*by P.J. Plauger*

# The Falutin' Index

What kind of programmer are you? Are you the kind who likes to curl up in a corner and bang out code? Do you prefer working in small groups? Or do you believe that all good software requires project plans, weekly progress reports, and CASE tools?

The interesting thing about our business is that we have room enough for all these working styles. Even in the narrow field of embedded programming, projects come in all sizes. You can usually gravitate to the kind of work environment you find most comfortable—but not always. We're in the midst of a recession, in case you hadn't noticed, which can often limit your choice of projects to work on. It also limits your ability to hop to another job that's more to your liking. And even if you're happily and stably employed, life is seldom simple. Problems come in all shapes and sizes. Try as you might, you can never make them all look exactly alike.

You probably try harder than you think to make your problems look alike. Most people do. It seems to be a major problem in our business. People repeatedly fail to accurately gauge the problems they face. Instead, they look for the size problem they *want* to solve and go solve that one. It's a recipe for wasted effort, at least. In extreme cases, it's a recipe for disaster.

The mismatch applies recursively as well. You attack a large project by dividing it into smaller chunks. Each of these chunks offers further opportunity to guess wrong. In an organization that doesn't work hard to improve its estimating skills, disasters pop up at all levels of complexity.

I find embedded systems programming to be particularly vulnerable to misjudgments about problem size. I think it's because the hardware and software are more closely intertwined than in other branches of the software business. You're more likely to have hardware-trained managers guessing about software complexity or the converse. We all know about the greater demands for high performance and reliable operation. Both goals up the stakes considerably.

The name of the game is congruence. The effort you bear had better be consistent with the intrinsic complexity of the problem. Don't try hard enough, and you don't meet your deadlines—not by a long shot. The classic disasters in our business involve gross underestimates of a problem's complexity. Or, they reveal a wholly inadequate management style for the problem being tackled. (A large enough project is primarily an exercise in management. The programming technology has only a minor effect on the outcome, or the total cost.)

So what happens if you try too hard? Well, you certainly waste money and time. You end up so preoccupied with specifying and reporting that you fail to notice whether the job gets done right. Here is the origin of those notorious $500 toilet seats that the Pentagon occasionally purchases. Most of the cost lies in proving that the trivial hardware meets the nontrivial specifications.

I'm still oversimplifying. More is involved in developing software-based technology than just problems and programmers. In fact, I can identify at least half a dozen aspects of the business. Each has an intrinsic complexity. Your goal is to keep all the aspects of a project at a comparable complexity.

## MEASURING COMPLEXITY

How do you measure complexity? If we could quantify that question reliably, we'd all be better at our jobs. However, a certain type of qualitative measure is widely used. I refer, of course, to the well-known "falutin'" index. A high-falutin' problem has a high degree of intrinsic complexity. A low-falutin' problem is relatively trivial. For the sake of subsequent discussion, I will talk in terms of this qualitative scale.

The first aspect to consider is the problem. Yes, each problem we face has an intrinsic complexity. In principle, it is the job of the systems analyst to capture this complexity in a readable specification. The analyst interviews the "customer" to capture the essence of the system to be modeled. A good analyst may recommend one or more possible implementations, but should resist the urge to dictate the one right way to do things.

In practice, of course, systems analysts only work on high-falutin' problems. For a middle-falutin' problem, you may put on your analyst hat long enough to write a page or two of specifications. You have every right to intermix analysis and design to your heart's content. After all, that's the hat you get to put on next.

For a low-falutin' problem, you probably won't even speak the "A" word. Analysis consists of a sentence that begins with the words, "What we need is " Complete the sentence and

you're done. No need to apologize. Just get on with solving the problem.

In all cases, your primary concern is guessing right about the degree of complexity (sorry—the falutin' index). Yes, I said "guess." In the software biz, you never do exactly the same thing twice. If the current problem looks very much like one you've solved before, you can guess pretty accurately that it has the same degree of complexity. The more new issues it raises, the more trouble you're in.

That's why divide-and-conquer is such an important approach to partitioning analysis problems. The sum of several wrong guesses about small problems is almost always less disastrously wrong than a wrong guess about the whole problem. (Isn't it reassuring to know that much of our success lies in limiting how wrong we are?)

The second aspect is the solution you design for the problem. Don't confuse it with the problem itself. The problem is real. The solution is an abstract model of the problem. It is the job of the system designer to form an abstract model that manages the problem's complexity. If the analysis has been done right, the designer knows how falutin' the problem is and will act accordingly. A good designer will plan an implementation that is as complex as it has to be, but no more.

**A high-falutin' design for a low-falutin' problem is tremendously expensive. Excess complexity means excess cost.**

A high-falutin' design for a low-falutin' problem is tremendously expensive. We are in the business of controlling complexity, so excess complexity means excess cost. It is not just a development cost but an ongoing burden to maintainers. Consider how rapidly complexity grows with apparent size. You soon learn that minimizing complexity is much more than an aesthetic goal. It lies at the heart of our business.

I once saw an embedded system that essentially ran a glorified vending machine. It did the moral equivalent of counting coins, dispensing goods, and making change. The analysis consisted (rightly) of a page of data-flow diagrams and another page of constraints written in English. The proposed design modelled the data transforms as half a dozen processes running under a commercial real-time operating system. I'm sure the designer enjoyed using what he had learned, but he overdid it. The final system used a simple polling loop, and it ran on an 8-bit processor instead of a 16-bitter, with much less RAM.

An equal danger lies in oversimplification. Beware of a low-falutin' design for a high-falutin' problem. A naive design causes expensive problems, because you often don't see the subtle cases that are mishandled until later in the development cycle. We now have ample data about the cost of finding and fixing design flaws. The price goes up dramatically with time. It's considerably cheaper to find and fix shortcomings in a design review than by responding to bug reports from the field.

## DESIGN AND IMPLEMENTATION

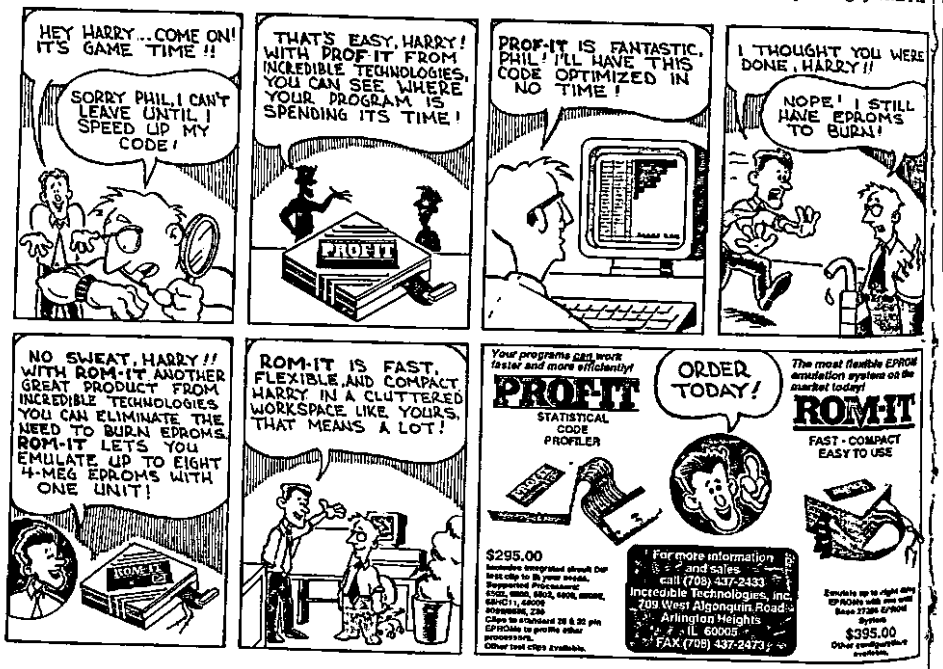
I check for two things every time I review a design for an embedded system. First, I look for synchronization points. If shared data is not protected from dual access, problems are inevitable. (Often, however, they only appear when the system is loaded.) Next, I check each module that must make decisions. If data that might affect the decision isn't available to the module, it will surely be ignored by the implementors. When the need is discovered later, the data will probably be smuggled to the right place by some undesirable channel.

The actual implementation is yet another aspect. Don't confuse implementation with design. The designers come up with a blueprint for solving the problem. The implementors make the solution work. Designers like to think that they dictate all the important details of an implementation. They are wrong.

A high-falutin' program for a low-falutin' design can mask a lot of simplicity. Here is one place where hotshot assembly-language programmers cause a lot of trouble. They like to turn five pages of C code into 30 pages of unreadable text, just to shave bytes and microseconds in a few "critical" places (they usually guess wrong about which places are critical, by the way).

You get the same problem at the other extreme. The latest fad is object-oriented programming (in case you've been in Antarctica for the past two years). You can do many good things with OOP, but you can also obscure problems. Between polymorphism and operator overloading, you can make a C++ program that actively misleads. It may look like C, but it acts like APL—with performance to match. As I've said repeatedly, save the big guns for the big problems.

A low-falutin' program for a high-falutin' design is even worse. This disease is common in large programming shops that are managed "top down." Management embraces the comfortable notion that programmers are a commodity. They hire undertrained programmers (so-called "Mongolian hordes") then train them only superficially, if at all.



Most competent programmers soon learn to command higher salaries as analysts and designers. Management tries to control the process by frequent progress reports, code reviews, and a form of quality assurance (QA). But the QA usually consists of still more undertrained programmers writing low-falutin' test cases to beat against the low-falutin' deliverable code. You don't have to guess the outcome. You've either experienced this sort of fiasco personally, had a friend who did, or read about the financial fallout.

Now we get to the programmer. Simply put, you can't expect low-falutin' programmers to pull off high-falutin' programming jobs. It's just not in the cards. Please understand—I'm not talking native intelligence here. The smartest people in the world with only a year of experience are going to be low-falutin' programmers. They simply lack the skills and experience. Give them a few years of training, guidance, and on-the-job successes, however, and their falutin' index will rise.

Should high-falutin' programmers be given low-falutin' jobs? Of course. A good programmer should be prepared to tackle anything. A really good programmer will know not to use a sledge hammer when a nutcracker will suffice. A good manager will know when to team a low-falutin' programmer with a high-falutin' one on a medium-falutin' job. Programming is a craft, and apprenticeship is a proven method for passing along craft knowledge.

Here is an important opportunity for self-examination. You should have a good sense of your falutin' index as a programmer. Some people have a natural ability to deal with lots of complexity (Ken Thompson and Dennis Ritchie, the designers of UNIX and C, are two notable examples). The rest of us must struggle with complexity in smaller chunks. Not to worry. The more techniques you learn for crafting programs, the more complexity you can manage. You may take smaller bites than others, but you learn to chew faster, as it were.

If you know your limits, you are less likely to get in over your head. You also have a better notion as to where you must stretch yourself to grow. And you are more aware of the bias I spoke of earlier—you know what size problems you want to solve.

**A medium-falutin' programmer is one who can work in a group.**

## HOW FALUTIN'?

I began this article with several questions. They were not rhetorical. They concern yet another aspect of the program development business—the kind of organization you like to work in. Just as you can have a mismatch between design and implementation, you can be a person in the wrong kind of organization (for you, that is).

Organizationally speaking, a low-falutin' programmer is a loner. That's not necessarily bad. Give a low-falutin' type a low-falutin' job, and it will get done quickly. I have worked with many competent people of this description with success. In fact, I favor the low-falutin' style myself, at least organizationally (I like to work on low-to-medium-falutin' problems).

Organizationally, a medium-falutin' programmer is one who can work in a group. You need these folks to tackle the medium-to-high-falutin' problems. Once a job gets too big to be handled by one person, communication skills become important. Programming skills are also important, but less so. Unless the pieces fit together, it doesn't make a bit of difference how good each piece is separately.

A high-falutin' programmer is one who can work on projects that require multiple groups. Here is where the management becomes more important than the technology, as I indicated earlier. You need folks who know the technology well enough but don't have to actually write code to feel fulfilled. These types sublimate their technical urges by helping others get the job done. I have never seen a large project succeed without a serious complement of good technical managers. Never mind what the business schools say, an M.B.A. can't cut it alone in our trade.

Low-falutin' programmers in high-falutin' organizations are literally in over their heads. They don't know how to behave in committee meetings. They don't understand the need for all those Mickey Mouse reports. They are, in short, accidents waiting to happen. Bear that in mind the next time you get a tempting job offer. Is the organization much more structured than the one you've worked in so far? In your effort to get ahead, make sure you don't walk off the end of the pier.

You also have to be wary if you're accustomed to a high-falutin' organization. That neat consulting opportunity may not be what you expect. Don't give up your data repository and QA department until you know you can work comfortably without them.

The falutin' index applies to other aspects. You have low-falutin' customers who are unsophisticated and high-falutin' ones who demand lots of deliverables. You may have a low-falutin' work environment in a high-falutin' organization (upper management may be cheapskates).

Given all these aspects, the number of falutin' profiles are practically endless. Think about the projects you have worked on. Were they successful? How consistent was the complexity profile? My experience is that these two answers tend to be highly correlated (you may know of a few exceptions). Successful projects are congruent, or they aren't successful.

*P J Plauger has been active in the development of standards, most notably for the C programming language. His latest books are* The Standard C Library, *published by Prentice-Hall in Englewood Cliffs, N.J., and (with Jim Brodie)* ANSI and ISO Standard C, *published by Microsoft Press in Redmond, Wash.*