11
Plan toThrow OneAway

There is nothing in this world constant but inconstancy.

SWIFT

It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all try ***something.***

FRANKLIN D. ROOSEVELT

Collapse of the aerodynamically misdesigned Tacoma Narrows Bridge, 1940
UPI Photo/The Bettman Archive

Pilot Plants and Scaling Up

Chemical engineers learned long ago that a process that works in the laboratory cannot be implemented in a factory in only one step. An intermediate step called the *pilot plant* is necessary to give experience in scaling quantities up and in operating in nonprotective environments. For example, a laboratory process for desalting water will be tested in a pilot plant of 10,000 gallon/day capacity before being used for a 2,000,000 gallon/day community water system.

Programming system builders have also been exposed to this lesson, but it seems to have not yet been learned. Project after project designs a set of algorithms and then plunges into construction of customer-deliverable software on a schedule that demands delivery of the first thing built.

In most projects, the first system built is barely usable. It may be too slow, too big, awkward to use, or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved. The discard and redesign may be done in one lump, or it may be done piece-by-piece. But all large-system experience shows that it will be done.[2] Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time.

The management question, therefore, is not *whether* to build a pilot system and throw it away. You *will* do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers. Seen this way, the answer is much clearer. Delivering that throwaway to customers buys time, but it does so only at the cost of agony for the user, distraction for the builders while they do the redesign, and a bad reputation for the product that the best redesign will find hard to live down.

Hence *plan to throw one away; you will, anyhow.*

The Only Constancy Is Change Itself

Once one recognizes that a pilot system must be built and discarded, and that a redesign with changed ideas is inevitable, it becomes useful to face the whole phenomenon of change. The first step is to accept the fact of change as a way of life, rather than an untoward and annoying exception. Cosgrove has perceptively pointed out that the programmer delivers satisfaction of a user need rather than any tangible product. And both the actual need and the user's perception of that need will change as programs are built, tested, and used.[3]

Of course this is also true of the needs met by hardware products, whether new cars or new computers. But the very existence of a tangible object serves to contain and quantize user demand for changes. Both the tractability and the invisibility of the software product expose its builders to perpetual changes in requirements.

Far be it from me to suggest that all changes in customer objectives and requirements must, can, or should be incorporated in the design. Clearly a threshold has to be established, and it must get higher and higher as development proceeds, or no product ever appears.

Nevertheless, some changes in objectives are inevitable, and it is better to be prepared for them than to assume that they won't come. Not only are changes in objective inevitable, changes in development strategy and technique are also inevitable. The throw-one-away concept is itself just an acceptance of the fact that as one learns, he changes the design.[4]

Plan the System for Change

The ways of designing a system for such change are well known and widely discussed in the literature—perhaps more widely dis-

cussed than practiced. They include careful modularization, extensive subroutining, precise and complete definition of intermodule interfaces, and complete documentation of these. Less obviously one wants standard calling sequences and table-driven techniques used wherever possible.

Most important is the use of a high-level language and self-documenting techniques so as to reduce errors induced by changes. Using compile-time operations to incorporate standard declarations helps powerfully in making changes.

Quantization of change is an essential technique. Every product should have numbered versions, and each version must have its own schedule and a freeze date, after which changes go into the next version.

Plan the Organization for Change

Cosgrove advocates treating all plans, milestones, and schedules as tentative, so as to facilitate change. This goes much too far—the common failing of programming groups today is too little management control, not too much.

Nevertheless, he offers a great insight. He observes that the reluctance to document designs is not due merely to laziness or time pressure. Instead it comes from the designer's reluctance to commit himself to the defense of decisions which he knows to be tentative. "By documenting a design, the designer exposes himself to the criticisms of everyone, and he must be able to defend everything he writes. If the organizational structure is threatening in any way, nothing is going to be documented until it is completely defensible."

Structuring an organization for change is much harder than designing a system for change. Each man must be assigned to jobs that broaden him, so that the whole force is technically flexible. On a large project the manager needs to keep two or three top programmers as a technical cavalry that can gallop to the rescue wherever the battle is thickest.

Management structures also need to be changed as the system changes. This means that the boss must give a great deal of attention to keeping his managers and his technical people as interchangeable as their talents allow.

The barriers are sociological, and they must be fought with constant vigilance. First, managers themselves often think of senior people as "too valuable" to use for actual programming. Next, management jobs carry higher prestige. To overcome this problem some laboratories, such as Bell Labs, abolish all job titles. Each professional employee is a "member of the technical staff." Others, like IBM, maintain a dual ladder of advancement, as Fig. 11.1 shows. The corresponding rungs are in theory equivalent.

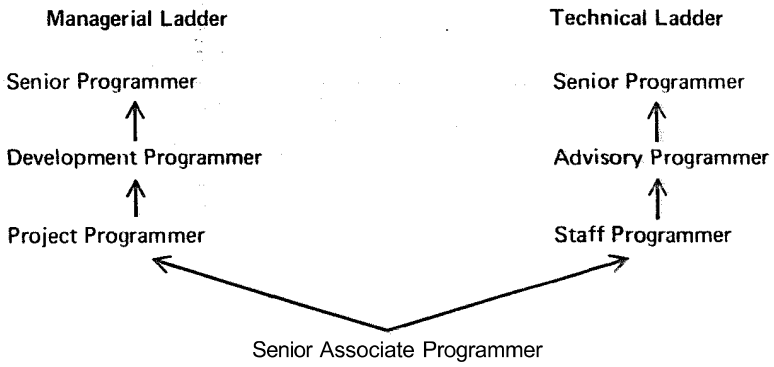

Fig. 11.1 IBM dual ladder of advancement

It is easy to establish corresponding salary scales for rungs. It is much harder to give them corresponding prestige. Offices have to be of equal size and appointment. Secretarial and other support services must correspond. A reassignment from the technical ladder to a corresponding level on the managerial one should never be accompanied by a raise, and it should be announced always as

The fundamental problem with program maintenance is that fixing a defect has a substantial (20-50 percent) chance of introducing another. So the whole process is two steps forward and one step back.

Why aren't defects fixed more cleanly? First, even a subtle defect shows itself as a local failure of some kind. In fact it often has system-wide ramifications, usually nonobvious. Any attempt to fix it with minimum effort will repair the local and obvious, but unless the structure is pure or the documentation very fine, the far-reaching effects of the repair will be overlooked. Second, the repairer is usually not the man who wrote the code, and often he is a junior programmer or trainee.

As a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire bank of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice such *regression testing* must indeed approximate this theoretical ideal, and it is very costly.

Clearly, methods of designing programs so as to eliminate or at least illuminate side effects can have an immense payoff in maintenance costs. So can methods of implementing designs with fewer people, fewer interfaces, and hence fewer bugs.

One Step Forward and One Step Back

Lehman and Belady have studied the history of successive releases in a large operating system.[8] They find that the total number of modules increases linearly with release number, but that the number of modules affected increases exponentially with release number. All repairs tend to destroy the structure, to increase the entropy and disorder of the system. Less and less effort is spent on fixing original design flaws; more and more is spent on fixing flaws introduced by earlier fixes. As time passes, the system becomes less and less well-ordered. Sooner or later the fixing

ceases to gain any ground. Each forward step is matched by a backward one. Although in principle usable forever, the system has worn out as a base for progress. Furthermore, machines change, configurations change, and user requirements change, so the system is not in fact usable forever. A brand-new, from-the-ground-up redesign is necessary.

And so from a statistical mechanical model, Belady and Lehman arrive for programming-systems at a more general conclusion supported by the experience of all the earth. "Things are always at their best in the beginning," said Pascal. C. S. Lewis has stated it more perceptively:

> *That is the key to history. Terrific energy is expended—civilizations are built up—excellent institutions devised; but each time something goes wrong. Some fatal flaw always brings the selfish and cruel people to the top, and then it all slides back into misery and ruin. In fact, the machine conks. It seems to start up all right and runs a few yards, and then it breaks down."*[1]

Systems program building is an entropy-decreasing process, hence inherently metastable. Program maintenance is an entropy-increasing process, and even its most skillful execution only delays the subsidence of the system into unfixable obsolescence.