1
The Tar Pit

Een schip op het strand is een baken in zee.
[*A ship on the beach is a lighthouse to the sea.*]

DUTCH PROVERB

C. R. Knight, Mural of La Brea Tar Pits
Photography Section, Natural History Museum of Los Angeles County

No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind's eye one sees dinosaurs, mammoths, and sabertoothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks.

Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems—few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty—any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it.

Therefore let us begin by identifying the craft of system programming and the joys and woes inherent in it.

The Programming Systems Product

One occasionally reads newspaper accounts of how two programmers in a remodeled garage have built an important program that surpasses the best efforts of large teams. And every programmer is prepared to believe such tales, for he knows that he could build *any* program much faster than the 1000 statements/year reported for industrial teams.

Why then have not all industrial programming teams been replaced by dedicated garage duos? One must look at *what* is being produced.

In the upper left of Fig. 1.1 is a *program.* It is complete in itself, ready to be run by the author on the system on which it was developed. *That* is the thing commonly produced in garages, and

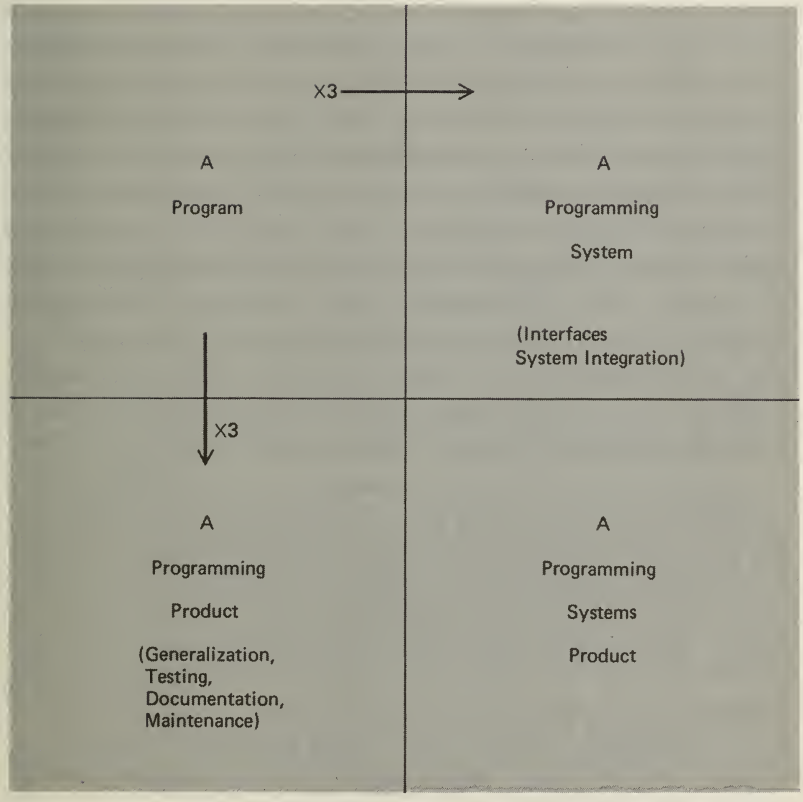

Fig. 1.1 Evolution of the programming systems product

that is the object the individual programmer uses in estimating productivity.

There are two ways a program can be converted into a more useful, but more costly, object. These two ways are represented by the boundaries in the diagram.

Moving down across the horizontal boundary, a program becomes a *programming product.* This is a program that can be run,

tested, repaired, and extended by anybody. It is usable in many operating environments, for many sets of data. To become a generally usable programming product, a program must be written in a generalized fashion. In particular the range and form of inputs must be generalized as much as the basic algorithm will reasonably allow. Then the program must be thoroughly tested, so that it can be depended upon. This means that a substantial bank of test cases, exploring the input range and probing its boundaries, must be prepared, run, and recorded. Finally, promotion of a program to a programming product requires its thorough documentation, so that anyone may use it, fix it, and extend it. As a rule of thumb, I estimate that a programming product costs at least three times as much as a debugged program with the same function.

Moving across the vertical boundary, a program becomes a component in a *programming system.* This is a collection of interacting programs, coordinated in function and disciplined in format, so that the assemblage constitutes an entire facility for large tasks. To become a programming system component, a program must be written so that every input and output conforms in syntax and semantics with precisely defined interfaces. The program must also be designed so that it uses only a prescribed budget of resources—memory space, input-output devices, computer time. Finally, the program must be tested with other system components, in all expected combinations. This testing must be extensive, for the number of cases grows combinatorially. It is time-consuming, for subtle bugs arise from unexpected interactions of debugged components. A programming system component costs at least three times as much as a stand-alone program of the same function. The cost may be greater if the system has many components.

In the lower right-hand corner of Fig. 1.1 stands the *programming systems product.* This differs from the simple program in all of the above ways. It costs nine times as much. But it is the truly useful object, the intended product of most system programming efforts.

The Joys of the Craft

Why is programming fun? What delights may its practitioner expect as his reward?

First is the sheer joy of making things. As the child delights in his mud pie, so the adult enjoys building things, especially things of his own design. I think this delight must be an image of God's delight in making things, a delight shown in the distinctness and newness of each leaf and each snowflake.

Second is the pleasure of making things that are useful to other people. Deep within, we want others to use our work and to find it helpful. In this respect the programming system is not essentially different from the child's first clay pencil holder "for Daddy's office."

Third is the fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles, playing out the consequences of principles built in from the beginning. The programmed computer has all the fascination of the pinball machine or the jukebox mechanism, carried to the ultimate.

Fourth is the joy of always learning, which springs from the nonrepeating nature of the task. In one way or another the problem is ever new, and its solver learns something: sometimes practical, sometimes theoretical, and sometimes both.

Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. (As we shall see later, this very tractability has its own problems.)

Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has

come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.

Programming then is fun because it gratifies creative longings built deep within us and delights sensibilities we have in common with all men.

The Woes of the Craft

Not all is delight, however, and knowing the inherent woes makes it easier to bear them when they appear.

First, one must perform perfectly. The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.[1]

Next, other people set one's objectives, provide one's resources, and furnish one's information. One rarely controls the circumstances of his work, or even its goal. In management terms, one's authority is not sufficient for his responsibility. It seems that in all fields, however, the jobs where things get done never have formal authority commensurate with responsibility. In practice, actual (as opposed to formal) authority is acquired from the very momentum of accomplishment.

The dependence upon others has a particular case that is especially painful for the system programmer. He depends upon other people's programs. These are often maldesigned, poorly implemented, incompletely delivered (no source code or test cases), and poorly documented. So he must spend hours studying and fixing things that in an ideal world would be complete, available, and usable.

The next woe is that designing grand concepts is fun; finding nitty little bugs is just work. With any creative activity come

dreary hours of tedious, painstaking labor, and programming is no exception.

Next, one finds that debugging has a linear convergence, or worse, where one somehow expects a quadratic sort of approach to the end. So testing drags on and on, the last difficult bugs taking more time to find than the first.

The last woe, and sometimes the last straw, is that the product over which one has labored so long appears to be obsolete upon (or before) completion. Already colleagues and competitors are in hot pursuit of new and better ideas. Already the displacement of one's thought-child is not only conceived, but scheduled.

This always seems worse than it really is. The new and better product is generally not *available* when one completes his own; it is only talked about. It, too, will require months of development. The real tiger is never a match for the paper one, unless actual use is wanted. Then the virtues of reality have a satisfaction all their own.

Of course the technological base on which one builds is *always* advancing. As soon as one freezes a design, it becomes obsolete in terms of its concepts. But implementation of real products demands phasing and quantizing. The obsolescence of an implementation must be measured against other existing implementations, not against unrealized concepts. The challenge and the mission are to find real solutions to real problems on actual schedules with available resources.

This then is programming, both a tar pit in which many efforts have floundered and a creative activity with joys and woes all its own. For many, the joys far outweigh the woes, and for them the remainder of this book will attempt to lay some boardwalks across the tar.