40

Chapter 3
Abstraction

Our first tool, divide and conquer, breaks enigmas into manageable problems. These leaf nodes are manageable partly because they are conceptually simple; the length of a classical symphony, roughly one hour, is a simple concept compared to the data capacity of a CDROM. Successful leaf nodes are manageable also because they are familiar. The length of a symphony, for example, might be familiar from attending a classical concert. A concert is typically 2.5 hours with a half-hour intermission (interval) in the middle, leaving one-hour blocks at the start or end for a full symphony.

Familiarity is the sibling of reuse. Successful divide-and-conquer reasoning breaks a problem not just into parts but into reusable parts. Discovering and constructing such parts is the purpose of **abstraction** – the second tool for organizing complexity. Abstraction is, according to the *Oxford English Dictionary* [24]:

> The act or process of separating in thought, *of considering a thing independently of its associations;* or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs. [my italics]

Abstraction thereby generates new ideas and new units of thought.

The tools taught in this book are themselves fruits of abstraction. For example, many estimation problems were solved by dividing the problem into small, manageable parts. This pattern needed a name – divide and conquer. The other tools, even abstraction, are similarly the fruits of abstraction.

3.1 Reusability

The most important characteristic of abstraction is reusability. As Abelson and Sussman [1, s. 1.1.8] eloquently describe:

40

> The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

To understand what makes a useful, reusable abstraction, let's examine a weak, barely reusable abstraction and compare its features with the features of a useful abstraction. So, imagine that the designers of UNIX had noticed that users often needed to count how often each word appears in a document, listing the most frequent words first (along with their frequencies). One solution is to provide a special utility called `sortedwordfreq`. This utility, however, cannot solve any other problem.

As an improvement, the problem could be broken into three steps:

1. break the document into words, one per line.
2. count how often each word appears
3. sort the frequency list by frequency

UNIX could provide three command-line programs, one for each step, and the user would connect the programs with pipes:

```
break_into_words < file.txt | wordfreq | sort_frequency_list
```

where `sort_frequency_list` sorts a list such as

```
 34 an
273 the
 12 where
 23 none
```

to produce

```
273 the
 34 an
 23 none
 12 where
```

This approach is an improvement on the monolithic solution because one of the three pieces might be used in solving a different problem.

The actual solution using the UNIX tools is even more reusable. Rather than provide a special program to break a document into words, one per line,

UNIX provides a utility called `tr`. It translates characters into other characters. So ask it to translate all non-alphabetic characters into newline characters, and then to squeeze repeated newline characters into one newline character. That command is

```
tr -s -c 'a-zA-Z' '\012'
```

The `-s` option tells `tr` to squeeze repeated newlines into one newline. The `-c` option tells `tr` to invert (complement) the following character set (the upper- and lowercase alphabet). It is simpler to specify the non-alphabetic characters by what they are not than by what they are. So this invocation of `tr` turns any non-alphabetic character into a newline, then squeezes repeated newlines into one newline.

It turns the first sentence of this paragraph into the following list of words, one per line:

```
The
actual
solution
using
the
Unix
tools
is
even
more
reusable
Rather
than
provide
a
special
program
to
break
a
document
into
words
```

The next step is to count how often each word appears. Perhaps UNIX provides a program called `count` that performs this task? Such a program

would have to look through the entire list and accumulating counts. It is simpler first to sort the list. Then identical words appear in clumps, which means that the counting program need not scan the entire list. Instead it can consider one clump at a time. The sorting step is accomplished by the familiar program `sort`. The clump counting is accomplished by a new program, `uniq` with the `-c` option.

Here is the result of taking the text of Gibbon's *Decline and Fall of the Roman Empire* (volume 1) and feeding it to the pipeline: first taking out all punctuation and turning it into a list of words, one per line; then sorting the list; then counting the clumps. The result is:

```
$ tr -cs 'a-zA-Z' '\012' < decline.txt | sort | uniq -c
   4452 a
    233 A
      2 Aaron
      9 ab
      1 Ab
      8 abandon
     30 abandoned
      1 Abandoning
      1 abandonment
      3 Abate
        ...
```

These are not the 10 most common words! Rather, they are the 10 alphabetically earliest words (along with their counts). To find the most common words, sort this output numerically by adding `sort -nr` to the end of the pipeline:

```
  $ tr -cs 'a-zA-Z' '\012' < decline.txt | sort | uniq -c |
sort -nr
   24241 the
   17920 of
    9097 and
    5951 to
    4452 a
    3869 in
    3171 was
    2904 his
    2737 by
    2711 The
    ...
```

We're almost there! But a problem is the appearance of 'The' on a separate line. We forgot about uppercase versus lowercase. So let's use `tr` one more time (what a useful abstraction), to turn uppercase into lowercase:

```
  $ tr -cs 'a-zA-Z' '\012' < decline.txt | tr 'A-Z' 'a-z' |
sort | uniq -c | sort -nr
    26960 the
    18099 of
     9168 and
     6050 to
     4685 a
     4217 in
     3171 was
     3081 his
     2815 by
     2396 that
     ...
```

The new count for 'the' is 29690. But the count for 'the' together the count for 'The' give a count of $24241 + 2711 = 26952$. What accounts for the discrepancy between 26952 and 26960? Let's ask UNIX to tell us about all forms of 'the' that showed up. To do so, use `grep` to match only lines reporting counts for 'the' or one its mixed-case variants. The `-i` option to `grep` tells `grep` not to care about upper versus lowercase. The pattern for `grep` to look for is then a space followed by 'the' followed by the end of line ($ in `grep` notation). So the pipeline with its output is:

```
  $ tr -cs 'a-zA-Z' '\012' < decline.txt | sort | uniq -c |
grep -i ' the$'
    24241 the
     2711 The
        8 THE
```

Ah, so there were eight appearances of 'THE' – which accounts for the discrepancy between 26952 and 26960.

3.2 Notation and hierarchy

Good notation promotes good thinking.