Forewords so as not to bias the reader up front. Mostly, the essays speak for themselves.

Other collections from "Programming on Purpose" deal with other themes. Besides program design, I have written essays on (among other things): programming technology, software standards development, the business of software, and the people who love and write computer software. Some essays are humorous, some are deadly serious. A few are gems, but I like to think that all are worth reading. If you enjoy what you find here, please consider the other collections as well.

The magazine business sees considerable turnover of editorial staff. Miller Freeman, the publisher of *Computer Language*, is no exception. I have thus enjoyed the services of many editors over the years. All have worked hard to rescue my prose from its more florid excursions. They have nevertheless permitted me to retain a certain colloquial illiteracy that I find comfortable. I thank all the people at Miller Freeman who, over the years, have helped make these essays more readable. You should too.

Two people in particular deserve oak-leaf clusters. Regina Starr Ridley, now a publisher at Miller Freeman, was one of my earliest editors. And Nicole Freeman, now a managing editor there, has cheerfully haunted my career in many editorial guises. I am happy to acknowledge their continuing assistance in making "Programming on Purpose" better. I am also happy to count both as good friends.

Having given credit where it is due, I must issue a warning. I re-edited these essays from the original machine readable. I certainly strove to recapture the spirit of *Computer Language* edits, but I make no pretense at following them to the letter. If any have lost ground as a result, you can blame me.

P.J. Plauger
Concord, Massachusetts

1 Which Tool is Best?

If you had to build a wooden table, which tool would you use? A saw gets you off to a good start, but it's lousy for shaping round legs and for driving screws. It also leaves much to be desired when it comes to finishing the surface and applying paint. With a lathe, you can do a great job of turning those legs. But I leave it to your comic imagination to envision how you would use it on the other jobs. And if those images don't brighten your day, replay the scenes with, in turn: a hammer, a screw driver, and a pair of pliers. The best compromise might be the proverbial Swiss Army Knife, which is equally poor at all operations.

It is ridiculous, of course, to even think of building something as elaborate as a table with just one tool. Try to convince a practicing carpenter to do so and you'll be dismissed as daft. Yet this is exactly what goes on in the programming profession every day. A handful of tool sellers keep trying to convince us that there is one right tool for developing software.

Speaking as someone who spent years selling programming development methods (read: Snake Oil Miracle Cure) with the best of them, I can tell you that life ain't that simple. I have since done ten years' penance for my sins, by writing hundreds of thousands of lines of commercial software. Most of that has been for my company, Whitesmiths, Ltd. More recently, I've watched others write still more. The experience has been humbling.

What we have learned collectively is that there are many good techniques for building software, but no one is "best." No one technique is even *adequate* when taken alone. We follow all the rules of *The Elements of Programming Style* (**K&P74, K&P78**), and then some. We write structured code, nearly all the time, and practice top-down design as much as possible. We use the latest program-development software as described in *Software Tools* (**K&P76, K&P81**), and then some. In short, we practice what I've preached for years. But that isn't enough.

There is the apocryphal story of the famous mathematician giving a lecture. He fills board after board with abstruse formulae, his audience slaving furiously to keep up with his leaps of logic. A few less hardy souls cringe when, for the sixth time, he begins a sentence with, "It is obvious that ..." But this time he hesitates. He repeats the dread phrase and hesitates again. Then he walks out of the room! Ten minutes later, just as the audience is getting restless, he returns. He picks up the chalk, says, "It *is* obvious." And continues with his lecture.

That's how we found that top-down design works much of the time. Once you have composed a program, it is easy to look at it and see how you *should* have arrived at it by the orderly process of stepwise refinement from a global statement of purpose. Getting there from a standing start is a different matter entirely. For a test tube sized problem, you can almost always succeed. For a problem whose "shape" is familiar, you just build a structure of similar shape and you have a good chance of getting to the bottom without going astray. But for something big enough (by the standards of your experience) and new enough (ditto), you can get just as lost starting at the top as you can at the bottom.

To put it crassly, top-down design is a great way to *redesign* a program you already know how to write.

Does that make it wrong to teach top-down design to programmers? Not at all. As you acquire experience, you get better and better at making good programs top down. And if you can succeed by stepwise refinement, you almost always get a better product than with the undisciplined approach most programmers adopted in the past. It wasn't even that programmers coded bottom up instead, for bottom-up design is a perfectly good discipline that also has its arena of applicability. No, most programmers of my acquaintance *just started writing code*, following no discipline whatsoever. And that is a technique that works well only for people with *lots* of experience.

I have a brother-in-law who is a skilled cartoonist. He can start in the upper left corner of a sheet of paper and elaborate a brilliant drawing. All the proportions are right and everything is in proper perspective. Us mortals must follow the usual art-school rules (see back of match cover for advertisement), block out the shapes, then fill in the detail. Ken Thompson, the originator of the UNIX operating system, has a similar skill with programming. He can write a chunk of code *in assembly language* that is half again more complex than you or I would tackle in C or Pascal, and get it right on the first draft. People like him give top-down design a bad name.

It is more fair to say that top-down design has not been practiced nearly enough, by every day programmers, unless it's really pushed. Now it has been oversold, in some circles, to the point that people feel obliged to use it even when it is not the best technique.

I had to write my third C compiler before I began to see how to design the whole thing from the top down. Don't misunderstand — great chunks of parsing code and symbol table management were designed top down from the outset. But then I'd done that sort of thing for various assemblers and editors in the past. The compiler as a whole did not make sense until I figured out that it had *at least three "tops."* And that insight I got from applying data-flow analysis, which is nominally a tool of structured analysis. (See **deM79**.)

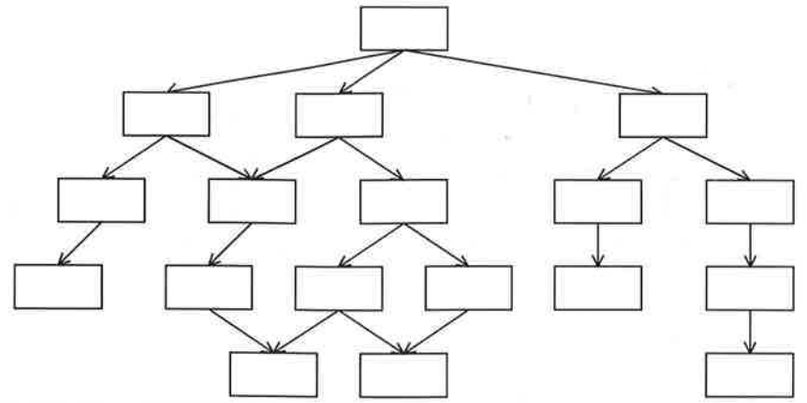

Figure 1.1 *A typical (small) structure chart.*

Every method for designing software that's worth its salt comes with some way to document your design. For top-down design, the ultimate goal is to produce a *structure chart* or, as IBM commercialized it, a *HIPO Chart*. HIPO stands for Hierarchy with Input, Process, and Output.

What it is is an upside down tree (root at the top) of boxes. Each box is a subroutine and all its subordinates in the tree are the subroutines that it calls. You can even document the passing of arguments up and down the tree by drawing little arrows with names on them, beside the bigger arrows connecting the boxes. (See Figure 1.1.) A structure chart reflects the fundamental belief of top-down design that you design a program by decomposing it into successively less abstract subroutines. When you get down to subroutines that do atomic operations, such as **`READ`** and **`WRITE`** statements or simple calculations, you are done.

There are some fundamentalist sects in top-down design. Those who take the acronym HIPO seriously insist that every box should have *exactly* three subordinates, one to get Input, one to do the Process, and one to emit the Output. (See Figure 1.2.) This Procrustean attitude reminds me of Frederic the Great's reputed approach to organizing the Prussian Army, which can be overstated as three principles:

1. Any officer who has fewer than seven subordinates hasn't delegated enough responsibility. Get rid of him.
2. Any officer who has more than seven subordinates can't keep track of them all. Get rid of him.
3. Any officer who can't guess the first two rules doesn't know how to organize an army. Get rid of him.

The result is obvious.

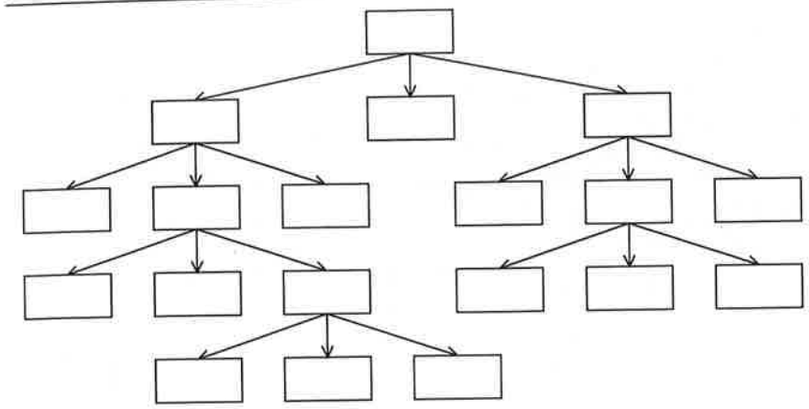

Figure 1.2 *A purist's HIPO chart.*

Another fundamentalist sect has it that each of the lower level boxes is called by only one superior, at the next level up. (See Figure 1.3.) This is the pure "stepwise refinement" approach that shuns any attempt to identify common subroutines and use them to advantage. In sooth, such attempts are akin to library building, which is one of the principal activities in bottom-up design.

In bottom-up design, the approach is to try to guess all of the low level routines you are going to need, then stockpile them. Build enough of them and you can see how to write fancier routines that call on the ones you wrote earlier. If you guess right, eventually you will be able to write a main routine that calls on your library of lower-level routines to do all the hard stuff. You have reached the top.

In real life, you do both top-down and bottom-up design on any non-trivial program. You practice stepwise refinement for a spell, until you start

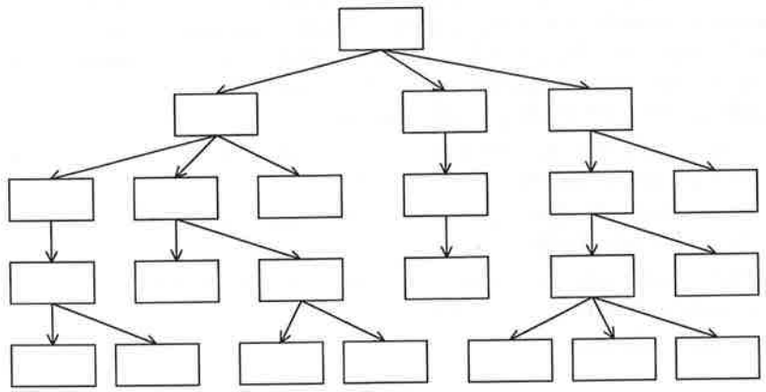

Figure 1.3 *A purist's stepwise refinement.*

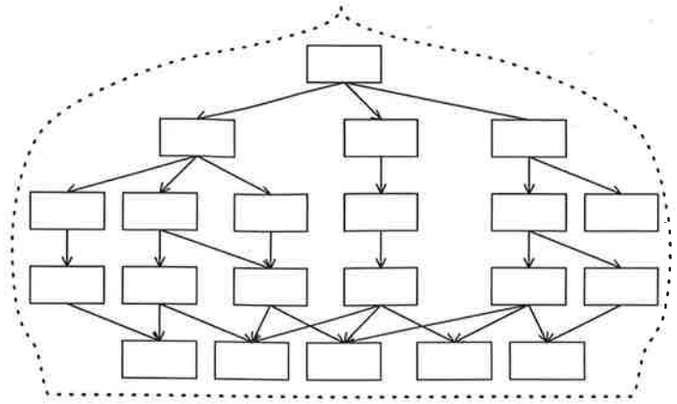

Figure 1.4 *A mosque-shaped structure chart.*

to notice that different parts of the program need to call upon the environment in similar ways. You then begin to compromise among various needs for, say, opening files or allocating chunks of memory, until you end up with a dozen or so *primitives* that perform all interactions with the environment. These dozen-odd primitives form the bottom level of a structure chart that may fan out to several times as many functions part way down.

So instead of ending up with a pyramid-shaped structure chart, as the stepwise-refinement purists would have it, you get a "mosque." (See Figure 1.4.) Like the turrets in Arabian architecture, your chart flares out from a point at the top, then necks back in at the bottom. Larry Constantine was the first person I know to describe mosque-shaped structure charts, and the forces that bring them into existence. (See **Y&C89** for some practical techniques for building programs.)

In summary, you judge a design method by the documentation it produces. For top-down design, you look at structure charts. The more dogmatic you are about applying a design method, the fewer real-life problems you are going to solve. I have never seen a real-life structure chart, for a program of significant size, that follows the HIPO "rule of three" or the stepwise refinement "fan-out-only rule." People who are best at doing what they call top-down design have a wealth of experience to draw upon. Part of this wealth is skill in one or more other design techniques, which they apply almost subconsciously.

With that as a preamble, I can now tell you where top-down design *really* falls down.

I drew a structure chart of the first C compiler I wrote. It was a clean, well organized program (as compilers go), but the structure chart was a mess. Why? Because of a little design choice I made. It was one that no COBOL programmer ever had to make, a design choice that is almost not

a choice at all for any writer of modern compilers. I made heavy use of recursion. Consider the simple grammar for an arithmetic expression:

```
expr := term | expr BINOP term
term := NUMBER | UNOP term | ( expr )
```

This defines an expression as either a term or (recursively) an expression followed by a binary operator (**BINOP**) and a term. A term is some **NUMBER**, or a unary operator (**UNOP**) followed (recursively) by a term, or a parenthesized expression.

The last item brings the recursion full circle. You can (and should) express this as two functions:

- **get_expr** — which obtains an expression, by calling itself and **get_term**
- **get_term** — which obtains a term, by calling itself and **get_expr**

The structure chart consists of two boxes, *each of which points at the other.* (See Figure 1.5.) You write **get_expr** as the higher-level function, because the rest of your program will always be looking for expressions, never terms. But this little loop back creates a tangle unanticipated by the developers of structure charts. There were few widely used recursive languages in those days.

Now imagine a C compiler with a much more elaborate subtree of functions for computing expressions. It has a few back loops in its structure chart, including at least one back to the top. That compiler also has a fairly complex subtree of functions for parsing types. These types are also recursively defined, so the structure chart for that part also has one or more backloops. And now for the fun part: C uses expressions in defining types, such as the values of enumeration constants, the size of bit fields, and the size of arrays. C also uses types in defining expressions, for writing type cast operators!

The structure chart for the whole works looks like the New York City subway system. (No figure attempted.)

One of the strong selling points for structured programming is that it makes for more readable flow charts. You use a flow chart to document the control flow *within* a routine much as you use a structure chart to document the control flow *among* separate routines. Unstructured routines were rightly accused of resembling a bowl of spaghetti, because their flow charts sprawled seemingly at random. Tracing one strand was an exercise in despair.

If a document doesn't help you control complexity by partitioning it into manageable chunks, it is not pulling its weight. It also makes you suspect the utility of the design strategy that leads to a document you find hard to understand. In this example, I see failure at two extremes. The simple grammar has a structure chart that is trivial, and captures almost none of

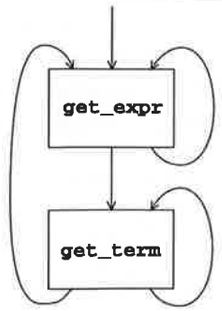

Figure 1.5 *Structure chart for the expression parser.*

the complexity of the problem. The original grammar was best at doing that. And the complex C compiler has a structure chart which is as tangled as many an unstructured program. Perhaps structure charts are not appropriate for documenting certain programs.

I have picked on top-down design at length because it is best entrenched as the Right Way to do things. If you have your own favorite design method, I can assure you that there are practical situations where it breaks down as well. Do you believe in data-structured design, as taught by Jean Dominique Warnier and/or Ken Orr(**War78, Orr77**)? Try it on a sort module and see how far you get. Do you believe that data-flow design, mentioned earlier, is best? See how much it helps you write a fast Fourier transform.

It is only natural that, when you discover a technique that helps you solve an interesting class of problems, you focus on the arena where you get successes, to the exclusion of others. It is also only natural for someone selling books or training on a given design method to play down its limitations. But neither of these tendencies is an excuse for misapplying a technique where it is useless at best and harmful at worst. As a colleague of mine likes to say, to a five-year-old with a hammer, everything looks like a nail.

My goal in writing this premiere essay is to convince you that no one tool is best for developing computer programs. My goal in coming essays is to introduce you to the many tools that I have added to my carpenter's bench over the years. For each I will show where you can use it, how it works, and how you know when you are done with it. I will also show you where it is not at its best, or where you should not use it at all.

I have chosen the title "Programming on Purpose" for two reasons. First, I want to contrast the methods described here with the programming by accident that happens altogether too often. And second, I intend to address my remarks to the serious, goal-oriented programmer. If you haven't got a clear purpose in writing code, then you are "hacking" in the worst sense of that word. I personally have little interest in that pastime.

In the near quarter century since I started programming for a living, I have seen the profession change dramatically. What was once regarded as a black art became in time an art form that many could master. It is now changing from an art to a craft, a trade that can be mastered by almost anyone with the proper patience and motivation. (The term "craft" also has overtones of producing something useful, not merely decorative.) Only parts of computer programming have become an engineering discipline, and fewer parts still have matured to a science. Like any earnest craftsman, I borrow freely from engineering and science where they help get a job done. I make no pretense at being either a software engineer or a computer scientist.

Several years ago, I gave a series of ACM lectures in New York City under the title, "Methods of System Design." Despite the dry title, the lectures were well received (or so it appeared to me). These essays draw upon that material, plus the experience I have acquired since then. I hope you find them useful. □

Afterword: This, obviously, was my premiere column. My fear as I was writing it was that I was out of touch with the business of software-design methodologies. All my references were old! It took a bit of hurried reading for me to learn that those old references were still current. Few books were published on the business of software design in the early 1980s. (The trade has since picked up, but still not to the level of the late 1970s.)

I find this essay to be a good overview of software-design methods. It also gives a succinct statement of intent for the columns that follow.

2 Writing Predicates

METHOD: Inside-out design.

DESCRIPTION: Inside-out design focuses on the actual expression of a module. It is invariably the last stage in the process of capturing an algorithm in executable code. The name suggests that you are working from the middle of a program (the processing of data) out to the edges (the input/output interface).

Since all programs deal with conditional logic, or predicates, the principal technology of inside-out design is aimed at getting predicates correct, readable, and efficient (in that order). Completeness checks, de Morgan's Rules, decision tables, and Karnaugh maps all serve this end.

DOCUMENTATION: Decision tables best capture the logic of a complex predicate, at a level that permits of multiple implementations and that serves as a more readable rendition of a given implementation.

LIMITATIONS: Inside-out design involves you in the full complexity of a program at the most detailed level. To apply it before a program has been reduced, by other methods, to a set of functionally cohesive modules is to invite confusion and disaster.

A predicate is that Boolean expression you write following a **`WHILE`** or **`IF`** to determine which statement gets executed next. Whatever discipline you use to elaborate the structure of your program, in the end you must get all the predicates right to have a working product. There's lots of technology you can bring to bear on getting predicates right, but I have never seen it put in one place before. This essay is my attempt at summarizing most of the techniques I see good programmers use every day in writing and debugging control-flow logic.

The preamble to this essay is the first in a series of *manual pages* covering various aspects of designing computer software. It may seem rather ponderous for a topic so mundane as expressing **`WHILE`** and **`IF`** statements correctly, but I believe that it contains several important reminders, particularly on the topic of limitations. In the company of its siblings, appearing in subsequent essays, it should appear somewhat less pompous.

To begin with a concrete example, consider the classic algorithm for computing the greatest common divisor, or GCD, of two integers. It has been a favorite of computer textbook writers for years, because it is elegant and compact. Or at least it can be. The basic idea is to divide the smaller integer into the larger and keep the remainder. If the remainder is zero, the