Pipeline Architecture

Software Architecture

Brae Webb

February 27, 2023

So far. . .

Layered architectures reduce the impact of changing a layer

Question

Why does the layer order matter?

Question

Why does the layer order matter?

Answer

Each layer implements a different interface.

So...

If every layer implements the same interface?

Extreme layered architecture

Pipeline Architectures[1]

[1] sorta

Definition 1. Pipeline Architecture

Components connected in such a way that the output of one component is the input of another.

Question

Can you think of a *pipeline archiecture*?

Question

Can you think of a *pipeline archiecture*?

Answer

How about *bash*?

```
>> cat assignment.py | grep "hack" | wc -l | tee code-quality.txt
```

```
>> cat assignment.py | grep "hack" | wc -l | tee code-quality.txt
```

Notice:

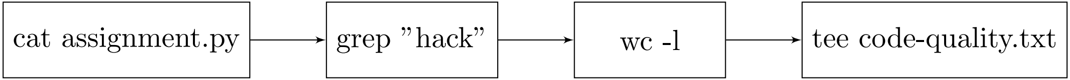
- Each program performs a small well-defined task.

```
>> cat assignment.py | grep "hack" | wc -l | tee code-quality.txt
```

Notice:

- Each program performs a small well-defined task.
- Each program implements the same interface (i.e. raw text).

```
>> cat assignment.py | grep "hack" | wc -l | tee code-quality.txt
```




Filter
Pipe
Filter
Pipe
Filter

Filters

Modular software components

Filters

Modular software components

Pipes

The flow of data between filters

Types of Filters

Producers

Source of data

Types of Filters

Producers

Source of data

Transformers

Transform data

Types of Filters

Producers

Source of data

Transformers

Transform data

Testers

Filter data

Types of Filters

Producers

Source of data

Testers

Filter data

Transformers

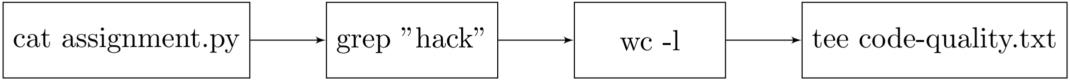
Transform data

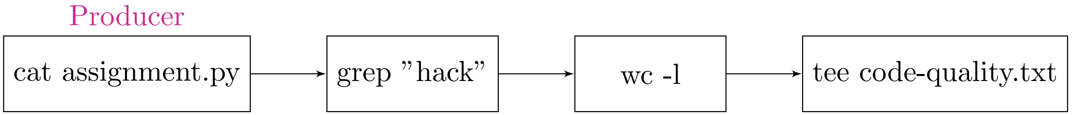
Consumers

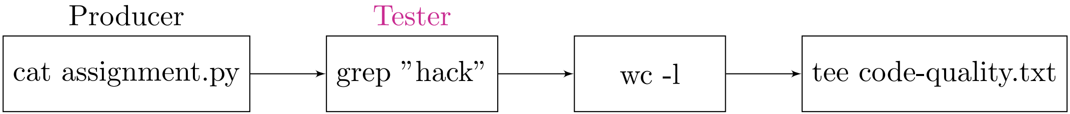
Target for results

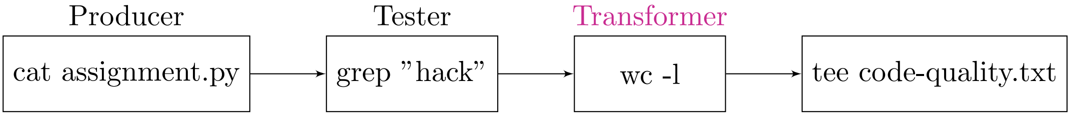
Exercise

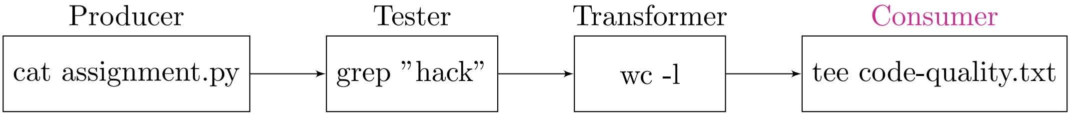
Label the bash pipeline


cat assignment.py
grep "hack"
wc -l
tee code-quality.txt


Producer
cat assignment.py
grep "hack"
wc -l
tee code-quality.txt


Producer
Tester
cat assignment.py
grep "hack"
wc -l
tee code-quality.txt


Producer
Tester
Transformer
cat assignment.py
grep "hack"
wc -l
tee code-quality.txt


Producer
cat assignment.py
Tester
grep "hack"
Transformer
wc -l
Consumer
tee code-quality.txt

Definition 2. One Direction Principle

Data should flow in one direction — *downstream*.

Definition 3. Independent Filter Principle

Filters should not rely on specific upstream or downstream components.

Corollary 1. Generic Interface

The interface between filters should be generic.

Corollary 2. Composable Filters

Filters (i.e. Transformers & Testers) can be applied in any order.

The Case Study

Bash

POSIX-2017

Shell & Utilities

Utilities

admin - create and administer SCCS files (DEVELOPMENT)
alias - define or display aliases
ar - create and maintain library archives
asa - interpret carriage-control characters
at - execute commands at a later time
awk - pattern scanning and processing language
basename - return non-directory portion of a pathname
batch - schedule commands to be executed in a batch queue
bc - arbitrary-precision arithmetic language
bg - run jobs in the background
c99 - compile standard C programs
cal - print a calendar
cat - concatenate and print files
cd - change the working directory
cflow - generate a C-language flowgraph (DEVELOPMENT)
chgrp - change the file group ownership
chmod - change the file modes
chown - change the file ownership
cksum - write file checksums and sizes
cmp - compare two files
command - execute a simple command
comm - select or reject lines common to two files
compress - compress data
cp - copy files
crontab - schedule periodic background work
csplit - split files based on context
ctags - create a tags file (DEVELOPMENT, FORTRAN)
cut - cut out selected fields of each line of a file
cxref - generate a C-language program cross-reference table (DEVELOPMENT)
date - write the date and time
dd - convert and copy a file
delta - make a delta (change) to an SCCS file (DEVELOPMENT)
df - report free disk space
diff - compare two files
dirname - return the directory portion of a pathname
du - estimate file space usage
echo - write arguments to standard output
ed - edit text
env - set the environment for command invocation
ex - text editor
expand - convert tabs to spaces
expr - evaluate arguments as an expression
false - return false value
fc - process the command history list
fg - run jobs in the foreground
file - determine file type
find - find files
fold - filter for folding lines
fort77 - FORTRAN compiler (FORTRAN)
fuser - list process IDs of all processes that have one or more files open
gencat - generate a formatted message catalog
getconf - get configuration values
get - get a version of an SCCS file (DEVELOPMENT)
getopts - parse utility options
grep - search a file for a pattern
hash - remember or report utility locations
head - copy the first part of files
iconv - codeset conversion
id - return user identity
ipcrm - remove an XSI message queue, semaphore set, or shared memory segment identifier
ipcs - report XSI interprocess communication facilities status
jobs - display status of jobs in the current session
join - relational database operator
kill - terminate or signal processes
lex - generate programs for lexical tasks (DEVELOPMENT)
link - call link function
ln - link files
localedef - define locale environment
locale - get locale-specific information
logger - log messages
logname - return the user's login name
lp - send files to a printer
ls - list directory contents
m4 - macro processor
mailx - process messages
make - maintain, update, and regenerate groups of programs (DEVELOPMENT)

Question

Who has heard of *literate programming*?

The Challenge — set by Jon Bently

1. Read a file of text.
2. Determine the n most frequently used words.
3. Print out a sorted list of those words along with their frequencies.

Knuth's Solution

17 pages of elegant and descriptive code.

programming pearls

by Jon Bentley
with Special Guest Oysters
Don Knuth and Doug McIlroy

A LITERATE PROGRAM

Last month's column introduced Don Knuth's style of "Literate Programming" and his WEB *system for building programs that are works of literature. This column presents a literate program by Knuth (its origins are sketched in last month's column) and, as befits literature, a review. So without further ado, here is Knuth's program, retypeset in* Communications *style.* —*Jon Bentley*

Common Words Section

Common Words	Section
Introduction	1
Strategic considerations	8
Basic input routines	9
Dictionary lookup	17
The frequency counts	32
Sorting a trie	36
The endgame	41
Index	42

1. Introduction. The purpose of this program is to solve the following problem posed by Jon Bentley:

> Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

Jon intentionally left the problem somewhat vague, but he stated that "a user should be able to find the 100 most frequent words in a twenty-page technical paper (roughly a 50K byte file) without undue emotional trauma."

Let us agree that a *word* is a sequence of one or more contiguous letters; `"Bentley"` is a word, but `"ain't"` isn't. The sequence of letters should be maximal, in the sense that it cannot be lengthened without including a nonletter. Uppercase letters are considered equivalent to their lowercase counterparts, so that the words `"Bentley"` and `"BENTLEY"` and `"bentley"` are essentially identical.

The given problem still isn't well defined, for the file might contain more than k words, all of the same frequency; or there might not even be as many as k words. Let's be more precise: The most common words are to be printed in order of decreasing frequency, with words of equal frequency listed in alphabetic order. Printing should stop after k words have been output, if more than k words are present.

2. The *input* file is assumed to contain the given text. If it begins with a positive decimal number (preceded by optional blanks), that number will be the value of k; otherwise we shall assume that $k = 100$. Answers will be sent to the *output* file.

```
define default_k = 100 {use this value if k isn't
        otherwise specified}
```

3. Besides solving the given problem, this program is supposed to be an example of the WEB system, for people who know some Pascal but who have never seen WEB before. Here is an outline of the program to be constructed:

```
program common_words (input, output);
  type ⟨Type declarations 17⟩
  var ⟨Global variables 4⟩
  ⟨Procedures for initialization 5⟩
  ⟨Procedures for input and output 9⟩
  ⟨Procedures for data manipulation 20⟩
  begin ⟨The main program 8⟩;
  end.
```

4. The main idea of the WEB approach is to let the program grow in natural stages, with its parts presented in roughly the order that they might have been written by a programmer who isn't especially clairvoyant.

For example, each global variable will be introduced when we first know that it is necessary or desirable; the WEB system will take care of collecting these declarations into the proper place. We already know about one global variable, namely the number that Bentley called k. Let us give it the more descriptive name *max_words_to_print*.


```
⟨Global variables 4⟩ ≡
max_words_to_print: integer;
    {at most this many words will be printed}
```

See also sections 11, 13, 18, 22, 32, and 36.
This code is used in section 3.

5. As we introduce new global variables, we'll often want to give them certain starting values. This will be done by the *initialize* procedure, whose body will consist of various pieces of code to be specified when we think of particular kinds of initialization.

```
⟨Procedures for initialization 5⟩ ≡
procedure initialize;
  var i: integer;  {all-purpose index for initializa-
        tions}
  begin ⟨Set initial values 12⟩
  end;
```

This code is used in section 3.

6. The WEB system, which may be thought of as a preprocessor for Pascal, includes a macro definition facility so that portable programs are easier to write. For example, we have already defined '*default_k*' to be 100. Here are two more examples of WEB macros; they allow us to write, e.g., '*incr*(*count*[*p*])' as a convenient abbreviation for the statement '*count*[*p*] ← *count*[*p*] + 1'.

```
define incr(#) ≡ # ← # + 1   {increment a vari-
        able}
define decr(#) ≡ # ← # − 1   {decrement a vari-
        able}
```

7. Some of the procedures we shall be writing come to abrupt conclusions; hence it will be convenient to introduce a '**return**' macro for the operation of jumping to the end of the procedure. A symbolic label '*exit*' will be declared in all such procedures, and '*exit*:' will be placed just before the final **end**. (No other labels or **goto** statements are used in the present program, but the author would find it painful to eliminate these particular ones.)

```
define exit = 30  {the end of a procedure}
define return ≡ goto exit  {quick termination}
format return ≡ nil  {typeset 'return' in boldface}
```

8. Strategic considerations. What algorithms and data structures should be used for Bentley's problem? Clearly we need to be able to recognize different occurrences of the same word, so some sort of internal dictionary is necessary. There's no obvious way to decide that a particular word of the input cannot possibly be in the final set, until we've gotten very near the end of the file; so we might as well remember every word that appears.

There should be a frequency count associated with each word, and we will eventually want to run through the words in order of decreasing frequency. But there's no need to keep these counts in order as we read through the input, since the order matters only at the end.

Therefore it makes sense to structure our program as follows:

```
⟨The main program 8⟩ ≡
  initialize;
  ⟨Establish the value of max_words_to_print 10⟩;
  ⟨Input the text, maintaining a dictionary with
     frequency counts 34⟩;
  ⟨Sort the dictionary by frequency 39⟩;
  ⟨Output the results 41⟩
```

This code is used in section 3.

9. Basic input routines. Let's switch to a bottom-up approach now, by writing some of the procedures that we know will be necessary sooner or later. Then we'll have some confidence that our program is taking shape, even though we haven't decided yet how to handle the searching or the sorting. It will be nice to get the messy details of Pascal input out of the way and off our minds.

Here's a function that reads an optional positive integer, returning zero if none is present at the beginning of the current line.

```
⟨Procedures for input and output 9⟩ ≡
function read_int: integer;
  var n: integer;  {the accumulated value}
  begin n ← 0;
  if ¬eof then
    begin while (¬eoln) ∧ (input↑ = '␣') do
      get(input);
    while (input↑ ≥ '0') ∧ (input↑ ≤ '9') do
      begin n ← 10*n + ord(input↑) − ord('0');
      get(input);
      end;
    end;
  read_int ← n;
  end;
```

See also sections 15, 35, and 40.
This code is used in section 3.

10. We invoke *read_int* only once.

```
⟨Establish the value of max_words_to_print 10⟩ ≡
  max_words_to_print ← read_int;
  if max_words_to_print = 0 then
    max_words_to_print ← default_k
```

This code is used in section 8.

11. To find words in the *input* file, we want a quick way to distinguish letters from nonletters. Pascal has

McIlroy's Solution

```
tr -cs A-Za-z '\n' | \
    tr A-Z a-z | \
    sort | \
    uniq -c | \
    sort -rn | \
    sed ${1}q
```

Question

Is literate programming bad?

Question

Is literate programming bad?

Answer

No, the Unix philosophy is just good.

The Unix Philosophy

- Write programs that do one thing and do it well.

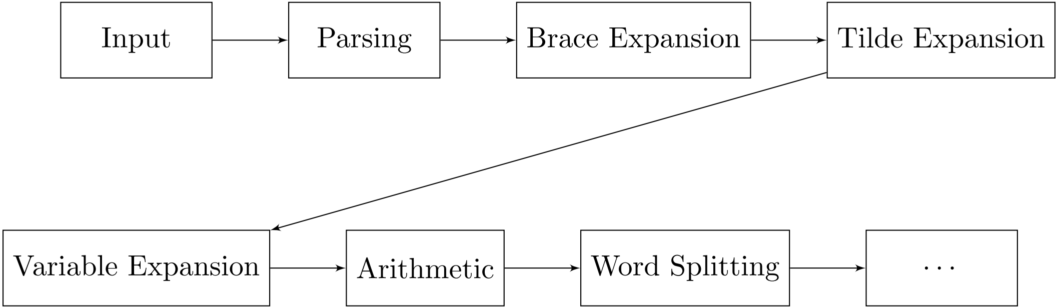
The Unix Philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.

The Unix Philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

Bash itself is a pipeline



Reading...

“Pipeline Architecture” Notes *[Webb and Thomas, 2023]*

References

[Webb and Thomas, 2023] Webb, B. and Thomas, R. (2023).
Pipeline architecture.
https://csse6400.uqcloud.net/handouts/pipeline.pdf.