On one hand, the software industry has made great strides since its inception 50 years ago. Programming languages are more sophisticated. Our development processes are more mature. The applications we are constructing are more complex. The problems that we tackle increase in complexity, size, criticality, and range every year.

On the other hand, when compared to any other mature discipline (e.g., medicine, building construction, any engineering discipline except so called “software engineering”), we find major differences in the degree of satisfaction by intended users. These other disciplines seem to produce solution after solution that work as expected. And although the software industry is tackling increasingly more complex problems, it seems to create more useless systems than useful solutions. The data from many studies supports this observation.

This section explores various aspects of the software industry as a whole. The first essay, Software Lemmings, chastises the professionals within our industry for acting just like lemmings. Rather than making sound decisions, we regularly take a course that “everybody” is taking. After all, there is safety in numbers. In the case of giant herds of wildebeest marching across the plains of Africa, an individual animal’s safety is increased because a lion attack will devour the weakest members of the herd. Our safety in numbers is a bit different. When software professionals do the same thing that everybody else is doing, they cannot be blamed for failure; all that has to be said is “but everybody is doing it!” On the other hand, if a software professional chooses to do something unique, he or she can be blamed. This latter argument is further explored in the next essay, Recovering from Method Abuse.

In Tomorrow’s Blacksmiths, I take a much larger look at the future of the software industry. It asks, “Is software really the future?” Or are we so blinded by our own local achievements that we are failing to see a bigger solution?

Politics and religion have always attracted a wide range of disciples ranging from fundamentalists to moderates to liberals. And each religion and each country has experienced power shifts among these various players. Thus for example, in the early 2000’s, Iraq was controlled by the power of Islamic fundamentalists, United States Jewry by liberal (reform) Judaism, and the South African federal government by liberalism. In the next essay, On Software Development Strategies, Politics, and Religion,
I explore the roles of conservative and liberal movements in the history of the software industry, with emphasis on power shifts.

The next essay, *Art or Engineering, One More Time*, places software development on a scale measured by engineering on one extreme and the visual arts on the other. The verdict: somewhere in between, in a position similar to home construction.

Many naïve individuals in the software industry develop metrics to measure this or that aspect of a software product or the process used to develop it. The essay, *Why Build Software?*, reminds us all that such metrics are of little significance relative to the “ultimate measure,” i.e., that of whether the software solves the problem for which it was intended.

Yogi Berra said, *It Feels Like Déjà vu All Over Again*. And this brief essay describes how we could conclude that our industry is making great strides if you wear rose colored glasses, and how we could conclude quite the opposite opinion if we remove those glasses.

The essay titled *Eras of Software Technology Transfer* describes how empiricism replaced rational thought in the late 1980’s, and how rationalism is slowly creeping back into vogue.

*Fifteen Principles of Software Engineering* is just what it says it is. I explore the 15 most important principles that software practitioners should follow.

I admit it. I have become discouraged by the way our industry uses software cost schedule estimation technology. I think it is great that folks like Barry Boehm, Capers Jones, and Larry Putnam have collected and analyzed reams of data about past projects. But I think that what people have done with this data tells only half the story. The essay called *Thoughts on Software Estimation* explores my seven biggest frustrations with the practices of software managers as they attempt to determine how long it will take to complete a software project.
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.

*Originally published as “Software Lemmingineering” in IEEE Software, 10, 5 (September 1993), pp. 79–81, 84.
### TABLE 1.1 Lemming Trails

<table>
<thead>
<tr>
<th>Trail</th>
<th>Active years</th>
<th>Goals</th>
<th>Time to achieve consistent results</th>
<th>Risk</th>
<th>Payoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured</td>
<td>1965–1986</td>
<td>Reduce development cost</td>
<td>1975</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase documentation quality</td>
<td>1975</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase user satisfaction</td>
<td>1975</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Object</td>
<td>1980–2000 (OOD and</td>
<td>Reduce development cost</td>
<td>1990</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td>OOP)</td>
<td>Increase reliability</td>
<td>1990</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase maintainability</td>
<td>1990</td>
<td>Low</td>
<td>Medium to High</td>
</tr>
<tr>
<td></td>
<td>1988–2000 (OOA)</td>
<td>Reduce development cost</td>
<td>1993</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Improve user satisfaction</td>
<td>1993</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>maturity</td>
<td></td>
<td>Increase user satisfaction</td>
<td>1993</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase quality</td>
<td>1993</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce development cost</td>
<td>1988</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase portability</td>
<td>1985</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase quality</td>
<td>Never</td>
<td>High</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase maintainability</td>
<td>Never</td>
<td>High</td>
<td>None</td>
</tr>
<tr>
<td>C</td>
<td>1980–2000+</td>
<td>Increase user satisfaction</td>
<td>1993</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce development cost</td>
<td>1980</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Improve user satisfaction</td>
<td>1985</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase quality</td>
<td>Never</td>
<td>High</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase maintainability</td>
<td>Never</td>
<td>High</td>
<td>None</td>
</tr>
<tr>
<td>Prototyping</td>
<td>1985–2000+</td>
<td>Increase user satisfaction</td>
<td>1993</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce development cost (by delivering throwaway prototypes to customers)</td>
<td>Never</td>
<td>High</td>
<td>None</td>
</tr>
<tr>
<td>CASE</td>
<td>1988–2000+</td>
<td>Moderately increase productivity</td>
<td>1993</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dramatically increase productivity</td>
<td>2013+</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase documentation quality</td>
<td>1993</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase quality</td>
<td>1993</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase quality</td>
<td>1996–2003</td>
<td>High</td>
<td>Application-dependent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase quality</td>
<td>1996–2003</td>
<td>High</td>
<td>Application-dependent</td>
</tr>
</tbody>
</table>
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 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 maintain-
ability 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.

**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 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; 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.

EPILOGUE

Since I wrote this article, a few of the fads I discussed have disappeared; software reuse and COTS have now taken on their proper roles: as useful techniques that no longer are treated with so many hullabaloos. In the mean time, the industry’s love affair with dot-com’s has come and gone, along with a significant redistribution of wealth. And new fads have arrived to take the places of those now gone: xml, .net, C#, Java, the search for a single notation to represent all concepts, and so on. These will all absorb huge amounts of energy and money, will make some contribution for
future generations (but nothing in proportion to the energy expended to perpetrate their use), and will then gently disappear.

SEEDS FOR DEBATE

1. Select any of the items described in this paper as a fad, and present a case for why it is not so. That is, argue for why the fundamental bases of that item are so critical to the future of software, that it will remain forever.

2. Develop and document an argument that the fads I discuss in this essay have been misrepresented. In actuality, only the words to describe those concepts behave like fads; the concepts themselves live forever.

3. Select some software concept that is popular today (i.e., when you are reading this book, not when I am writing it). Present an argument for why it is but a fad. Also present the opposite argument: that it is a major step in evolution and its concepts and practices will endure.