
Can structured formatters prevent train crashes?

JACQUES ANDRÉ

IRISA/INRIA
Campus de Beaulieu
F-35042 Rennes, France

SUMMARY

A typographic layout error is analysed for its likely effect as being one of the causes of a train crash. Arguments are put forward to show that this error could not have occurred if a structured text formatter had been used.

KEY WORDS Structured formatters Document reliability Typographic errors

Paris, Gare de Lyon, 27 June, 1988, 18:47. A crowded suburban train is ready for departure when, suddenly, another train arrives in front of it. There is a crash with 56 dead and hundreds injured. Obviously the killer train had no brakes. Why?

The French government immediately set up a commission to analyse the disaster. This commission published its analysis in a report in September 1988 [1]. First of all it appears that ‘Such a disaster is not due to a single cause. Rather to a chain of different circumstances’. Someone pulled down the emergency lever which caused the train to stop in an unscheduled station; an air-brake pipe was faulty and the train driver was unable to bleed it; the radio alarm system was out of order, etc. However the commission noticed that *the maintenance manuals were particularly complex to use*, and it even noticed *an error in the layout of the document describing the process of repairing brakes*.

The French text of that document, in its original layout, is shown in Figure 1. Figure 2(a) exhibits the main points of the document in that original layout, while Figure 2(b) shows how they should have appeared¹.

Whenever the ‘1st CASE’ of Figure 2(a) applies, the layout tells us that the driver processes only the *xxx* actions and nothing else. Apparently he does not need to look at the second case, nor to notice the hidden phrase ‘In both cases’ with its associated actions. Thus, he does not switch on the taps, nor does he check the air brakes, and so on. By contrast, the layout of Figure 2(b) indicates that, after obeying the ‘1st CASE’ actions, the driver has to follow on with the ‘In both cases ...’ actions which, in turn, involve checking the brakes.

We can surmise that the original document was typeset using a second-generation typesetting machine. Let us assume that, on such a machine, there are three tags for controlling indentation, of the form:

- <IX> to right-indent the text that follows (i.e. to shift right the left margin),
- <QL> to feed the current line and to start the next line at the current margin,
- <EP> to feed the current line and to restore the left margin to its previous value.

¹ Though it could be argued that the lines *1st CASE* and *2nd CASE* should be even more indented

- b) Plusieurs véhicules sont bloqués, le mécanicien :
- S'assure que ce blocage n'est pas la conséquence de la fermeture d'un robinet d'arrêt de la conduite générale situé avant la partie de train bloquée :
- 1^{er} CAS : Aucun robinet d'arrêt CG n'est fermé :
- Il actionne la commande de la valve de purge le temps suffisant pour provoquer le desserrage sur chaque véhicule bloqué.
- 2^e CAS : Un (ou plusieurs) robinet d'arrêt est trouvé fermé :
- Il ouvre le robinet
- Dans les 2 cas, le mécanicien :
- ouvre le robinet d'arrêt CG situé en arrière du dernier véhicule relié à la CG.
 - vérifie le serrage des freins du dernier véhicule freiné.
 - referme le robinet CG
 - vérifie en se dirigeant vers la tête du train :
 - le desserrage des freins de tous les véhicules,
 - que le blocage n'a pas provoqué d'avarie aux roues.
- Il applique les mesures concernant le signalement et la reprise de marche (article 385).

Figure 1. Part of a train maintenance manual with a layout error: the second line after 2^e CAS (Dans les 2 cas = In both cases) should not be indented

Figure 3(a) shows how the text has probably been typeset, to give Figure 2(a), while Figure 3(b) shows how it should have been typeset. This example shows how a very small bug (<EP> instead of <QL> on line 5) can cause a disaster.

Two questions are relevant to this kind of instructional text:

1. How can we make-up a text to be both legible and understandable?
2. How can we edit reliable texts?

According to T.R.G. Green and S.J. Payne, when they were studying concurrency [2], 'the well-documented use of typographical cues to illuminate instructional text has in the past been limited to illustrating *containment* relations (sections within chapters or subsections within sections) and *succession* relations (after one chapter we come to the next). No other relations have been studied'.²

It is worthwhile to consider the layout rules for a specific set of instructional texts, namely, those for programming languages. Thanks to Dijkstra's paper 'go to statement considered harmful' [5], many studies have been made of the 'best way' to edit conditional statements (such as *if* statements or *case* statements, loops etc.). All the proposed solutions are based on horizontal indentation of internal parts of the statement, and on

² Since that time, other papers have been published, e.g. Pat Norrish [3] uses a typographical approach while Virbel [4] looks at lists using a linguistic approach.

| | |
|---|---|
| <pre> 1 The driver ... checks ... 2 1st CASE: xxx 3 xxx 4 2nd CASE: yyy 5 yyy 6 In both cases the driver 7 - zzz 8 - zzz 9 Then, the driver restarts the train.</pre> | <pre> The driver ... checks ... 1st CASE: xxx xxx 2nd CASE: yyy yyy In both cases the driver - zzz - zzz Then, the driver restarts the train.</pre> |
| (a) | (b) |

Figure 2. (a) How the text was printed; (b) how it should have been printed

| | |
|--|---|
| <pre> 1 The driver checks ... <EP> 2 1st CASE: <IX> xxx <QL> 3 xxx <EP> 4 2nd CASE: <IX> yyy <QL> 5 yyy <QL> 6 In both cases the driver 7 <IX> - zzz <QL> 8 - zzz <EP> 9 Then, the driver restarts ...</pre> | <pre> The driver checks ... <EP> 1st CASE: <IX> xxx <QL> xxx <EP> 2nd CASE: <IX> yyy <QL> yyy <EP> In both cases the driver <IX> - zzz <QL> - zzz <EP> Then, the driver restarts ...</pre> |
| (a) | (b) |

Figure 3. (a) How the text was edited; (b) how it should have been edited

the vertical lining up of ‘parentheses’ (where an **end** would be the right parenthesis of a **begin**). However, a given statement may be edited in many ways, depending on the language, and in the end there is no definitive preferred template. There is still much work to be done on program understandability and its relationship to legibility factors.

The same problems occur in the formatting and make-up of texts. For example, many papers have been published on the legibility of bibliographical references, but how many of those definitive answers, using electronic publishing facilities, would be useful and usable today? Equally, one cannot give precise indentation rules for conditional statements in repair manuals for train brakes. Indeed, it may be the case that understanding would be helped by other visual clues, in addition to the indentation.

We can give rather more guidance regarding the second question, of how to edit reliable texts. More precisely, we wish to be sure that an error cannot occur when editing a manual. First of all, let us imagine that our manual for repairing train brakes was formatted using a WYSIWYG system. Obviously, when typing in the rules, errors of the kind we have just described would be quite impossible (one sees on the screen that the indentation is wrong). However, no one can guarantee that such errors might

```

1 The driver checks ...
2 \begin{description}
3 \item[1st CASE:] xxx\\
4 xxx
5 \item[2nd CASE:] yyy\\
6 yyy
7 \end{description}
8 In both cases, the driver
9 \begin{itemize}
10 \item zzz
11 \item zzz
12 \end{itemize}
13 Then, the driver restarts the train.
```

Figure 4. \LaTeX code for figure 2.a,

not occur if the text is subjected to some global ‘search and replace’ operation (or any operation without human check), on some given text string.

Again, let us turn to examples from the realm of programming languages. J.J. Horning [6] tells us that ‘The experience of the last thirty years shows that it is not easy to produce nearly-correct programs. ... The goal of reliable programming is to minimise the number of faults in completed programs. ... Checking always relies on a certain amount of redundancy built into the language’. We note that the high level of redundancy in structured programming languages allows them to be practicable solutions to this problem.

For the same reason, structured formatters allow text to be formatted safely. Indeed, parenthesized structures (with **end** completing some part of text starting with **begin**), recursivity (parentheses are to be balanced) and redundancies (an **end** must have the same ‘label’ as the corresponding **begin**, e.g. **begin{section}** has to be closed by an **end{section}**) are good tools for reliability, even if they are boring to type in. More information on structured documents is given in [7].

Let us now imagine that the train manual for repairing brakes had been written in some structured formatter such as \LaTeX [8]. A standard way for editing the text of Figure 2(a) is shown in Figure 4. Let us assume also, that the user made an error, e.g. forgetting the **end{description}** at line 7 or mistyping it. In these circumstances an error would be detected by \LaTeX due to mis-matched parentheses, and although that error message would not be too easy for a beginner to understand, the fact remains that a message would be posted and no printed output would be produced. Redundancies of this sort lead to fault intolerance, and hence to reliability.

By using programming systems such as Mentor, the newer document manipulation systems such as Grif are even going further. Thanks to their internal structure and to the use of windows, one no longer needs to type in ‘parentheses’: they are now so implicit in the overall document structure that it is very difficult to use them erroneously.

REFERENCES

1. *Rapport de la Commission d'étude de l'accident de la Gare de Lyon*, Ministère des Transports et de la Mer, Paris 1988. See also J. André, 'L^AT_EX ou SGML pouvaient-ils faire éviter l'accident de la gare de Lyon?', *Cahiers GUTenberg*, **1**(1), 21–25, (April 1989).
2. T.R.G. Green and S.J. Payne, 'The woolly jumper: typographic problems of concurrency in information display', *Visible Language*, **XVI**(4), 391–403, (1982).
3. P. Norrish, 'Semantic Structures of Text', in *Structured Documents*, eds. J. André, R. Furuta and V. Quint, Cambridge University Press, Cambridge, 1989, pp. 143–159. See also P. Norrish, *The Graphic Translatability of Text*, British Library R & D Report 5854, 1988.
4. J. Virbel, 'The contribution of linguistic knowledge to the interpretation of text structures', in *Structured Documents*, eds. J. André, R. Furuta and V. Quint, Cambridge University Press, Cambridge, 1989, pp. 161–189.
5. E.W. Dijkstra, 'Go to statement considered harmful', *Communications of the ACM*, **11**(3), 147–148 (1968).
6. J.J. Horning, 'Programming languages', in *Computing Systems Reliability*, eds. T. Andersen and B. Randell, Cambridge University Press, 1979, pp. 109–152.
7. J. André, R. Furuta and V. Quint (eds.), *Structured Documents*, Cambridge University Press, 1989.
8. L. Lamport, *L^AT_EX, A Document Preparation System*, Addison-Wesley, Reading, Mass., 1986.