Alan Davis, Editor

SOFTWARE LEMMINGINEERING

lemmingineering, \lem-'en-je-'ni ər-iŋ\ n : the process of engineering systems by blindly following techniques the masses are following, without regard to the appropriateness of those techniques.

IN THE SUMMER OF 1958, I SAT IN FRONT of my parents' bungalow at Rockaway Beach, armed with a pencil and a pad of paper. Day and evening, I collected data on the never-ending parade of subway trains that stopped at the station at the end of the block.

Every day I collected long columns of data. The neighbors were in awe. The neighbors' children were jealous every time I shouted out a new all-time record.

Obviously, the Davis boy had great talent with numbers. Obviously, he was going to be a great mathematician. Obviously, he was on the path to wisdom, truth, knowledge. I was encouraged to continue. If I was collecting data, it must be good.

What data was I collecting that entire summer 35 years ago? Arrival time, departure time, train number, number of passengers who got on board, and the number of passengers who got off. Then I added all these together to get a grand total for that train, gleefully announcing each new high. Very valuable stuff.

I was not doing anything more worthwhile, positive, or smart than the next kid. But I had a lot of people thinking otherwise for awhile.

It seems to me that software developers, in their search for the Holy Grail, are like all those who were convinced I was doing something meaningful that summer. Like children, we envy others who seem to know something we don't know. Like lemmings, we tend to follow the leader, without ever asking ourselves if we want to go where the leader is taking us. We tell ourselves that if everybody else is taking a certain path, it must be the right way to go.

I would like to play the devil's advocate for a moment and examine some software-engineering paths that have led to cliffs or box canyons in the past, paths we now believe will lead to the Promised Land, and those that have not yet been blazed. My goal is to get us all to think more about our chosen path so that we can make an intelligent decision to remain on it or perhaps to take a road less traveled.

Table 1 summarizes the lemming paths explored in this column and gives you an idea of their popularity, goals, risks and payoffs.

LIKE CHILDREN, WE ENVY THOSE WHO MAY KNOW SOMETHING WE DON'T. LIKE LEMMINGS, WE TEND TO FOLLOW THE LEADER.

STRUCTURED PATH. In the 1970s, structured techniques were sold as a solution to the software industry's apparently escalating costs and growing customer dissatisfaction. First it was structured programming, then structured analysis and structured design. For a decade, we blindly marched in formation to the beat of a structured drum. By the end of the '70s, "structured" had become synonymous with "good." We were barraged with the adjective "structured."

In retrospect, the collection of widely disparate programming practices that have at various times been called "structured" has clearly been infused into our professional culture. Today, we no longer say "structured programming," we simply say "programming."

Structured analysis and structured design have not fared as well. The original structured analysis paralyzed us through overanalysis and was found unsuitable for complex real-time systems. Later versions, proposed by Paul Ward, Derek Hatley, and Edward Yourdon solved the overanalysis problems and added control mechanisms to more easily model real-time systems.

But no one stopped to ask why we were doing structured analysis in the first place. If we had, we may have realized that the goal of the requirements phase — a document that describes the system as a black box — cannot be described with hierarchical dataflow (or control-flow) diagrams.

And structured design, a simple set of practices to transform a structured analysis into a calling hierarchy, is only as effective as structured analysis.

Still, the lemmings that stampeded down the structured path had little choice. There was nowhere else to run! Now, many survivors have joined

Editors: Alan Davis
University of Colorado
1867 Austin Bluffs Pkwy., Suite 200
Colorado Springs, CO 80933-7150
Internet adavis@vivaldi.uccs.edu

Winston Royce
TRW
1 Federal Systems Park Dr.
STC 7165U
Fairfax, VA 22030
(703) 803-5025
fax: (703) 803-5108

new herds heading down various other paths.

OBJECT PATH. In the late 1980s and early 1990s, object-oriented techniques were sold as a solution to the software industry's cost and customer-satisfaction problems. First it was object-oriented programming, then object-oriented design, and now object-oriented analysis. Now, "object-oriented" has become synonymous with "good," and we are barraged with this adjective.

Object-oriented programming is based on very sound principles of quality programming discovered 20 to 30 years ago: data abstraction, information hiding, encapsulation, inheritance, and so on. It is great to see these proven principles become popular. By the year 2000, they will be so infused into our culture that we'll no longer say "object-oriented programming;" we'll just say "programming."

Object-oriented design (most schools of it, anyway) is based on the same solid foundations. Object-oriented designs exhibit a trait Michael Jackson preached 20 years ago: A design that fails to mimic the real world's structure is not just bad, it is wrong!

So what's at the end of the paths marked OOP and OOD? Improved maintainability and reliability, and probably reduced development costs. These are certainly sufficient reasons to stay on this track. But don't believe the road signs that promise dramatic increases in productivity and incredible success at reuse. Both might be true, but then again they might not be! Most who have chosen this path have unrealistic expectations.

Object-oriented analysis is another story. Keeping in mind the goal of the requirements phase — a specification — let's ask ourselves if we can achieve that by performing OOA. The answer is no — no OOA technique helps you achieve that goal. Even when we augment objects with state and behavioral descriptions, the problem we encountered with structured analysis remains: How can we ascertain the black-box behavior of the overall system by examining the individual behavior of a set of objects?

The claim that OOA is great because the transition to OOD is so easy is almost insane. No engineering discipline claims to have techniques that make transitioning from requirements to design easy. That very transition is 90 percent of what engineering is!

Claims of increased maintainability and reliability from object-oriented techniques make little sense in the context of requirements. The unproved (though possible) claims of dramatic improvements in productivity and reuse from using object-oriented programming and design techniques are less justified at requirements time.

AT THIS YEAR'S ICSE, I GOT THE IMPRESSION THAT EVERYONE HAS AT LEAST ONE FOOT IN THE PROCESS-MATURITY STAMPEDE.

PROCESS-MATURITY PATH. I got the impression at this year's International Conference on Software Engineering that everybody has at least one foot in this stampede. The goal should be to consistently produce high-quality (reliable, maintainable, usable...) software that satisfies customer and user needs within schedule and on budget. This calls for engineers with the right background, training, and skills; managers with the right background, training, and skills; the ability to select the correct development process for the project at hand; and enough resources (time, money, tools...) to get the job done. Many are following the process-maturity-model path, trying to achieve this goal.

Such models emphasize the importance of a repeatable, measurable development process. These are certainly two very important aspects of achieving the goal, but they are definitely not sufficient alone, and they may not even be necessary. With the right people, you can succeed without process repeatability or measurability. With the wrong people, you can't succeed even *with* process repeatability and measurability.

For example, hospitals routinely collect data on the success rates of surgeons, to compare and contrast physicians. However, good surgeons are good even without data. And extremely difficult operations yield very poor data regardless of how good the physicians are. Furthermore, having hard data doesn't help the surgeons get better; it simply enables the medical profession to more accurately inform the patient of the prognosis.

At this point in the industry's development, it is more important to select an appropriate process model for each project than to expect a tailored version of a generic process model to work for all projects in an organization. In short, the process-maturity path is only the first leg of a very long journey on the path to quality software, but it is just that and nothing more.

C LANGUAGE PATH. Here is another incredibly popular path. And a very dangerous one, because it leads us away from proven quality-instilling programming practices toward hacking.

That is not to say it's impossible to produce quality programs in C. Certainly it is possible. But programmers who prefer C are probably more interested in producing software fast. They'd rather not take the time to avoid error-prone constructs; they don't mind playing around at the bit or pointer level. (Programmers who prefer Ada, on the other hand, tend to avoid tricks in favor of producing error-free, fail-safe software.)

Why is the C path so crowded? The reason is simple: Unix has become a de facto standard across many hardware platforms, so writing code in C gives companies maximum flexibility in response to evolving hardware.

There is no real reason to leave this path right now, but don't deceive yourself as to your reasons for taking it. It is for market share; it is for portability; it is for short-term revenue (all very good reasons). It is not to produce a quality product; it is not for long-term market penetration; it is not for long-term profit.

PROTOTYPING PATH. Software prototyping started to become popular in the mid to late 1980s. As originally conceived, prototypes provide an early version of a system to a user, helping to uncover things like necessary or unnecessary functions and effective or ineffective interfaces. Because so many software systems built today fail to solve the users' problems, prototyping helps ensure that requirements are known before we build a full-scale system.

Prototypes are helpful only if they are built quickly, so a plethora of techniques and tools appeared to help us rapidly build prototypes. However, by 1990 the quick-and-dirty

TABLE 1
LEMMING TRAILS

Trail	Active years	Goals	Time to achieve consistent results	Risk	Payoff
Structured	1965-1986	Reduce development cost	1975	Low	Low
		Increase documentation quality	1975	Low	Low
		Increase user satisfaction	1975	Low	Low
Object	1980-2000 (OOD and OOP)	Reduce development cost	1990	Low	Medium
		Increase reliability	1990	Low	Medium
		Increase maintainability	1990	Low	Medium to High
	1988-2000 (OOA)	Reduce development cost	1993	High	Low
		Improve user satisfaction	1993	High	Low
Process maturity	1990-2005	Improve process	1993	Low	Medium
		Increase user satisfaction	1993	High	Low
		Increase quality	1993	High	High
		Reduce development cost	1993	High	Medium
C	1980-2000+	Reduce development cost	1980	Medium	Medium
		Increase portability	1985	Low	High
		Increase quality	Never	High	None
		Increase maintainability	Never	High	None
Prototyping	1985-2000+	Increase user satisfaction	1993	Low	High
		Reduce development cost (by delivering throwaway prototypes to customers)	Never	High	None
CASE	1988-2000+	Moderately increase productivity	1993	Low	Medium
		Dramatically increase productivity	2013+	High	High
		Increase documentation quality	1993	Low	Medium
		Increase quality	1993	Low	Low
Reuse	1988 - ∞	Reduce development cost	1996–2003	High	Application-dependent
		Increase quality	1996–2003	High	Application-dependent
Comquats	1988 - ∞	Reduce development cost	1996-2003	High	Application-dependent
		Increase quality	1996-2003	High	Application-dependent

prototype was also seen as a solution to escalating software costs and slipping schedules. The logic was, "If we can produce software prototypes quickly, then we can deliver all software quickly simply by calling the prototype a product."

Think about it. If we are ineffective at producing quality software when we try, how awful will our products be if we don't try? If we don't know how to build in quality at reasonable cost, how can we expect to build in quality at negligible cost? And we certainly don't know how to retrofit quality into a prototype.

In typical lemmingineering fashion, we have taken a great idea and misapplied it. Prototyping is a great way to help ensure user satisfaction and thus reduce cost. It is a terrible way to reduce development costs by eliminating all the techniques we know ensure quality.

The lemmings on the prototyping path have suddenly veered toward a cliff.

CASE PATH. Originally, most CASE tools were graphical editors that had rudimentary syntax-checking capability. If you are using structured analysis (or any other graphical technique) intelligently, CASE tools offer considerable improvement in productivity but only superficial improvement in quality.

CASE tools help a software engineer in the same way a word processor helps an author. A word processor does not make a poor novelist a good one, but it will make every author more efficient and their material more grammatical. A CASE tool does not make a poor engineer a good one, but it will make every engineer more efficient and their product prettier.

Don't get me wrong. CASE tools have considerable value, as do word processors. Unfortunately, because it is such a competitive market, CASE tools are being sold not for their primary value, but for a whole variety of other features: automatic code generation, automatic prototype generation, automatic test generation. In evaluating CASE technology, let's try to keep in mind that there's no such thing as a free lunch.

The lemmings are (and should be) stampeding on this path, but, once again, many are not aware that what lies at the end of the path has been oversold.

REUSE PATH. All engineering disciplines encourage synthesis through the use of building blocks. Home builders use prefabricated doors and windows; bridge builders use prefabricated steel beams; circuit-board makers use off-the-shelf integrated circuits;

Continued on p. 84

process than the broad inquiry in *Whelan*, and its immediate effect is to eliminate a good deal of expression from copyright protection. In essence, *Altai* gave larger scope to the generally accepted merger doctrine.

In another case, *Brown Bag Software* v. *Symantec*, the Ninth Circuit Court of Appeals used the abstraction-filtration-comparison process to dismiss a claim that Symantec had infringed Brown Bag's outlining program. Expert evidence had been presented to show that numerous specific features of the two programs were substantially similar. Nevertheless, after the court dissected the Brown Bag program and applied the merger doctrine, it determined that much of the program consisted of "concepts fundamental to computer programs" such as the need to access files, edit work, and print — none of which was copyright-protected.

In the celebrated case of *Apple Computer, Inc.* v. *Microsoft, Inc.*, a US Federal District Court rejected the claim by Apple that Microsoft's Windows and Hewlett-Packard's New Wave infringed the graphical user interface of the MacIntosh computer. The court relied heavily on the merger doctrine in rejecting Apple's claim that the overall "look and feel" of its user interface was protected expression. Such familiar devices as the use of windows, icons, menus, and the opening and closing of on-screen objects were found to be examples of expression merged with idea, protected only from identical copying. The court rejected the claim that the look and feel of the program was protected, finding that such an imprecise and indefinite standard would allow Apple to sweep within its proprietary embrace screen windows and other user interfaces that used such standard features. In the court's view, this was not supported by copyright law or desirable as a matter of public policy.

MIXED EMOTIONS. What effect is the *Altai* process likely to have on future computer copyright cases? Clearly, it will be more difficult to prove program infringement. More reliance will be placed on expert witnesses for abstraction, filtration, and comparison of each program element. The decision may prove to be a disincentive to programming research and development because it will be riskier to invest time and money in a new program that may be subject to shrinking copyright protection. On the other hand, the decision may spur new creativity by encouraging others to develop programs with less fear that a broad programming field has been "locked up" by another.

As a result of this uncertainty, programmers may increasingly look to other methods of legal protection such as patent or trade secret. The need for distinction between idea and expression is well-rooted in the law of intellectual property and will likely remain so. The policy-level contest pits reward for developers against encouragement of new development — though arguably the former produces the latter. The last word has not yet been uttered; in the meantime, developers should consult their attorneys regarding program protection while we wait for further litigation to clarify the issue. ◆

MANAGER

SOFTWARE LEMMINGS

Continued from page 81

IC makers use standard cells. In these disciplines, this practice is called "use" or "engineering," never "reuse." Only in our discipline is "reuse" such a buzzword.

Clearly, if we could use prefabricated software components (that are larger than statements) in producing new systems, development costs and schedules would be reduced and product quality should be enhanced. We are now busily populating repositories of potentially reusable components and constructing new components for systems with an eye toward their eventual reuse. An amazing amount of effort is being expended on such an immature technology.

Do we really know what a prefabricated, reusable component should look like? Of course, the marketers of repositories will say yes. Clearly, we will have to experiment with many components and many repositories before we learn what such components should look like and how best to store, retrieve, and compose them. I fully endorse vast experimentation by researchers and practitioners alike. However, the technology is still very new.

Unlike many other paths, the goals here are quite realistic and will eventually be achieved. But the path is not yet paved. Step carefully and don't be overconfident in the short term.

COMQUATS [SIC] PATH. Superimposed on our efforts to reduce cost and increase quality is our repeated disappointment with the degree of user satisfaction we have achieved with custom-built software systems. One branch of the reuse path is the commercial-off-the-shelf path. COTS is the ultimate in reuse. We take an existing, complete "system" and add custom software to build a new application, thus greatly reducing costs.

The resulting system may not be perfect, but then neither are fully custom systems. Increased quality (in terms of satisfying user needs) is fully dependent on the degree to which the COTS software is suitable and how easy it is to customize. Increased quality (in terms of reliability, safety, and availability) is dependent on the quality in the COTS software, hence the name Comquat, or commercial off-the-shelf quality software.

LOOK BEFORE YOU LEAP. Just because everybody's doing it doesn't make it right. On the other hand, just because everybody's doing it doesn't make it wrong. In conclusion, I offer this advice.

- Set realistic goals and be realistic about the likelihood of success.
- Don't believe the hype. Determine for yourself if a path makes sense for you and your organization. Pick a path after clearly understanding its potential risks and rewards, in both degree and probability.
- Be cautious, but don't ignore every path. You can't afford to.
- Don't forget your goal — to solve user needs with a reliable, maintainable, usable, safe system within schedule and budget. Your goal is not to "reuse maximally" or "use object-oriented analysis."
- When you achieve your goal after 17 steps of a 25-step recipe, stop!
- Whatever you do, do not follow any path just because everybody's doing it. ◆