Documentation Tools and Techniques

J. R. Mashey
D. W. Smith

Bell Laboratories
Piscataway, New Jersey 08854

Keywords: Text processing, text formatting, UNIX.

Abstract: In a software development project of any appreciable size, the production of usable, accurate documentation may well consume more effort than the production of the software itself. Several years of experience on many Programmer's Workbench projects have shown that document preparation should not be separated from software development and that the combination of a flexible operating system, a powerful command language, and good text processing facilities permits quick and convenient production of many kinds of documentation which might be otherwise unobtainable, impractical, or very expensive. Our basic approach has been to develop techniques for effective combination of existing UNIX facilities. A number of case histories are given to illustrate the flexibility, convenience, and general usefulness of these techniques.

1. INTRODUCTION

The Programmer's Workbench (PWB) [DOL76A] is a working example of a system specialized for program development. Many activities performed with the PWB's help can be considered to be in some sense document preparation or text processing:

- editing source code or documents;
- scanning output returned from other systems;
- scanning files for certain data (e.g., cross-referencing, data collection and summarization);
- recording changes to source or document text using SCCS [ROC75A];
- preparing scripts for load or regression tests [DOL76B];
- examining output from load or regression tests;
- preparing input for and producing many kinds of reports;
- entering and maintaining information on activities and schedules.

From our observation of actual PWB usage, it appears that flexible, integrated text processing is an absolute requirement for any program development facility. This requirement is *not* satisfied by adding a simple text formatter to an existing system; the *entire* system must support text manipulation in a consistent, convenient way. Text formatting becomes more flexible and adaptable when implemented as part of a more general environment, rather than as an isolated, specialized "add-on." In addition, many normal programming activities are better supported when they have good text processing facilities available in the same way as other tools. Specific advantages include:

- The same editor is used to enter and modify documentation, source programs, and procedures for manipulating them both. This unification helps lower training and support costs.
- Likewise, it is possible to use existing tools to manipulate either source programs or text without regard for the intended use of these tools. Obvious examples include file scanners, sorts, and cross-reference commands. The PWB user has access to all the tools provided by UNIX [RIT74A]. The set of tools is extensive [KER75A], and unexpected uses are constantly found for them.
- When a document must include samples of source programs, it is easy to assure accuracy of such samples because they can be automatically copied from actual source program files.
- No artificial distinctions need exist among source programs, documentation, and data bases. A set of files and command procedures may appear to be a small interactive data base management system; viewed another way, it is just a mechanism for producing a large, complex document. Finally, information from these files may be used to automatically generate or verify parts of source programs.
- It is often *more* important to automate the management and control of text than it is to format it. It is especially necessary to be able to do this quickly and cheaply, because procedures and needs not only differ among projects but change quickly and often.

In this paper we illustrate some of the facilities one gains by utilizing a concept unique to the UNIX system—that text processing is an integral part (not an "add-on") of an effective program development environment. We first briefly survey some of the useful tools provided by UNIX. We then show the desirability, convenience, and (in our opinion) the *necessity* of this approach by examining case histories of several projects implemented on the PWB in the last few years.

2. BACKGROUND

2.1 Environment

Many aspects of the UNIX environment make it especially suitable for text processing.

First, a UNIX file is just a string of characters, whose format and interpretation are under control of the user, rather than UNIX. The most common file format is that of an *ASCII file*, i.e., a sequence of arbitrary-length *lines* terminated by *new-line* characters. This format avoids any need to worry about blocking or blank stripping. Explicit allocation or de-allocation of storage is avoided by the ability of files to grow and shrink dynamically.

Second, UNIX supplies a flexible hierarchical file structure that is easily adaptable to user needs. Convenience of document organization is improved by the ease of grouping large numbers of files. Users may share files and commands in various ways.

Third, the UNIX *pipe* mechanism is particularly desirable for text processing because it provides an effective way of manipulating data streams. A pipe connects two executing programs, causing the output of the first to be read as the input to the second, while all synchronization and buffering details are handled by UNIX itself. Two or more programs can thus be connected together to form a *pipeline*, an object commonly found in UNIX text processing applications.

Finally, although some types of terminals are more convenient than others for UNIX usage, UNIX is easily adaptable to many different kinds of terminals. It is rarely necessary to modify the operating system to take advantage of some new type of terminal. All that is usually necessary to take advantage of any unusual terminal features is to write a single, small, non-privileged command to cater to the specific terminal.

2.2 Specific Tools

Although almost all UNIX commands turn out to have some text processing use sooner or later, some are especially useful.

2.2.1 Editing System. *Ed* is the standard editor. A related command, *sed,* is a *stream editor,* whose use is more efficient and convenient than that of *ed* in one-pass applications, i.e., when used in a pipeline.

2.2.2 Text Formatting Software. The simplest version of the "run off" formatting software is *roff,* whose formatting power is roughly equivalent to that of IBM's ATS, although its input appearance is radically different. *Nroff* is much more powerful than *roff,* permitting the use of sequences of formatting codes (known as macros) whose behavior may be modified by arguments specified when such a macro is interpreted. The typesetting software, *troff,* is similar to *nroff* with respect to programming, but its output is intended for a Graphic Systems, Inc., phototypesetter, which can produce various font styles and character sizes as well as numerous special characters for mathematical expressions and for other purposes.

2.2.3 Output Filters. Most *roff* and *nroff* output can be printed on most terminals, but some output takes advantage of special features and escape sequences found only on the Teletype® Model 37. To make certain sequences (such as reverse line feed, half-line motion, or extended character sets) function properly on other output devices, the output of *roff* or *nroff* must be processed ("filtered") by other programs before being physically printed. For example, *gsi* is used to make output acceptable to certain terminals that use the Diablo HyType® print mechanism; *hp* is used for output on a Hewlett Packard 2640A. *Col* is used to "normalize" such output for terminals lacking reverse line feed, especially when multi-column output is desired.

2.2.4 Input Filters. Just as output filters process output before it is printed, input filters can process text before it is formatted. Such filters often handle complex or unusual requirements without requiring modification of the formatters. For example, *eqn* and *neqn* permit convenient formatting of mathematical expressions [KER75B], and *tbl* helps align and format tabular matter and its associated headings.

2.2.5 Data Storage Facilities. An *archive file* is a collection of files created and maintained via the *ar* command. It is often used to store large numbers of small files in an efficient way, and may be used in some data base applications. The commands that are part of SCCS [ROC75A] can be used to maintain control over document changes, record such changes, and recreate the document as it existed at any point in time.

2.2.6 File Examination Software. The *grep* program is used to extract from text files lines containing a specified sequence of characters. *Cref* produces cross-reference listings of the C and UNIX assembler programming languages. The *diff* program reports differences between two files, listing the lines that have been added, deleted, or changed. The *comm* program lists lines that are common to two files. *Typo* helps the user quickly identify possible typographical errors.

2.2.7 Command Language. The UNIX command language is interpreted by a program called the *shell.* It provides a quick, inexpensive way to combine other tools into the desired procedures [MAS76A]. For example, the procedure in Figure 1 formats a set of files according to a standard *nroff* macro set "−mm", allowing an optional first argument "−g" to request use of the *gsi* filter and 12-pitch output; otherwise output is produced for a terminal without reverse-line motion (e.g., a G.E. TermiNet 300).

```
if $1 = −g then
   shift
   nroff −h −rT1 −mm $1 $2 $3 $4 $5 $6 $7 $8 $9 | gsi +12
   exit
endif
nroff −mm $1 $2 $3 $4 $5 $6 $7 $8 $9 | col
```

Figure 1

3. PROGRAMMER'S WORKBENCH MEMORANDUM MACROS

The *nroff* and *troff* are unusually flexible and powerful text formatters. However, to make the best use of this software, one must become quite familiar with all the basic formatting requests. Learning the requests and the useful ways of combining them is a full-time *programming* task. For the PWB, we spent considerable effort in examining what kinds of things people would want to do in manipulating and stylizing their documentation—whether preparing letters, technical reports, user manuals, or software documentation. With the design priorities as described below, we developed the Programmer's Workbench Memorandum Macros (PWB/MM), a unified, consistent, and flexible set of formatting codes. PWB/MM minimizes the differences between *nroff* and *troff,* making it easier to enter a document without concern for its eventual output medium. This package brings much of the inherent power of the formatters to the user who does not want to spend a great deal of time learning their intricacies. It is heavily used by clerks and typists, as well as by technical personnel. PWB/MM was used to prepare this paper, as well as the five companion PWB papers [BIA76A, DOL76A, DOL76B, KNU76A, MAS76A].

3.1 Design Priorities

The following are the qualities we emphasized in the design and implementation of PWB/MM, in approximate order of importance:

- *Robustness in the face of error*—When the input is incorrect, either a reasonable interpretation is made of an error, or a message is produced when an error is detected.
- *Ease of use for simple documents*—It is not necessary to specify complicated sequences to produce simple documents, especially those following any reasonable format. Default values are provided where at all possible.
- *Parameterization*—People have many different preferences in the area of document styling. Many parameters are provided so that users can customize output to their needs over a wide range of styles. These parameters are set once at the beginning of the document. The formatting codes are entered consistently, regardless of the parameter settings.
- *Extension by moderately expert users*—We have made a strong effort to use mnemonic naming conventions and consistent techniques in the construction of the macros. A user can add new macros or redefine existing ones if necessary. There is also the provision, in certain cases, to have user-defined macros called at appropriate points within the provided macros.
- *Device independence*—The most common use of PWB/MM is printing documents on hard-copy typewriter-like terminals, using *nroff.* Output can be printed on either 10-pitch or 12-pitch terminals and can be examined on an appropriate CRT terminal. Finally, phototypeset output can be produced using *troff.*
- *Minimization of input*—The design of the macros lessens repetitive, unnecessary typing. For example, to obtain a blank line after all first or second level headings, the user need only set a specific parameter once at the beginning of a document, rather than add a blank line after each such heading. A table of contents, as well as other information derived from the input text, can be generated automatically.

3.2 Features

PWB/MM features permit the user to concentrate on the *logical structure* of the document, not on its eventual *appearance.* We feel this is a desirable direction of evolution for text processing. Some specific examples include the implementations of headings, various styles of lists, and footnotes.

3.2.1 Heading Styles. The formatting code ".H" specifies a new heading. The arguments to this code indicate the level of the heading (first or top level, second level, etc.) and the text of the heading itself. For example, the two headings immediately above were generated by:

```
.H 2 Features
.H 3 "Heading Styles."
```

One does not need to specify the number to be assigned to a heading. Headings are automatically numbered when the document is formatted. Thus, if sections of text are rearranged (which is very easy to do with the UNIX editor *ed*), one need not worry about renumbering any affected headings.

The default heading style produces numbered headings, as illustrated by this paper. By setting some parameters before the first heading, the user can easily modify this style to obtain, for example:

- any of various outline styles, replacing the default sequence "1., 1.1, 1.1.1" by, for instance, the sequence "I., A., 1.";
- centered headings in place of left-justified ones;
- different fonts (or underlining style) for each level;
- different choices of pre- and post-spacing.

What must be noted is that although the style may vary, the way of typing a heading does *not*. A few global parameters control the final overall appearance.

This approach not only contributes to uniformity of style within a document, but also allows the user to make radical changes in style *after* the document has been entered. Finally, the same text can be included in several documents that must adhere to differing standards, as in the case when an internal report is submitted to a journal that requires another format.

3.2.2 Lists. Various kinds of lists are often needed:

Ordered lists	Items within the list ("list items") are numbered or "lettered" sequentially.
Unordered lists	Items in the list are marked in some way, say, with a dash or a bullet.
Definition lists	Each item in the list is a word or a phrase followed by an explanation or definition. (This list is an example of a definition list.)
Reference lists	Lists of references used in the body of the text.

As an example of the ease with which such lists can be entered, Figure 2 shows the input text for the above list.

```
.VL 18
.LI Ordered\ lists
Items within the list ("list items") are
numbered or "lettered" sequentially.
.LI Unordered\ lists
Items in the list are marked in some way, say,
with a dash or a bullet.
.LI Definition\ lists
Each item in the list is a word or a phrase
followed by an explanation or definition.
(This list is an example of a definition list.)
.LI Reference\ lists
Lists of references used in the body of the text.
.LE
```

Figure 2

All formatting codes begin with a period in column 1. Each kind of list in PWB/MM consists of the following three parts—a "list begin" which specifies the type of list (.VL in this case), one or more "list items" (.LI followed by the text), and the "list end" (.LE) marking the end of the list. All information about the kind of list appears in the "list begin." It is very simple to add, delete, or reorder the items within a list. For ordered lists, the numbering or lettering is handled automatically so that, in inserting or deleting items, one need not manually renumber any of the items. Lists can be nested to a depth of six levels.

3.2.3 Footnotes. To enter the text of the footnote and to generate the next footnote number, one need only type *F after the word to be noted, followed by the text of the footnote delimited by two formatting commands. For example, the following input text:

```
This is the word\*F
.FS
The text of the footnote
goes here.
.FE
that is to have a footnote.
```

will produce the following output:

This is the word[1] that is to have a footnote.

as well as the footnote at the bottom of this page.

PWB/MM automatically records the footnote for subsequent inclusion at the bottom of the current page. If there is not enough room on the current page for the entire footnote, the remainder of the footnote is saved and printed at the bottom of the next page. All the mental processes of the typist in handling footnotes (gauging the length of the footnote text, remembering to bring up the bottom margin by the proper amount, etc.) are handled automatically.

3.3 PWB/MM Summary

PWB/MM supplies many services in addition to those described above. The end user obtains the benefits of complex formatting procedures without having to learn the techniques of their implementation. Note that PWB/MM represents a *layered* approach to text formatting: it is a second layer built on *nroff* and *troff*. We feel that a common formatter design error is that of building *too much* into the formatter itself, rather than using a layered approach to formatting complex documents. For example, the fact that the footnote mechanism was not locked into our formatters allowed us much more flexibility in creating variant footnote styles.

4. CASE HISTORIES

The UNIX environment permits quick construction of working systems from existing components, easy adaptation of existing files to new purposes, and inexpensive modification of systems to meet changing needs. To support these assertions, we describe very briefly a few applications chosen from the many that have been built in the last two years.

4.1 A Text Processing Application

One of the Bell System companies had a special text processing requirement. The company conducts a weekly management seminar and is required to prepare timely reports describing each session—what material was presented, what comments were made, a list of any unresolved questions, etc.

4.1.1 Text Production and Analysis. The initial need of the seminar is a text processing facility—the ability to enter, update, and print various kinds of reports for each session. The reports are printed on hard-copy terminals for editing and author review. The final reports are phototypeset for distribution to top management.

4.1.1.1 Report Production. The reports are entered using the UNIX editing system, embedding within the text of the reports certain formatting codes. A precursor to PWB/MM, these formatting codes were designed to be used for either *nroff* or *troff*. The *typo* program is also utilized to help find typographical errors.

Each session closes on a Friday; by the following Wednesday, a 70-page booklet containing the phototypeset originals of all the material from that session is ready for reproduction. The staff in

1. The text of the footnote goes here.

charge of the seminar has been quite pleased with the documents produced. They have stated that there would have been no other way to produce the same high-quality output in the same period of time.

4.1.1.2 Analysis of Content. With several weeks' data stored on the system, the question arose as to possible ways of extracting portions of that data. It appeared reasonable to use the system to produce another report consisting of all paragraphs containing the word *marketing,* for example.

This analysis was accomplished by considering the structure of the text in the report files. Due to the use of the formatting codes, it turned out that all sections of text (paragraphs, items in lists, etc.) were preceded and terminated by specific character strings. Although a paragraph might contain the context word (like *marketing* or *Marketing*) several times, such a paragraph is to appear only once in the output report. Thus, it was necessary to use the editor and the *sort* command to obtain a unique list of the line numbers of "paragraph beginnings." Another editor script alters this list of unique numbers to obtain yet a third editor script that causes all the lines of the paragraph to be printed, taking care of cases such as paragraphs that are followed by lists. Another pass through the original file using this last *generated* script produces the desired output.

Once these scripts were developed, it became a very simple matter to put them together in a *shell* procedure. All that the user needs to specify in invoking this procedure is the name of the file containing the data and the context word by which the paragraphs are to be selected.

4.1.2 Conferee Roster. Another text processing task is to maintain a list of conferees scheduled to attend the seminar. This list is periodically updated since persons are often rescheduled due to business demands. The roster of conferees for a given session might not actually be finalized until a few days before the beginning of the session.

The data for all the conferees attending a given session are entered into the same file, whose name is the week during which the session is held. The data for each conferee is entered as shown in Figure 3, using formatting macros for each item of information needed. The "company" data includes special two-letter codes for all Bell System companies.

```
last name           .LN  Smith
first name          .FN  "Dale W."
title               .TL  "Member of Technical Staff"
company             .CO  \*(BL
address             .AD  "6 Corporate Pl., Piscataway, NJ"
telephone           .TN  "201 981 7315"
department          .DP  "Support Products and Systems"
level               .LV  0
name/address flag   .NA
```

Figure 3

After several sessions of the seminar, the accumulated rosters took on added importance. It is possible to scan the data to determine whether or not a particular person has ever attended the seminar and, if so, the week of attendance. For instance:

```
grep "^\.LN.*name" filenames
```

will list all files containing an entry for a person whose last name is *name.* Providing this information by manually scanning printed rosters is time-consuming and error-prone.

With this data, reports listing such information as:

- Who has attended from the marketing division?
- How many fifth level managers from XYZ Company have attended?
- What engineering personnel from XYZ Company have attended?

are very easy to obtain. What is needed is a way to say "Give me all the data about conferees who have this property." This is accomplished by having the user specify, to a shell procedure, what the desired property is (utilizing, of course, information about the structure of the data). To obtain all the data about the conferees from any given week who are in the marketing department, one would execute:

```
ed – filename  >output
g/^\.DP.*Marketing/?^\.LN?,/\.NA/p
q
```

By first placing all the conferee data into a temporary file and then executing the above editor script on that file, one obtains the data about *all* conferees from marketing.

One can further categorize the extracted data by executing a similar editor script looking for the desired property. For example, if the file *mkt* contains all the marketing conferees, one could obtain all fifth level marketing people by

```
ed – mkt  >fifth
g/^\.LV.*5/?^\.LN?,/^\.NA/p
q
```

By providing alternate definitions for the formatting macros, one can generate a data record for each conferee. These records can then be sorted in various ways (by company, by name, etc.). Again, the ability to easily combine existing UNIX facilities proved invaluable in extracting additional information from existing data.

4.2 A Data Base Application

Any programming project that involves more than a few people seems to generate innumerable small data bases to record information needed for documentation and control of the project. Such data bases are often small (a megabyte or less), need to be implemented in a timely fashion, and vary greatly in nature from project to project. To handle this sort of data base, one can either utilize an existing general-purpose data base management system (DBMS) or write several programs to implement the desired application. The former approach often yields a severe form of overkill, while the latter may require too much programming effort and unacceptable delays.

A third approach is that of building an extremely specialized DBMS for the specific application. Programming costs and lead time are minimized by writing command language procedures that combine existing tools in appropriate ways. The advantages of this approach are: speed of implementation, satisfaction of specific needs of the application, small size of code (no compiled code at all), ease of maintenance, reliability, and good output appearance (e.g., by using *nroff*). The main disadvantage is the possible lack of execution speed. Also, there may be some inconvenience in handling larger data bases or those whose structure has a bad match to the UNIX file system. On the whole, this approach appears to be quite worthwhile; it has been applied very successfully a number of times.

As an example, the first such system known to the authors was built for a large programming project during September, 1974. The project had been using an obsolete DBMS to store information regarding error messages produced by the project, and a better DBMS was desperately needed. The project includes a group of 400 modules, each of which can produce error messages of two different types. Certain information must be maintained for each error message, such as the cause of error, actual message text, response required, responsible person, etc. There are about 3000 messages altogether, requiring about a megabyte of disk storage, organized into 550 UNIX files. One author (Mashey) spent about three work-days (spaced over a period of two weeks) as follows:

- Discussions were held with project members to determine their requirements.

- Data formats were designed.
- Procedures were written to add, update, delete, and print individual items in the data base.
- Procedures were built to produce various kinds of summaries and large reports.
- Procedures were included to maintain logs of data base changes.
- All of the procedures were tested and (more or less) debugged.

By using existing tools and by modifying the package according to changing user perceptions, a usable system existed at the end of two weeks. Over the next few months, the data base was loaded with information from the old system—the major part of the time being required for data purification. The total time spent by the author on this project was about two work-weeks, of which half was documentation time. By February 1975, the entire project had been turned over to a relatively new UNIX user who has had little trouble in modifying and extending it. At turnover time, the entire package of procedures and formatting macros consisted of nothing but text files, occupying a total of 30,000 bytes.

This system is used in various ways. In one sense, it is a small DBMS; in another, it is just a way of manipulating a large, complicated document. Finally, it is also used to verify the consistency of the actual programs with their documentation. A selected subset of the information is extracted and sent to the machine on which the application project executes. It is then compared with the actual message text.

It is interesting to note that data base management software *per se* has not proved popular on UNIX to date, because more flexibility can be obtained through the appropriate use of *sort, ed,* and other UNIX commands.

4.3 Production of Revision Bars

Existing tools can often be combined to produce novel results, although sometimes a new tool must be built to fill a gap between some others. An example is the process used to generate automatic revision bars (or other marks) for modified documents. When manuals are revised and distributed to users, such revision bars are placed in the margins to indicate the locations of changed text. The formatters support a mode of operation in which a specified character is placed in the margin of every line of output text. This mode may be turned on and off by requests in the formatter input. The only problem is that of determining where to insert these requests. The following command line illustrates the entire process:

```
diff – old new | diffmark temp | ed – old; nroff temp
```

Diff compares the two files, and generates an editor script to convert file *old* into file *new.* The resulting script is dynamically piped ("|") into *diffmark,* which rearranges this script and then passes it to the editor. The editor, in turn, edits the *old* file, but writes the result into *temp. Temp* is identical to *new,* except that formatter change-mark requests are included in appropriate places. *Nroff* then formats the result with the needed revision bars.

Diffmark is a small, simple program written specifically to bridge the gap between the other commands and was intended only for use with documentation. As commonly happens, a number of people immediately applied the same process to other kinds of text, i.e., source programs and data base schemas.

In some cases, a user simply makes a copy of a file before editing it, performs the process described, and removes the *old* and *temp* files after *temp* has been printed and distributed. If more extensive control is desired, SCCS [ROC75A] can be used to maintain complete control of change history and recreate any needed version of a file. This is a good example of the unexpected advantages of keeping documentation and source together in an environment where they can both be handled with the same tools.

4.4 Miscellaneous Applications

4.4.1 Viewgraph Production. With the phototypesetting software *troff* and a set of formatting macros developed for the PWB, one can easily prepare viewgraphs for lectures, demonstrations, etc. The advantages for doing this are as previously stated—the ability to easily modify the data, the ability to include actual programming statements, and the assurance of accuracy during revision.

4.4.2 Bibliography Management. To avoid wasted effort when papers are written, especially by groups of people, a bibliography data base system was built from ordinary text files and a few shell procedures. A unique key is assigned to each reference as it is added to the data base. A document is scanned for keys of this form, and a list of unique keys extracted and sorted. The list is used to search the bibliography data base and extract the items to be included in the reference list. The procedure to perform all of these actions was written, coded, and debugged in 20 minutes. It was used to simplify the preparation of the references for this and the related papers.

4.4.3 Milestones and Scheduling. A need arose for a simple, easy to use way of managing schedule dates and activities for individuals and groups of various sizes. It was an afternoon's work to create a package of formatter macros, text files, and shell procedures to accomplish this. Schedules can be summarized at various levels, and ordered according to half a dozen sort keys. The entire package consists of about six pages of text.

5. CONCLUSIONS

We have stated the opinion that a software development facility needs a good, integrated set of text processing tools. We have summarized the tools most commonly used, and described a few of their applications to support our opinion. They are but a very small sample of those available; UNIX makes it almost too easy to invent and implement useful applications on the spur of the moment. From our experience, this a very desirable situation since many useful, effective aids have been quickly and cheaply built this way. In many other environments, they might well be too small, too difficult, or too expensive to ever be done.

REFERENCES

All references cited in this paper appear at the end of *"An Introduction to the Programmer's Workbench,"* by Dolotta, T. A., and Mashey, J. R., in these Proceedings.