
</xsl:function>
```

The final JSON index looks like this:

```
{
  …,
  "mila_hoffmann_lederer": {
    "name": "Mila Hoffmann-Lederer",
    "sort-name": "Hoffmann-Lederer, Mila",
    "aliases": [ { "name":"Mila Hoffmannlederer", "sort-
name":"Hoffmannlederer, Mila" } ]
  },
  …
}
```

Combining all the index files generated each source file is trivial, and we do it in the program that runs the transforms. Once all the source files are transformed,

we have a directory full of entry XML documents, and a complete index of all the entries across all the source files.

4. Generating a website

Generating the HTML pages for entries themselves is relatively straightforward, but in addition to generating those, we're also generating the index from a JSON file, so we'll take a quick tour through the process.

4.1. Generating the A–Z index page

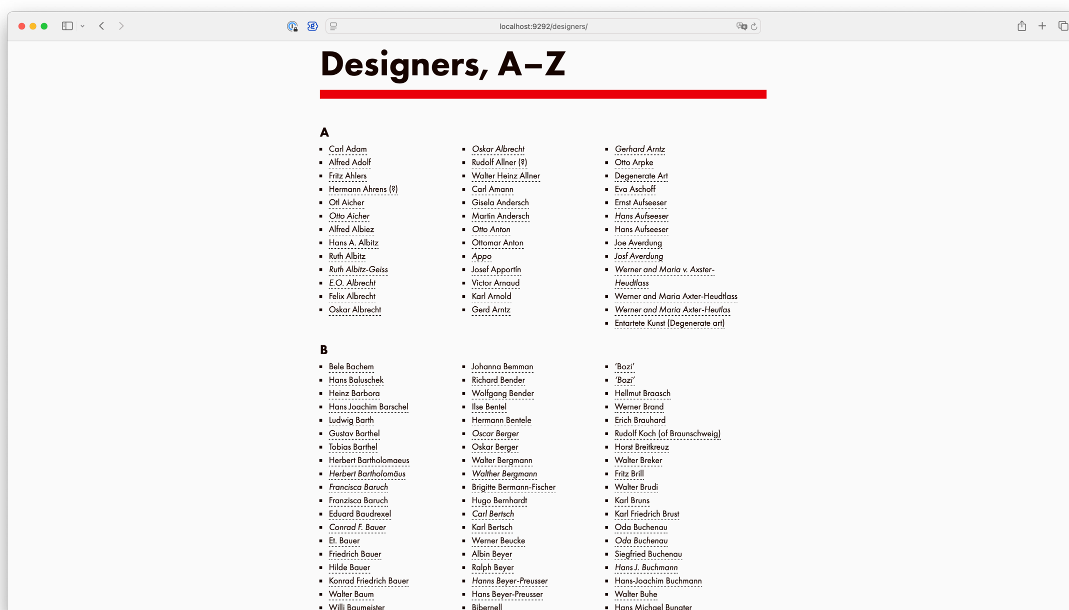


Figure 6. The A-Z index page

The index page is generated directly from the `index.json` file built from combining the indices built by processing the sources. To do this requires parsing the JSON file using a Saxon s9api.JsonBuilder, and then, in our case, calling a named template with the XdmMap just created as the global context item. We have to call a template rather than simply applying templates because we're importing a base layout stylesheet that controls the generation of the HTML skeleton, and a template that matches / won't match an XdmMap.

The layout stylesheet is very straightforward. It defines several modes, one for each 'hole' in the page that needs filling, generates the HTML skeleton and calls apply-templates in the right hole with right mode, which the importing stylesheets need to support. Here's how that looks for the index page, showing the areas for the masthead, content, and footer:

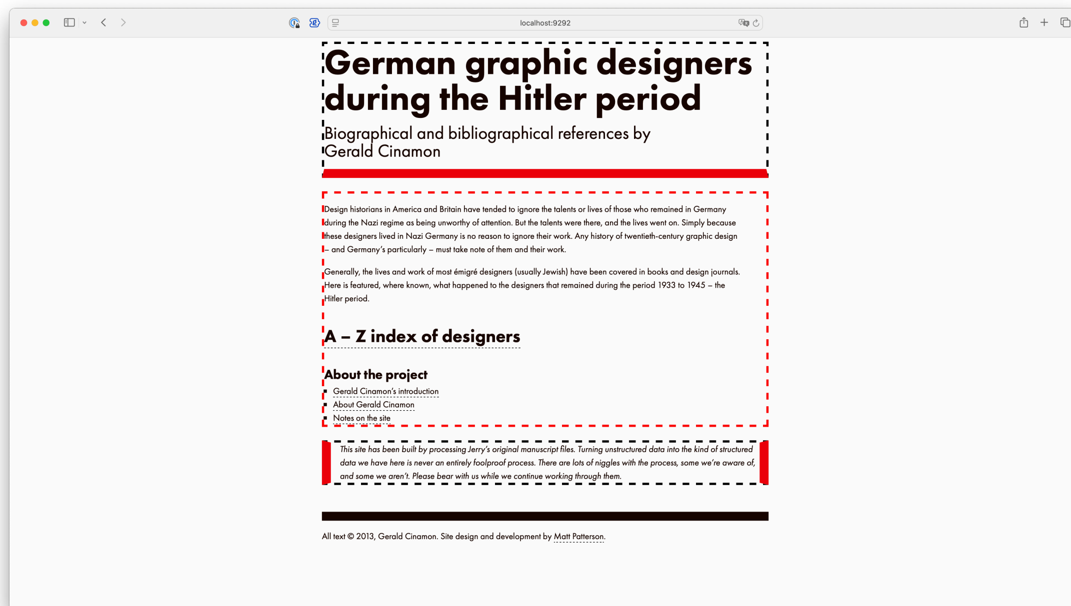


Figure 7. The layout skeleton

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:math="http://www.w3.org/2005/xpath-functions/math"
  exclude-result-prefixes="xs math"
  version="3.0">

  <xsl:mode on-no-match="shallow-skip"/>
  <xsl:mode name="content" visibility="public" on-no-match="shallow-
skip"/>
  <xsl:mode name="masthead" visibility="public" on-no-match="shallow-
skip"/>
  <xsl:mode name="footer" visibility="public" on-no-match="shallow-
skip"/>

  <xsl:output method="html" html-version="5.0"/>

  <xsl:template name="index-template">
    <html>
      <head>
        <title>…</title>
        <link rel="stylesheet" media="all" href="/css/main.css" />
       </head>
      <body>
```

```
        <div id="container">
          <div class="masthead">
            <xsl:apply-templates mode="masthead" select="."/>
          </div>
          <div id="content">
            <xsl:apply-templates mode="content" select="."/>
          </div>
          <footer>
            <xsl:sequence>
              <xsl:apply-templates mode="footer" select="."/>
              <xsl:on-empty>
                <nav>
                  <div class="nav-group">
                    <h3><a href="/">Home</a></h3>
                  </div>
                  <div class="clearfix"/>
                </nav>
              </xsl:on-empty>
            </xsl:sequence>
            <div id="copyright">
              <p>…</p>
            </div>
          </footer>
        </div>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="/">
    <xsl:call-template name="index-template"/>
  </xsl:template>
</xsl:stylesheet>
```

Probably of most interest here is that we can provide navigation fallback through the use of `<xsl:on-empty>`, so if the footer templates generate nothing, a useful nav link back home is generated instead. The A-Z page stylesheet itself looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:math="http://www.w3.org/2005/xpath-functions/math"
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:array="http://www.w3.org/2005/xpath-functions/array"
  xmlns:gd="http://germandesigners.net"
  exclude-result-prefixes="xs math map array gd"
  version="3.0">
```

```
  <xsl:import href="../layouts/base.xsl"/>

  <xsl:template match=".[. instance of map(*)]" mode="masthead">
    <h1>Designers, A-Z</h1>
  </xsl:template>

  <xsl:template match=".[. instance of map(*)]" mode="content">
    <xsl:variable name="index" select="."/>

    <xsl:variable name="unordered-ungrouped-entries" as="element()*">
      <xsl:sequence select="map:for-each($index, gd:a-z-entries#2)"/>
    </xsl:variable>

    <xsl:for-each-group select="$unordered-ungrouped-entries"
        group-by="upper-case(substring(@sort-name, 1, 1))">
      <xsl:sort select="@sort-name"
          collation="http://www.w3.org/2013/collation/UCA?
strength=secondary"/>
      <h2 class="letter-index"><xsl:value-of select="current-grouping-
key()"/></h2>
      <xsl:variable name="max-col-entries" select="ceiling(count(current-
group()) div 3)"/>
      <xsl:for-each-group select="gd:sort-entries(current-group())"
        group-by="if (position() le $max-col-entries) then 1
            else if (position() le (2 * $max-col-entries)) then 2 else
3">
        <ul class="letter-index">
          <xsl:apply-templates select="current-group()"/>
        </ul>
      </xsl:for-each-group>
    </xsl:for-each-group>
  </xsl:template>

  <xsl:function name="gd:a-z-entries" as="element(entry)+">
    <xsl:param name="url-slug" as="xs:string"/>
    <xsl:param name="entry" as="map(*)"/>

    <entry url-slug="{$url-slug}" sort-name="{$entry?sort-name}"
name="{$entry?name}"/>
    <xsl:for-each select="array:flatten($entry?aliases)">
      <entry url-slug="{$url-slug}" sort-name="{.?sort-name}" name="{.?
name}" alias="true"/>
    </xsl:for-each>
  </xsl:function>
```

```
  <xsl:function name="gd:sort-entries" as="element()*">
    <xsl:param name="input" as="element()*"/>

    <xsl:perform-sort select="$input">
      <xsl:sort select="@sort-name" collation="http://www.w3.org/2013/
collation/UCA?strength=secondary"/>
    </xsl:perform-sort>
  </xsl:function>

  <xsl:template match="entry[@alias]">
    <li class="alias"><a href="/designers/{@url-slug}"><xsl:value-of
select="@name"/></a></li>
  </xsl:template>

  <xsl:template match="entry">
    <li><a href="/designers/{@url-slug}"><xsl:value-of
select="@name"/></a></li>
  </xsl:template>
</xsl:stylesheet>
```

The key stages are:

1. Output the page heading into the `masthead` hole.
2. To populate the `content` hole with the A–Z list, first generate a sequence of `<entry>` elements from `index.json`'s map, exploding out aliases so that each also has an `<entry>`.

   Because maps are unordered in XSLT 3, this sequence is unordered.
3. Use `<xsl:for-each-group>` to group alphabetically, by the first letter of the sort name. This gives us our A–Z buckets.
4. We want three columns of entries for each letter, so take the count of entries in a bucket, divide by 3 and round up to the next integer to get a maximum. Use `<xsl:for-each-group>` again, this time passing it a sorted sequence of entries, grouping into columns by checking if the position of the entry is <= the maximum (column 1), twice the maximum (column 2), or more (column 3).
5. Use `<xsl:apply-templates>` to generate the list items for each column, adding a different class to aliases

We use several XSLT 3-specific approaches here. Firstly, we use atomic-value template matching in order to allow the standard base layout to call apply-templates, selecting and matching on the map itself. In order to generate the exploded list of entries plus their aliases, we use `map:for-each()`, which calls the provided function for every key/value pair in the map, returning a single sequence containing the return values of the provided function. While you can use a anonymous function defined in line (`map:for-each($m, function($k,`

`$v) { … }))`, we're using higher-order functions to pass the `gd:a-z-entries()` function in. The notation `gd:a-z-entries#2` says to select the two-argument form, in order to deal with choosing between function overloads. This approach allows us to effectively combine XSLT's standard element constructors with XPath expressions, which allows for readable, idiomatic, code.

4.2. A page for the entry

The entries themselves vary wildly in length and complexity. We've already done the majority of the work needed to generate the HTML during the creation of the entry XML, but there are a few things that need to be done: Wrapping list-items in the biography section in a `<ul>`, resolving and generating links for cross-references, and generating the previous / next links.

Here's an entry that contains very little. There's no biography, and there are no cross-references, so we're just left with the previous and next links.

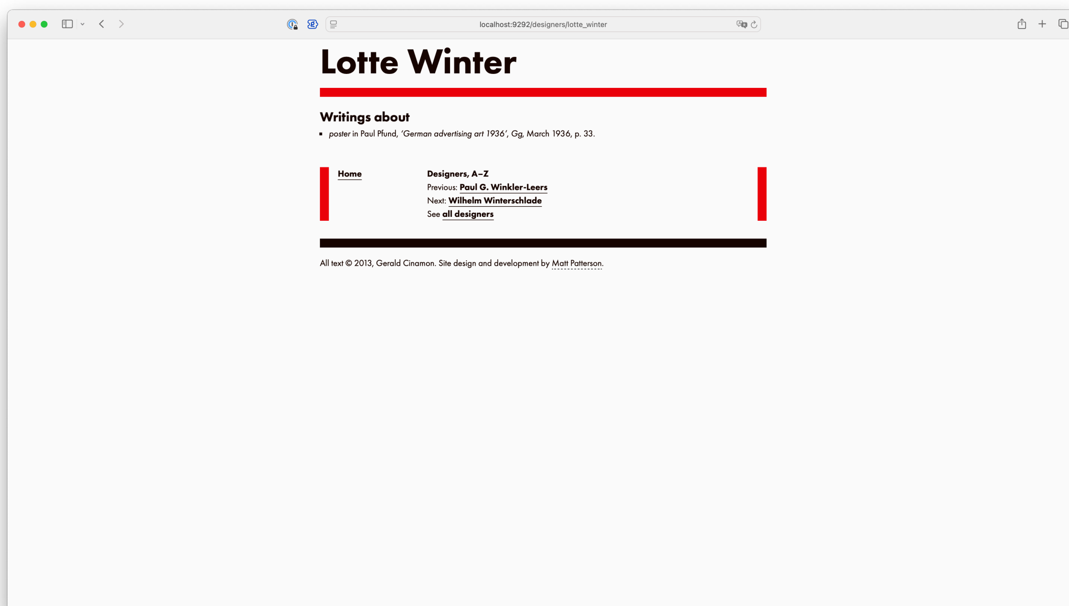


Figure 8. A short entry

To generate the previous and next links, we need to know the name and URL slug of both. While I'm sure we could do that with a query over a collection returned by `fn:collection()`, that seems a little wasteful: we'd need to perform it again for every entry. Instead, we're going to post-process our `index.json` and generate a new index that maps an entry (via its URL-slug) to the previous and next entries.

The XSLT to do this is quite compact, but demonstrates a couple of other nice XSLT 3 features, so I'll reproduce it here in full:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:math="http://www.w3.org/2005/xpath-functions/math"
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:gd="http://germandesigners.net/"
  exclude-result-prefixes="xs math map"
  version="3.0">

  <xsl:output method="json" indent="yes"/>

  <xsl:function name="gd:sorted-index" as="array(*)*">
    <xsl:param name="index" as="map(*)"/>

    <xsl:sequence
      select="sort(map:for-each($index, function($key, $value) { [ $key,
$value ] }),
        'http://www.w3.org/2013/collation/UCA?strength=secondary',
        function($item) { $item(2)?sort-name })"/>
  </xsl:function>

  <xsl:function name="gd:sibling" as="map(xs:string, map(xs:string,
xs:string))">
    <xsl:param name="rel" as="xs:string"/>
    <xsl:param name="entry" as="array(*)"/>

    <xsl:sequence select="map { $rel: map {
          'url-slug': $entry(1), 'name': $entry(2)?name }
        }"/>
  </xsl:function>

  <xsl:variable name="gd:previous" select="gd:sibling('previous', ?)"
    as="function(array(*)) as map(*)" />
  <xsl:variable name="gd:next" select="gd:sibling('next', ?)"
    as="function(array(*)) as map(*)" />

  <xsl:template match=".[. instance of map(*)]" as="map(*)*">
    <xsl:map>
      <xsl:iterate select="gd:sorted-index(.)">
        <xsl:param name="previous" as="array(*)?" select="()"/>
        <xsl:on-completion select="map:entry($previous(1),
$previous(2))"/>

        <xsl:sequence select="if (empty($previous)) then ()
```

```
            else map:entry($previous(1), map:merge(($previous(2),
$gd:next(.))))"/>

        <xsl:next-iteration>
          <xsl:with-param name="previous"
            select="if (empty($previous)) then .
                else array { .(1), map:merge((.(2),
$gd:previous($previous))) }"/>
        </xsl:next-iteration>
      </xsl:iterate>
    </xsl:map>
  </xsl:template>
</xsl:stylesheet>
```

As you can see, we're using the same map/reduce through `<xsl:iterate>` technique from the original index-generation stylesheet. Again, we have to sort the input map because maps are unordered in XPath 3. We are using it in a different way, though. We need to add the previous and next references to every entry. The brute-force approach is to use `<xsl:for-each>` to iterate over the range `1 to count($sorted-entries)`, and subtract 1 from the current value to find the previous, and add 1 to find the next. You also need special cases for the first and last items, which won't have a previous or next item, respectively. The `<xsl:iterate>` approach here removes almost all the special casing and messing about. The approach relies on `<xsl:next-iteration>` and `<xsl:map>`.

`<xsl:map>` expects a sequence of maps as the result of evaluating its body, and it then merges them all into a single map. We'll return single-entry maps from the body of `<xsl:iterate>` so that we end up with the complete index map we need. Because maps are unordered, and we need to supply the entries correctly sorted (so that we can know the previous entry) we need to use an array or sequence. `map:for-each()` makes it trivial to output a sequence of key/value pairs, as a 2-item array. In addition to the current item in the sequence we pass the previous entry map as the `$previous` param, through the `<xsl:next-iteration>` mechanism. Now we can add the current entry to the previous as its `next` relation, and output from the body of the `<xsl:iterate>`. We then add the entry from `$previous` to the current one as its `previous` relation, and then pass that to the next iteration as the value of `$previous` through `<xsl:next-iteration>`. At the end of the iteration `<xsl:on-completion>` outputs the final entry, and we're done. Obviously, the value of `$previous` is `()` for the first entry, so that first step just doesn't output anything.

The function that generates the previous and next maps is `gd:sibling()`, but we don't use that directly. Instead we use partial function application to create new function objects, stored in the `$gd:previous` and `$gd:next` variables, that keep the invocation concise. Calling `$gd:next(.)` is the same as calling

`gd:sibling('next', .)`. We could, of course, write wrapper functions, but the end result would be the same and there would be more boilerplate. This case, with a simple single-variable binding is particularly suited because we don't need to process the variable being bound.

The `<xsl:map>` then creates a single map from all the output entries, and that gets serialized to disk to be used later.

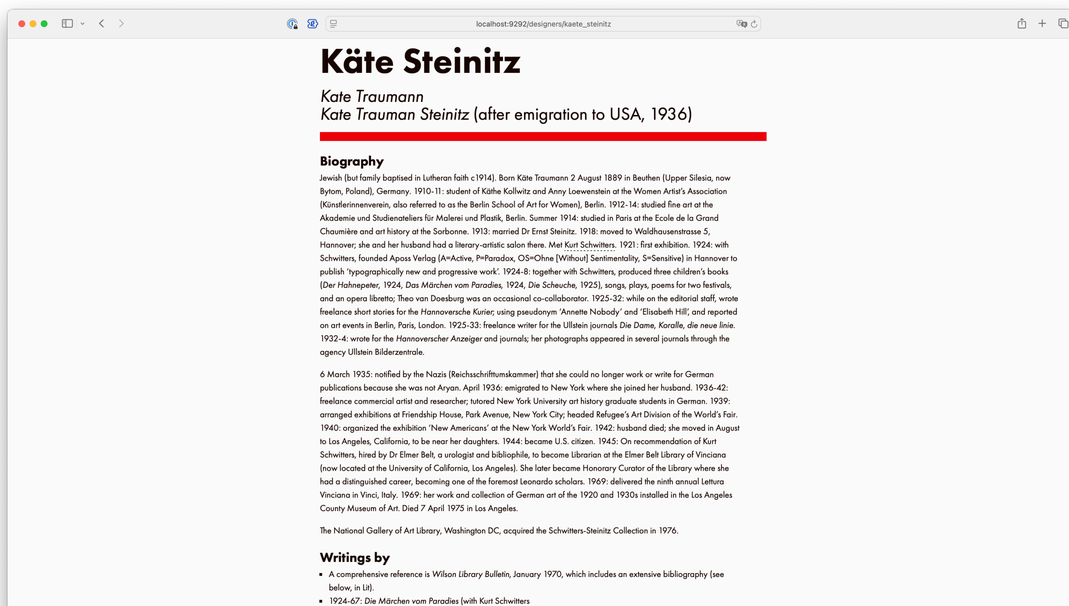


Figure 9. A long entry

We also need to look up cross-references. Again, while we could make this work with queries over an `fn:collection()`, what we're talking about is string comparison of the text content of a cross-reference `<span>` and the set of names we already have in the JSON index we produced for the A–Z page. We do that transformation in the build script, which is in Ruby [8]. That looks like this, where the collection (`index`) being operated on by `reduce` is passed an initial value, an empty map, which will be the left-most parameter in the block, `links`. The current item from `index` being evaluated is destructured to the `url_slug` and `entry` variables, added to the map in `links`, and the return value of the block is passed as the `links` parameter to the next iteration:

```
index.reduce({}) { |links, (url_slug, entry)|
  links[entry['name']] = url_slug
  if entry.has_key?('aliases')
    entry['aliases'].each { |aka|
      links[aka['name']] = url_slug
```

```
    }
  end
  links
}
```

Another map/reduce that winds up spitting out a very simple map with the name as key and URL-slug as value. With that in hand, let's start looking at the XSLT for generating the page, just including the bits relevant to the next/previous links and cross-refs for now:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:array="http://www.w3.org/2005/xpath-functions/array"
  exclude-result-prefixes="xs map array"
  version="3.0">

  <xsl:import href="../layouts/base.xsl"/>

  <xsl:param name="entry-links" as="map(*)" select="map {}"/>
  <xsl:param name="cross-refs" as="map(*)" select="map {}"/>

  <xsl:mode on-no-match="shallow-skip"/>
  <xsl:mode name="html" on-no-match="shallow-copy"/>

  <xsl:template match="span[@class = 'Crossref-designer'][$cross-
refs(string(.))]" mode="html">
    <span class="Crossref-designer"><a href="{ $cross-refs(string(.)) }">
      <xsl:apply-templates mode="#current"/>
    </a></span>
  </xsl:template>

  <xsl:template match="entry" mode="footer">
    <nav>
      <div class="nav-group">
        <h3><a href="/">Home</a></h3>
      </div>
      <div class="nav-group">
        <h3>Designers, A-Z</h3>
        <xsl:variable name="links" select="$entry-links(@url-slug)"/>
        <xsl:if test="map:contains($links, 'previous')">
          <p class="previous">Previous: <a href="/designers/{ $links?
previous?url-slug }">
            <xsl:value-of select="$links?previous?name"/></a>
          </p>
        </xsl:if>
```

```
        <xsl:if test="map:contains($links, 'next')">
          <p class="next">Next: <a href="/designers/{ $links?next?url-
slug }">
            <xsl:value-of select="$links?next?name"/></a>
          </p>
        </xsl:if>
        <p class="atoz">See <a href="/designers/">all designers</a></p>
      </div>
      <div class="clearfix"/>
    </nav>
  </xsl:template>
</xsl:stylesheet>
```

We pass the cross-ref and the next/previous index indices in as global params. They are, after all, the same dataset needed across all the entries we'll be processing. We can parse them once in the controlling program and reuse them in every transformation.

Cross-ref processing is very dumb: it just uses the text of the node as the string key into the $cross-refs map, and the template only matches if there's a match, so we don't even need to handle match and non-match behaviour, we just add a `<a href="…">` wrapper with the correct URL slug (they're relative links, so the URL-slug by itself is correct) and we're done. Some more processing needs to be done to the node text to make the lookup process more robust – currently, trailing whitespace would stop a lookup – but it demonstrates the approach.

The lookup needed for the next/previous links is much more robust – the URL-slug key is not pulled from a text node with dubious whitespace: it's exact, and unique. With the entry's links in hand, generating the HTML is a simple matter of adding `<p>`'s for the previous and next links, if they're there. Because the map keys are simple `xs:string`s whose value we already know, we can use the lookup operator `?` to pull them out: `$links?next?url-slug` being equivalent to `$links('next')('url-slug')` or `map:get(map:get($links, 'next'), 'url-slug')`.

Generating the Subject name in the masthead region is very straightforward:

```
<xsl:template match="name[parent::subject]" mode="masthead">
    <h1><xsl:apply-templates mode="html"/></h1>
</xsl:template>

<xsl:template match="aliases" mode="masthead">
    <ul class="aka">
        <xsl:apply-templates mode="#current"/>
    </ul>
</xsl:template>

<xsl:mode name="alias" on-no-match="shallow-skip"/>
```

```
<xsl:template match="alias" mode="masthead">
    <li><xsl:apply-templates mode="alias"/></li>
</xsl:template>

<xsl:template match="name" mode="alias">
    <xsl:apply-templates mode="html"/>
</xsl:template>
```

The most complex part is handling any aliases, because we need to wrap that in a list, but because of the `<aliases>` container there's a very obvious place to hang that.

Dealing with the various sections is essentially the same thing repeated several times with different headings, so we'll just look at the biography section, before looking at the `html` mode:

```
<xsl:template match="biography" mode="content">
    <xsl:call-template name="section">
        <xsl:with-param name="class" select="'description'"/>
        <xsl:with-param name="heading">Biography</xsl:with-param>
    </xsl:call-template>
</xsl:template>

<xsl:template name="section" expand-text="yes">
    <xsl:param name="heading" as="xs:string"/>
    <xsl:param name="class" as="xs:string"/>

    <div class="{$class}">
        <h2>{$heading}</h2>
        <xsl:for-each-group select="*"
            group-adjacent="if (self::li) then self::li/@class else
false()">
            <xsl:choose>
                <xsl:when test="current-grouping-key()">
                    <ul>
                      <xsl:apply-templates select="current-group()"
mode="html"/>
                    </ul>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:apply-templates select="current-group()"
mode="html"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:for-each-group>
    </div>
</xsl:template>
```

```
<xsl:template match="p|li" mode="html">
    <xsl:copy>
        <xsl:apply-templates mode="html"/>
    </xsl:copy>
</xsl:template>
```

Because the biography section con contain lists as well as standard paras, we make use of `<xsl:for-each-group group-adjacent="…">` for the last time. If the element is an `<li>`, then we use its `@class` for the grouping key, otherwise just return `false()`, which is a value we can guarantee that none of the `xs:string` class attribute values will be, and which allows us to use the delightfully simple `<xsl:when test="current-grouping-key()">` to switch between list-wrapping and normal processing.

Normal processing is also very simple, by design. In the entry XML generation phase we created content which used HTML elements, so all we need to do, unless we have a special case, is copy the nodes into the output. The special cases are cross-refs, which we've covered already, and stripping extra attributes from the `<p>` and `<li>` elements – the class attributes we pulled over from Word but no longer need now we've wrapped list items. We use `<xsl:copy>` to copy the element, but not its attributes, and go on our way - the `html` mode is `on-no-match="shallow-copy"`, so everything we don't explicitly process just gets copied over.

5. In conclusion

This paper aimed to stand as a useful behind-the-scenes exploration of all the stages of preparation and processing the German Designers project required, explaining the choices we made and showing the techniques we used, in the hope that they are useful to people.

After a number of years of stable deployment without updates, the need to move from the original dynamic website, the desire to move to statically generated pages, and the existing XSLT 2 WordprocessingML transforms, gave us the motivation, and an ideally sized playground, to explore the possibilities XSLT and XPath 3 far more thoroughly in a real-world setting than we expected. The results, in terms of the relative simplicity of the XSLT and build system code, and its speed of development, was delightful. An area like JSON processing, which has not been traditionally regarded as a strong point of XSLT / XPath was easy to integrate and allowed not only easy sharing of conveniently-queryable data structures between processing stages, but also easy interoperability with non-XSLT tooling.

We look forward to being able to explore some of the initial design ideas for the project that previously seemed too complex, now that we have tooling that fits the problem space, and tooling that permits such easy iterative development.

6. Bibliography

Bibliography

[1] Gerald Cinamon: *Gerald Cinamon, Graphic Designer and Author*. `https://geraldcinamon.com/`

[2] ICA, London: *Gerald Cinamon: Collected Work Since 1958*. `https://archive.ica.art/whats-on/gerald-cinamon-collected-work-1958/`

[3] Gerald Cinamon: *German graphic designers during the Hitler period*. `http://www.germandesigners.net/`

[4] Evan Lenz: *The WordprocessingML Vocabulary*. `https://lenzconsulting.com/wordml/`

[5] Wikipedia: *Office Open XML file formats*. `https://en.wikipedia.org/wiki/Office_Open_XML`

[6] ISO: *Office Open XML File Formats — Part 1: Fundamentals and Markup Language Reference*. `https://www.iso.org/standard/71691.html`

[7] ECMA International: *ECMA-376*. `http://ecma-international.org/publications-and-standards/standards/ecma-376/`

[8] Ruby contributors: *The Ruby programming language*. `https://www.ruby-lang.org/`

[9] Ruby on Rails contributors: *Ruby on Rails*. `https://rubyonrails.org/`