# The Art of the Framework

## Framing the problem

### Building customized editing environments and workflows

*Framing the Problem* by Wendell Piez

Balisage 2016

North Bethesda, Maryland

Wednesday August 3, 2016

Problem statement #1

# So … what kind of critter are these exactly?

"Semantic publishing"
(sometimes proprietary, shh!)

"Literate programming"

Editorial Schematron

Office Open XML (DOCX / XSLX) extraction

"Chunking" and content (re) allocation

Data conversion pipelines

XML QA frameworks

Publishing applications

Editorial / publication pipelines and support applications

Metadata crosswalks

Expense accounting application
(coordinates my accounting)

Digital Humanities Quarterly
(open-access journal)

`TIMESHEET`: homemade time logging application
(drives my billing)

Balisage Proceedings

My home-grown slide development thing …
*(makes old-fashioned PDFs via SVG, authored in XML)*

The-thing-that-has-no-name-yet,
the "Un-CMS" task tracking / ticketing application …

# A business view

## What is the unit of production?

The application artifact? the application?

or both together: the application in operation?

## What is the unit of analysis?

A “solution” is a solution to what (sort of) problem?

## What is the value offering?


Antique sewing machine in a London shop window, 2010
(Photo by the author.)

### Context

XML tools continue to improve and mature

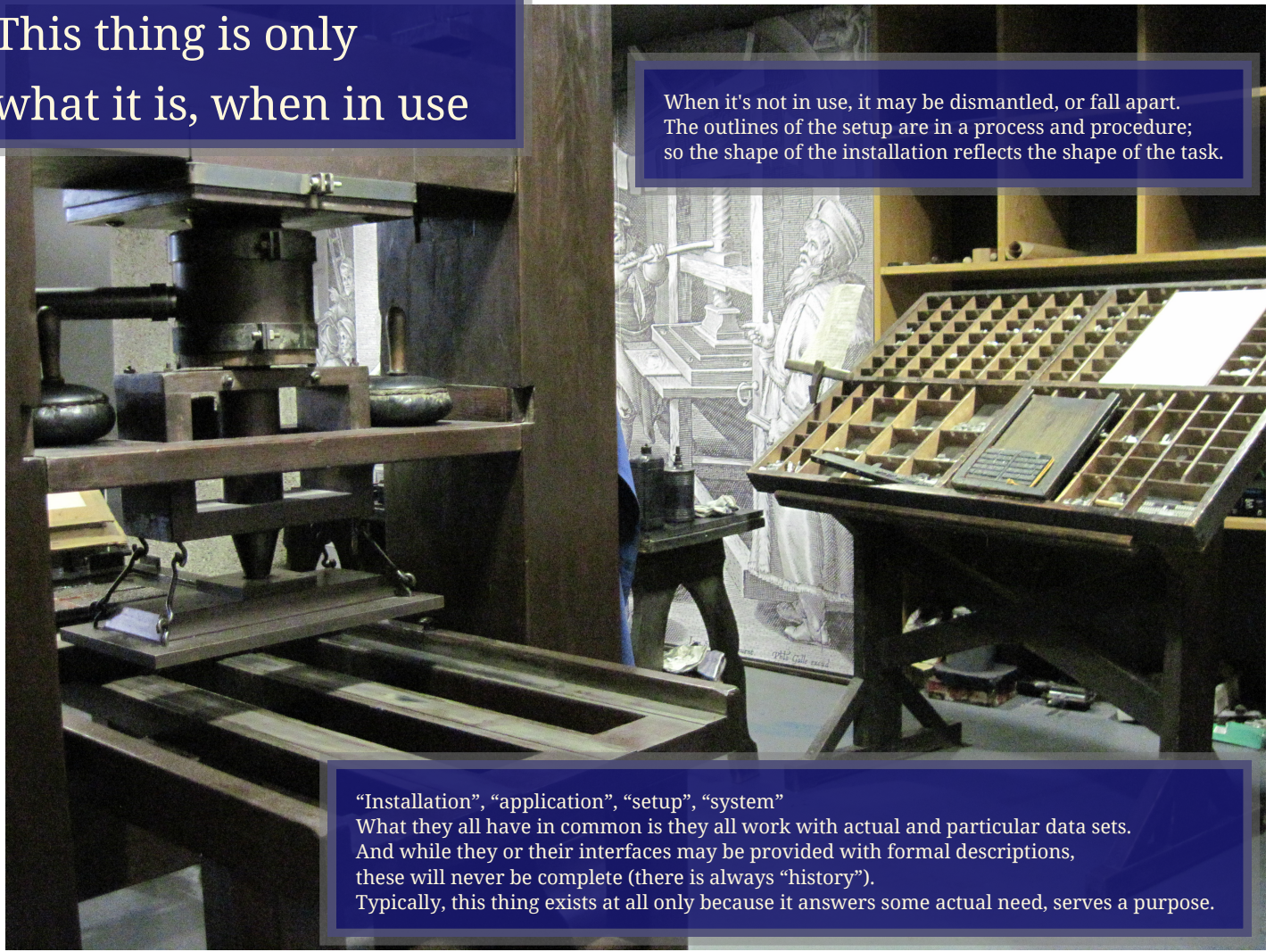
XML-based systems have demonstrated strengths

Yet some problems just don't have generic solutions

“Tailoring” is still necessary to get the best results

Is there a market for tailors?

# This thing is only what it is, when in use



When it's not in use, it may be dismantled, or fall apart.
The outlines of the setup are in a process and procedure;
so the shape of the installation reflects the shape of the task.

“Installation”, “application”, “setup”, “system”
What they all have in common is they all work with actual and particular data sets.
And while they or their interfaces may be provided with formal descriptions,
these will never be complete (there is always “history”).
Typically, this thing exists at all only because it answers some actual need, serves a purpose.

A reproduction Gutenberg press in a museum display (Gutenberg Museum Mainz). Photo by the author.

# Increase and evolution

Batch and shell scripts calling DSSSL or XSLT processor

GNU make

Shell utilities called from a text editor

XSLT+serializer writing shell scripts

Apache Ant calling XSLT etc.

Apache Cocoon + XSLT (+ shell utils)

XQuery + XSLT + XML database

XProc processor

XML IDE / environment

running CSS, XSLT, XProc etc.

(More to come)

Developments in

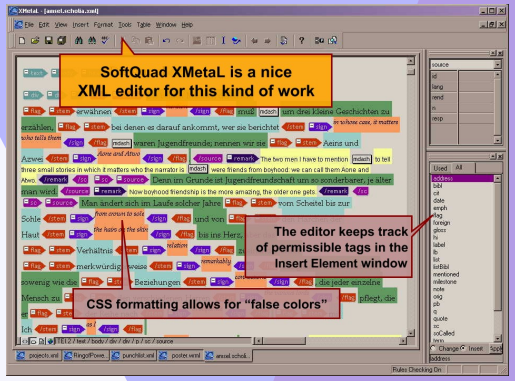
- Scale / speed / throughput
- General capability (scope of applicability)
- Front end / back end
- Ease of use / accessibility
- Portability
- Sophistication / complexity

# Continuities and convergences

Two distinct streams of development

have converged into one

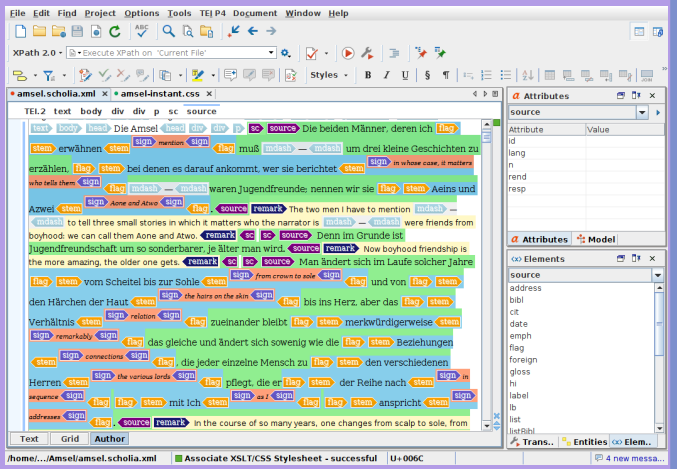
2002




Structured XML editing environments
are already mature and successfully deployed.
(Here: **XMetaL** showing a 2002 demo.)
*Out of reach for many (due to up front costs).*

*Or:* you build a homemade pipeline
Stitching together calls to the system
With ad-hoc patched-together code including
build utilities (e.g. Apache Ant)
XSLT + extensions
shell utilities
(This lineage leads eventually to xmlsh, XProc etc.)
*Cheap and agile, but has its down sides*
including long-term maintenance risks -
(these systems are stable until the day they are not)

2016



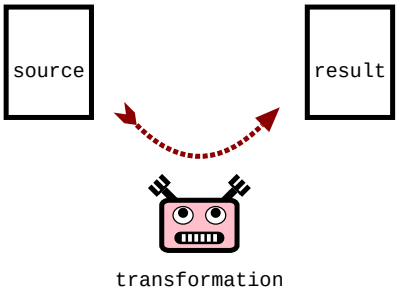
An XML IDE like oXygen XML Editor combines
the functionalities of the structured editor with
an open-toolkit approach to extensibility.
More functionality for significantly *lower* cost.
(BTW, yes, oXygen “cheats” by wiring in
a commodity toolkit.)
Also: noteworthy “free” advances in speed,
size, power, back end technology ...

# Commonalities across generations

XSLT entered very early

and is still central

(I do not believe this is entirely an accident of perspective.
Working systems that do not have XSLT, have something like it.)




Components have proven to be long-lived
*because* specifications have been published
(standardized more or less formally)
and industry-leading implementations
have tended to be rigorous.

The web is a critical enabler of this work

HTML and CSS (including CSS as applied to XML) are invaluable

Even when we aren't “publishing”

# Transformations, pipelines, and ...?

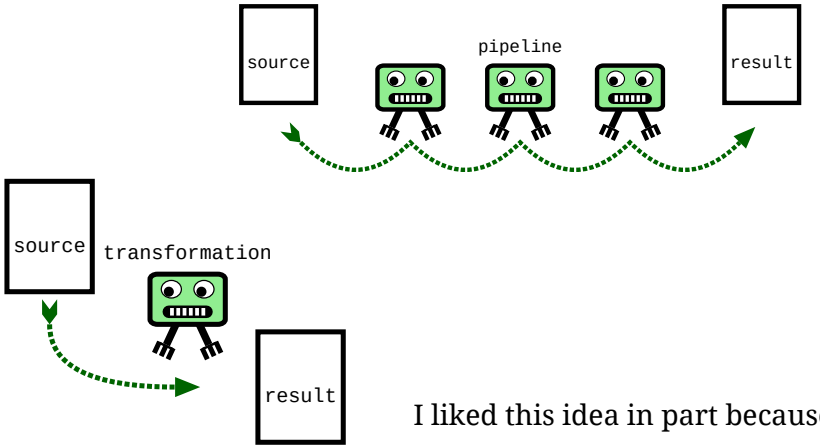
**“framework”**

Is there such a thing as a working *collection* of pipelines and transformations?

**“rig”**

Partly designed, partly configured out of available components.
Working together with ends in view, in a *workflow* of some kind.
(What we are looking at when we look at our flowcharts.)

**“assembly”**




I liked this idea in part because of its fractal quality:
the logic of the whole applies also to its parts.
And because it corresponds to what I see in the real world.

# Cost of transformation

A transformation should tend to pay for itself over more runs ...
(Leveraging the application of the logic of the type, to the instance)
... but external as well as intrinsic factors determine when and how soon this happens ...
Formula also applies to pipelines and higher-level assemblies.

***D*** = development cost
***S*** = (first time) setup cost
***count*** = piece count (runs, discrete results, or other)
***ave*** = (average) operations and "materials" per piece
(i.e. ***total operations / count***)

(average) cost / count:

$$((D + S) / count) + ave$$

Note: ***D*** can sometimes be applied (to an executing pipeline) to bring ***ave*** down.
(Composability of transformations.)

Transformation is more cost effective when piece count is high
and "materials + operations" (per piece) is low (if not near zero).

Or, when development and setup are inexpensive or already paid for,
and the results are worth the per-piece (operations) costs.

# Exploring this model

***D*** = development cost
***S*** = (first time) setup cost
***count*** = piece count (runs, discrete results, or other)
***ave*** = (average) operations and "materials" per piece

cost / count:

***((D + S) / count) + ave***

**Hidden costs**

Cost of development includes cost of specifications
i.e. defining mappings from source(s) to target(s)
or risks when they are unstated or incorrect.
Plus: costs of developing your developers!

Operations costs (per piece or aggregate) may include the cost of checking results
of any transformations that you do not trust (and all that may follow)

Also these costs say nothing about the quality of the results,
only about what it takes to produce them.

**Secrets of Success**

Lower ***D*** by relying on standards and standards-conformant toolkits
Lower ***S*** by investing (however modestly) in an environment catering to users
Lower operations and "task overhead" (***ave***) to zero whenever possible
*This may imply improvements in quality of inputs*
Distinguish automatable from non-automatable and do not burden operators with the former
*This may require refactoring a problem*
First, get the scope and requirements right
Then focus on quality and operations, not count

# What is not a rig

***… what is not a boat …?***

*A design for a boat*

*Sailing, boat building or naval science*

*Boat parts, components or fittings*

*Its cargo, passengers, sailors or shipping line*

Standards

Libraries

Schemas and validation regimes (including “frameworks”)

Applications, tools and toolkits (including “frameworks”)

“One size fits all” 80% solutions

(such as several of my projects on github)

APIs, markup languages or other interfaces

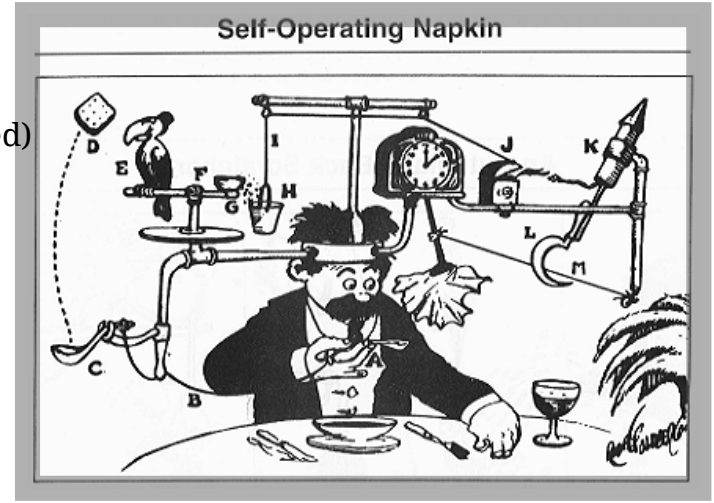
Rather, these constitute components and (when generally shared) the intellectual and technological *commons*:
A shared resource enriched by the contributions of everyone and lowering **Development**, **Setup** and **Operations** for everyone.

Interestingly, these effects are felt the most when **count** is low (“the little guy”?) even as they benefit everyone.

# How to recognize a XXXXX when you see one

Update - this is no longer always true!
"Framework" in the sense of "patchwork of routines"
has merged with framework as structured editing environment.
So, oXygen and similar applications combine what was done 15 years ago
in both XMetaL and Emacs together, for a fraction of the cost.
Entire, well-integrated environments can be designed,
assembled, and packaged for tasks both specific and general.
Plus, bells and whistles!

Embarrassing glue code
Sits on some embarrassing external dependency
Two or more languages
Little (or big) parts not implemented yet
(yet the whole thing still works)
Sometimes, Rube Goldberg complexity
(this is not a goal but it is sometimes tolerated)
Somehow the job gets done
Users view it with mixture of admiration
and skepticism, pride and fear
Sometimes, elegance in surprising places



# Paradoxes of working systems

We can focus attention on actual not just hypothetical problems and data sets.

Since focus will not be on the system as such, but on the work it supports, a successful solution disappears into the background.

(Like typography, it is working best when not noticed.)

We achieve success when we get the job done – and we actually know when that is.

It is always easy to ask your users when they are right there!

In designing and building such workflows and environments, what considerations regarding user needs, end products, interim work states, validation needs, and constraints on the system need to be taken into account?

Architecture

How can we best take advantage of existing libraries, tools, specifications, and platforms?

Environment

And how can we achieve and communicate a clear understanding of the framework we are constructing, as distinct from the tools, tool libraries, platforms and specifications which support it and help realize it?

Outlook (Planning)

# Contrary Impulses

## The perennial question

| Devil | Angel |
| --- | --- |
| **We do what we must.** | **Let's not get into trouble.** |
| **We can learn as we go!** | **Figure out the right way, then do it!** |
| **Tomorrow is another day.** | **Doing it wrong is worse than not doing it at all.** |

Describe any formal and informal (implicit) specifications of inputs

Same for outputs

Input side: what is the data acquisition model for all sources

Output side: Is there a publication model?
Who are the consumers or clients and how is data presented to them?

For each processing pipeline, describe:
- Interfaces (in and out)
- Resources (static and dynamic)
- Mapping requirements
- Any preliminary analysis
- Workflow model

Identification of failure points
- How are bugs / failures detected?
- What is the process/sequence when failures are detected?
- Would early detection help?
- Are there any points where detection/remediation
  - costs more than it is worth?

# Frame, wrap up and take away

In designing and building such workflows and environments, what considerations regarding [illegible] constraints?

Cultivate a tolerance for “organic” solutions.

Know the difference between “up” and “down” and why it matters.

Look for opportunities at the edges of the system (design matters).

Architecture

How can we best take advantage of existing libraries, tools, specifications, and platforms?

Consider impacts of environment on Setup, Development and Operations.

Support standards by using them, in public if possible.

Cultivate local technical knowhow esp “commons”-based.

If you pick your escalator correctly you can get off on any floor.

Environment

And how can we achieve and communicate a clear understanding of the framework we are constructing, as distinct from the tools, tool libraries, platforms and specifications which support it and help realize it?

Look to the goal! *Love your data.*

The framework you need is the framework that takes good care of it.

Help to frame your solution by giving it a name.

Outlook (Planning)

***Framing the Problem* by Wendell Piez**

**Balisage 2016**

**North Bethesda, Maryland**

**Friday August 5, 2016**