The life and times of **ded**, text display editor

Richard Bornat

Harold Thimbleby

Scanned from and originally published in *Cognitive Ergonomics and Human Computer Interaction*, pp225–255, edited by J. B. Long and A. Whitefield, Cambridge University Press, 1989. ISBN 0-521-37179-1.

8.1 Introduction

Ded is a text display editor designed by computer scientists. The design is characterised by simplicity and adherence to user interface principles. This has led to a good design but with contentious features. This chapter highlights the conflict between principles and features, particularly in the social context in which ded was designed.

The primary intent of this chapter is to present the context and design decisions lying behind a particular interactive system. Neither the design itself nor its description here has been directly influenced by the book's framework; it seems more honest that way. However, the following salient points may be noted:

- Science is considered as a form of explicit, public knowledge. The public (i.e., user) accessibility of explanation of **ded** was a criterion of satisfactory design. As designers we also explained **ded** to ourselves by way of principles; thus, the role of principles was paramount, as we hope will be made clear in this chapter.
- The designers of **ded** were embedded in the environment for which **ded** was designed, though with certain concessions for some users at the periphery. Thus the acquisition representation was phenomenological.
- The development paradigm is not seriously tested by a close-coupled two-designer system. In our case, the various distinctions (analysis; generalisation; application; synthesis) were not explicitly formulated, and need not have been for a successful conclusion.
- The design of a practical system is clearly an engineering endeavour. Our design of **ded** was, however, analogous to a scientific endeavour: we used principles (qua scientific representations). We made predictions about interaction between principles, explored them, and synthesised new principles.

Ded is a text editor that was designed at a time when computers were becoming much more accessible to certain groups of people, mostly located in university departments of computer science, who found that they were using computers for long periods at a time, perhaps for the whole of their working day, with no practical limitation on the amount of 'computer time' they consumed other than a vague obligation to be fair to other simultaneous consumers. At the same time the hardware they touched—the 'terminal' was changing as slow hardcopy printer/keyboard terminals were replaced by much faster visual display terminals (henceforth VDTs).

Ded has been very popular with the people who have used it, but many who were already used to more sophisticated editors have never been tempted to change allegiance. We do not know whether this is because **ded** is really worse for the applications those programmers have in mind or because people imprint on their first editor. There are many other plausible explanations, such as peer pressure and the so-called learning paradoxes caused by narrow horizons where users want to get their present work completed *now* in preference to investing the time to learn a new system which might improve their *future* performance (Carroll, 1987).

Certainly, some people have found **ded** good enough to go to the trouble of porting, which is no easy task. **Ded** has been ported to many machines and has formed the basis for a study in the formal

specification of interactive systems (Sufrin, 1981).

8.1.1. Why write about a 1970's system?

Ded is years old; and user interfaces, their sophistication and our understanding of them, have moved on. Why write—or read—about it? First, ded is worth talking about. It is a good system, and we want to try and explain why.

We suspect that a lot of software products are designed along the lines that if something is imaginable and possible then it is certainly permissible, and may as well be provided as another feature of the program. With **ded** we wanted to provide more than was possible (we were using small machines by present standards): we had to do less than this, and we turned to principles for guidance. Of course, we had to invent the principles too. This design strategy—analyse the task and its constraints, devise theories (we called them principles) and then synthesise a design consistent with that theory—is basically the theme of this book. Following design principles is essentially 'top-down' design; but of course, devising the principles themselves is 'bottom-up' (plus experience and inspiration). Design by searching for principles is a method approved by Descartes, "Each problem that I solved became a rule which served afterwards to solve other problems" (Œuvres, vol. VI, Discours de la Methode). It is interesting to see how it works out in practice.

Ded is a very simple system—which is very hard to appreciate by *reading* about it, especially since we have emphasised design details rather than the subjective feel of using it. **Ded** doesn't have many features, but an enormous effort went into its design. In fact, **ded** existed in almost complete, but unpolished, form early on: we spent most of the design time (about twelve man-years part-time, three or four full-time) improving and polishing **ded** rather than adding new features. We wanted to perfect rather than camouflage and palliate, which would indeed have been a much easier route.

Because **ded** is so simple we can talk about a significant and useful part of its design in a short chapter. Nevertheless we have hardly done full justice either to history (there is a lot that is not recorded here) nor have we done full justice to **ded** itself (there are many design issues not addressed in this chapter). As we explore design alternatives in this chapter, social influences, principles, hardware limitations, user tasks...try and imagine how much more difficult it would be to design a big system—say like an airline reservation system—*really carefully*. The most interesting thing in design is not the principles, but how they are compromised, because they surely will be—indeed, it has been suggested that complex design cannot be done properly but only faked (Parnas & Clements, 1986).

Like many other computer scientists we believe that programming has nothing to do with computers: that it is an abstract business, to do with the description of the behaviour of abstract machines. Equally it seems to have nothing to do with social factors. But the design of software products is heavily influenced by the characteristics of computer hardware and the ways in which they are commonly used. Thus, we are interested in exploiting technology to its utmost. VDTs don't cost money to run, because they don't print on paper, and they work fast enough to be a plausible alternative to paper and pencil. By using up lots of computer time you can make them do some very interesting tricks indeed. The combination of almost unlimited computer time with free printing at the terminal made tempting overtures to a software designer. **Ded** was designed to play the nicest tunes we could on the best hardware we could lay our hands on.

8.2 Ded: a quick overview

Text editors are as-it-were 'stripped down' word processors, with out any provision for other office procedures such as mail merging, printing, drawing diagrams, checking spelling; they are possibly one of the most widely used forms of interactive system. They are most often used in environments where the text, once edited, will be further processed by other programs (e.g., text formatters, spelling checkers, compilers etc).

The main feature of **ded** is that it is so simple it can be used straight away in almost complete ignorance of its full repertoire of commands. It is not a 'space-cadet' system. You cannot type a single key and expect to get to the end of the known universe. You *can* get there if you really want to, but you cannot get there quickly by a simple typing mistake!

The basic operations of **ded** can be presented very simply. The keyboard has keys labelled with up,

down, left and right arrows and a RUBOUT (or DELETE) key. The explanation to the novice user would say that you type normally, you use the arrow keys to move up, down, left and right in the stuff you are writing and use the RUBOUT key to remove bits you don't like. When you have finished you press the ESCAPE key, type ok and press RETURN. That explanation is all you need to be able to construct letters to the bank manager, simple programs and so on. If you need to know more you can ask somebody eke. The rapid feedback (for *every* key press[1]) indicates to the user exactly what they are doing. In summary, you can use **ded** like a magic typewriter.

<table>
<tr><td>START LINE
<<<</td><td>SCROLL UP</td><td>END LINE
>>></td><td></td></tr>
<tr><td>WORD LEFT
<<</td><td>UP
↑</td><td>WORD RIGHT
>></td><td>GO TO
COMMAND LINE</td></tr>
<tr><td>LEFT
←</td><td>CENTRE</td><td>RIGHT
→</td><td>EXECUTE
COMMAND</td></tr>
<tr><td>NEXT LINE
↲</td><td>DOWN
↓</td><td></td><td rowspan="2">NEW
COMMAND</td></tr>
<tr><td colspan="2">SCROLL DOWN</td><td></td></tr>
</table>

Figure 8.1: Typical keypad arrangement

On some terminals, such as those modified at Queen Mary College (by Ben Salama and Derek Coppen), the user may have further hints from the labels on other keys—for instance that the cursor may be moved in units of words, or that things other than the character under the cursor can be deleted. The 'numeric keypad' to the right of the main QWERTY keys on PC-style keyboards, may be arranged for **ded** as in Figure 8.1. Sometimes the keys will be labelled, but on some terminals the user has to memorise where each key is; they will be laid out fairly logically so this is not difficult. For the PC keyboard, pressing the shift key labelled ALT causes the motion keys (such as word left) to delete what they would have moved over. Thus it is possible to delete left and right in units of character, word and line from the cursor position, as well as to delete lines up and down. The central key scrolls the screen so that the cursor is positioned in the middle, to provide the user with equal amounts of context above and below the current line. This sort of keypad arrangement, combined with **ded**'s carefully-designed word and line motion behaviour (Thimbleby, 1981) greatly facilitates touch typing—none of the awful hand-repositioning and cursor-overshoot of a mouse-based system.

A user invokes **ded** by specifying a file to edit. **Ded** reads the file (or creates it if it does not already exist) and shows a display something like Figure 8.2; though the text displayed would normally be as large as the terminal screen, typically around 25 lines, rather than the 9 we show here.

```
    The main feature of ded is that it is so simple
    it can be used straight away in almost complete
    ignorance of commands.
    All a user needs to know is that what he types gets
    inserted into the file, that arrow keys move the cursor around
    (so that he can insert text at any point),
    that the delete (or rubout) key deletes text, and
    lastly how to leave the editor using the 'ok' command.
>
```

Figure 8.2: Screen layout (on a small screen!)

The screen is split into four regions. The top right region, the biggest of the four (about twenty lines deep and seventy-eight characters wide on typical terminals), displays the user's text. The cursor is shown in Figure 8.2 at the top left hand corner: the arrow keys can be used to move it anywhere within the region. The bottom line is a command line, where the user can type special commands to **ded** that are not intended to form part of the text.[2] For example, '`ok`'—the command to leave **ded**—would be

[1] If a key does nothing visible to the user, it *really* does nothing (but it rings a bell to let the user know).

[2] The command line is on the bottom rather than the top for reasons discussed in Granada and Teitelbaum (1982)— though we had already decided before reading the paper.

typed on the command line. The '>' symbol is fixed, and helps to make the command line appear separate from the rest of the display. The left hand column is for more advanced use: it is used for naming regions of text. A special command names a line or several lines of text; other commands can be used to do their work within such a region of text rather than on the entire file.

```
      The main feature of ded is that it is so simple
      it can be used straight away in almost complete
      ignorance of commands.
a     All a user needs to know is that what he types gets
a     inserted into the file, that arrow keys move the cursor
a>    (so that he can insert text at any point),
a     that the delete (or rubout) key deletes text, and
a     lastly how to leave the editor using the 'ok' command.
>     ok_
```

Figure 8.3: Screen layout as really seen

The lines between regions in Figure 8.2 don't really appear on the screen. Figure 8.3 shows what would actually be shown after the user has named a few lines 'a' and is about to execute the 'ok' command to leave **ded**. The bold '**>**' (darkened in this book, but bright on white-on-black terminals) mid way up the screen indicates the line the cursor used to be on, before the user moved it to the command line at the bottom.

You might mistype 'ok' (as 'ko' perhaps). The command that you type is visible on the screen so that if the editor didn't recognise it you can at least read it to see what might be wrong. It is natural to consider that it might be changed: indeed, it can be treated just like the main text and by using the same keys. You can insert and delete characters within a command and then ask for it to be carried out.

Complete novices can use **ded** successfully after a few minutes experimentation. They have much more trouble with the QWERTY keyboard, with the idea of files and of storing texts, with the notion of constructing UNIX commands in sequence, than they do with constructing and modifying texts with **ded**. The idea of using programs—like **ded** and text formatters—to process text held in files is difficult to grasp at first. The fact that UNIX commands don't mean anything to **ded** and **ded** commands don't mean anything to UNIX is confusing too, and the conventions of most formatters are incomprehensible to anybody in the limit. **Ded** is simple because it concentrates on a part of the documentation processing job, because it can only be approached by somebody who has understood the basic mystery of file storage and is prepared to be introduced to a corner of the impenetrable mysteries of formatting. In part we are hiding behind the complexity of other parts of the UNIX system: we could make **ded** as simple as we wished, so long as some bit of UNIX can be relied upon to do the rest of the job.

There is no 'help text' in **ded**, although we were often urged to provide some. We were and are quite happy with the fact that naïve users of **ded** use only a portion of its facilities and have no idea that the rest of it exists. They find out more about it from talking to other users who know more—ded was designed for a timesharing environment where you are never far away from somebody else who is using the same machinery. After a few years we wrote a full manual—it ran to about seventy pages and the only thing we remember about it is that some wag said that pencil, paper and rubber don't need SEVENTY pages to explain.[3]

8.3 Historical background

Historically, text editors were heavily oriented to the medium carrying the text. Thus, in the 1960s and early 1970s paper tape was a quite popular medium for storing text: users could edit paper tape by taking scissors, poking holes and using masking tape to obscure unwanted holes. The user clearly had to be *au fait* with the binary character code, but otherwise the operation of the 'editor' (i.e., splicer and punch) was 'direct manipulation'. The tape itself did not record the history of the editing: all that could be seen from the tape was the *new* text (and some bits of masking tape). Searching for the right place to fiddle was greatly aided by running the tape row-by-row through a reader/printer/punch

[3] Writing the manual helped uncover a few minor bugs, so it was not a totally wasted effort. On the other hand, if we had not written the manual, perhaps nobody would have even thought of trying the things that did not work properly or even known what they should have done!

terminal so that you could read the corresponding text.

It soon became popular to use interactive computer terminals. Now the computer, rather than the user, could perform the searches to find the places in the text where changes were wanted. Edits would be specified by instructions: for example, to delete a line of text or to replace a line with some new text. Hardcopy terminals, such as electronic typewriters, make a record of each instruction. Now the record of an edit is the history of the changes: the sequence of instructions that together dictated the edit. Here is an interesting change brought about by the technology: on-line terminals certainly support a more flexible and easier-to-use way of text editing, but the representation of the editing is now more complex. For many users the history of the editing became distracting—a price to pay for the extra flexibility. Figure 8.4, a typical editing session in this style, cannot be understood without knowing the detailed *history* of the user's interaction; indeed some of the text might even be the result of an earlier print command (i.e., the apparent interaction might be text in the user's file) and then you would be thoroughly confused.

The history could exhibit lines of text in any order on the paper: the fact that one line of text preceded another would not mean that it *textually* preceded the other, merely that the user had *examined* it earlier. It was easy for the pressurised user to confuse the two and make mistakes. There is a trade-off to be made, between the simple-but-tedious text editing of the paper tape era and the complex-but-fast text editing of the on-line terminal era.

When the visual display terminal became widely available in the late 1970s it became possible to reappraise the design of text editors. Now it was possible to display a page of text on the screen and to show how it changed under each instruction to edit it. Just like the old paper tape editor, there was no need to actually show the sequence of instructions used for making the changes.

Such text editors should be easier to use (assuming that the user is not especially interested in the history of their session)—so long as users can reliably construct the commands for editing the text without knowing the history of previous edits. Since the editor does not display the history, there should be no reason for the user to have to remember it.

Conventionally, a 'history-free' interactive system is called *declarative*, to distinguish it from a *procedural* (or *imperative*) interactive system. Very often it turns out to be difficult to provide a purely declarative system—the compromises are often called *modes*, being information the system keeps to itself. There is some debate—that will be touched on in this chapter—over the merits of modes: some modes for certain purposes appear to be necessary and useful; others appear to be superfluous and damaging for the user. Modes mean that the user has to do or say less, but run the risk that the mode information hidden in the computer is not what the user thinks it is. The fewer modes, the more declarative a system and (as is now widely argued in support of 'fifth generation' computing projects, such as Prolog) the easier systems should become to do powerful things.

```
/changes/
and show how it changes under each instruction to edit it.
-
Now it was possible to display a page of text on the screen
+
and show how it changes under each instruction to edit it.
s/changes/changed/p
and show how it changed under each instruction to edit it.
/othe /
Unlike many othe Unix tools it is generally
s//other/
?on-line computer?
It soon became popular to use on-line computer terminals
c
It soon became popular to use interactive
computer terminals.
```

Figure 8.4: Transcript of an editing session
Notice that it is not obvious what text is typed by the user

Ded is a declarative low-mode editor: quite a novel idea for its time. It was so novel a concept, we did not know what it was called when we did it; we called it *picture editing*, an idea that we discuss more fully below in section 8.6.5. The declarative nature, and our deliberate adherence to it, makes **ded**

radically simpler than every other editor known to us.

8.4 Design versus evolution

The most important thing about the design of ded is that it never happened, in the sense that there was never a time when a group of people—computer scientists or ergonomists or anybody else—sat down together and thought about what the new editor product would be like. Perhaps it would be truer to say that there was no one time when we sat down together and that we didn't sit down together *before the editor existed*. There were lots of times when people sat down together. There was a lot of design activity, but it didn't precede production, it suffused it. It was rather like what is nowadays called 'prototyping' except that in our case there was always a fully working version of the product to be discussed.

We like to believe that **ded** wasn't designed, it evolved. Richard wrote the first version because he needed it for his work, colleagues begged to be allowed to use it as well, and the bandwagon was rolling. We worked out design principles during the evolution of the design: **ded** was rewritten at least three times to fit new ideas of what those design principles should be, and at least one of those rewritings was thrown away when we reverted to previous versions of our principles. Our evolutionary process was a sort-of genetic engineering rather than natural selection.

Our notion of the target population of users changed over time. From a program written for one person to use, we eventually believed we had developed one which could be useful to the secretarial staff in our departmental office, who spent too much time retyping documents which could easily be word processed. Nowadays they use **ded** all the time. Later still we believed it would be useful to novice undergraduate computer scientists, who have problems understanding just about everything they come into contact with[4]. We were much less responsive to the needs, or at least the expressed demands, of fellow programmers: they could write their own editor if they didn't like the way we did things.

Social factors helped to make the design process evolutionary. The Computer Systems Laboratory at QMC tried to be a non-hierarchical institution, at least so far as academics and research students were concerned. Nobody felt empowered to say what other people should be doing. Sometimes they felt they should tell other people what they should not do, for example because they felt it was wasteful of lab resources, but Richard's position as a senior academic meant that he could get away with most things. So we could design **ded** without asking anybody's permission beforehand and we could make it do just what we wanted without taking notice of anybody else's feelings or opinions.

Computer programmers don't keep their opinions to themselves very much and we were often informed of ded's supposed shortcomings and ways in which they might be overcome—sometimes we feel that we talked about little else, though Richard made himself responsible for design decisions. That made ded a very 'principled' program. If he thought a suggestion to improve it didn't fit with the current set of design principles, he didn't make that improvement unless and until he could be persuaded. He was often persuaded, but sometimes obdurate.

Ded was very much a product of us and our environment. We don't know what we would have done somewhere else, or what we might have done if we had had a different idea of the target users. Different approaches are possible under different circumstances: the design and motivation behind **ded** (partly covered in Thimbleby, 1981, 1982 &; 1983) can be contrasted with that of **`sam`** (Pike, 1987) and EMACS (Stallman, 1984). Meyrowitz and van Dam (1982) provide a general survey of text editing.

8.5 Early developments

Richard was in the middle of writing a book when he came to QMC in 1977. This was the major impetus for the design of ded. The rest of this section, section 8.5, describes Richard's background; it has been written in the first person.

My design ideas have come largely from experience of and dissatisfaction with other people's

[4] We also had some idea that the program would be too computationally demanding to be used by lots of students sharing one machine. But it seems that the students actually spent less time compiling and correcting incorrect programs: overall they were better off.

*software products. In the case of **ded** my dissatisfaction was with the way that contemporary software appeared not to exploit the capabilities of the new VDT technology and to waste the free computer time which I was beginning to enjoy. I had already accumulated about a hundred pages worth of text in machine-readable form. I wanted to be able to review this text, to modify it and to add new text, to reorganise its structure of sections, chapters and paragraphs, to write whole new chapters. I wanted to use what is nowadays called a word processor and since nothing like that existed I had to invent my own.*

I had constructed my hundred pages in the first instance by using an imperative line-based editor, though in the later stages I had used a declarative picture editor which displayed twenty lines or so of text on the screen and allowed simple overstriking and insertions. All that had been done using what now seems ridiculously inadequate hardware, which could print between ten and thirty characters per second. The hardware at QMC could handle up to a thousand characters a second. I could see that it would be possible to compose text on screen, to type paragraphs and to read them in their context without having to specify what the context was and for the first time to move upwards *in the text, to show what was off the top edge of the screen.*

In inventing my machine for writing the book I drew upon experience with all sorts of earlier text editors which I had used. Major influences were the line-based editor **SOS***, which I had used on the DEC PDP-10, my experiences in using an editor called* **SYRUP** *I had written for my own use on the PDP-10 and* **em***, which was a version of the* UNIX *text editor* **ed** *devised by George Coulouris running on the PDP-11. Apart from* **SYRUP***, all of these were imperative history-based editors*

Like many other people at the time, I was fascinated by the possibility of producing 'typeset' text. UNIX *had and still has programs called* **nroff** *and* **troff** *which do this job. They work with text which is interspersed with* formatting commands*; for example the previous paragraph was produced from the text shown in Figure 8.5.*

```
.PP
\f2In inventing my machine for writing the book I drew upon
experience with all sorts of earlier text editors
which I had used.
Major influences were the line-based editor \f4SOS\fP, which
I had used on the DEC PDP-10, my experiences using an editor
called \f4SYRUP\fP I had written for my own
use on the PDP-10 and \f4em\fP, which was a version of the
\e*U text editor \f4ed\fP running on the PDP-11 devised by
George Coulouris. ...
```

Figure 8.5: **troff** typesetting commands

The most important thing about text to be given to a typesetter is that the original layout hardly matters. The typesetting program works out where words are to go on the page, places headings, decides on margins and so on: the result is quite different from what you type. When original textual layout does matter it is in order to place particular characters at particular positions on the page, to produce a particular character-picture in the formatted text, and character-pictures can quite easily be built up character by character. That meant that I could use a 'no-frills' text editor to construct my book, an editor without facilities to arrange and format words on a page.

8.5.1. Idea: Only one mode[5]

I wanted to be able to compose English text on screen. I wanted to survey the text as I typed it, so I wanted it at every instant to show me exactly what I had typed. I didn't know what to do about insertions and overstrikes—that is, whether a key depression should signal the editor to replace the character at the editing position (overstriking) or to insert a new character before the one at the editing position (insertion).

At the time, because of limitations in VDT hardware and shortages in computer time, many

[5] Pedants may notice that we use the terms 'one mode' and 'modeless' apparently interchangeably. Technically a system must have at least one mode, but we can understand a system appearing to be modeless if its user can cease to be aware (and not need to be aware) of the one mode it is in.

declarative (history-free) editors worked normally as overstrike editors, but could be commanded to make an insertion: then they inserted a wide space in the text into which the insertion was typed by overstriking, closing up the gap when the insertion was completed. I didn't like that idea because it meant that you couldn't easily read a sentence you were modifying: the big space made it hard to read and quite often part of the sentence fell off the right-hand edge of the screen. I'd experimented, in the editor **SYRUP***, with a more computationally expensive insertion technique which put in a character every time you pressed a key, and in particular rubbed out the last character you had typed whenever you pressed the* DELETE *key. But I didn't know what to do about deletion when the editor wasn't in insertion mode.*

Here a happy accident intervened. In a conversation with Jon Rowson, one of my colleagues in the lab, I was introduced to the idea of a 'modeless' editor, one which always worked using insert/delete and never allowed overstriking. It just seemed a good idea, and I adopted it immediately.

In order that I could type rapidly I made the editor put in an automatic line-splitting command whenever the text got close to the right-hand margin of the screen, but I made no further concessions to layout.

8.5.2. Idea: Search for anything

The hundred pages of book text that existed in 1977 were in **runoff** *format: runoff was the remote ancestor of* **nroff**, **troff**, *TEX and countless other typesetting programs. QMC's* UNIX *machine had* **nroff**, *and* **nroff** *would enable me to typeset pages which were much more readable than* **runoff***'s attempts. Both programs use typesetting commands which appear on single lines within the text to be set, but their command sets are different. In one very important and influential case—footnoting—it was necessary to change more than one command at a time.* **Runoff***'s footnoting commands were as shown in Figure 8.6.* **Nroff** *recognised more compact commands and allowed automatic footnote numbering, so I wanted to alter that text to the form shown in Figure 8.7.*

```
information is prepared about the arguments[8.1]
.FN9
.Bl .F -NJ
-------------------------------------------------------------------
[8.1] I have used 'argument' to describe information passed in
the procedure call and 'parameter' when used by the called
procedure.
.END FOOTNOTE
of the call;
```

Figure 8.6: Footnoting in **runoff**

```
information is prepared about the arguments\c
.fn
I have used 'argument' to describe information passed in
the procedure call and 'parameter' when used by the called
procedure.
.ef
of the call;
```

Figure 8.7: Footnoting in **nroff**

None of the text editors available at QMC could recognise patterns which extended outside a single line, let alone do substitutions which altered more than one line of the text. I wanted to be able to change my text systematically and I believed that it would be easier to write a program to do it for me rather than to go through the text laboriously, and probably unreliably, changing it piecemeal.

So I decided that I wanted to be able to search for and replace textual patterns which extended over a single line. I found that it was very difficult to type the commands which recognised something as complicated as the footnote pattern illustrated above. In its first incomplete form, whose notation took some time to evolve, this command would be as shown in Figure 8.8 below.

I don't think it's necessary to give an explanation of this notation—just feel the complexity!—but Appendix 3 gives some explanation for the curious.

```
x/['[-]']*]'^'.'*'^'.'*'----'.'*^['[-]']*]/\c'$.fn'$
```

Figure 8.8: A ghastly search & replace

8.5.3. Idea: Make everything visible and editable

Many editors do not make the command line visible; but if ded allows commands as complex as in Figure 8.8 then I needed to be able to see what I was doing! Once I had constructed a command which was recognisable to **ded***, it was probable that I hadn't constructed the right one. Usually I'd got the search pattern wrong, so that it didn't find anything in the text to replace. So I wanted to be able to edit the command line, using the same editing features as for editing other text.*

8.5.4. Idea: Make changes interactively

A more dangerous possibility was that my command described a replacement of some sort, but not the one I intended. That made me design an interactive replacement command which would show the effect of each replacement, in its context, for my confirmation or rejection. I borrowed this idea from the line-based editor **SOS** *which I had used on the PDP-10.*

8.6 Design principles

Because **ded** evolved, the design principles which we believed at the end weren't the ones we started with. There may be important features of the editor which are only explainable in terms of principles which we have forgotten. So far as we can tell from the notes we made at the time, we did have a number of principles in mind. Some of them may have been little more than platitudes, but we believed we followed them at the time—and we still believe in most of them now. The ones we believe in are described below.

8.6.1 Always show something is happening

When editing text something happens every time you press a key: a mark appears or disappears from the screen, the editing position cursor moves or in the extreme the terminal bleeps. Sometimes a command, particularly a search in a large file, can take a long time to carry out. It was very important to show that ded was working away in the background, even when the screen wasn't changing. We made it display a white blob mark at the end of the command line after hitting RETURN. Very simple, but quite important, not least because it dissuaded people from pressing the RETURN key again and again in the belief that ded hadn't noticed them the first time, which could produce nasty effects when the command finally finished.

8.6.2 Show the context, but keep the screen still

Because we wanted to compose and modify text on-screen we wanted always to see the context within which we were working. We wanted always to be able to read the text around an insertion, because what we were typing would typically be some sort of extended explanation of something else that was already in place. The problem is that 'context' is maintained by **ded**, but exactly what the context is depends on what the user is wanting to do—and **ded** knows nothing about that.

Showing as much context, text above and below the cursor, as possible seems a sensible strategy. It would seem to mean redrawing the screen image so that the insertion point was on the middle line of the screen, and making sure thereafter that it stayed there as the user moved the cursor around. In practice this had some disadvantages. If the screen image jumped about whenever one started an insertion it didn't feel like typing on a piece of paper. Also, it was by no means true that the relevant context for the user was equally above and below the insertion point: it might be mainly the text above or mainly the text below that was important. So we rejected the notion that insertion should trigger a change of context.

If it is important to display as much context as possible from moment to moment, then we should at least arrange for the user to be able to take full advantage of what context he can see. That is, we should keep each context as long as possible, only changing it when really necessary. How to keep context was more difficult to decide. It seemed that to move down the screen as the text grows felt more like a typewriter, more 'natural'. We rationalised this to say that users would prefer

modifications, whether insertion or deletion, to leave as much as possible of the screen unaltered. With hindsight we may have been influenced by the limitations of our hardware, so it may have been a concern to update the screen as rapidly as possible which was paramount. At any rate the principle 'keep the picture as still as possible' became a design principle and is even preserved as a comment in the text of the program—this is Hansen's principle of display inertia (Hansen, 1971).

Trying to show as much context above and below as possible obviously conflicts with trying to keep the display as still as possible. In the limit, using the techniques employed in **ded**, you reach the top or the bottom of the screen. We decided that close to the screen margins it was more important to preserve the typing context than to make editing instantaneous.

We kept at least two lines above and below the editing position visible—hang the response delays and the movement in the picture. Because an attempt to move the cursor into the top or bottom margins causes the screen to scroll and by doing this **ded** generally centres the cursor, the next scroll (and the delay it causes redrawing the screen) will be postponed as long as possible—until the user has managed to reach a screen margin by moving from the middle.

8.6.3 Fast interaction

It was important that typing new text to the editor should be as fast as typing on a typewriter, or else it would be better to be working with pencil, paper, scissors and sticky tape. The delays involved in processing commands from the keyboard and sending commands to the screen were sometimes quite severe. We found that when our time-shared computer was overloaded the delays were so extended that it was impossible to work unless you could touch-type with your eyes closed. The only rational policy was to make the delay as close to zero as possible and then hope that it was small enough.

We designed an editor which would do all the things that we thought it should and made it go as fast as possible on the hardware we actually had. When we got more capable hardware, we altered **ded**'s internal workings so that interaction became faster, but we didn't change its capabilities at all.

8.6.4 Ignore deficiencies of the terminal hardware

When **ded** was being written most of the computing world had standardised on the ASCII character set of ninety-six printable and thirty-two control characters, but it hadn't standardised on how VDTs should work. (Now it has, more or less, but oh dear, that's another story!) The VDT on which **ded** first operated could recognise commands to overstrike a character, to move down a line and to move the editing position (the 'cursor') to any position on the screen. The 'move down' command worked like a 'line feed' when the editing position was at the bottom of the screen and moved the whole picture up one line. The VDT couldn't recognise commands to move up a line, to insert a blank space or a blank line or to pan the picture left or right. Despite these deficiencies it was quite easy to write a program which would produce any desired change of a displayed piece of text.

One design principle, first articulated to us by Bruce Anderson, was that **ded** shouldn't be designed not to do something just because it was difficult or slow to produce that change on our particular VDT hardware. It was clearly desirable to be able to insert a character or split a line anywhere within the displayed text: it would be intolerable if it could only be done on the top or the bottom line of the picture, for example, even though those might be the only places where the VDT hardware could do it fast enough to be apparently instantaneous.

8.6.5 Edit a picture of the text

A central decision in **ded**'s design was to be declarative—the user edits a picture of his text. "Editing a picture of your text" is how we would describe the idea to a user (and described it to ourselves at the time); "declarative" is how we would now describe the idea pompously. The dual form of the idea constitutes a *guep* (Thimbleby, 1984).

If the user edits a picture, he can only edit what can be pictured! If the text contains formatting characters like newlines, tabs, blanks and so on—as it inevitably will—characters where there is no one-to-one correspondence between the text itself and the picture it describes, **ded** has to choose a pictorial convention that can be edited directly as picture. **Ded** provides no way whatsoever of editing

the text itself, that is the ASCII text, directly. For more background to this decision see Appendix 2.

An immediate and contentious consequence is that **ded** discards blanks at the end of lines of text, and tab characters are treated as multiple spaces. The newline character has no significance for picture editing; the user can position the terminal cursor anywhere on the picture without regard for the corresponding notional cursor position in the text it is representing. Thus the cursor may be placed 'beyond' the end of a line without special commands. A pleasant consequence, unfamiliar to users of some other editors, is if the '↑' key is pressed repeatedly the cursor moves up vertically (rather than zig-zagging along the end of lines).

```
A pleasant consequence, unfamiliar
to users of           _
some other editors, is if the '↑' key is
pressed repeatedly the cursor moves
up vertically         _
(rather than zig-zagging along the end of lines).
```

Figure 8.9: Possible cursor positions moving upwards in **ded** (starting from the '`i`' in '`zagging`')

```
A pleasant consequence, unfamiliar_
to users of_
some other editors, is if the '↑' key is_
pressed repeatedly the cursor moves_
up vertically_
(rather than zig-zagging along the end of lines).
```

Figure 8.10: Possible cursor positions moving upwards in a silly editor (again starting from the '`i`' in '`zagging`'). Once the cursor 'attaches' to the end-of-line it stays there!

Notice the non-declarative (history sensitive) nature of the design represented in Figure 8.10. Suppose at some stage during an interaction the cursor is at the end of the second line (i.e., just after 'of'). When the user moves the cursor up, it will either move to 'c' on the line above or to just after 'r': depending on the user's earlier actions and whether there are invisible spaces to the right of the 'f ' on the second line. A popular compromise is a design half way between Figures 8.9 and 8.10: and is worse than either. The cursor would move as in Figure 8.9, but as soon as the user types a character, it has to go in the text, so the cursor jumps to the corresponding position as in Figure 8.10.

Further advantages can be deduced from Appendix 2, and include the ability to edit the full ASCII character set—a principle that has section 8.6.6, next, to itself.

8.6.6 Full ASCII character set

Programmers like to be able to type control characters (the ASCII code allows for one hundred and twenty-eight characters, but only ninety-six are printable text). Since it is extremely inconvenient to be unable to edit and correct a corrupt text file (for that is what we shall call a file with normally-unprintable character codes in it), **ded** should provide some means to edit non-printing characters. Of course, 'corrupt' files arise most frequently when programmers are at work! **Ded** uses a convention that a character following '◊' is converted to a control character of the same name. Thus '`◊a`' (or equivalently '`◊A`') represents the unprintable control-A.

Since **ded** is a picture editor, the user can treat '◊A' as two separate characters. For example, he could move the cursor between '◊' and '`A`' and insert a '`G`': this would create control-G followed by `A`. Most usefully, the user can search for *any* control character by searching for '◊' directly—it is simply treated as any other character on the screen.

8.6.7 Fit in the memory of a PDP-11

Ded had to run on the hardware we could afford, because we needed to use it. The PDP-11 was a very small machine and, in the early days, programs and data had to fit into 64K bytes. This was always a tiny space. It meant that we had 'hardware support' for any refusal to add new facilities to **ded** because there was never room in the memory space available. The only way to add new stuff was to find out ways to generalise what was already there, to make the new addition operationally and

conceptually consistent with what was already there.

Because **ded** evolved it was always rather a messy program and there were always ways in which it could be rewritten as a smaller program to do the things that it could already do. So there was often room for an improvement if we really wanted to make it, but the constraint of space was always a potent one.

Note that the PDP-11's limited address space did not favour particular features over any others. If we had had a different sort of computer it might have had curious hardware that restricted certain aspects of **ded**'s design more than others. In some ways we were fortunate to have an impartial restriction imposed on us: without the restrictions it would have been far easier to extend **ded**, adding features, rather than get what we had right. So we were forced into 'survival of the fittest'.

8.6.8 Simple operations should be simple, and the complex possible

We've seen that **ded**'s operations are simple, but we bowed to programmer's pressure by providing global replacements, which can occur without visual confirmation of effects.

Global replacements are repeated replacements made by the editor throughout a document (i.e., globally). For example, the user may wish to correct the spelling of 'commitment' everywhere he spelt it 'comittment'. If we follow our principles, the user should be able to see each change. We might also want the user to confirm each change the editor proposes to make. However, if a document is very long, watching (and possibly confirming) each change can be very tedious. Programmers were particularly insistent on the need to avoid tedium even at the expense of security and as a compromise, **ded** has two forms of global replacement—one makes changes one-at-a-time, querying the user at each step; the other performs all changes 'at once' unconditionally.

8.6.9 Be safe to use

We do not believe, still less require, users to be perfect typists. **Ded** follows a 'shaky fingers' model: typing mistakes are easily seen and corrected. Some typing mistakes or slips (such as trying to move left when the cursor is already as far left as its going to go) cause the terminal bell to ring. The bell also helps copy typists, interviewers... people who may not want to pay full attention to the screen. There are good psychological reasons for using sound to notify users of simple errors, though there are equally good sociological reasons to prefer silence (do you want the whole typing pool to know your error rate?), so **ded** provides a command to silence the bell if necessary.

To complete an edit, type '`ok`'. To abandon an edit, type '`q`'; but if the file has been changed it is sensible to require some sort of confirmation before abandoning possibly hours of editing. Many editors ask the user to confirm by typing '`y`' (for 'yes'), or to type the `q` command again—but it is all too easy to slip into a habit of typing '`qy`'! Instead **ded** requires the user to type the long command '`reallyquit`', in the hope that one is less likely to type it by accident.

We have seen that some features in **ded** permit a happy co-existence of expert and naive-user oriented features. Many editors provide 'levels' or other features for customising to specific user types; levels can be provided very easily by a programmable user interface, such as that provided by the editor EMACS (Stallman, 1984). It is clear that this is a very useful feature for programmers, most especially in research and development environments. But it is also clear that customisation, even when not as radical as might be achieved by programming, can change a user interface (of course, this is what the programming is for!) so much that a user cannot rely on his colleagues for help—for he uses a different editor. When does a customised editor become a different editor? Is customisation a designer's let-out for an incomplete design?[6]

The user interface of **ded** is fixed and all features are available to all users at all times. Interestingly, we have not yet found a programmable editor that could emulate **ded**: most serious problems arise with the notion of picture editing and cursor motion (EMACS, for instance, insists on editing ASCII text). Some operations, such as word motion, become too inefficient even if they can be specified.

[6] To appease those people who like customisation—it *is* a let-out, but then designs are *never* complete...

8.6.10 Undo anything

We both felt that **ded** should have an undo command. Although Harold left QMC in 1981 and we implemented undo independently, our ideas were remarkably similar: sufficiently so for the following paragraphs to correctly describe both approaches.

It is remarkably easy to delete a line by mistake, and then you cannot see what it was, let alone remember it to retype it. Undo commands allow the user to recover from such disastrous mistakes. Our philosophy was that it was better to be able to undo anything, even if the user had to wait a little while, than (as most other editors work) undo only certain things on certain days (but perhaps faster than our more general scheme). Some editors can only undo the last thing you did (if it is the sort of thing that is undoable!); often you cannot undo more than one thing, because the editor is arranged so that an attempt to undo two things just undoes the first undo.

A constraint on the undo command was that it must be simple (we don't want the user making mistakes with the undo command that in turn have to be undone by another error-prone meta-undo command), and that it should fit in with the declarative low-mode nature of **ded**. The mode problem with undo commands is quite serious: the undo command can mean anything—the reverse of whatever has to be undone. The solution was to make undoing a mini-dialogue that displayed something of what commands would be undone: thus, at least, the user can see what would happen.

An important property of the undo command is that if you want to undo a lot of work, it will take you an equally long time (because you can only undo one keystroke at a time). This helps ensure that you do not make catastrophic undos which you then need to undo...

8.7 Problems with the design

There are all sorts of ways in which **ded**'s design might be improved. Indeed it might still be under improvement, still evolving, if we hadn't simply lost interest and moved on to do other things. There are certainly some ways in which its design ought to have been better.

Problems in the design can be classed as:

- *weaknesses in implementing the principles* e.g., the search notation
- *difficulties in implementing the principles* e.g., editing tables
- *unfortunate consequences of adhering to the principles* e.g., editing tables
- *other compromises made under pressure* e.g., non-interactive substitution

We shall explore these problems below.

A major disaster area is the notation for searching in the text (see Figure 8.8 or Appendix 3). Novice users don't master it beyond simple searches for words, and its adherence to exact character matches makes it not so useful for word processing. **Ded**'s simple search commands are simple enough, but the complicated search patterns are horrid. It *is* possible for the expert user to do almost anything, but it is very hard to work out how to do it. A regular expression grammar isn't a good programming notation and in some respects the design was far too complicated to be good. Only the two of us completely understood it and that was a long time ago.[7]

Ded does have some modes. The simplest example is that the RETURN key, pressed when editing text, breaks the current line but when editing the command line, signals the execution of the command. For example, to search for '`xyz`' you press ESCAPE, `/`, `x`, `y`, `z` and then RETURN. For a time we resisted this double use of RETURN but it is so ingrained in the mind of the UNIX user that to execute a command you type it and press RETURN that in the end we had to capitulate.

Another capitulation to public pressure is the non-interactive substitution. The normal **ded** substitution ('`x`' for 'eXchange') shows each substitution as it occurs, asking for confirmation or rejection. The abnormal command ('s' for 'Substitute') does them all regardless. The effect is often devastating, but that serves them right for asking for such a nasty facility.

[7] At least one of the people who read a draft of this chapter claimed to understand it as well. When we have enough candidates we will set an examination.

8.7.1 Unfortunate consequences of the picture approach

Ded edits a two-dimensional picture of a text in which a line has position vertically and a character or a word has a position within a line. In many ways it is convenient to view a text one-dimensionally, as a sequence of characters, words or sentences. An example of **ded**'s two-dimensional fixation, taken over from the line-based editors which it supplanted, is that it is only possible to command it to mark, move or copy blocks of lines, not characters, words or sequences of words.

Long lines are messy, so it is important that they do not occur through normal editing or text entry using **ded**. To this end, **ded** provides a 'wordwrap' feature (which is a standard feature nowadays for any word processor). Simply, **ded** inserts 'start-a-newline' instructions *between words* in the stream of the user's typing whenever it gets too close to the bell margin. The effect is that the user may continually copy type, and **ded** will fill text lines well enough. In fact, *good* wraparound is a difficult feature to specify: our 'good enough' wraparound strategy tends to result in occasional uneven line lengths.

When a session with **ded** is completed, the edited text has to be reconstructed from the text picture and this cannot guarantee restoration of the original byte sequence. If the user inserted a tab, **ded** would have inserted spaces into the picture. The user may then edit these spaces or move the cursor across or around them. When the user has finished editing, **ded** assumes the spaces really are spaces and not tabs. **Ded** is *idempotent*: if a file is once edited by **ded**, **ded** changes it into a picture form (i.e., by removing invisible text such as spaces at the right hand ends of lines, removing blanks lines at the end of the file, replacing tabs by spaces) and it will then stay in that form. Furthermore, a user cannot construct a non-pictorial file by using **ded**. Thus the ASCII/picture transformations made by **ded** should be of no concern to the user (unless possibly he is worried about data compression—**ded**'s strategy penalises programmers who use lots of tabs).

Ded is awkward to use to construct tables. It is easy to construct a preliminary picture in which various parts line up in tabular form,[8] for example as in Figure 8.11.

```
    Table of contents

Footnoting              44
Crumbing                46
Headaches               50
```

Figure 8.11: Tabular layout

But when changing the spelling of 'Grumbing' to 'Grumbling', the page reference on that line is pushed one place to the right. That is a trivial example of a serious difficulty which could not be solved within our set of design principles—and it gets worse when you want to use symbols like π in your text. Again, the problems can be pushed over into other software tools, in this case to **tbl**, a table formatting system that can do this and much more besides that would never be attempted by an editor.

8.7.2 Scrolling

The terminal screen **ded** uses is of course far smaller than most texts that users will want to edit: it is not as high, nor as wide. The conventional solution to simulate indefinite height is to scroll the image vertically on the screen. This gives the appearance of a window onto a 'scroll' of text extending above and below the physical screen. The same technique could be used to scroll sideways (like a real papyrus scroll—so the rolls don't get in the scribe's way), but it turns out that few terminal manufacturers felt it an important enough feature to provide. It was simply too computationally expensive for **ded** to support left-right scrolling: it is very slow to use, the text picture 'falls apart' during the scroll operation and it would have been tedious to program. Thus **ded** scrolls vertically, but uses a different technique to handle arbitrarily wide text.

Vertical scrolling is an interesting problem that has been widely studied. The user can move the cursor *within the screen* by the '↑' and '↓' instructions. What should they do at the edges? **Ded** takes them to have the 'same' meaning relative to the text: that is, for '↑' the screen will scroll downwards,

[8] We refer to an earlier age, when computers printed with fixed-width characters. Nowadays when printers use proportionally-spaced fonts things are much more difficult.

so revealing the previously hidden text *above* the cursor. The cursor will then move up one line as required.

Wide text has to be fitted within the width limits of the screen. Since **ded** cannot assume to know what the text means (**ded** does not aspire to be a word processor), it must not make an arbitrary 'line break', particularly one that cannot be seen. We can either have the text indicated as 'off the screen' (perhaps by displaying a '⇒' symbol at the right hand end of a truncated line), or we can put the continuation of the line immediately beneath:

```
...t indicated as 'off the screen' (perh⇒
...t the right hand end of a truncated
...on of the line immediately beneath:
```

or

```
...t indicated as 'off the screen' (perh◊
aps by displaying
...t the right hand end of a truncated
...on of the line immediately beneath:
```

Figure 8.12: Alternative ways to handle long lines

There are disadvantages to either solution. In the first case, if the user scrolls the screen to the right most of the contextual information—probably contained in short lines—will be lost off on the left. In the second case, at least the user can see the full context, but moving the cursor vertically is problematic: we either have to compromise the picture idea, or have the cursor jump a couple of lines at a time. **Ded** uses the fully visible, wrapped, approach but we are not sure that we ever thought it out properly. Perhaps we wanted to discourage long lines? William Newman's remark at the time that panning editors are confusing to use, because you lose context when panning along a long line as the rest of the text slips off the left of the screen was comforting. We never experimented with panning in **ded** until much later (when it was too late to do anything about it); however our experience then, and with other panning editors since, supports William Newman's opinion.

An interesting feature of the wrapped approach is that the user can use the *same* notation himself to join lines. If ded used the '⇒' notation, typing '⇒' (even if the user could type it) would have no effect because there would not be any text already to its right (and therefore **ded** should not display it). But with the '◊' notation, the user can type '◊' at the end of any line in order to join it to the following line, whatever is on either line. This is a simple example of 'equal opportunity' (Runciman & Thimbleby, 1987).[9]

8.7.3 Integration with other software

A text editor is a cut-down word processor: but we still want all those useful features. A word processor comes with all the features integrated into the same system: this means that the users (and designers!) are faced with a larger, harder-to-understand user interface. On the other hand, a separate text editor such as **ded** has to comply—or rather its users have to comply—with the arbitrary and generally unrelated conventions of those programs providing the extra features.

It is generally impractical to run **ded**, or indeed any display editor, by other programs. One would like to integrate the editor of one's choice with all interactive systems (such as mail systems, spreadsheets etc.) that involve text input or editing. Unfortunately display editing commands generally have incremental effects on the editor state which means that programs either have to maintain an equivalent state or parse the display output of **ded**. Neither is practical, and programs that use **ded** are reduced to simply invoking it and trying to recover from any errors later.

The picture issue is also a serious impediment to integration because **ded** may be used to prepare data for other programs which want to distinguish between (say) tabs and spaces. We can escape

[9] Not everything is equal opportunity. The '`file`' command tells the user what the current file is called. **Ded** responds by displaying '`>file` *name*' on the command line. Unlike many other commands, the user cannot edit what he can see—the name, to change it or to edit the whole line into another command. This is not equal opportunity and a bug we frequently regret.

criticism by claiming that **ded** is a text editor, not a data editor—but programmers might claim that if **ded** is so good, why can't they use it for all editing? Which sort of consistency is more important: simplicity or universality? We felt that a good user interface was more important than superficial consistency with existing programs. At QMC we had access to the source of all programs and we were able to make changes in many cases. Systems programmers elsewhere may be less fortunate or less motivated.

8.8 Conclusions

What can be learnt from studying the design activity which led to the production of **ded**? First, being aware that every issue deserves detailed examination: we are surprised that examining minutiae led to such improvements and had such interesting consequences; equally, we are worried that it is too easy just to build without designing. Being aware that one can design and make explicit tradeoffs is half the battle.

Secondly, that social factors are important. In the case of **ded**, there were significant advantages in Richard's proprietary ownership of the design. 'Egoless programming' was a popular slogan of the 1970s (Weinberg, 1971): our activity in designing **ded** was quite the opposite. We don't think that it would have been so principled a design if Richard hadn't had absolute control over what happened to it.

Thirdly, that design evolution works: **ded** was still evolving up to about three years ago. We believe that in an innovative activity, which is what the design of **ded** was in its time, sit-down-design wouldn't have worked at all. Several times we designed an improvement, tried it out and found after some weeks of experimentation that it didn't work well in practice, so we took it out. Given that we were breaking new ground and that there were so many variables necessarily being altered at the same time, we couldn't have got anywhere without experiment. An ergonomist might have suggested some novel improvements, but we can't imagine how they could have anticipated any of our experimental 'findings'. Perhaps our view is blinkered, but it seems that first of all you need a reasonable system to start experimenting with, and that is where all the effort goes to.

Some of **ded**'s design decisions were quite arbitrary, while some were due to practical issues of implementation. For example, the difficulty of implementing a powerful editor on a small machine precluded a very complex design. Some decisions were initially arbitrary but later constrained other decisions; some, however, are independent and universally arbitrary. We think it is fair to claim that **ded** is easy to use because it has a principled design. Even if the principles are ergonomically awry, the user at least knows what to expect—or at least he will see the precise effect in a few milliseconds—in principle he can see everything pertinent to his use of the editor. No such claims have been made for any other editors, but in the final analysis it would take theories and experiments to establish claims of ease-of-use. Even so, we expect effects would be eclipsed by previous experience. If that is the case, then the sooner principled interactive system design catches on, the sooner users will cease to obtain their first experience on unprincipled systems.

John Long has proposed that the problem facing cognitive ergonomics can be characterised as how to increase compatibility between the computer's representation and the user's representations. With a declarative user interface design, the user need have no model or understanding of the history of a session with **ded**. There can be no mode changes in the past that can possibly affect the present. Editing a picture immediately increases user/computer compatibility.

Computer scientists get bored with a subject when the technology moves past it. We really don't care what happens to **ded** any more. A good word processor on a home micro is in many ways more useful than **ded** already— but not necessarily easier to use!—so hardly any users care about it either.

Those that are forced to use it (we used it to write this chapter) know that it is elderly and put up with its little quirks. We would guess that now, when ergonomists might be able to say what a text editor should really be like, nobody would want to listen because users want the latest technology and computer scientists want to think about the next step forward.

Appendix 1. Ded command summary

Apart from the single-keystroke direct manipulation commands (e.g., Figure 8.1), **ded**

provides the following commands:

- search, search-and-replace, search-and-replace-everywhere, interrupt current search or replacement
- mark lines with a letter, copy marked region, delete marked region, move marked region, do any command within a marked region (e.g., search-and-replace), go to start or end of marked region.
- write file, read file, append file, write current file and edit another
- quit (abandon changes, or update file), pause (to resume at a later date), ok-exit
- undo (undo any number of keystrokes—including search/replace interrupts, back to the last file write)
- scroll screen by a page (up or down), go to a named or numbered line, go to first or last line
- execute UNIX commands, execute commands (e.g., compilers) and save diagnostics, go to next error line (in any file)
- various minor commands, such as: silence bell, change word-wrap behaviour, set tabs, find name of current file, redraw screen, find last usage of a given command etc.

Appendix 2. Why edit a picture?

In the mid 1970s we were just emerging from the Teletype™ era when most computer-readable and computer printed text was in upper case with very few different punctuation marks.[10] But UNIX had been designed to utilise the full ASCII character set—upper and lower case letters and various punctuation marks. Various conventions had been developed so that UNIX programmers could type commands to UNIX even when their terminals were deficient. On our VDTs there were no keys to print any of the characters left curly bracket '`{`', right curly bracket '`}`', vertical bar '`|`', tilde '`~`' nor opening quote '`‘`'. The terminal didn't recognise commands to print those characters either. The first three at least of the characters were essential in order to program in **C**, the main UNIX programming language, so there was a real problem when constructing, editing, printing or otherwise reviewing program texts.

The convention we used at QMC was to type a left curly bracket as two characters '`\(`'—backslash followed by left bracket. In just the same way a text would be printed with '`\(`' in every position where '{' should have appeared. This seemed quite rational, but for reasons of 'consistency' since the two marks in the printed text represented *one* character, the two keystrokes were understood to have constructed only *one* character. Under the normal convention that RUBOUT deletes the last character typed, the sequence of keystrokes `a`, `b`, `\`, `(`, RUBOUT, `c` would be understood as '`abc`' since the RUBOUT deleted the curly bracket you had typed. All very well, so far as it goes.

The designers of this convention had had some difficulty in putting it into practice. In particular, perhaps in order to reduce the number of key depressions, they had taken the totally daft decision that backslash was only part of a composite character when it needed to be. So '`\a `' was interpreted as a *two* character sequence—backslash followed by `a`—but '`\(`' was the single character '{'. The consequence was that a line which contained the four characters '`a\(b`'—`a`, backslash, left bracket, `b`—would be printed in *exactly* the same way as a line containing the three characters '`a{ b`'. More amazing still was that backslash followed by RUBOUT meant 'insert the RUBOUT key code in the line'. It was possible to type a line which contained a backslash followed by a left bracket: you typed `\`, SPACE, RUBOUT, `(`. The RUBOUT eliminated the space and left the backslash as a single character, which by now the system has forgotten about. You have now typed a line that means `\(` but looks exactly like what you would have if you had simply typed '`\)`', which actually means `{`. In other words, the meaning depends on what you did, not on what it looks like. This is history sensitive, or imperative; it is not declarative, it is certainly not editing a picture—and a bad thing for users.

All this was so confusing that the decision was made that **ded** wouldn't edit an ASCII text but a *picture* of the ASCII text. In that picture a left curly bracket would appear as *two* characters and would *behave* as two characters. When you had finished working on the text the picture would be translated

[10] UNIX practice is still to refer to terminal connections as *ttys*, with obvious etymology.

back into ASCII: note that the picture is translated, not how you made the picture (which would be being history sensitive).

This decision to work on a picture of the original text, not the text itself, opened up lots of possibilities. It made it possible to explain where the editing position went when you moved it up, down, left or right—it went up, down, left or right in the picture without worrying whether the cursor was exactly over an ASCII character (what would have happened if the cursor had been moved up in between '\' and '(', or—worse—moved beyond the end of the line where there were *no* ASCII characters at all?). The cursor in **ded** could be moved to any of these positions if you wanted it to be there.

It was an enormous simplification. Layout characters—line feeds, carriage returns, tabs—could all be ignored once the input had been translated into a picture. Its drawback was that the translation from ASCII text to picture was many-to-one, so information was lost. In particular tab layout information was lost, blank spaces at the end of lines disappeared and, at least in the early stages of **ded**'s evolution, so did blank lines at the end of a text. But the simplicity of the explanation overcame most people's objections. **Ded** is what is nowadays known as a 'WYSIWYG' editor, from the phrase 'what you see is what you get'. This is, though, a slightly idiosyncratic use of the phrase: for (apart from programmers) few users would ever want to see on paper what they had got, they would rather pump it through a text formatter first.

Appendix 3. Ded's search notation

Search notations are often based on so-called regular expressions. Many text editors use a period '.' (or even an invisible control character) to describe 'match any character' and an asterisk '*' to describe repetition.

For example, the following **ed** command would find a place in a text where the word 'the' was followed after some number of characters by the word 'stop':

```
/ the .* stop /
```

There are several problems associated with using regular expressions for specifying searches in natural language text; we mention three, our design of **ded** solves two. First, the commands specify *exact* matches: editors often have various conventions e.g., to permit searching for upper and lower case letters conveniently. This causes the second problem, to provide worthwhile utility *lots* of special symbols are needed—and the user might use them by accident or incorrectly. Thirdly, since period and asterisk (and lots of other characters) have special meanings it is necessary to use a conventional description of them: e.g., to find a line which actually contained a period followed by an asterisk—like the line in the illustration above—it was necessary to type the command

```
/\.\*/
```

This *overloads* many characters: the special characters have *two* meanings, themselves and themselves-as-special-symbols. This can cause confusion for the user, especially since the characters are likely to be infrequently used ones such as '@', '*', '[' and '{'—so the user can easily forget that they have a special meaning. We were often tripped up by this convention because we often wanted to look for particular **nroff** commands in text, and **nroff** commands always start with a period (and they have lots of backslashes in them as well which cause similar problems we needn't go into). But in any case the novice shouldn't have to learn to do something complicated just to save the expert a few keystrokes. So we decided that in **ded**'s search patterns period an asterisk and all the rest should have no special meaning. That made one less thing to explain to novices.

In order to describe repetitions and such like we turned the standard convention on its head. A period preceded by a prime means 'match any character' and an asterisk preceded by a prime means repetition. So in **ded**'s notation the two searches illustrated above become

```
/ the '.'* stop /
```

`/.*`

The advantage of this approach is threefold: the probability of accidental use of a special symbol is minimised; typing errors resulting in invalid symbol combinations are easily detected (e.g., `'x` has no meaning); many special symbols can be provided. Thus **ded** provides a simple searching language for naïve users (with little chance of their invoking advanced features accidentally), and at the same time it is able to provide many search and replacement features found on no other editors.

Anyway, the notation works, and it works well enough.

Bibliography

J. M. Carroll (1987), *Paradox and the Active User: Implications of Learning for Design*, Complex Learning Workshop, University of Lancaster.

R. E. Granada, R. C. Teitelbaum & G. L. Dunlap (1982), *Effects of VDT Command Line Location on Data Entry Behaviour*, Proceedings Human Factors Society.

W. J. Hansen (1971), *User Engineering Principles for Interactive Systems*, AFIPS Conference Proceedings, **39**, pp523–532, Las Vegas.

N. Meyrowitz & A. van Dam 1982, *Interactive Editing Systems*, ACM Computing Surveys, **14**(3), pp321–415.

D. L. Parnas, P. C. Clements (1986), *A Rational Design Process: How and Why to Fake It*, IEEE Transactions on Software Engineering, **SE-12**(2), pp251–257.

R. Pike (1987), *The text editor* `sam`, Software—Practice and Experience, **17**(11), pp813–845.

C. Runciman & H. W. Thimbleby (1986), *Equal Opportunity Interactive Systems*, International Journal of Man-Machine Studies, **25**(4), pp439–451.

R. M. Stallman (1984), *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*, in Interactive Programming Environments, eds. D. R. Barstow, H. E. Shrobe & E. Sandewall, pp300–325, McGraw-Hill Pub.

B. Sufi-in (1981), *Formal Specification of a Display Editor*, PRG-21, Programming Research Group, Oxford University.

H. W. Thimbleby (1981), *A Word Boundary Algorithm for Text Processing*, Computer Journal, vol. **24**, pp249–255.

H. W. Thimbleby (1982), *A Text Editing Interface: Definition and Use*, Computer Languages, vol. **7**, pp25–40.

H. W. Thimbleby (1983), *Guidelines for 'Manipulative' Editing*, Behaviour and Information Technology, vol. **2**, ppl27–161.

H. W. Thimbleby (1984), *Generative User-Engineering Principles for User Interface Design*, Proceedings First IFIP Conference on Human Computer Interaction, INTERACT '84, London, volume **2**, ppl02–107.

G. M. Weinberg (1971), *Psychology of Programming*, Van Nostrand Reinhold, London.

Acknowledgements

The authors gratefully acknowledge helpful criticism from Tony Fisher, David Laukee, John Long, and Andy Whitefield. (This copy was scanned and edited from the original by Brock Craft in 2004.)