Chapter 8: User-Friendly Programming

So far we've discussed the use of style to make your code clear and easy to read. But style doesn't stop at the printed page. A program is not only edited, debugged, and compiled; it is also used. In this chapter we extend our discussion of style to include how the program appears when it is in use.

What Does User-Friendly Mean?

As programmers, we encounter a large number of tools, utilities, and other programs. Some are a joy to use, letting us get our work done with a minimum of fuss. Others are a nightmare, with obscure and complex command sets.

What is a user-friendly program? Simply a program that the user considers a friend instead of an enemy.

In the early days of computing, machines cost millions of dollars and programmers cost only a few thousand. Companies could afford to keep several specialists around to translate management requests into language the computer could understand.

For the programmers, the early computers were very user-unfriendly. IBM's OS/360 required the programmer to interface with it using a particularly brutal language called JCL. The commands were cryptic; for example, “copy” was “IEBGENER”, and specifying a file could easily take up three to five lines of JCL code.

Over the years, computers have dropped in price, and the cost of programmers has increased. Low prices have meant that more and more people can buy computers. High salaries have meant that fewer and fewer people can afford to pay a full-time programmer to run them.

Software has had to evolve with the times, too. Programs have had to become easier to use in order to accommodate newer, less computer-literate clients.

Today, people with no computer training at all can go into Radio Shack, plunk down $1000 and walk out with a computer that is faster and more powerful than an early IBM that cost millions of dollars.

Law of Least Astonishment

For years, people have tried to come up with a set of laws to define what is user-friendly and what is not. Many of them involve complex standards and lots of rules; but the best law that I've seen governing program design is the Law of Least Astonishment: the program should act in a way that least astonishes the user.

Rule 8-1:

> *Law of Least Astonishment: The program should act in a way that least astonishes the user.*

Modeling the User

Computers intimidate many people. (Those who aren't intimidated tend to become programmers.) Your first step in writing a user-friendly program is to put yourself in the shoes of the user. What does a user want from the system?

Always remember that users have a job to do. They want the computer to do that job their way, with a minimum of effort.

Almost all tasks done by computer were at one time done by hand. Before word processing, there was the typewriter. Before databases, there was the card file. A good program should be designed to emulate a manual task that the user knows. For example, a good word processor lets the user treat it like a typewriter. True, it adds a great many features not found on a typewriter, but at heart it still can be used like a typewriter.

A good example of a program imitating a manual procedure occurred when a business school graduate student was attending a financial analysis class. He noticed that the professor had a set of figures arranged in a neat set of rows and columns on the blackboard. Every time the teacher changed one number, he had to recalculate and write a new set of numbers.

The student figured that a computer could perform the work automatically, so he invented VisiCalc, the first spreadsheet program. Successful modeling brought this observant programmer a million-dollar idea.

Error Messages

Sooner or later, every user makes a mistake. When that happens, an error message usually appears. Writing a good error message is an art. Care and thought need to go into the creation of these messages.

Examples of poor error messages abound. I once ran a FORTRAN program and was surprised to see the following message at the end of my run:

```
JOB KILLED BY IEH240I
```

So I consulted the book called Messages and Codes (aka The Joke Book), which was supposed to contain a complete list of errors, and it did—for all the codes except the IEH series, which was in the FORTRAN manual. Going to the FORTRAN book, I discovered that IEH240I meant “Job killed by fatal error.” Of course, I knew it was a fatal error the moment it killed my job.

It turns out that the program tried to do a divide by 0, which resulted in a “Divide by 0" message followed by the IEH240I.

Error messages should not be cryptic. The IEH240I code sent me on a wild goose chase through two books, only to wind up where I started.

You cannot expect the user to know computer terminology. For example, a message like this:

```
FAT table full
```

means nothing to most users. “What do I do? Put the computer on a diet?”

Remember that most users are not programmers, and they won't think like programmers. For example, a secretary was having trouble saving a memo and complained to the computer center. “Do you have enough disk space?” asked the programmer. The secretary typed for a second and said, “Yes, I see a message disk space OK.” The programmer looked at the screen, and sure enough, there was the message:

```
Disk space: OK
```

After a few files were deleted, the message read:

```
Disk space: 16K
```

and the secretary was able to save the memo.

Sometimes an error message baffles even experienced programmers. For example, I'm still trying to figure out this one:

```
Error: Success
```

Sometimes it is difficult to pick out the error messages from all the other noise being produced by the computer. A solution is to clearly identify error messages by starting them with the word Error:.

A good error message tells the user what's wrong in plain English, and suggests corrective action. For example:

```
Error: Disk full.
```

Delete some files or save the data on another disk.

Rule 8-2:

Begin each error message with Error:. Begin each warning message with Warning:.

The classic IBM PC self test follows this rule, sort of:

```
Error: Keyboard missing
Press F1 to continue
```

One student programmer who took particular pride in his program created a work with the most interesting and obsequious error message I've seen:

```
This humble program is devastated to report that it cannot
accept the value of 200 for scale because the base and
thoughtless programmer who implemented this program has
thoughtlessly limited the value of scale to between 0.01 and
100.0. I implore your worthiness to reduce the scale and run
this miserable program again.
```

The Command Interface

MS/DOS has a very strange command interface. It appears to be built out of bits and pieces stolen from other operating systems, which results in a command language that is far from consistent.

For example, to get rid of a file, you use the command ERASE. But to get rid of a directory, the command is RMDIR. This is one of the many reasons MS/DOS is considered user-unfriendly. The command interface should be consistent. If you are going to use ERASE to get rid of a file, use ERASEDIR to get rid of a directory.

The GNU ispell program is another example of a program with a problem in consistency. This program checks spelling, and when it detects a misspelled word it produces a numbered list of suggested corrections:

```
Misspelled word: Oualline
1. Hauling
2. Mauling
3. Pauling
```

To select a replacement, you just type in the number. Type a 2, and “Oualline” becomes “Mauling.” The problem is that there can be more than 10 suggestions. In such cases, 1 is ambiguous. It can mean 1 or the first digit of 10. So the program forces you to type <ENTER> if you really want to select 1. Let's review the command interface:

To select a word, type its number, unless there are more than 10 displayed and you want number 1, then type the number 1 and <ENTER>.

How much simpler it would be to say:

```
Type the number and <ENTER>.
```

This example demonstrates the main strength of consistency: You don't have to remember very much. With computer manuals consisting of 1000+ pages, you must have consistency or you'll get insanity.

Help

Early programs were designed to save disk and memory space, not to be user-friendly. It was difficult to get more than code out of them, much less a help screen.

As user-friendly programming has gained acceptance, help systems has improved as well. Today there are help compilers to aid the programmer produce context-sensitive help screens. The compiler also allows the programmer to embed cross-references in the text that let the user jump immediately to a related subject. Finally, there is an index of all topics that the user can search.

Help compilers are available for Borland's compiler and Microsoft's Windows development system. But even without a help compiler, every program needs to provide some help. More complex programs need context-sensitive help. Far too often, help systems are not designed into programs from the start, but instead as “if we have time” projects. This makes programs very unfriendly.

Safety Nets

Occasionally a user will try to do something that causes permanent damage and loss of data to their system. A user-friendly program provides users with a safety net preventing them from doing something stupid unless they really want to.

For example, if the user tries to write over an existing file, the message:

```
About to overwrite the file START.TXT.
Are you sure [n]?
```

This gives the user a chance to abort the operation without damage.

Rule 8-3:

Don't let users do something stupid without warning them.

Accelerators

Some users eventually develop into power users. You know the type—they know every command in the program, have an amazing set of tricks for getting around program limitations, and can quote long passages from the reference manual.

The user interface for the power user is different from that needed by the novice. Many programs provide accelerator keys, which allow the user to perform common commands with a single keystroke. For example, to run a program in the Borland C compiler you must type Alt-R to bring up the run menu, and then R to run the program. Power users can hit Control-F9.