Getting It Wrong—A Cautionary Tale

Michael Jackson

There is a very simple kind of programming problem that appears everywhere in data processing—you have a serial file of transactions, sorted into ascending order by some group identifier, and you have to print a summary showing the total for each group. It's the kind of problem everyone knows about—at least everyone who has been around in programming for more than a month or two. Well, this story is about a programmer who was such a novice that he hadn't seen the problem before. So when it came his way he just did a nice simple piece of top-down design, and arrived at the following structured pseudocode:

```
PA:    open INFILE; display 'SUMMARY'; read INFILE;
    do while not eof-INFILE
       if new group
          end old group;
          start new group
       else process record
       endif;
       read INFILE
    enddo;
    display 'END SUMMARY';
    close INFILE
end PA;
```

Now you, my learned readers, can probably see at once that something here is not quite right; but our hero was only a novice. So he went right ahead and coded his solution in COBOL, and compiled it, linked it, and ran it on some test data. What came out was a little surprising (to our hero, though not to any of us more experienced programmers):

```
SUMMARY
..V/*     V..K$/K
A172632       +15
A195923       -60
...
Z749321     +8755
END SUMMARY
```

What, he wondered, could that curious second line be? In quite a short time he understood: it was, of course, the effect of ending the old group before the first group—only there was no old group before the first group. But how to make it disappear? Luckily, he knew his COBOL quite well. The first part of the curious line was of course, the group identifier, and could be removed by setting a VALUE clause in the WORKING-STORAGE item

```
02 PREVIOUS-RECORD-ID PIC X(7) VALUE SPACES.
```

while the right-hand part of the line was the total, and required slightly more subtle treatment:

```
02 GROUP-TOTAL PIC S9(6)
    VALUE ZERO
    BLANK WHEN ZERO.
```

After a recompilation he reran the test, and was delighted to see:

```
SUMMARY

A172632       +15
A195923       -60
A198564
A200135      -157
Z749321     +8755
END SUMMARY
```

He was poring happily over this printout, when his manager happened to come by. “What’s this?” asked the manager, pointing to the total for group A198564, “why is the total blank?” “That’s zero,” said the programmer. “No it isn’t,” said the manager, “it’s blank, and I want it to print as zero, not as blank.” Our hero, although a novice, was very quick to learn the tricks of the programmer’s trade, and, as quick as a flash, he replied “well, there are technical reasons why it has to print as blank.” This was a testing moment for the manager; he might have made the catastrophic mistake of asking “what technical reasons?” But he was cleverer than that; so he just said, “technical reasons or no technical reasons, it must print as zero.”

So it was back to the drawing board for our hero. Luckily, he happened to overhear one of his colleagues at lunch talking about something called a “first-time switch.” He had never heard of such a thing before, but, being very intelligent, he saw at once what the name implied and how such a device could help him. That afternoon he added one to his program:

```
PA: open INFILE; display 'SUMMARY'; read INFILE;
    move 0 to switch1;
    do while not eof-INFILE
       if new group
          if switch1 = 1
             end old group
          else move 1 to switch1
          endif;
          start new group
       else process record
       endif;
       read INFILE
    enddo;
    display 'END SUMMARY';
    close INFILE
end PA;
```

When he ran it again, the output looked really fine, and the program went into production running. It ran happily for six months, and then one day a clerk from the user department came to see our hero. “Look,” said the clerk, pointing to the end of that week’s report, “there’s no total for the last group.” Well, I expect that all of you readers knew all along that this was going to happen: after all, the end-group and start-group operations had originally been paired, and the introduction of switch1 removed one of the end-groups; so there must have been a group that was being started but not ended. And indeed there was, and it was the last group in the file.

Of course, the error had been there on every one of the twenty-five reports produced since the program had been put into production, but no one had noticed it before. There’s a lot of computer printout that nobody reads in most installations. The programmer, of course, didn’t like to say anything about this to the clerk, so he just said “OK, I’ll fix it.” Why had the programmer not noticed the error in testing? Really, because he was too conscientious. He had decided to test the program thoroughly, by using the whole of last year’s actual data as test data. This is a special kind of testing, called “volume testing” or “soak testing”. It’s a special kind of testing because you run the program on the input, but you don’t look at the output. After all, who could look through a pile of paper five inches high? What you do when you get five inches of output is this: you throw away the front two sheets, because they’re JCL, which no ordinary mortal understands (experts sometimes look to see that the system completion code is zero, but not everyone’s an expert); you check to see that there is no core dump; you look at the first and last lines; you spot-check a few of the totals; then you riffle through the whole five inches by raising the edge of the pile and letting the sheets peel off against your thumb, to make sure that if there is anything really nasty, like forty consecutive pages on which every line is printed with zeros all across the page, it will force itself on your attention. And that’s all So the error had remained undetected.

But it was very easily fixed. At the end of the program the programmer inserted the statements to end the last group:

```
         ...
         read INFILE
      enddo;
      end last group;
      display 'END SUMMARY';
      close INFILE
end PA;
```

All went well for the next 17 months. And then the program showed that it was sensitive to something you wouldn't have expected a program to be sensitive to: the Bicentennial celebrations. What happened was that the company gave everyone a week's vacation; the next Monday morning they came back to work and ran the program, just like they ran it every Monday morning. What came out was:

```
SUMMARY

$$..V/*      ..D»K./
END SUMMARY
```

It was immediately obvious that something was wrong. There had been no transactions the previous week, so there were no groups, and the mystery line was, of course, the result of ending the last group—only there wasn't a last group. But again the problem was easily solved.

Our hero, more experienced now, was tempted to do something clever with switch1, but he resisted the temptation. He had heard about "defensive programming." Defensive programming is a theory based on the idea that when you are programming you don't have the faintest idea whether what you are doing is right or not, so you ought to do something that won't cause too much harm. In accordance with this admirable principle, which he felt was clearly applicable in his own case, he resisted the temptation to do something with switch1 and instead introduced switch2:

```
PA; open INFILE; display 'SUMMARY'; read INFILE;
    move 0 to switch1; move 0 to switch2;
    do while not eof-lNFlLE
       if new group
          if switch1 = 1
             end old group
          else move 1 to switch1
          endif;
          start new group
       else process record;
          move 1 to switch2
       endif;
       read INFILE
    enddo;
    if switch2 = 1
       end last group
    endif;
    display 'END SUMMARY';
    close INFILE
end PA;
```

After recompiling, he ran the program again on the empty file and produced the hoped-for result:

```
SUMMARY

END SUMMARY
```

Clearly, his tribulations were at an end. The program ran happily and successfully for the next 19 months, and no complaints were heard.

Then, one day, he was sitting quietly in his programming cubicle—Gerry Weinberg would have had something to say about that, I think—reading the job advertisements, when along came the clerk from the user department. "Look," he said, "the last group has been left off the printout again." And indeed it had. The diagnosis was easy. Somehow the program library had been messed up, and it

was an old, incorrect version of the program that had been run that week. But not so. After a couple of days of detective work our hero established beyond doubt that it was the current version that had been run. However, he also established that the program had been recently recompiled with the new version 6 compiler, for which the installation was a field-test site. It must have been the compiler that was at fault! With the help of the systems programmer, our hero went through the object code hexadecimal character by hexadecimal character, and related it to the source code. The job took only nine hours, which they did in one marathon stretch, thus earning a bonus from their appreciative management. But the result was to prove that the object code was a perfectly reasonable compilation of the source COBOL text! Only one possibility was left: it had to be a transient error in the hardware or the operating system. Now, people don't like to come to that conclusion if they can avoid it; but I have to say that I have never found a programmer who, in private conversation, alone with only one fellow programmer, could not remember at least one occasion in the past when something funny happened in one of his programs that had to be put down to that cause. And this seemed to be one of those occasions.

Certainly the error hadn't occurred before—at least, since the "end last group" statements were added to the program. And it didn't happen again.

However, something funny did happen recently. The clerk from the user department came to see the programmer. "I've been wondering," he said "about that week we lost the last group from the printout. I got hold of the card input-which we always keep for a few months—and I discovered that there were exactly 843 cards."

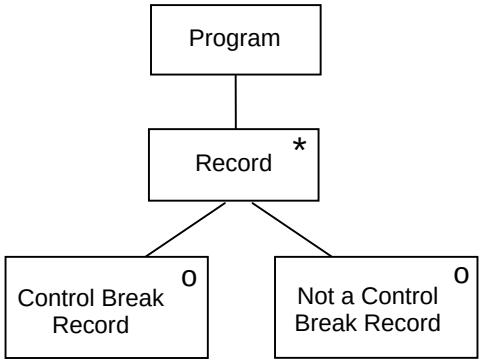
"So?" said the programmer. "Well," the clerk went on, "there were 842 group totals on the printout that week, and there should have been 843." "I don't see that that's relevant," said the programmer, "but it's kind of you to mention it. Thank you for taking the trouble." That night he took the program listing home secretly, and looked it over carefully. Suddenly the answer dawned on him. Of course! If there were 843 cards and 843 groups, then each group contained exactly one card; so the condition "new group" would be true on every card, and it would always be the "if" that was executed and never the "else." But the instruction to set switch2 was only in the "else" clause! So switch 2 was never set on, and the last group was never ended.

The problem, once identified, was easily solved. In accordance with the principles of defensive programming, another statement "move 1 to switch2" was added to the first clause of the "if-else", leaving the original statement untouched in the "else" clause. So the program was now:

```
PA: open INFILE; display 'SUMMARY'; read INFILE;
    move 0 to switch1; move 0 to switch2;
    do while not eof-INFILE
       if new group
          move 1 to switch2;
          if switch1 = 1
             end old group
          else move 1 to switch1
          endif;
          start new group;
          move 1 to switch2
       else process record;
          move 1 to switch2
       endif;
       read INFILE
    enddo;
    if switch2 = 1
       end last group
    endif;
    display 'END SUMMARY';
    close INFILE
end PA;
```

We can be confident that the program is now perfect and correct. But, of course, it has been running only for three months in its new form, and I will keep you posted if there are any new developments.

And the moral? Well it's this. The basic structure of the program, as originally designed, looks like this:



The condition "new group" in the program pseudocode tests each record to determine whether it is a control break record or not. This structure is wrong. Not inferior; not inelegant; just plain wrong.

Here's why. The difficulties were all caused by the "end group" instructions. Now, how often should we end a group? Why, once per group! Where, in this program structure, is there a component that processes each group? There is no such component, and the "end group" instructions cannot therefore be correctly allocated to the program structure. That's what all the difficulty was about.

Now, all of you good readers are experienced folk, and you wouldn't have made these mistakes — not, at least, on this small and well-known problem, and certainly not if you have read *Principles of Program Design*. But I know some very experienced programmers who made just this kind of mistake on bigger and more obscure programs; in fact, a lot of the mistakes they make are of exactly this kind—having the wrong program structure. But I don't suppose that any of those programmers are reading this cautionary tale, are they?

Originally included in JSP course material by Michael Jackson Systems Limited, 1974-1980. Also reproduced in: *John Cameron; JSP & JSD: The Jackson Approach to Software Development; IEEE CS Press, 1989.*