Introduction

In this essay, I walk through the implementation of a simple software project in a sequence of discrete design steps

- Software Architecture
- Software Engineering
- Software Implementation.

I intend to show how to apply the principles of *divide and conquer* and *superposition* to the design of a system.

I use diagrams to design each phase of the project and show how to manually convert the diagrams to code.

Background: I use draw.io to create the diagrams in this essay.

Problem

The Problem that I want to solve is using the Scrivener[1] writing tool to produce blog posts that are compatible with Github Pages.

Github Pages uses the Jekyll[2] tool and the file formats prescribed by Jekyll.

I decompose the problem into several parts:

1) Writing a blog post using the Scrivener tool

[1] Scrivener is a tool for writing books, etc. It goes well beyond what can be done with WYSIWYG editors, such as Word®. Scrivener can produce output in many formats. We will stick with .html (web page) output, for this project.

[2] Jekyll is a "static site generator", in other words, it produces a website containing only pure .html files and requires no access to "dynamic" technologies like databases. Jekyll is used to produce blogs for github pages. Github pages is a website generator that runs Jekyll whenever a new post is pushed to the appropriate repo. My blog, as an example, can be seen at guitarvydas.github.io and the corresponding repo is at https://github.com/guitarvydas/guitarvydas.github.io.

2) Magic transformer
3) Using the Jekyll tool to build a blogging website and having Jekyll add the above blog post (written using Scrivener) to the website
4) The "rest" - storing the blog in *git*, uploading the blog to github pages, etc.

Parts (1) and (3) are handled by the respective tools (Scrivener and Jekyll). Part (4) uses standard, available technology.

I need to write a program that performs step (2), which transforms the result of Scrivener into something that is acceptable as input to the Jekyll tool.

Step (2) has one input and one output:

- input: a blog post written, and saved into the file system, using the Scrivener tool
- output: a file(s) saved into the file system that is acceptable to the Jekyll tool.

In a very basic form, Fig. 1 shows this top-level breakdown.

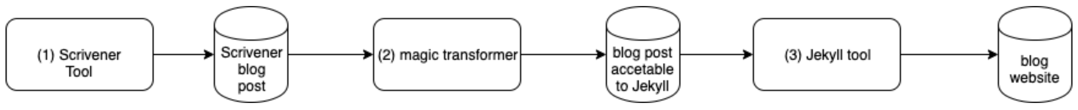


Fig. 1 Basic Architecture

We can redraw this using parts, by adding *ports* to the processes, and skipping the intermediate files, as in Fig. 2

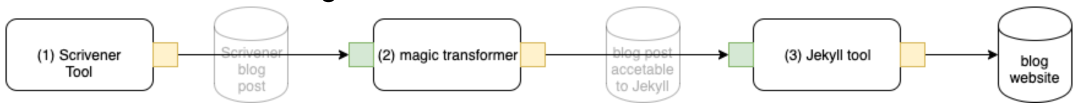


Fig. 2 Basic Architecture Redrawn

(I choose, arbitrarily, to color output *ports* yellow and input *ports* green, Side-effects appear as red *ports*.).

Before we do anything else, we need to understand what (1) produces and what (3) consumes.

Scrivener Output

The Scrivener tool is like *Word*®, on steroids.

For this example, we are only concerned with the shape of the output created by Scrivener.

In particular, we are only interested in the shape of HTML output by Scrivener.

To output a document in HTML format, we use the *File >> Compile* option, and we tell Scrivener to compile to a *Web Page (.html).*

The resulting HTML depends on

- whether the document only contains text
- whether the document contains text and .PNG files.

[*Scrivener has many more options and output formats, but for this project, we only care about HTML output and we are willing to constrain our blog posts to contain only text and .PNG files. Computer Science researchers might proceed to generalize and include more options, but, we are interested in only a practical result* and *we are willing to stick to a few conventions in order to simplify the degrees of freedom in the problem. Later, after this works, we can consider new projects and use this one as a starter template for more complicated variations. Non-programmers like the idea of pulling an existing project off of the shelf and tweaking it to produce new results. (Git shows that projects can be tweaked and that D.R.Y. in-the-large can be handled automatically)*].

Scrivener Output for Pure Text

For text-only documents, Scrivener outputs a single .HTML file.

This is shown in Fig. 3

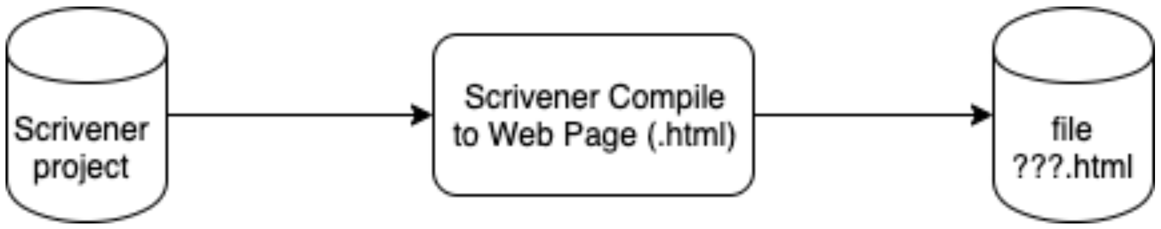


Fig. 3 Scrivener Output - Text Only Case

Scrivener Output for Text Which Includes Images

For text documents that contain images (.PNG files), Scrivener outputs a directory containing one .html file and a directory called "Images" (spelled with a capital "I" and lower case "mages"). The .html file refers to images using an href element which contains a relative reference to the .png file in the Images/ directory.

This situation is shown in Fig. 4.

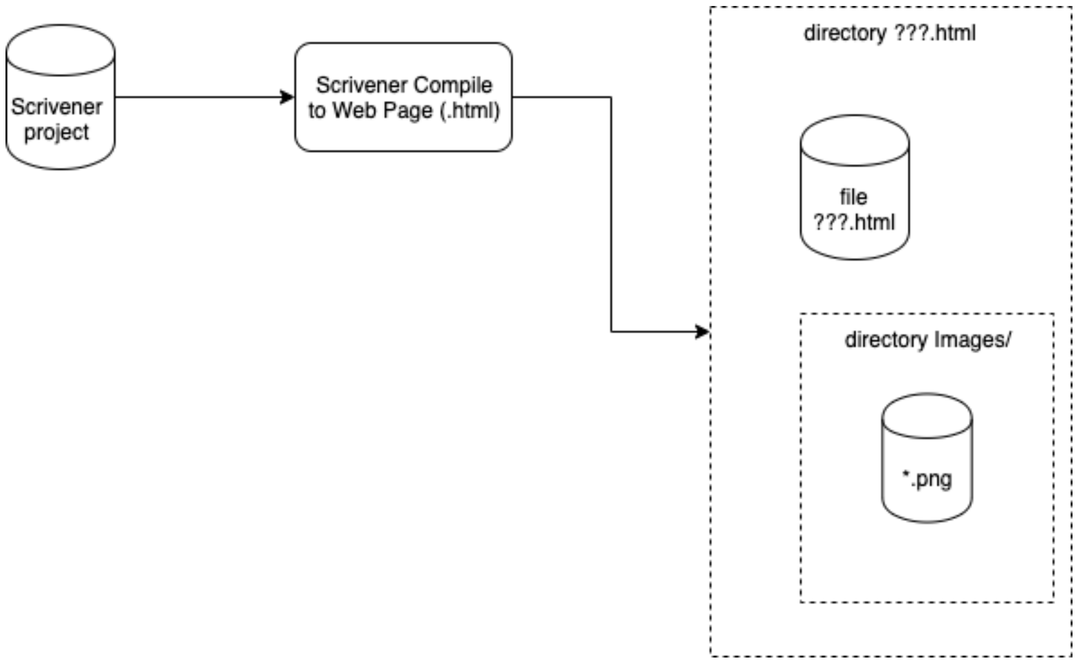


Fig. 4 Scrivener Output - Text and Images Case

Detail: the name of the directory has the .html extension, for example `abc.scriv` is output as `abc.html/`, which contains `abc.html/abc.html` and `abc.html/Images/`.

At first, this seems bizarre, but it is legal.

The choice of *where* the files are saved is made during the *Scrivener Compile* action.

Jekyll Input

Jekyll expect files to be in a subdirectory called `_posts` with each file name being prepended with the date, e.g. _posts/2021-01-09-blog.html.

This situation is shown in Fig. 5

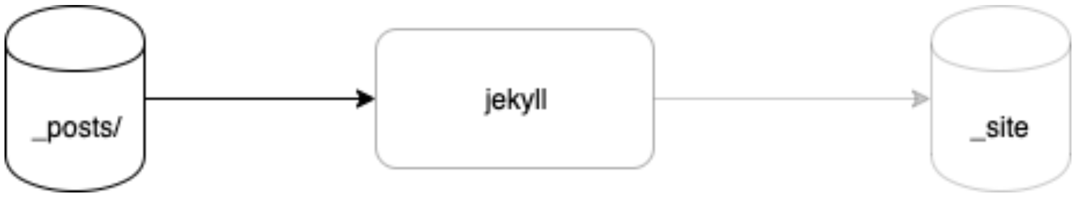


Fig. 5 Jekyll Input

[*Jekyll is usually used to create* static html sites *from blog posts. Blog posts can be raw .html files, or, (the usual case) blog posts can be markdown files, .md, containing* front matter. *When Jekyll processes .md files with front matter, it filters them using the* Liquid[3] *tool and puts the result in _site/. When processing raw .html (the case we are interested in), Jekyll simply copies the file(s) into the _site/ without processing them.*]

When a .html file contains images, it is customary to put the images in a directory called "assets" and to have the .html file refer to images in that directory, as in Fig. 6.

[3] Liquid was developed by Shopify. https://shopify.dev/docs/themes/liquid/reference

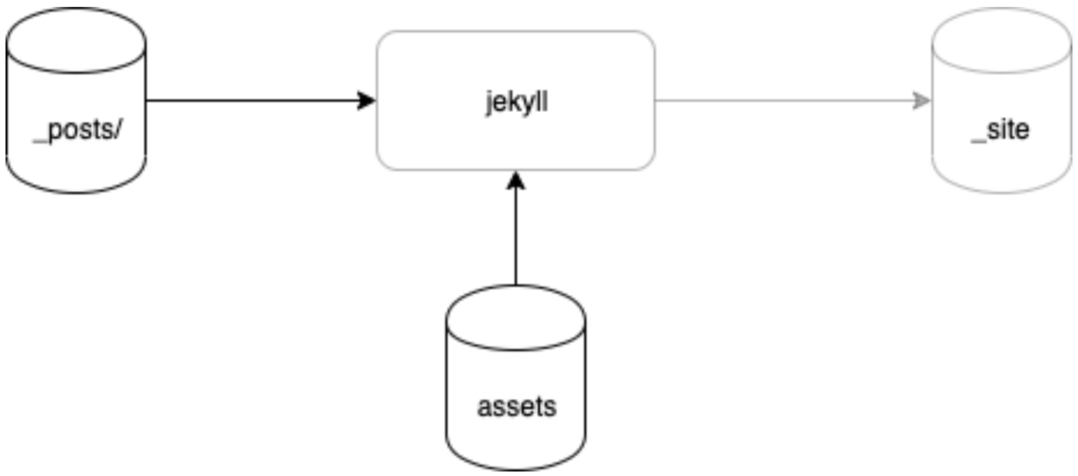


Fig. 6 Jekyll Input Using Images in Assets/

Background:: Jekyll expects a certain directory structure, https://jekyllrb.com/docs/structure/, and produces a static website in the subdirectory called "_site".

Background: Jekyll does not process any files contained in directories the are named "_*". The exceptions are "_posts", "_data", "_site", etc.

Solution

At this point, we know what the input to step (2) looks like and what output it must produce.

We can solve this problem using a layered approach, using any PL (Programming Language).

We will call the layers (1) Architecture, (2) Engineering and (3) Implementation.

Layers (1) and (2) contain no code. All of the code is only written in layer (3).

The point of layer (2) (Engineering) is to add enough detail to allow code to be written in the Implementation layer (3).

The point of layer (1) is to collect information about the problem and to arrange that information in a way that indicates the preferred solution.

We iterate between steps (1) and (2) until we know enough about the problem. "Enough" in this case means that Engineers can draw detailed diagrams ("the spec") that can be used by Implementors in step (3). [*You guessed it - if Implementors (3) don't have complete information, there is an iteration between (2) and (3). This iteration might cause new iterations between (1) and (2). Experienced Engineers know what questions to ask to keep such bubbling-up at a minimum. N.B. Agile encourages iteration between (1) and (3) in 2-week cycles. This can only work if the problem is "simple" or if Implementors are, also, Engineers (in which case their time is being wasted and the customer is over-paying high-priced talent to produce menial work). In state-of-the-art practice, all of the above stages are combined and implemented by a single person or a single team. This results in a higher cost for the overall project and tends to slow implementation (too much detail at every stage). Experienced developers break a project down "in their heads", resulting in uneven implementations that depend on the biases of the developers. Any structuring of projects works "better" than what we have.*]

The strict separation into multiple layers, like Architecture, Engineering and Implementation makes the whole enterprise work more like business organizations and allows the sub-units of the organization to be completely isolated. An organization is broken if it allows information/questions to leak between non-adjacent layers. Information that leaks through non-adjacent layers is not isolated. Strict separation stops dependencies from working their way through an organization in a spaghetti-like manner. Such leakage inhibits scaling.

Architecture

It is the Architect's responsibility to make a design clear and understandable to other readers.

An overview of what we want is in Fig. 7.

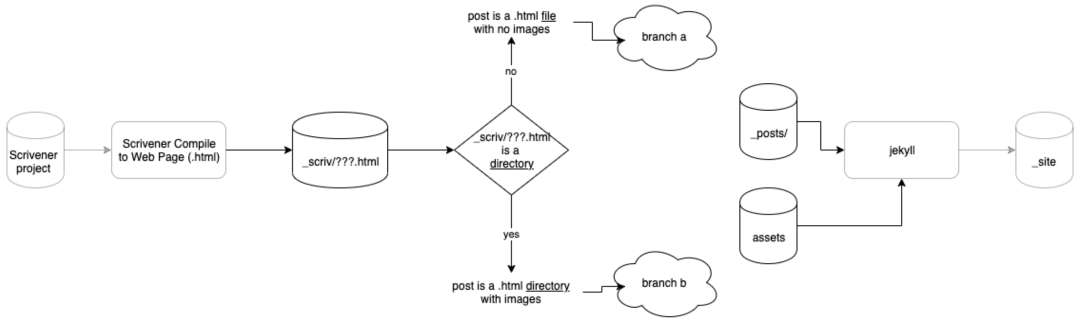

Scrivener project
Scrivener Compile to Web Page (.html)
_scriv/???.html
_scriv/???.html is a directory
no
post is a .html file with no images
branch a
yes
post is a .html directory with images
branch b
_posts/
assets
jekyll
_site

Fig. 7 Overview

As stated before, we don't have to worry about the existing tools - Scrivener and Jekyll. This project needs only to implement the stuff in the middle.

Fig. 8 shows the proposed Architecture for the stuff in the middle

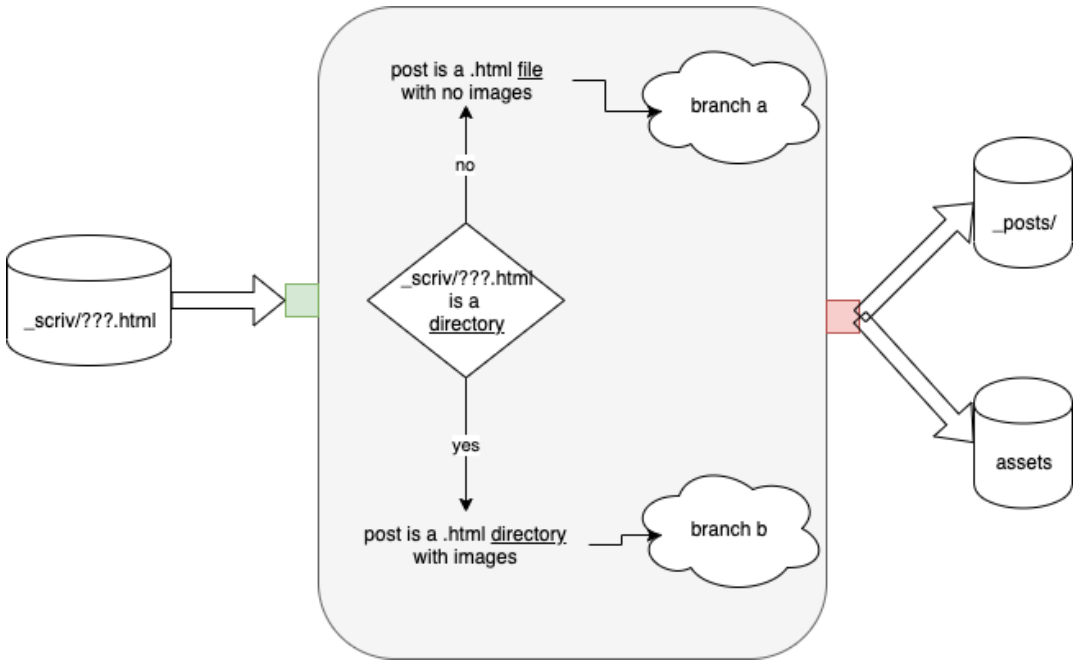

_scriv/???.html
post is a .html file with no images
branch a
no
_scriv/???.html is a directory
yes
post is a .html directory with images
branch b
_posts/
assets

Fig. 8 Architecture

We, also, need to say something about the *input port* (green) and the *output port* (yellow) and side-effects (red).

input: A filename. The filename might represent the name of (a) a .html file or (b) a directory. In case (b), the directory contains a .html file and another directory called "Images".

output: none[4]

side effects: A .html file created in the _posts/ directory and image files created in directory "assets" with unique names. The arefs in the .html file are fixed up to point to corresponding images in the "assets" directory.

[*As it stands, the above "description" is written in a* hand waving *manner. Not all terms are defined. It is unreasonable to ask the Architect for all of the details. It is the job of the Engineer to determine if there are* enough *details present to allow implementation. Some details can be interpolated by the Engineer and some can't. In the latter case, the Engineer must iterate with the Architect and, in the former case, the Engineer does not need to iterate with the Architect. For example, in a corporation, a CEO steers the ship, but does not micro-manage. Cottage industries do not scale because the CEOs (owners) do "everything" and, effectively, micro-manage every aspect of the business. Large corporations scale up (to being large corporations) only when micro-management is expunged. To ready a small business for scaling, the owner must chop up the work into identifiable layers, then, hire workers to perform work in each layer, allowing the workers to concentrate on the work in his/her layer only (instead of "trying to do everything").*]

Engineering

Engineers are responsible to produce enough detail to allow implementation.

Engineers are responsible to display details in a way that is understandable to the implementors.

Engineers off-load thinking, about minutiae, from the Architect(s).

Engineers try to interpolate details for the design.

When interpolation does not work, or requires extrapolation, Engineers

[4] Earlier, I said that this project had one output. I was simplifying.

interace with Architects to glean further details.

Engineers have the experience and training, to think about details without writing code. Currently, Universities train Software Professionals to become Implementors. Training for Software Architecture and Software Engineering does not exist and is expected to "rub off" through experience and involvement in many projects.

In other fields, say Construction, Universities can train Architects and Engineers in only a few years. Implementors (construction workers) are trained in vocational schools or through apprenticeship.

Back to this example. In this specific example, the Architect has indicated that two branches need to be considered:

- Scrivener documents that contain text only
- Scrivener documents that contain text and images.

Abstraction is a form of *optimization*[5] that is taught in Universities. Architects, and Engineers, must concentrate on the problem-at-hand and <u>strip abstraction away</u> from the instructions given to implementors.

For now (maybe forever) we will continue with the sub-divisions - branch A (text only) and branch B (text plus images) - suggested by the Architect.

The Architect has suggested a form of Divide and Conquer.

[5] One form of this optimization is often called D.R.Y. (abbreviated as DRY). DRY is appropriate for Maintenance Engineering, Test Engineering and Implementation and Maintenance, etc., but, DRY is *not* appropriate for Architecture and Realization Engineering. As an example, Architects and Engineers need to indicate that a *solution* is *like* some other solution with changes (see Paul Bennett's Framing Software Reuse). Inheritance, as currently implemented, does not capture this kind of relationship in an easy-to-understand manner.

Considering only one branch - branch A, text only - we see that two operations need to occur

- create a file prefix, where the prefix begins with the date in YYYY-MM-DD- form, and,
- move the newly-named .html file into the _posts directory.

Considering only the other branch - branch B, text plus images - we see the following break down:

- create a file prefix, in YYYY-MM-DD- format
- edit the arefs in the .html file to refer to images in the assets/ directory
- move the .html file into the _posts/ directory
- move the images into assets/.

Furthermore, at this Engineering layer, we will make the decision that we will keep the Images/ directory intact, but will prepend it with the prefix and move it, whole, into the assets directory.

This is interpolation. The Architecture is not changed by this decision. The Engineer does not need to iterate with the Architect regarding this decision.

More interpolation: the Engineer looks at the .html format generated by Scrivener and notices that the file-creation-date is embedded in the file as meta-data. The Engineer decides (arbitrarily) that this file-creation-date will be used to generate the required file prefix. In the UNIX® file system, it is possible to get the file creation date from the operating system, but the Engineer decides not to use this and decides to use the Scrivener meta-data instead. This decision is made by the Engineer on a "gut feel", knowing that various operating systems provide such creation-date information in various forms. It is deemed "better" by the Engineer to let Scrivener worry about this aspect.

I show diagrams of these branches, below. I leave grayed-out portions on the diagram that act as reminders of what the ultimate goal is. The grayed-out portions are not required in the Implementation.

Top Level

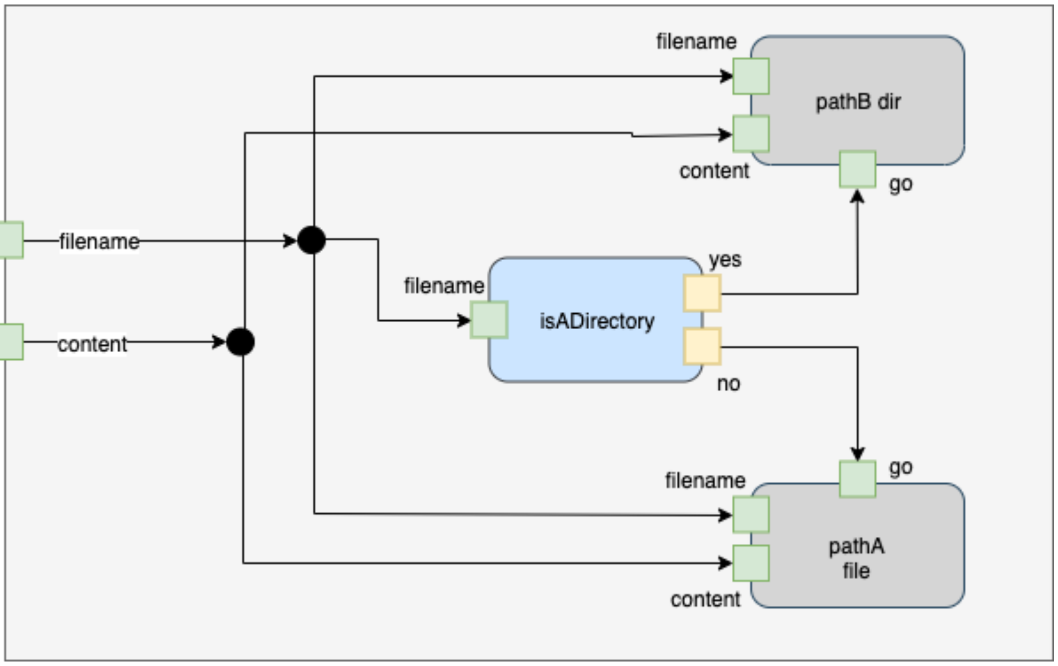


Fig. <$n:figure:Engineering Top Level> Engineering Top Level

The Engineer has created a top level diagram that contains

- one blue box (Leaf) called "isADirectory'
- one gray box (Schematic) called "pathA"
- one gray box (Schematic) called "pathB"
- one gray box that is the composition of the above boxes
- two Dots (wire splitters) that are used internally to route "filename" and "content".

In the most extreme case, this implies that six (6) things need to be implemented:

- 2 x wire splitter
- 1 x "isADirectory"
- 1 x "pathB"
- 1 x "pathA"
- 1 x top level.

[*N.B in Arrowgrams, dots are so common that we build them into the Arrowgrams* *notation, whereas in Bash, we will need to build dots explicitly. In FBP, dots are not allowed (they can, though, be explicitly implemented as FBP Components). In Electronics, dots are common and are shown as dots on a schematic diagram (whereas cross-overs are shown only as two wires crossing over, without a dot). Arrowgrams is based on ideas from Electronics drawings. Arrowgrams "borrows" from Electronics. This is a high-level form of reuse.*]

The gray boxes, e.g. pathA and pathB, might be broken down further, later. We might add new Parts to the list of things that need to be implemented, later.

is A Directory?

A simple function within the top level is to determine whether a given filename represents a file or a directory.

The Engineer does not need to break this down any further, since this functionality can be done in *bash*. If the implementation language does not make this easy, the first-cut implementation can just shell-out to a *bash* script.

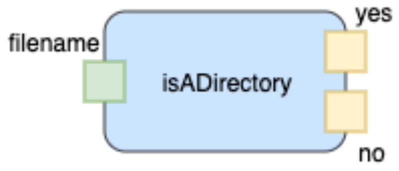
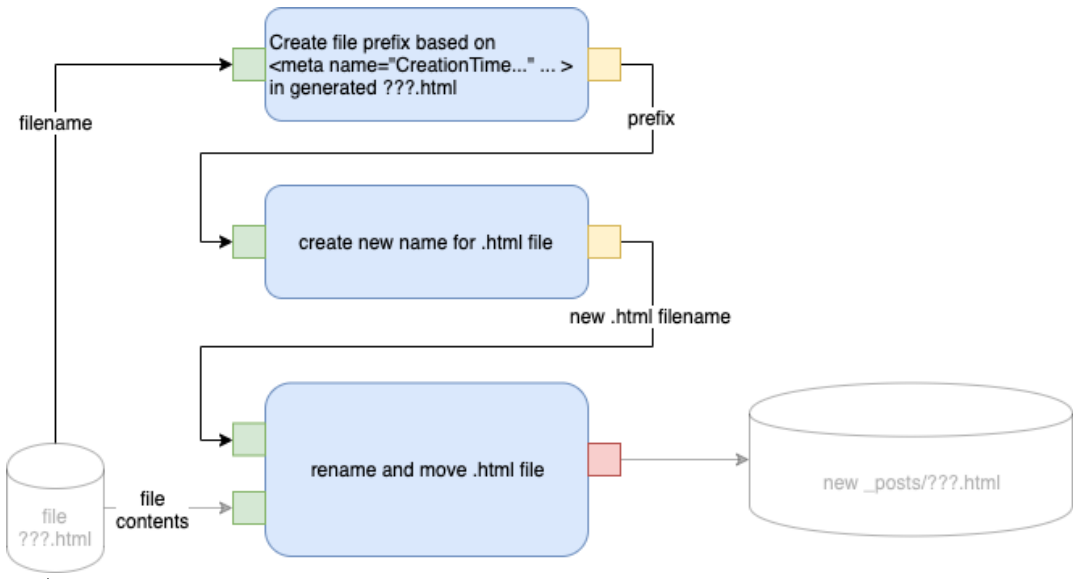


Fig. 9 Part: Is A Directory?

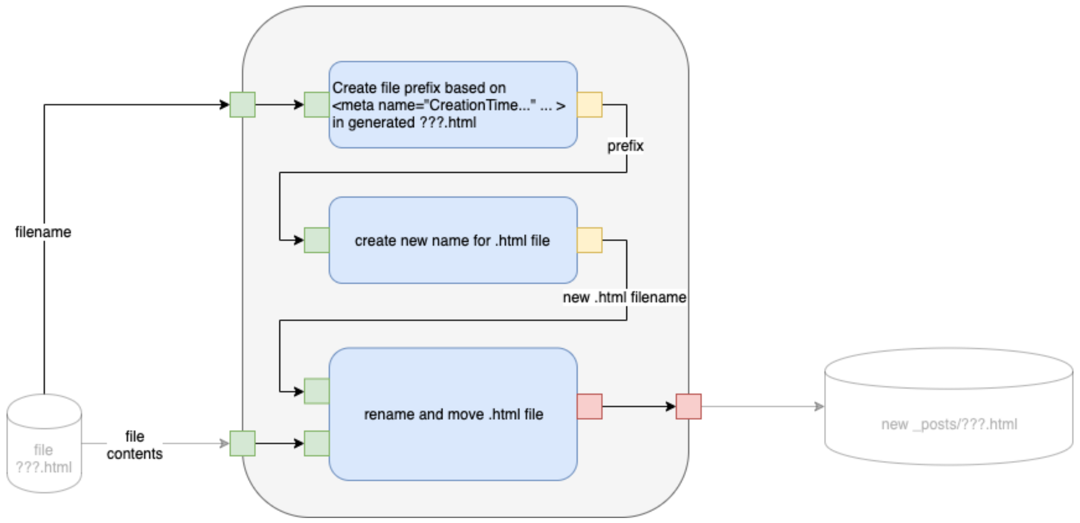
Branch A - Text Only



Create file prefix based on <meta name="CreationTime..." ... > in generated ???.html
filename
prefix
create new name for .html file
new .html filename
rename and move .html file
file ???.html
file contents
new _posts/???.html

Fig. <$n:figure:Engineering Path A> Engineering Path A

The Engineer refines this diagram further…



Create file prefix based on <meta name="CreationTime..." ... > in generated ???.html
filename
prefix
create new name for .html file
new .html filename
rename and move .html file
file ???.html
file contents
new _posts/???.html

Fig. 10 Engineering Path A Refined

"Blue" boxes mean "code".
"Gray" boxes mean composition of blocks. "Gray" boxes are call Schematics.

N.B. there are four (4) boxes to be implemented:

1. "create file prefix" (blue box)
2. "create new name for .html file" (blue box)

3. "rename and move .html file" (blue box)
4. compose the above into a Schematic part (gray box)

The Engineer creates a black box from the above:

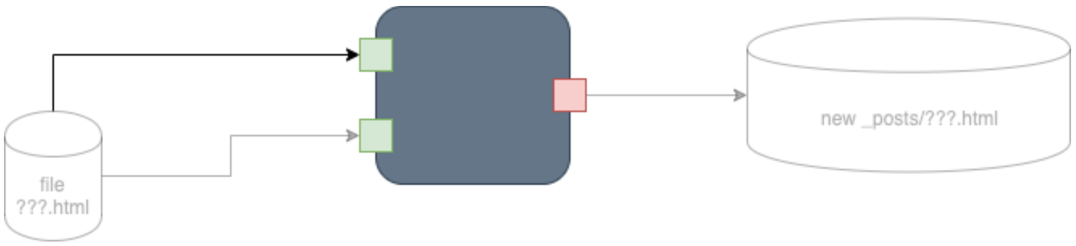
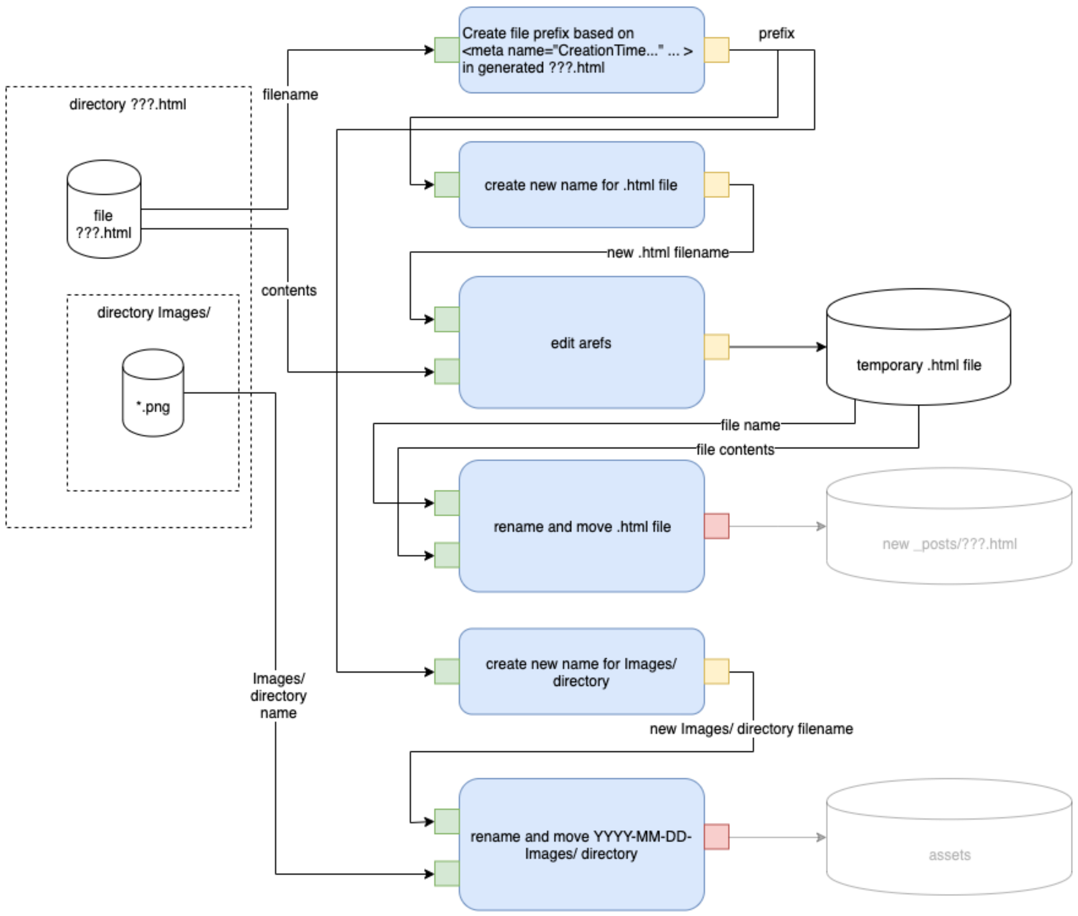


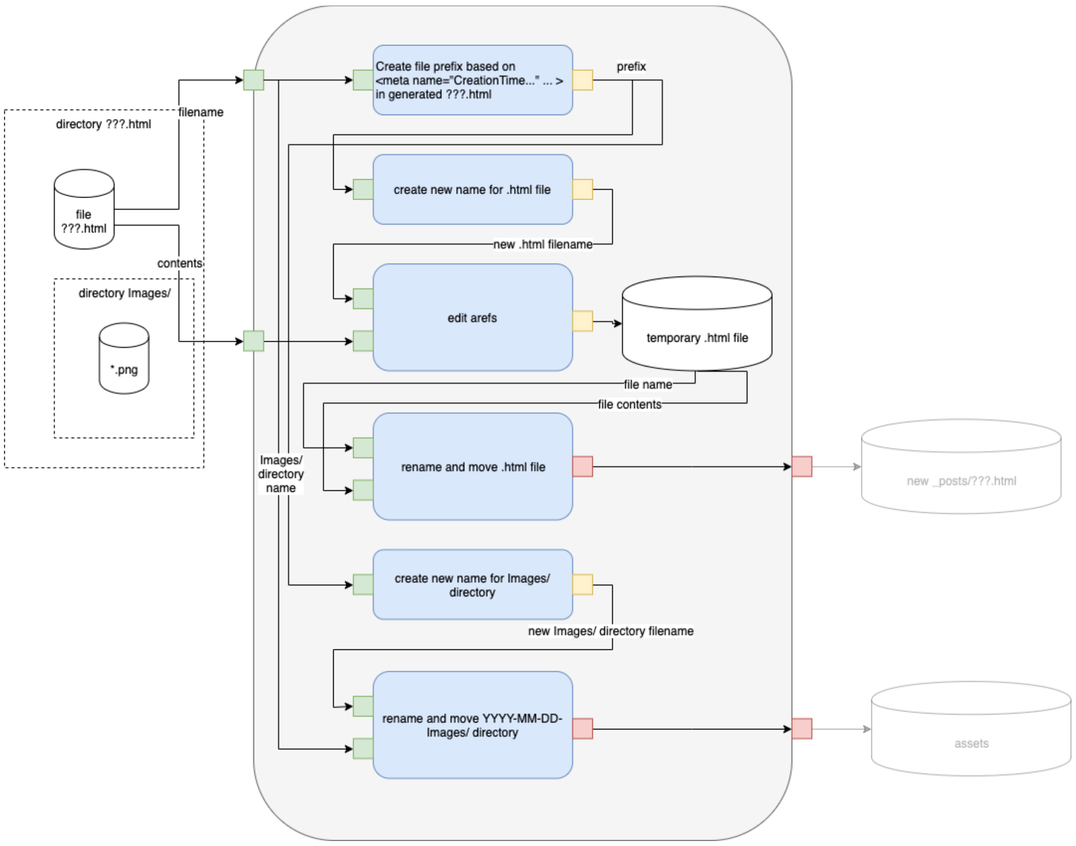
Fig. 11 Black Box Engineering Path A

Branch B - Text Plus Images



Create file prefix based on <meta name="CreationTime..." ... > in generated ???.html
prefix
filename
directory ???.html
file
???.html
create new name for .html file
new .html filename
contents
directory Images/
edit arefs
temporary .html file
*.png
file name
file contents
rename and move .html file
new _posts/???.html
create new name for Images/ directory
Images/
directory
name
new Images/ directory filename
rename and move YYYY-MM-DD-Images/ directory
assets

Fig. 12 Engineering Path B Refined



directory ???.html
file ???.html
filename
contents
directory Images/
*.png
Create file prefix based on <meta name="CreationTime..." ... > in generated ???.html
prefix
create new name for .html file
new .html filename
edit arefs
temporary .html file
file name
file contents
rename and move .html file
new _posts/???.html
Images/ directory name
create new name for Images/ directory
new Images/ directory filename
rename and move YYYY-MM-DD-Images/ directory
assets

Fig. 12 Engineering Path B Refined

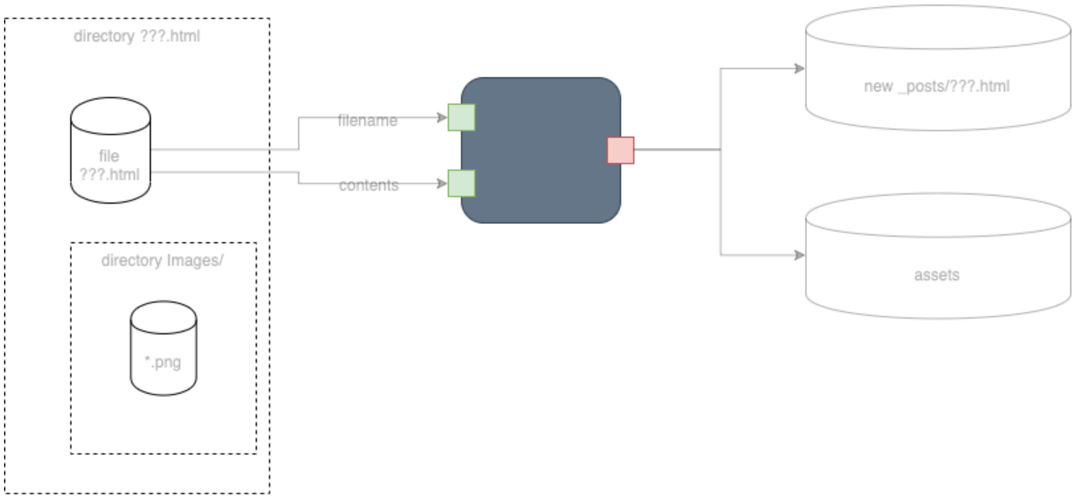


Fig. 13 Black Box Engineering Path B Refined

Implementation

At this point, we have enough of a design, and code can be implemented.

Q: The Implementor asks the Engineer whether the temporary file, specified in Branch B, needs to be made available.

A: In this design, the temporary directory does not need to be visible to the outside world, although it might assist in debugging the logic. Intermediate results, stored in human readable form, can help.

Given that the temporary file is not actually needed, the Implementor can choose any base language for the implementation.

In this particular example, we will use *bash* and write scripts that represent the Parts. Parts must be completely isolated from one another and need to run as asynchronous processes. We will implement *wires* as named pipes. We will implement *ports* using file redirection.

It turns out that UNIX® and Bash can create isolated processes, but, running the processes in an asynchronous manner is slightly tricky. We will need to use Bash "&" in many places.

Bash scripts use a *rendezvous* protocol for starting up pipes - both ends must exist before the script can proceed. This means that Bash scripts will appear to "hang" if both ends of a pipe are not attached to processes. This will drive us towards needing an "&" on just about every line of the Bash scripts.

Background: in modern forms of UNIX®, e.g. Linux and MacOS, file descriptors (FDs) can be accessed like files, e.g. `/dev/fd/3` refers to FD 3, `/dev/fd/4` refers to FD 4, and so on. Pipes can be bolted on to FDs using Bash file redirection syntax.

Background: Bash provides a `read` builtin command, and assigns the incoming data to a Bash variable, for example:

```
read -u 3 xxx
```

Reads from FD 3 and assigns the data to Bash variable ${xxx}.

This same action can be performed by referencing the FD with file redirection, e.g.

```
read xxx </dev/fd/3
```

Both methods[6] will be used, depending on our mood on a given day.

Parts will be built as bash scrips and FDs will be assigned using redirection. For example:

```
./create-file-prefix 3<${wire1a} 4<${wire2a} 5>${wire3} &
```

This says that Part "create-file-prefix" has two inputs and one output. The inputs come from FD[3] ad FD[4] whereas the output is sent to FD[5]. We use bash variable ${wire1a}, ${wire2a} and ${wire3} to represent the named pipes.

Background: to implemented Arrograms semantics - a Part is not "done" until all of its children are "done" - we use bash `wait`.

We will avoid using FD[0], FD[1] and FD[2], since these have special meaning to UNIX® commands.

The above invocation of Part "create-file-prefix" demonstrates a vital property

[6] More methods of doing this in bash exist. For example, we could use backtick syntax.

of the Arrowgrams way of thinking. The part, itself, can not know where it is getting input from and where it is outputting to. Every Part can only refer to its own array of file descriptors (FDs). A Part cannot call other parts by name.

Let the previous statement "sink in".

Most PLs currently bake the callee's name into the code, e.g. xyz(…), and this inhibits flexible Architecting.[7]

All Parts get inputs from FDs and send their outputs to FDs[8].

For debugging, we call scripts on entry and exit in each part. The on-the-fly data used for debugging is supplied by a string passed in as $1. Concatenation of debug information is done by using $0 (the command's name) in addition to the string given in $1. For example, the above code is written as:

```
./create-file-prefix "[$1] $0" 3<${wire1a} 4<${wire2a} 5>${wire3}
&
```

No Connection - NC - must be handled explicitly in bash. We implement a

[7] Even worse, the return address is baked into the code using a dynamic call-chain, called the *Stack*.

DLLs try to solve this direct-call problem using indirection, but still allow the *Stack* to determine returns.

Current functional PLs are attacking this problem by disassociating definition from implementation. UNIX® was capable of doing this in the 1970's.

FP ignores practical issues like divide and conquer. Peter Lee showed how to apply divide and conquer to Denotational Semantics, in 1989. https://www.amazon.ca/Realistic-Compiler-Generation-Peter-Lee/dp/0262121417

[8] Schematic Parts work the same way as Leaf Parts. One cannot tell how a Part is implemented - as a Leaf or as a Schematic - when using a Part.

Part called nc as a (simple) bash script that reads its input port and does nothing with the data. (Redirecting to /dev/null works only for output NCs, not input NCs, so we implement a Part to perform the no-op action).

Wire-splitters - dots - need to implemented as explicit Parts.

Normally, dots would be implemented by the Arrowgrams IDE and wouldn't need to be explicitly implemented.

We stick to the strategy of avoiding generalization, so we build two kinds of wire-splitters, one that splits[9] an incoming *event* onto two output wires and another that splits the *event* to three output wires.

In both cases, Branch A and Branch B, the Implementation of creating a new filename is trivial and the 2nd box is dropped.[10]

[9] In electronics, splitting signals involves only the calculation of *impedances.* In software, splitting signals involves copying. In this project, we use bash and get copying "for free". If we were to implement Parts in C, though, we would have to worry about the implications of copying (shallow copy, deep copy, pass-by-reference, etc).

[10] In fact, this might be a case of premature and unnecessary optimization on my part. The bash code for Branch A and Branch B was written early in the project, before all of the diagram-to-bash kinks had been worked out. This kind of change should have been iterated with the Engineer. In this case, the project was so small that I executed all tasks in this project. I was the Architect, Engineer and Implementor. I cut corners in Engineering because I Implemented the bash scripts. Had I exercised more discipline, this optimization might not have occurred. In a larger project, different people would fill the roles of Architecting, Engineering and Implementation - which would have put a damper on such egregious changes.

In a full IDE, which converted diagrams to code, this optimization would not have occurred (it is possible to transpile diagrams to code, but this will be the

Annotated Diagrams

Annotations - the Implementor puts a name on every wire and an FD number on every pin.

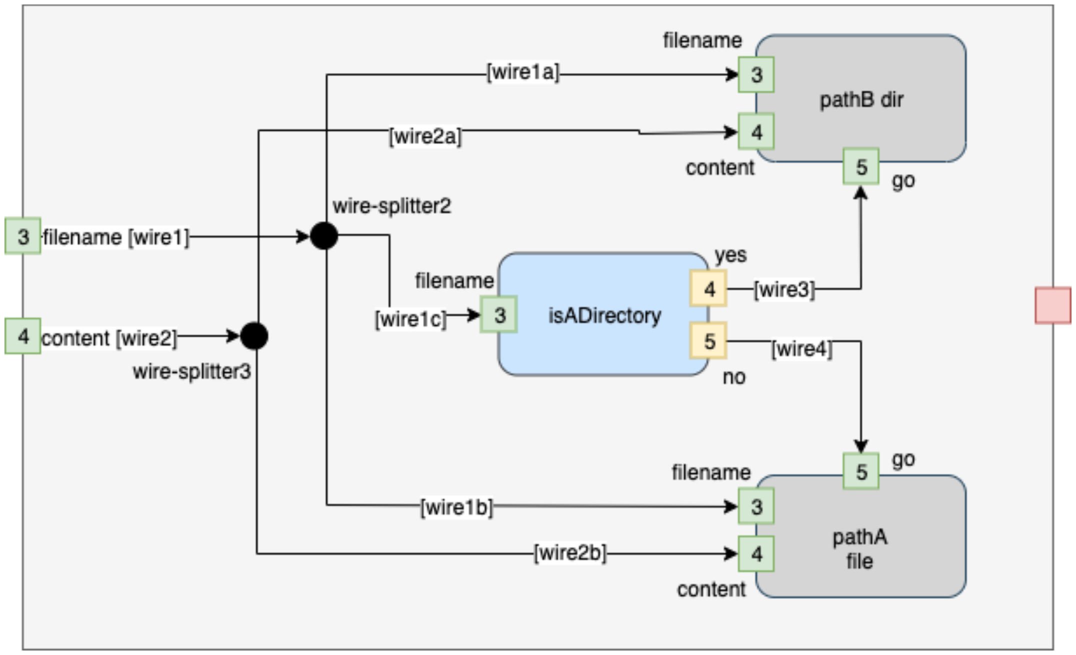


Fig. 14 Annotated Top Level Diagram

subject of another essay. See the section "Extra Marks" for clues on how transpilation from diagrams to code might be accomplished.

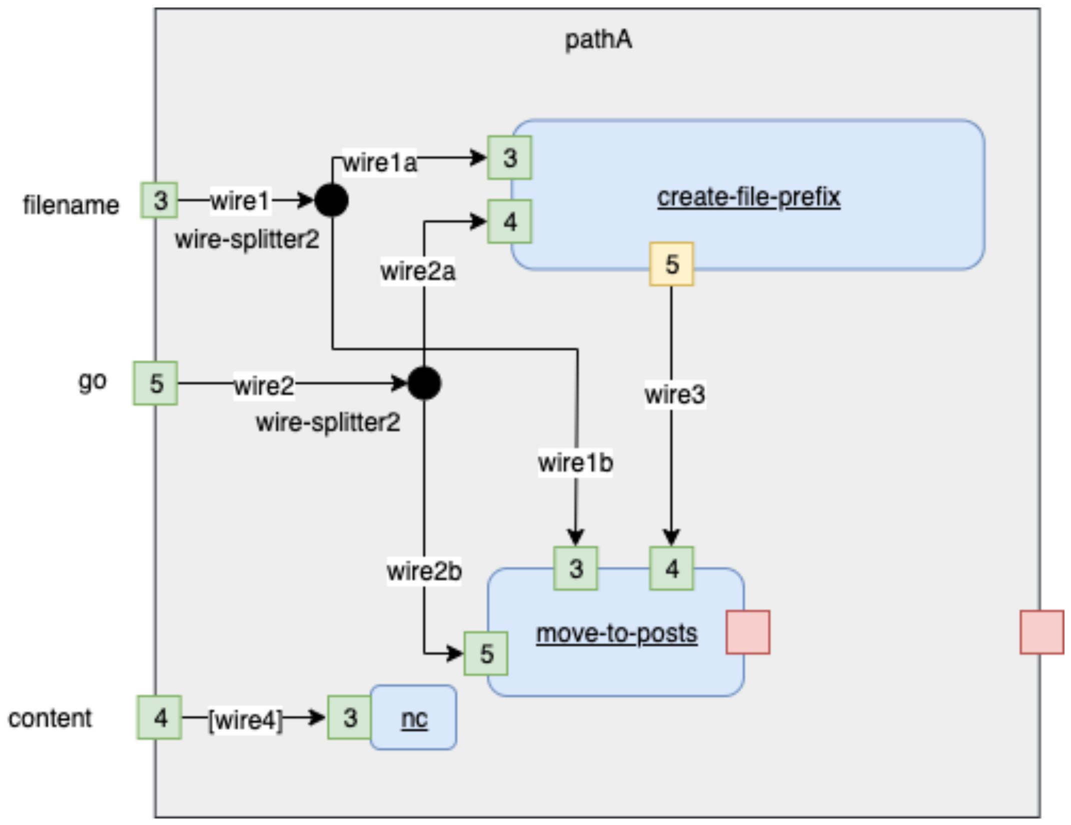


Fig. 15 Annotated Diagram for Branch A

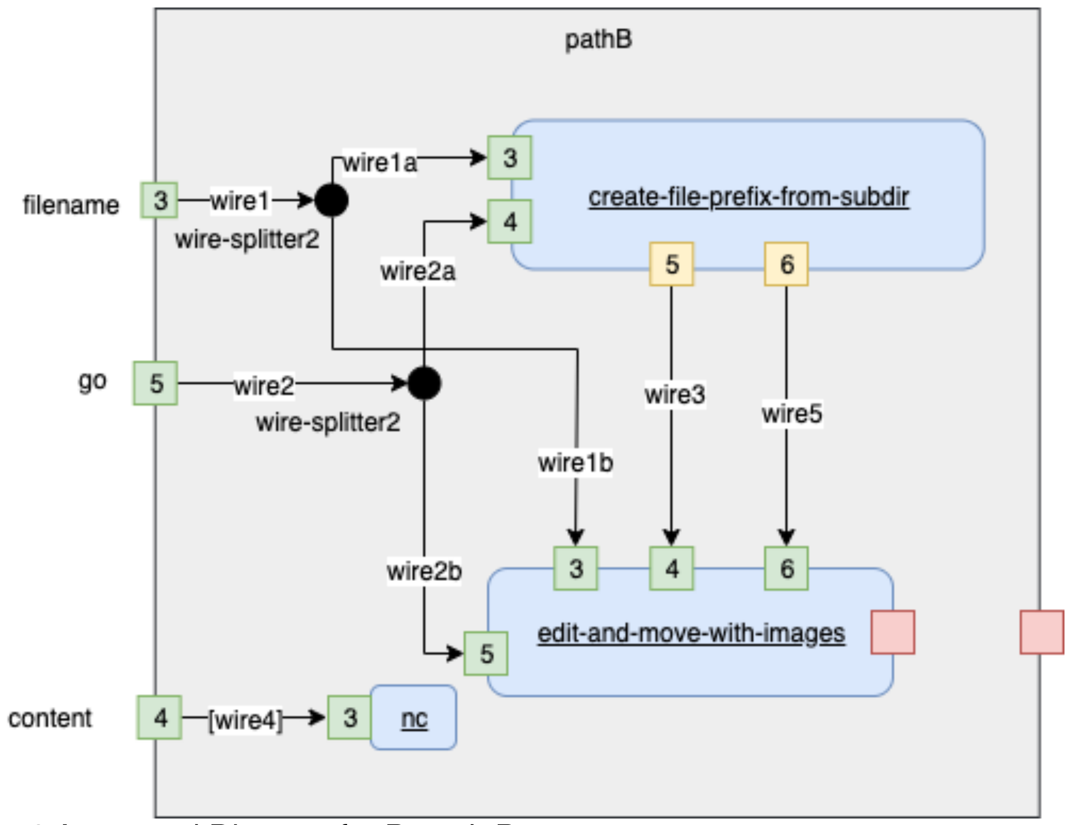


Fig. 16 Annotated Diagram for Branch B

BOM

In *electronics* design, one creates a "bill of materials", called a BOM.

The BOM for this project consists of 9 Parts:

- nc
- wire-splitter2
- wire-splitter3
- create-file-prefix
- move-to-posts
- create-file-prefix-from-subdir
- edit-and-move-with-images
- pathA
- pathB.

Code

The parts, implemented as bash scripts can be found in

NC https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/nc
wire-splitter2 https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/wire-splitter2
wire-splitter3 https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/wire-splitter3
create-file-prefix https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/create-file-prefix
move-to-posts https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/move-to-posts
pathA https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/pathA
create-file-prefix-from-subdir https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/create-file-prefix-from-subdir
edit-and-move-with-images https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/edit-and-move-with-images
pathB https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/pathB

I include the code, for the parts, below. The code is in branch AEI of `https://github.com/guitarvydas/scrivener-to-jekyll`.

nc

```
#~/bin/sh
read junk </dev/fd/3
```

wire-splitter2

```
#!/bin/bash
./debug-entry "[$1] $0"
read data </dev/fd/3
echo "${data}" >/dev/fd/5
echo "${data}" >/dev/fd/4
./debug-exit "[$1] $0"
```

wire-splitter3

```
#!/bin/bash
./debug-entry "[$1] $0"
read data </dev/fd/3
echo "${data}" >/dev/fd/4
echo "${data}" >/dev/fd/5
echo "${data}" >/dev/fd/6
./debug-exit "[$1] $0"
```

create-file-prefix

```
#!/bin/bash
#
```

```
    ## inputs:
    ## filename (fd 3)
    ## go (fd 4)
    #
    ## outputs:
    ## prefix (fd 5) (YYYY-MM-DD)
    #
    ./debug-entry "[$1] $0"


    read filename </dev/fd/3
    read go </dev/fd/4


    if [ "go" == "${go}" ]
    then
       prefix=`date "+%Y-%m-%d"`
       # error check needed here - filename should not be "" at this
point
       creation_time=`grep "<meta name=\"CreationTime\"" "$
{filename}"`
       if [ ! -z "${creation_time}" ]
       then
        prefix=`echo "${creation_time}" | sed -e 's/<meta
name="CreationTime" content="\(202.-..-..\)T..:..:..Z">/\1/'`
       fi
       echo "${prefix}" >/dev/fd/5
    else
       echo "???" >/dev/fd/5
    fi
    ./debug-exit "[$1] $0"
```

move-to-posts

```
#!/bin/bash
#
## inputs:
## filename (fd 3)
## prefix (fd 4)
## go (fd 5)
#
## outputs:
## <none> - causes side-effect of mv'ing prefix/filename
to ./_posts/
#
./debug-entry "[$1] $0"
read filename </dev/fd/3
read prefix </dev/fd/4
read go </dev/fd/5
if [ "${go}" == "go" ]
then
   basename=`basename "${filename}"`
echo mv "${filename}" "../_posts/${prefix}-${basename}"
   mv "${filename}" "../_posts/${prefix}-${basename}"
fi
./debug-exit "[$1] $0"
```

pathA

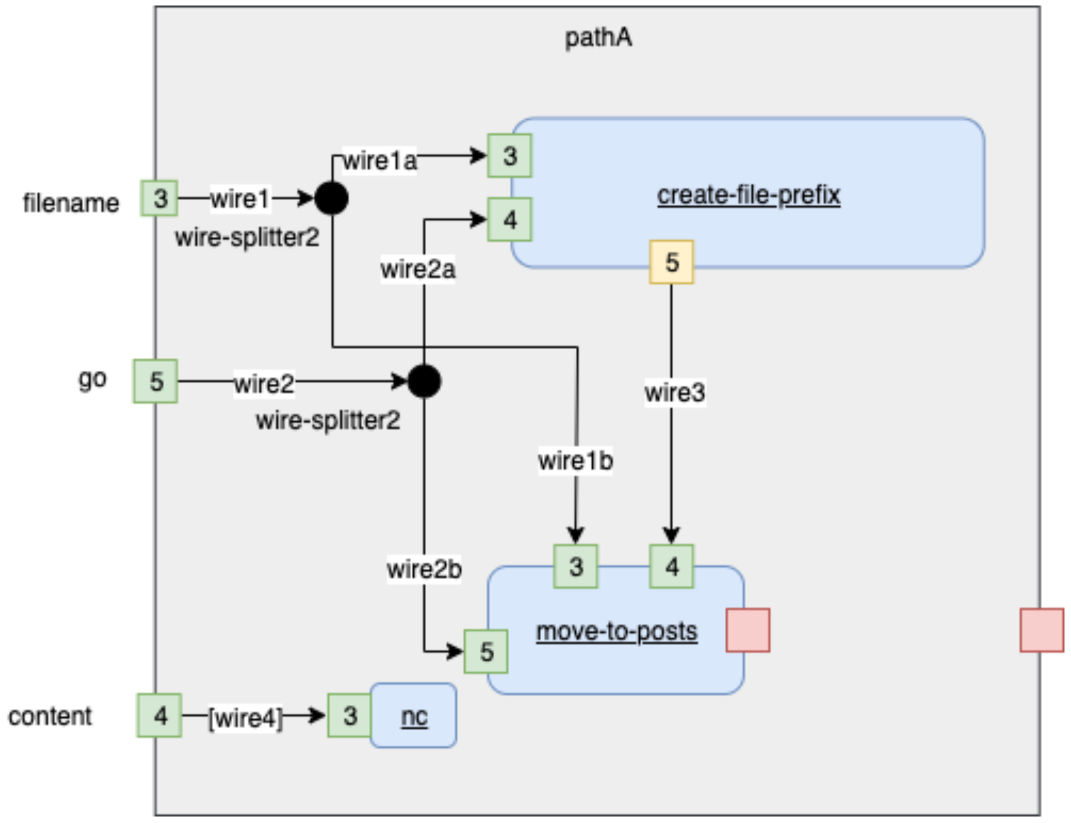


Fig. 17 Implementation Branch A

```
#!/bin/bash
./debug-entry "[$1] $0"


#
## inputs:
## filename (fd 3)
## content (fd 4) N/C
## go (fd 5)
#
## outputs:
## <none> - side-effect - file.html moved to ./
_posts/<prefix>_filename.html
#


wire1=wire_pA_1
wire1a=wire_pA_1a
wire1b=wire_pA_1b
wire2=wire_pA_2
```

```
wire2a=wire_pA_2a
wire2b=wire_pA_2b
wire3=wire_pA_3
wire4=wire_pA_4
mkfifo ${wire1} ${wire1a} ${wire1b} ${wire2} $
{wire2a} ${wire2b} ${wire3} ${wire4}


./wire-splitter2 "[$1] $0" 3<${wire1} 4>${wire1a}
5>${wire1b} &
./wire-splitter2 "[$1] $0" 3<${wire2} 4>${wire2a}
5>${wire2b} &
./create-file-prefix "[$1] $0" 3<${wire1a} 4<$
{wire2a} 5>${wire3} &
./move-to-posts "[$1] $0" 3<${wire1b} 4<${wire3}
5<${wire2b} &


read filename </dev/fd/3
echo "${filename}" > ${wire1} &


read content </dev/fd/4
echo "${content}" > ${wire4} &
./nc 3<${wire4} &


read pin_go </dev/fd/5
echo "${pin_go}" > ${wire2} &


./debug-exit "[$1] $0"
wait
```

create-file-prefix-from-subdir

```
#!/bin/bash
#
## inputs:
## filename (fd 3)
## go (fd 4)
#
```

```
## outputs:
## prefix (fd 5) (YYYY-MM-DD)
## fullprefix (fd 6) (YYYY-MM-DD-HH-MM-SS)
#
./debug-entry "[$1] $0"


read filename </dev/fd/3
read go </dev/fd/4


if [ "go" == "${go}" ]
then
    prefix=`date "+%Y-%m-%d"`
    base=`basename "${filename}"`
    # error check needed here - filename should not be
"" at this point
    creation_time=`grep "<meta name=\"CreationTime\""
"${filename}/${base}"`
    if [ ! -z "${creation_time}" ]
    then
   prefix=`echo "${creation_time}" | sed -e 's/<meta
name="CreationTime" content="\(2020-..-..
\)T..:..:..Z">/\1/'`
   fullprefix=`echo "${creation_time}" | sed -e 's/
<meta name="CreationTime" content="\(2020-..-..\)T\(..
\):\(..\):\(..\)Z">/\1-\2-\3-\4/'`
    fi
    echo "${prefix}" >/dev/fd/5 &
    echo "${fullprefix}" >/dev/fd/6 &
else
    # this is required because bash does not actually
support concurrency
    # bash imposes a rendezvous regimen on named pipes
and blocks processes until all pipes have 2 ends
    # this causes deep dependency chains
    # (in true concurrency, we would be allowed to
create pipes even when one end has not yet been
created, and we would not need these two lines of code)
    # (rendezvous causes dependencies, which causes
accidental complexity, which causes lots of head-
scratching)
    # (rendezvous can be implemented using an ACK/NAK
protocol)
    # (at the least, this pertains to bash under MacOSX
Catalina, probably elsewhere, too)
    echo "???" >/dev/fd/5 &
    echo "???" >/dev/fd/6 &
fi
```

```
    ./debug-exit "[$1] $0"
```

edit-and-move-with-images

```
#!/bin/bash
#
## inputs:
## filename (fd 3)
## prefix (fd 4) YYYY-MM-DD
## go (fd 5)
## fullprefix (fd 6) YYYY-MM-DD-HH-MM-SS
#
## outputs:
## <none>
##   - edit _new/filename/basename change all occurences
of Images to ../assets/prefix-Images --> /tmp/basename
##   - moves /tmp/basename to ../_posts/filename/prefix-
basename
##   - moves _new/filename/Images   to ../assets/prefix-
Images
#
## specific to Scrivener: Compile the document as a Web
Page (.html) and leave it in ../_new
## specific to Scrivener: if a .scriv file contains
images, a directory will be created - it contains
the .html file plus a sub-directory Images/, for example,
abc.sriv --> abc.html/abc.html and abc.html/Images (yes,
the directory is named abc.html)
#
./debug-entry "[$1] $0"
read filename </dev/fd/3
read prefix </dev/fd/4
read go </dev/fd/5
read fullprefix </dev/fd/6
if [ "${go}" == "go" ]
then
    base=`basename "${filename}"`
    # edit to new file and move it to _posts
    sed -e "s@<img src=\"Images@<img src=\"/assets/$
{fullprefix}-Images@g" <"${filename}/${base}" >"../
_posts/${prefix}-${base}"
    # move Images folder to ../assets/prefix-Images
    mv "${filename}/Images" "../assets/${fullprefix}-
Images"
    # delete source(s), completing the mv (mv is not cp)
```

```
        rm -rf "${filename}"
    fi
    ./debug-exit "[$1] $0"
```

pathB

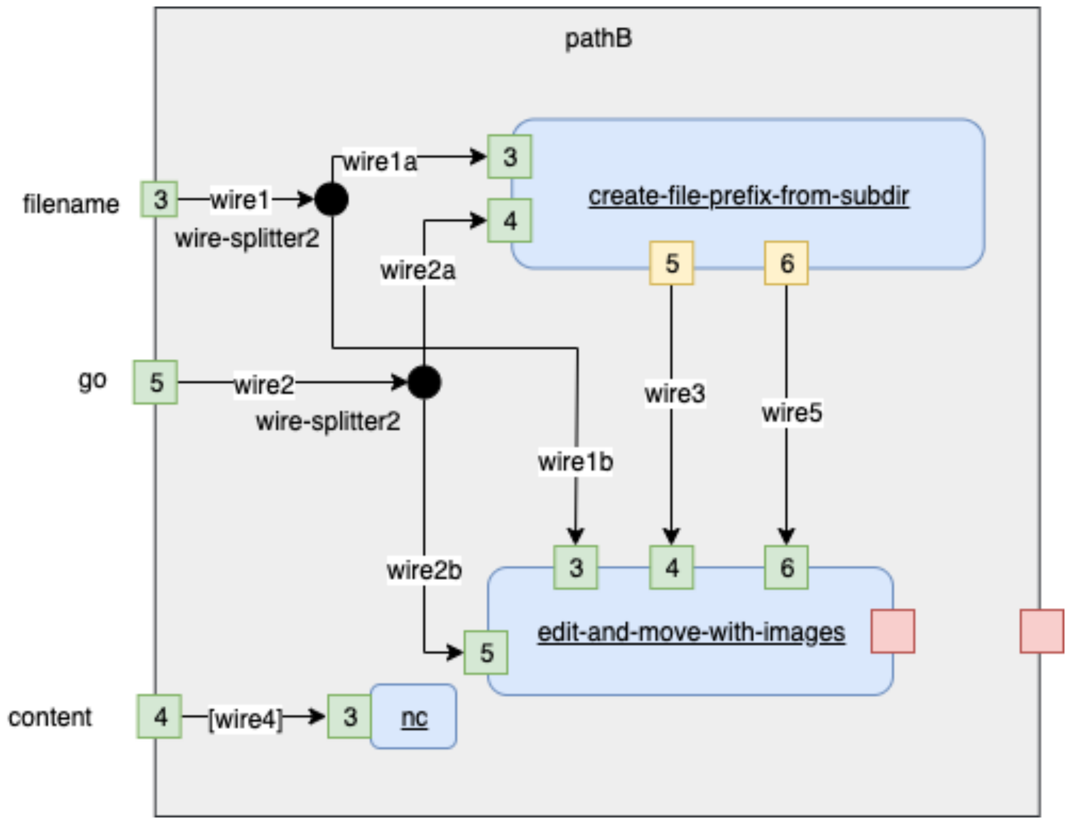


Fig. 18 Implementation Branch B

```
#!/bin/bash
./debug-entry "[$1] $0"


#
## inputs:
## filename (fd 3)
## content (fd 4) N/C
## go (fd 5)
#
## outputs:
```

```
## <none> - side-effect - file.html moved to ./
_posts/<prefix>_filename.html
#


wire1=wire_pB_1
wire1a=wire_pB_1a
wire1b=wire_pB_1b
wire2=wire_pB_2
wire2a=wire_pB_2a
wire2b=wire_pB_2b
wire3=wire_pB_3
wire4=wire_pB_4
wire5=wire_pB_5
mkfifo ${wire1} ${wire1a} ${wire1b} ${wire2} $
{wire2a} ${wire2b} ${wire3} ${wire4} ${wire5}


./wire-splitter2 "x [$1] $0" 3<${wire1} 4>${wire1a}
5>${wire1b} &
./wire-splitter2 "y [$1] $0" 3<${wire2} 4>${wire2a}
5>${wire2b} &
./create-file-prefix-from-subdir "[$1] $0" 3<$
{wire1a} 4<${wire2a} 5>${wire3} 6>${wire5}&
./edit-and-move-with-images "[$1] $0" 3<${wire1b}
4<${wire3} 5<${wire2b} 6<${wire5} &


read filename </dev/fd/3
echo "${filename}" > ${wire1} &


read content </dev/fd/4
echo "${content}" > ${wire4} &
./nc 3<${wire4} &


read pin_go </dev/fd/5
echo "${pin_go}" > ${wire2} &


./debug-exit "[$1] $0"
wait
```

Extra Marks - Transpiling Diagrams

Some clues about how to transpile diagrams to code:

- Convert the diagram into XML form. I drew the diagrams in draw.io. It outputs a compressed XML file. Each tab on the diagram is contained in its own element delimited by <diagram> … </diagram>
- Copy/Paste the compressed data into the tool https://jgraph.github.io/drawio-tools/tools/convert.html and press the *decode* button. This should result in human-readable XML of an mxGraph structure.
- The structure contains graphical information about every item in the drawing.
- Weed out the syntactic sugar.
- Normalize the data - I like factbases (see another one of my essays).
- Use code to transform the data into some very convenient form, e.g. JSON. I like using PEG-based parsers (Ohm-js for Javascript, ESRAP for Common Lisp).
- N.B. the graphical data can be considered to be a textual programming language, where details like X, Y, Width and Height have been added.
- Convert the 2D graphical information into 1D textual information. E.g. find all boxes, then, find all boxes that intersect boxes (high school math). Small boxes that sit on the edges of larger boxes are "ports".
- Use standard text-compilation technique from that point on.

Some clues on how to draw diagrams that can be transpiled:

- Ensure that all boxes are *concurrent*. The sequential paradigm (call/return) does not work for diagrams.
- Allow text to be included in the diagram (text is better than diagrams for a certain class of programs, e.g. "a = b + c" should be written and not drawn).
- Use an editor that allows drawing any shape and does not need to know details about what is being drawn.

- Imagine Emacs (vim, vscode, etc.) for diagrams. Emacs knows only how to edit characters, in general. Emacs doesn't check that the code is consistent - that's the job of the compiler in later stages. Emacs allows you to save inconsistent programs - for example, it doesn't check for consistency and stop one from saving an inconsistent program. The compiler will raise error messages, later.
- Imagine that diagrams are the same as text code.
- Think box/arrow/ellipse/text instead of pixels.
- Programmers' editors edit a grid of cells. Cells may not overlap. Cells are also called "characters".
- Programmers' editors are not-quite-2D. They allow 2D layout of text, but insist on arranging cells in lines and columns.
- Diagram editors, OTOH, allow cells to overlap. No strict grid structure is imposed.
- Don't think in terms of pixels, think about larger constructs, like boxes. One doesn't need pixel/raster recognition algorithms to effectively use diagrams as syntax (DaS).

White Box Testing

During development, most parts were tested using white-box testers.

The topmost white-box tester is

https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/whitebox-testall.

The remaining white-box testers can be viewed at:

https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/whitebox-test-create-file-prefix
https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/

whitebox-test-isADirectory

https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/whitebox-test-move-to-posts

https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/whitebox-test-pathA

https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/whitebox-test-pathA-fail

https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/whitebox-test-scrivener-to-jekyll

https://github.com/guitarvydas/scrivener-to-jekyll/blob/main/_bin/whitebox-test-wire-splitter2