Innovative Technology for Computing Professionals
Computer

Essence and Accidents of Software Engineering

Computer Classics

No Silver Bullet: Essence and Accidents of Software Engineering

by Frederick P. Brooks, Jr.

Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.

The familiar software project, at least as seen by the nontechnical manager, has something of this character; it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet—something to make software costs drop as rapidly as computer hardware costs do.

But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity. In this article, I shall try to show why, by examining both the nature of the software problem and the properties of the bullets proposed.

Skepticism is not pessimism, however. Although we see no startling breakthroughs—and indeed, I believe such to be inconsistent with the nature of software—many encouraging innovations are under way. A disciplined, consistent effort to develop, propagate, and exploit these innovations should indeed yield an order-of-magnitude improvement. There is no royal road, but there is a road.

The first step toward the management of disease was replacement of demon theories and humours theories by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions. It told workers that progress would be made stepwise, at great effort, and that a persistent, unremitting care would have to be paid to a discipline of cleanliness. So it is with software engineering today.

Does It Have to Be Hard?—Essential Difficulties

Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any—no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware. We cannot expect ever to see twofold gains every two years.

First, one must observe that the anomaly is not that software progress is so slow, but that computer hardware progress is so fast. No other technology since civilization began has seen six orders of magnitude in performance price gain in 30 years. In no other technology can one choose to take the gain in *either* improved performance *or* in reduced costs. These gains flow from the transformation of computer manufacture from an assembly industry into a process industry.

Second, to see what rate of progress one can expect in software technology, let us examine the difficulties of that technology. Following Aristotle, I divide them into *essence*, the difficulties inherent in the nature of software, and *accidents*, those difficulties that today attend its production but are not inherent.

The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed.

I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems.

If this is true, building software will always be hard. There is inherently no silver bullet.

Let us consider the inherent properties of this irreducible essence of modern software systems: complexity, conformity, changeability, and invisibility.

Complexity. Software entities are more complex for their size than perhaps any other human construct because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into a subroutine—open or closed. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.

Digital computers are themselves more complex than most things people build: They have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders-of-magnitude more states than computers do.

Likewise, a scaling-up of a software entity is not merely a repetition of the same elements in larger sizes, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly.

The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence. For three centuries, mathematics and the physical sciences made great strides by constructing simplified models of complex phenomena, deriving properties from the models, and verifying those properties by experiment. This paradigm worked because the complexities ignored in the models were not the essential properties of the phenomena. It does not work when the complexities are the essence.

Many of the classic problems of developing software products derive from this essential complexity and its nonlinear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, schedule delays. From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability. From complexity of function comes the difficulty of invoking function, which makes programs hard to use. From complexity of structure comes the difficulty of extending programs to new functions without creating side effects. From complexity of structure come the unvisualized states that constitute security trapdoors.

Not only technical problems, but management problems as well come from the complexity. It makes overview hard, thus impeding conceptual integrity. It makes it hard to find and control all the loose ends. It creates the tremendous learning and understanding burden that makes personnel turnover a disaster.

Conformity. Software people are not alone in facing complexity. Physics deals with terribly complex objects even at the "fundamental" particle level. The physicist labors on, however, in a firm faith that there are unifying principles to be found, whether in quarks or in unifiedfield theories. Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary.

No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which his interfaces must conform. These differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people, rather than by God.

In many cases, the software must conform because it is the most recent arrival on the scene. In others, it must conform because it is perceived as the most conformable. But in all cases, much complexity comes from conformation to other interfaces; this complexity

cannot be simplified out by any redesign of the software alone.

Changeability. The software entity is constantly subject to pressures for change. Of course, so are buildings, cars, computers. But manufactured things are infrequently changed after manufacture; they are superseded by later models, or essential changes are incorporated into later-serial-number copies of the same basic design. Call-backs of automobiles are really quite infrequent; field changes of computers somewhat less so. Both are much less frequent than modifications to fielded software.

In part, this is so because the software of a system embodies its function, and the function is the part that most feels the pressures of change. In part it is because software can be changed more easily—it is pure thought-stuff, infinitely malleable. Buildings do in fact get changed, but the high costs of change, understood by all, serve to dampen the whims of the changers.

All successful software gets changed. Two processes are at work. First, as a software product is found to be useful, people try it in new cases at the edge of or beyond the original domain. The pressures for extended function come chiefly from users who like the basic function and invent new uses *for* it.

Second, successful software survives beyond the normal life of the machine vehicle for which it is first written. If not new computers, then at least new disks, new displays, new printers come along; and the software must be conformed to its new vehicles of opportunity.

In short, the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product.

Invisibility. Software is invisible and unvisualizable. Geometric abstractions are powerful tools. The floor plan of a building helps both architect and client evaluate spaces, traffic flows, views. Contradictions and omissions become obvious. Scale drawings of mechanical parts and stick-figure models of molecules, although abstractions, serve the same purpose. A geometric reality is captured in a geometric abstraction.

The reality of software is not inherently embedded in space. Hence, it has no ready geometric representation in the way that land has maps, silicon chips have diagrams, computers have connectivity schematics. As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs superimposed one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. These graphs are usually not even planar, much less hierarchical. Indeed, one of the ways of establishing conceptual control over such structure is to enforce link cutting

until one or more of the graphs becomes hierarchical.[1]

In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication among minds.

Past Breakthroughs Solved Accidental Difficulties

If we examine the three steps in software technology development that have been most fruitful in the past, we discover that each attacked a different major difficulty in building software, but that those difficulties have been accidental, not essential, difficulties. We can also see the natural limits to the extrapolation of each such attack.

High-level languages. Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility.

What does a high-level language accomplish? It frees a program from much of its accidental complexity. An abstract program consists of conceptual constructs: operations, data types, sequences, and communication. The concrete machine program is concerned with bits, registers, conditions, branches, channels, disks, and such. To the extent that the high-level language embodies the constructs one wants in the abstract program and avoids all lower ones, it eliminates a whole level of complexity that was never inherent in the program at all.

The most a high-level language can do is to furnish all the constructs that the programmer imagines in the abstract program. To be sure, the level of our thinking about data structures, data types, and operations is steadily rising, but at an ever decreasing rate. And language development approaches closer and closer to the sophistication of users.

Moreover, at some point the elaboration of a high-level language creates a tool-mastery burden that increases, not reduces, the intellectual task of the user who rarely uses the esoteric constructs.

Time-sharing. Time-sharing brought a major improvement in the productivity of programmers and in the quality of their product, although not so large as that brought by high-level languages.

Time-sharing attacks a quite different difficulty. Time-sharing preserves immediacy, and hence enables one to maintain an overview of complexity. The slow turnaround of

batch programming means that one inevitably forgets the minutiae, if not the very thrust, of what one was thinking when he stopped programming and called for compilation and execution. This interruption is costly in time, for one must refresh one's memory. The most serious effect may well be the decay of the grasp of all that is going on in a complex system.

Slow turnaround, like machine-language complexities, is an accidental rather than an essential difficulty of the software process. The limits of the potential contribution of time-sharing derive directly. The principal effect of timesharing is to shorten system response time. As this response time goes to zero, at some point it passes the human threshold of noticeability, about 100 milliseconds. Beyond that threshold, no benefits are to be expected.

Unified programming environments. Unix and Interlisp, the first integrated programming environments to come into widespread use, seem to have improved productivity by integral factors. Why?

They attack the accidental difficulties that result from using individual programs together, by providing integrated libraries, unified file formats, and pipes and filters. As a result, conceptual structures that in principle could always call, feed, and use one another can indeed easily do so in practice.

This breakthrough in turn stimulated the development of whole toolbenches, since each new tool could be applied to any programs that used the standard formats.

Because of these successes, environments are the subject of much of today's software-engineering research. We look at their promise and limitations in the next section.

Hopes for the Silver

Now let us consider the technical developments that are most often advanced as potential silver bullets. What problems do they address—the problems of essence, or the remaining accidental difficulties? Do they offer revolutionary advances, or incremental ones?

Ada and other high-level language advances. One of the most touted recent developments is Ada, a general-purpose high-level language of the 1980's. Ada not only reflects evolutionary improvements in language concepts, but indeed embodies features to encourage modern design and modularization. Perhaps the Ada philosophy is more of an advance than the Ada language, for it is the philosophy of modularization, of abstract data types, of hierarchical structuring. Ada is over-rich, a natural result of the process by which requirements were laid on its design. That is not fatal, for subsetted working vocabularies can solve the learning problem, and hardware advances will give us the

cheap MIPS to pay for the compiling costs. Advancing the structuring of software systems is indeed a very good use for the increased MIPS our dollars will buy. Operating systems, loudly decried in the 1960's for their memory and cycle costs, have proved to be an excellent form in which to use some of the MIPS and cheap memory bytes of the past hardware surge.

Nevertheless, Ada will not prove to be the silver bullet that slays the software productivity monster. It is, after all, just another high-level language, and the biggest payoff from such languages came from the first transition — the transition up from the accidental complexities of the machine into the more abstract statement of step-by-step solutions. Once those accidents have been removed, the remaining ones will be smaller, and the payoff from their removal will surely be less.

I predict that a decade from now, when the effectiveness of Ada is assessed, it will be seen to have made a substantial difference, but not because of any particular language feature, nor indeed because of all of them combined. Neither will the new Ada environments prove to be the cause of the improvements. Ada's greatest contribution will be that switching to it occasioned training programmers in modern software-design techniques.

Object-oriented programming. Many students of the art hold out more hope for object-oriented programming than for any of the other technical fads of the day.[2] I am among them. Mark Sherman of Dartmouth notes on CSnet News that one must be careful to distinguish two separate ideas that go under that name: *abstract data types* and *hierarchical types*. The concept of the abstract data type is that an object's type should be defined by a name, a set of proper values, and a set of proper operations rather than by its storage structure, which should be hidden. Examples are Ada packages (with private types) and Modula's modules.

Hierarchical types, such as Simula-67's classes, allow one to define general interfaces that can be further refined by providing subordinate types. The two concepts are orthogonal_one may have hierarchies without hiding and hiding without hierarchies. Both concepts represent real advances in the art of building software.

Each removes yet another accidental difficulty from the process, allowing the designer to express the essence of the design without having to express large amounts of syntactic material that add no information content. For both abstract types and hierarchical types, the result is to remove a higher-order kind of accidental difficulty and allow a higher-order expression of design.

Nevertheless, such advances can do no more than to remove all the accidental difficulties from the expression of the design. The complexity of the design itself is essential, and such attacks make no change whatever in that. An order-of-magnitude

gain can be made by object-oriented programming only if the unnecessary type-specification underbrush still in our programming language is itself nine-tenths of the work involved in designing a program product. I doubt it.

Artificial intelligence. Many people expect advances in artificial intelligence to provide the revolutionary breakthrough that will give order-of-magnitude gains in software productivity and quality.[3] I do not. To see why, we must dissect what is meant by "artificial intelligence."

D.L. Parnas has clarified the terminological chaos:[4]

> Two quite different definitions of AI are in common use today. AI-1: The use of computers to solve problems that previously could only be solved by applying human intelligence. Al-2: The use of a specific set of programming techniques known as heuristic or rule-based programming. In this approach human experts are studied to determine what heuristics or rules of thumb they use in solving problems.... The program is designed to solve a problem the way that humans seem to solve it.
>
> The first definition has a sliding meaning.... Something can fit the definition of Al-1 today but, once we see how the program works and understand the problem, we will not think of it as Al any more.... Unfortunately I cannot identify a body of technology that is unique to this field.... Most of the work is problem-specific, and some abstraction or creativity is required to see how to transfer it.

I agree completely with this critique. The techniques used for speech recognition seem to have little in common with those used for image recognition, and both are different from those used in expert systems. I have a hard time seeing how image recognition, for example, will make any appreciable difference in programming practice. The same problem is true of speech recognition. The hard thing about building software is deciding what one wants to say, not saying it. No facilitation of expression can give more than marginal gains.

Expert-systems technology, AI-2, deserves a section of its own.

Expert systems. The most advanced part of the artificial intelligence art, and the most widely applied, is the technology for building expert systems. Many software scientists are hard at work applying this technology to the software-building environment.[3,5] What is the concept, and what are the prospects?

An *expert system* is a program that contains a generalized inference engine and a rule base, takes input data and assumptions, explores the inferences derivable from the rule base, yields conclusions and advice, and offers to explain its results by retracing its

reasoning for the user. The inference engines typically can deal with fuzzy or probabilistic data and rules, in addition to purely deterministic logic.

Such systems offer some clear advantages over programmed algorithms designed for arriving at the same solutions to the same problems:

- Inference-engine technology is developed in an application-independent way, and then applied to many uses. One can justify much effort on the inference engines. Indeed, that technology is well advanced.
- The changeable parts of the application-peculiar materials are encoded in the rule base in a uniform fashion, and tools are provided for developing, changing, testing, and documenting the rule base. This regularizes much of the complexity of the application itself.

The power of such systems does not come from ever-fancier inference mechanisms but rather from ever-richer knowledge bases that reflect the real world more accurately. I believe that the most important advance offered by the technology is the separation of the application complexity from the program itself.

How can this technology be applied to the software-engineering task? In many ways: Such systems can suggest interface rules, advise on testing strategies, remember bug-type frequencies, and offer optimization hints.

Consider an imaginary testing advisor, for example. In its most rudimentary form, the diagnostic expert system is very like a pilot's checklist, just enumerating suggestions as to possible causes of difficulty. As more and more system structure is embodied in the rule base, and as the rule base takes more sophisticated account of the trouble symptoms reported, the testing advisor becomes more and more particular in the hypotheses it generates and the tests it recommends. Such an expert system may depart most radically from the conventional ones in that its rule base should probably be hierarchically modularized in the same way the corresponding software product is, so that as the product is modularly modified, the diagnostic rule base can be modularly modified as well.

The work required to generate the diagnostic rules is work that would have to be done anyway in generating the set of test cases for the modules and for the system. If it is done in a suitably general manner, with both a uniform structure for rules and a good inference engine available, it may actually reduce the total labor of generating bring-up test cases, and help as well with lifelong maintenance and modification testing. In the same way, one can postulate other advisors, probably many and probably simple, for the other parts of the software-construction task.

Many difficulties stand in the way of the early realization of useful expert-system advisors to the program developer. A crucial part of our imaginary scenario is the

development of easy ways to get from program-structure specification to the automatic or semiautomatic generation of diagnostic rules. Even more difficult and important is the twofold ,task of knowledge acquisition: finding articulate, self-analytical experts who know why they do things, and developing efficient techniques for extracting what they know and distilling it into rule bases. The essential prerequisite for building an expert system is to have an expert.

The most powerful contribution by expert systems will surely be to put at the service of the inexperienced programmer the experience and accumulated wisdom of the best programmers. This is no small contribution. The gap between the best software engineering practice and the average practice is very wide_perhaps wider than in any other engineering discipline. A tool that disseminates good practice would be important.

"Automatic" programming. For almost 40 years, people have been anticipating and writing about "automatic programming," or the generation of a program for solving a problem from a statement of the problem specifications. Some today write as if they expect this technology to provide the next breakthrough.[5]

Parnas[4] implies that the term is used for glamour, not for semantic content, asserting,

> In short, automatic programming always has been a euphemism for programming with a higher-level language than was presently available to the programmer.

He argues, in essence, that in most cases it is the solution method, not the problem, whose specification has to be given.

One can find exceptions. The technique of building generators is very powerful, and it is routinely used to good advantage in programs for sorting. Some systems for integrating differential equations have also permitted direct specification of the problem, and the systems have assessed the parameters, chosen from a library of methods of solution, and generated the programs.

These applications have very favorable properties:

- The problems are readily characterized by relatively few parameters.
- There are many known methods of solution to provide a library of alternatives.
- Extensive analysis has led to explicit rules for selecting solution techniques, given problem parameters.

It is hard to see how such techniques generalize to the wider world of the ordinary software system, where cases with such neat properties are the exception. It is hard even to imagine how this breakthrough in generalization could occur.

Graphical programming. A favorite subject for PhD dissertations in software

engineering is graphical, or visual, programming—the application of computer graphics to software design.[6,7] Sometimes the promise held out by such an approach is postulated by analogy with VLSI chip design, in which computer graphics plays so fruitful a role. Sometimes the theorist justifies the approach by considering flowcharts as the ideal program-design medium and by providing powerful facilities for constructing them.

Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will.

In the first place, as I have argued elsewhere[8], the flowchart is a very poor abstraction of software structure. Indeed, it is best viewed as Burks, von Neumann, and Goldstine's attempt to provide a desperately needed high-level control language for their proposed computer. In the pitiful, multipage, connection-boxed form to which the flowchart has today been elaborated, it has proved to be useless as a design tool—programmers draw flowcharts after, not before, writing the programs they describe.

Second, the screens of today are too small, in pixels, to show both the scope and the resolution of any seriously detailed software diagram. The so-called "desktop metaphor" of today's workstation is instead an "airplane-seat" metaphor. Anyone who has shuffled a lap full of papers while seated between two portly passengers will recognize the difference—one can see only a very few things at once. The true desktop provides overview of, and random access to, a score of pages. Moreover, when fits of creativity run strong, more than one programmer or writer has been known to abandon the desktop for the more spacious floor. The hardware technology will have to advance quite substantially before the scope of our scopes is sufficient for the software-design task.

More fundamentally, as I have argued above, software is very difficult to visualize. Whether one diagrams control flow, variable-scope nesting, variable cross references, dataflow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant. If one superimposes all the diagrams generated by the many relevant views, it is difficult to extract any global overview. The VLSI analogy is fundamentally misleading—a chip design is a layered two-dimensional description whose geometry reflects its realization in 3-space. A software system is not.

Program verification. Much of the effort in modern programming goes into testing and the repair of bugs. Is there perhaps a silver bullet to be found by eliminating the errors at the source, in the system-design phase? Can both productivity and product reliability be radically enhanced by following the profoundly different strategy of proving designs correct before the immense effort is poured into implementing and testing them?

I do not believe we will find productivity magic here. Program verification is a very

powerful concept, and it will be very important for such things as secure operating-system kernels. The technology does not promise, however, to save labor. Verifications are so much work that only a few substantial programs have ever been verified.

Program verification does not mean error-proof programs. There is no magic here, either. Mathematical proofs also can be faulty. So whereas verification might reduce the program-testing load, it cannot eliminate it.

More seriously, even perfect program verification can only establish that a program meets its specification. The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification.

Environments and tools. How much more gain can be expected from the exploding researches into better programming environments? One's instinctive reaction is that the big-payoff problems— hierarchical file systems, uniform file formats to make possible uniform program interfaces, and generalized tools—were the first attacked, and have been solved. Language-specific smart editors are developments not yet widely used in practice, but the most they promise is freedom from syntactic errors and simple semantic errors.

Perhaps the biggest gain yet to be realized from programming environments is the use of integrated database systems to keep track of the myriad details that must be recalled accurately by the individual programmer and kept current for a group of collaborators on a single system.

Surely this work is worthwhile, and surely it will bear some fruit in both productivity and reliability. But by its very nature, the return from now on must be marginal.

Workstations. What gains are to be expected for the software art from the certain and rapid increase in the power and memory capacity of the individual workstation? Well, how many MIPS can one use fruitfully? The composition and editing of programs and documents is fully supported by today's speeds. Compiling could stand a boost, but a factor of 10 in machine speed would surely leave thinktime the dominant activity in the programmer's day. Indeed, it appears to be so now.

More powerful workstations we surely welcome. Magical enhancements from them we cannot expect.

Promising Attacks on the Conceptual Essence

Even though no technological breakthrough promises to give the sort of magical results with which we are so familiar in the hardware area, there is both an abundance of good

work going on now, and the promise of steady, if unspectacular progress.

All of the technological attacks on the accidents of the software process are fundamentally limited by the productivity equation:

$$\text{time of task} = \sum_i (\text{frequency})_i \times (\text{time})_i$$

If, as I believe, the conceptual components of the task are now taking most of the time, then no amount of activity on the task components that are merely the expression of the concepts can give large productivity gains.

Hence we must consider those attacks that address the essence of the software problem, the formulation of these complex conceptual structures. Fortunately, some of these attacks are very promising.

Buy versus build. The most radical possible solution for constructing software is not to construct it at all.

Every day this becomes easier, as more and more vendors offer more and better software products for a dizzying variety of applications. While we software engineers have labored on production methodology, the personal-computer revolution has created not one, but many, mass markets for software. Every newsstand carries monthly magazines, which sorted by machine type, advertise and review dozens of products at prices from a few dollars to a few hundred dollars. More specialized sources offer very powerful products for the workstation and other Unix markets. Even software tools and environments can be bought off-the-shelf. I have elsewhere proposed a marketplace for individual modules. [9]

Any such product is cheaper to buy than to build afresh. Even at a cost of one hundred thousand dollars, a purchased piece of software is costing only about as much as one programmeryear. And delivery is immediate! Immediate at least for products that really exist, products whose developer can refer products to a happy user. Moreover, such products tend to be much better documented and somewhat better maintained than home-grown software.

The development of the mass market is, I believe, the most profound long-run trend in software engineering. The cost of software has always been development cost, not replication cost. Sharing that cost among even a few users radically cuts the per-user cost. Another way of looking at it is that the use of n copies of a software system effectively multiplies the productivity of its developers by n. That is an enhancement of the productivity of the discipline and of the nation.

The key issue, of course, is applicability. Can I use an available off-the-shelf package to perform my task? A surprising thing has happened here. During the 1950's and 1960's,

study after study showed that users would not use off-the-shelf packages for payroll, inventory control, accounts receivable, and so on. The requirements were too specialized, the case-to-case variation too high. During the 1980's, we find such packages in high demand and widespread use. What has changed?

Not the packages, really. They may be somewhat more generalized and somewhat more customizable than formerly, but not much. Not the applications, either. If anything, the business and scientific needs of today are more diverse and complicated than those of 20 years ago.

The big change has been in the hardware/software cost ratio. In 1960, the buyer of a two-million dollar machine felt that he could afford $250,000 more for a customized payroll program, one that slipped easily and nondisruptively into the computer-hostile social environment. Today, the buyer of a $50,000 office machine cannot conceivably afford a customized payroll program, so he adapts the payroll procedure to the packages available. Computers are now so commonplace, if not yet so beloved, that the adaptations are accepted as a matter of course.

There are dramatic exceptions to my argument that the generalization of software packages has changed little over the years: electronic spreadsheets and simple database systems. These powerful tools, so obvious in retrospect and yet so late in appearing, lend themselves to myriad uses, some quite unorthodox. Articles and even books now abound on how to tackle unexpected tasks with the spreadsheet. Large numbers of applications that would formerly have been written as custom programs in Cobol or Report Program Generator are now routinely done with these tools.

Many users now operate their own computers day in and day out on various applications without ever writing a program. Indeed, many of these users cannot write new programs for their machines, but they are nevertheless adept at solving new problems with them.

I believe the single most powerful software-productivity strategy for many organizations today is to equip the computer-naive intellectual workers who are on the firing line with personal computers and good generalized writing, drawing, file, and spreadsheet programs and then to turn them loose. The same strategy, carried out with generalized mathematical and statistical packages and some simple programming capabilities, will also work for hundreds of laboratory scientists.

Requirements refinement and rapid prototyping. The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Therefore, the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements. For the truth is, the client does not know what he wants. The client usually does not know what questions must be answered, and he has almost never thought of the problem in the detail necessary for specification. Even the simple answer_"Make the new software system work like our old manual information-processing system"_is in fact too simple. One never wants exactly that. Complex software systems are, moreover, things that act, that move, that work. The dynamics of that action are hard to imagine. So in planning any software-design activity, it is necessary to allow for an extensive iteration between the client and the designer as part of the system definition.

I would go a step further and assert that it is really impossible for a client, even working with a software engineer, to specify completely, precisely, and correctly the exact requirements of a modern software product before trying some versions of the product.

Therefore, one of the most promising of the current technological efforts, and one that attacks the essence, not the accidents, of the software problem, is the development of approaches and tools for rapid prototyping of systems as prototyping is part of the iterative specification of requirements.

A *prototype software system* is one that simulates the important interfaces and performs the main functions of the intended system, while not necessarily being bound by the same hardware speed, size, or cost constraints. Prototypes typically perform the mainline tasks of the application, but make no attempt to handle the exceptional tasks, respond correctly to invalid inputs, or abort cleanly. The purpose of the prototype is to make real the conceptual structure specified, so that the client can test it for consistency and usability.

Much of present-day software-acquisition procedure rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software-acquisition problems spring from that fallacy. Hence, they cannot be fixed without fundamental revision—revision that provides for iterative development and specification of prototypes and products.

Incremental development—grow, don't build, software. I still remember the jolt I felt in 1958 when I first heard a friend talk about *building* a program, as opposed to *writing* one. In a flash he broadened my whole view of the software process. The metaphor shift was powerful, and accurate. Today we understand how like other building processes the construction of software is, and we freely use other elements of the metaphor, such as *specifications*, *assembly of components*, and *scaffolding*.

The building metaphor has outlived its usefulness. It is time to change again. If, as I

believe, the conceptual structures we construct today are too complicated to be specified accurately in advance, and too complex to be built faultlessly, then we must take a radically different approach.

Let us turn nature and study complexity in living things, instead of just the dead works of man. Here we find constructs whose complexities thrill us with awe. The brain alone is intricate beyond mapping, powerful beyond imitation, rich in diversity, self-protecting, and selfrenewing. The secret is that it is grown, not built.

So it must be with our software-systems. Some years ago Harlan Mills proposed that any software system should be grown by incremental development. [10] That is, the system should first be made to run, even if it does nothing useful except call the proper set of dummy subprograms. Then, bit by bit, it should be fleshed out, with the subprograms in turn being developed—into actions or calls to empty stubs in the level below.

I have seen most dramatic results since I began urging this technique on the project builders in my Software Engineering Laboratory class. Nothing in the past decade has so radically changed my own practice, or its effectiveness. The approach necessitates top-down design, for it is a top-down growing of the software. It allows easy backtracking. It lends itself to early prototypes. Each added function and new provision for more complex data or circumstances grows organically out of what is already there.

The morale effects are startling. Enthusiasm jumps when there is a running system, even a simple one. Efforts redouble when the first picture from a new graphics software system appears on the screen, even if it is only a rectangle. One always has, at every stage in the process, a working system. I find that teams can *grow* much more complex entities in four months than they can *build*.

The same benefits can be realized on large projects as on my small ones.[11]

Great designers. The central question in how to improve the software art centers, as it always has, on people.

We can get good designs by following good practices instead of poor ones. Good design practices can be taught. Programmers are among the most intelligent part of the population, so they can learn good practice. Hence, a major thrust in the United States is to promulgate good modern practice. New curricula, new literature, new organizations such as the Software Engineering Institute, all have come into being in order to raise the level of our practice from poor to good. This is entirely proper.

Nevertheless, I do not believe we can make the next step upward in the same way. Whereas the difference between poor conceptual designs and good ones may lie in the soundness of design method, the difference between good designs and great ones surely does not. Great designs come from great designers. Software construction is a *creative*

process. Sound methodology can empower and liberate the creative mind; it cannot inflame or inspire the drudge.

The differences are not minor—they are rather like the differences between Salieri and Mozart. Study after study shows that the very best designers produce structures that are faster, smaller, simpler, cleaner, and produced with less effort.[1,2] The differences between the great and the average approach an order of magnitude.

A little retrospection shows that although many fine, useful software systems have been designed by committees and built as part of multipart projects, those software systems that have excited passionate fans are those that are the products of one or a few designing minds, great designers. Consider Unix, APL, Pascal, Modula, the Smalltalk interface, even Fortran; and contrast them with Cobol, PL/I, Algol, MVS/370, and MS-DOS.

Table 1. Exciting vs. useful but unexciting software products

Exciting Products	
Yes	**No**
Unix	Cobol
APL	PL/1
Pascal	Algol
Modula	MVS/370
Across chip	MS-DOS
Smalltalk	
Fortran	

Hence, although I strongly support the technology-transfer and curriculumdevelopment efforts now under way, I think the most important single effort we can mount is to develop ways to grow great designers.

No software organization can ignore this challenge. Good managers, scarce though they be, are no scarcer than good designers. Great designers and great managers are both very rare. Most organizations spend considerable effort in finding and cultivating the management prospects; I know of none that spends equal effort in finding and developing the great designers upon whom the technical excellence of the products will ultimately depend.

My first proposal is that each software organization must determine and proclaim that great designers are as important to its success as great managers are, and that they can be expected to be similarly nurtured and rewarded. Not only salary, but the perquisites of recognition—office size, furnishings, personal technical equipment, travel funds, staff support—must be fully equivalent.

How to grow great designers? Space does not permit a lengthy discussion, but some steps

are obvious:

- Systematically identify top designers as early as possible. The best are often not the most experienced.
- Assign a career mentor to be responsible for the development of the prospect, and carefully keep a career file.
- Devise and maintain a careerdevelopment plan for each prospect, including carefully selected apprenticeships with top designers, episodes of advanced formal education, and short courses, all interspersed with solo-design and technicalleadership assignments.
- Provide opportunities for growing designers to interact with and stimulate each other.

Acknowledgments

I thank Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, Robert Patrick, and, most especially, David Parnas for their insights and stimulating ideas, and Rebekah Bierly for the technical production of this article.

References

1. D.L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, Vol. 5, No. 2, March 1979, pp. 128-38.
2. G. Booch, "Object-Oriented Design," *Software Engineering with Ada*, 1983, Menlo Park, Calif.: Benjamin/ Cummings.
3. *lEEE Transactions on Software Engineering* (special issue on artificial intelligence and software engineering), l. Mostow, guest ed., Vol. 11, No. 11, November 1985.
4. D.L. Parnas, "Software Aspects of Strategic Defense Systems:" *American Scientist*, November 1985.
5. R. Balzer, "A 15-year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering* (special issue on artificial intelligence and software engineering), J. Mostow, guest ed., Vol. 11, No. 11 (November 1985), pp. 1257-67.
6. *Computer* (special issue on visual programrning), R.B. Graphton and T. Ichikawa, guest eds., Vol. 18, No. 8, August 1985.
7. G. Raeder, "A Survey of Current Graphical Programming Techniques," *Computer* (special issue on visual programming), R.B. Graphton and T. Ichikawa, guest eds., Vol. 18, No. 8, August 1985, pp. 11-25.
8. F.P. Brooks, *The Mythical Man Month*, Reading, Mass.: Addison-Wesley, 1975, Chapter 14.
9. Defense Science Board, *Report of the Task Force on Military Software* in press.

10. H.D. Mills, "Top-Down Programming in Large Systems," *in Debugging Techniques in Large Systems*, R. Ruskin, ed., Englewood Cliffs, N.J.: Prentice-Hall, 1971.
11. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," 1985, TRW Technical Report 21-371-85, TRW, Inc., 1 Space Park, Redondo Beach, Calif. 90278.
12. H. Sackman, W.J. Erikson, and E.E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM*, Vol. 11, No. 1 (January 1968), pp. 3-11.

From the April 1987 issue of *Computer.*

The Cathedral and the Bazaar

Eric Steven Raymond$Date: 2000/08/24 22:37:44 $

Revision History

Revision 1.5124 August 2000 Revised by: esr
First DocBook version. Minor updates to Fall 2000 on the time-sensitive material.
Revision 1.495 May 2000 Revised by: esr
Added the HBS note on deadlines and scheduling.
Revision 1.5131 August 1999 Revised by: esr
This the version that O'Reilly printed in the first edition of the book.
Revision 1.458 August 1999 Revised by: esr
Added the endnotes on the Snafu Principle, (pre)historical examples of bazaar development, and originality in the bazaar.
Revision 1.4429 July 1999 Revised by: esr
Added the "On Management and the Maginot Line" section, some insights about the usefulness of bazaars for exploring
Revision 1.4020 Nov 1998 Revised by: esr
Added a correction of Brooks based on the Halloween Documents.
Revision 1.3928 July 1998 Revised by: esr
I removed Paul Eggert's 'graph on GPL vs. bazaar in response to cogent aguments from RMS on
Revision 1.31February 10 1998 Revised by: esr
Added "Epilog: Netscape Embraces the Bazaar!"
Revision 1.29February 9 1998 Revised by: esr
Changed "free software" to "open source".
Revision 1.2718 November 1997Revised by: esr
Added the Perl Conference anecdote.
Revision 1.207 July 1997 Revised by: esr
Added the bibliography.
Revision 1.1621 May 1997 Revised by: esr
First official presentation at the Linux Kongress.

I anatomize a successful open-source project, fetchmail, that was run as a deliberate test of some surprising theories about software engineering suggested by the history of Linux. I discuss these theories in terms of two fundamentally different development styles, the "cathedral" model of most of the commercial world versus the "bazaar" model of the Linux world. I show that these models derive from opposing assumptions about the nature of the software-debugging task. I then make a sustained argument from the Linux experience for the proposition that "Given enough eyeballs, all bugs are shallow", suggest productive analogies with other self-correcting systems of selfish agents, and conclude with some exploration of the implications of this insight for the future of software.

1. The Cathedral and the Bazaar

Linux is subversive. Who would have thought even five years ago (1991) that a world-class operating system could coalesce as if by magic out of part-time hacking by several thousand developers scattered all over the planet, connected only by the tenuous strands of the Internet?

Certainly not I. By the time Linux swam onto my radar screen in early 1993, I had already been involved in Unix and open-source development for ten years. I was one of the first GNU contributors in the mid-1980s. I had released a good deal of open-source software onto the net, developing or co-developing several programs (nethack, Emacs's VC and GUD modes, xlife, and others) that are still in wide use today. I thought I knew how it was done.

Linux overturned much of what I thought I knew. I had been preaching the Unix gospel of small tools, rapid prototyping and evolutionary programming for years. But I also believed there was a certain critical complexity above which a more centralized, a priori approach was required. I believed that the most important software (operating systems and really large tools like the Emacs programming editor) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time.

Linus Torvalds's style of development - release early and often, delegate everything you can, be open to the point of promiscuity - came as a surprise. No quiet, reverent cathedral-building here – rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who'd take submissions from *anyone*) out of which a coherent and stable system could seemingly emerge only by a succession of miracles.

The fact that this bazaar style seemed to work, and work well, came as a distinct shock. As I learned my way around, I worked hard not just at individual projects, but also at trying to understand why the Linux world not only didn't fly apart in confusion but seemed to go from strength to strength at a speed barely imaginable to cathedral-builders.

By mid-1996 I thought I was beginning to understand. Chance handed me a perfect way to test my theory, in the form of an open-source project that I could consciously try to run in the bazaar style. So I did – and it was a significant success.

This is the story of that project. I'll use it to propose some aphorisms about effective open-source development. Not all of these are things I first learned in the Linux world, but we'll see how the Linux world gives them particular point. If I'm correct, they'll help you understand exactly what it is that makes the Linux community such a fountain of good software – and, perhaps, they will help you become more productive yourself.

2. The Mail Must Get Through

Since 1993 I'd been running the technical side of a small free-access Internet service provider called Chester County InterLink (CCIL) in West Chester, Pennsylvania. I co-founded CCIL and wrote our unique multiuser bulletin-board software – you can check it out by telnetting to locke.ccil.org (telnet://locke.ccil.org). Today it supports almost three thousand users on thirty lines. The job allowed me 24-hour-a-day access to the net through CCIL's 56K line – in fact, the job practically demanded it!

I had gotten quite used to instant Internet email. I found having to periodically telnet over to locke to check my mail annoying. What I wanted was for my mail to be delivered on snark (my home system) so that I would be notified when it arrived and could handle it using all my local tools.

The Internet's native mail forwarding protocol, SMTP (Simple Mail Transfer Protocol), wouldn't suit, because it works best when machines are connected full-time, while my personal machine isn't always on the net, and doesn't have a static IP address. What I needed was a program that would reach out over my intermittent dialup connection and pull across my mail to be delivered locally. I knew such things existed, and that most of them used a simple application protocol called POP (Post Office Protocol). POP is now widely supported by most common mail clients, but at the time, it wasn't built-in to the mail reader I was using.

I needed a POP3 client. So I went out on the net and found one. Actually, I found three or four. I used one of them for a while, but it was missing what seemed an obvious feature, the ability to hack the addresses on fetched mail so replies would work properly.

The problem was this: suppose someone named 'joe' on locke sent me mail. If I fetched the mail to snark and then tried to reply to it, my mailer would cheerfully try to ship it to a nonexistent 'joe' on snark.

Hand-editing reply addresses to tack on '@ccil.org' quickly got to be a serious pain.

This was clearly something the computer ought to be doing for me. But none of the existing POP clients knew how! And this brings us to the first lesson:

1. Every good work of software starts by scratching a developer's personal itch.

Perhaps this should have been obvious (it's long been proverbial that "Necessity is the mother of invention") but too often software developers spend their days grinding away for pay at programs they neither need nor love. But not in the Linux world – which may explain why the average quality of software originated in the Linux community is so high.

So, did I immediately launch into a furious whirl of coding up a brand-new POP3 client to compete with the existing ones? Not on your life! I looked carefully at the POP utilities I had in hand, asking myself "which one is closest to what I want?" Because

2. Good programmers know what to write. Great ones know what to rewrite (and reuse).

While I don't claim to be a great programmer, I try to imitate one. An important trait of the great ones is constructive laziness. They know that you get an A not for effort but for results, and that it's almost always easier to start from a good partial solution than from nothing at all.

Linus Torvalds (http://www.tuxedo.org/~esr/faqs/linus), for example, didn't actually try to write Linux from scratch. Instead, he started by reusing code and ideas from Minix, a tiny Unix-like operating system for PC clones. Eventually all the Minix code went away or was completely rewritten – but while it was there, it provided scaffolding for the infant that would eventually become Linux.

In the same spirit, I went looking for an existing POP utility that was reasonably well coded, to use as a development base.

The source-sharing tradition of the Unix world has always been friendly to code reuse (this is why the GNU project chose Unix as a base OS, in spite of serious reservations about the OS itself). The Linux world has taken this tradition nearly to its technological limit; it has terabytes of open sources generally available. So spending time looking for some else's almost-good-enough is more likely to give you good results in the Linux world than anywhere else.

And it did for me. With those I'd found earlier, my second search made up a total of nine candidates – fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail and upop. The one I first settled on was 'fetchpop' by Seung-Hong Oh. I put my header-rewrite feature in it, and made various other improvements which the author accepted into his 1.9 release.

A few weeks later, though, I stumbled across the code for 'popclient' by Carl Harris, and found I had a problem. Though fetchpop had some good original ideas in it (such as its background-daemon mode), it could only handle POP3 and was rather amateurishly coded (Seung-Hong was at that time a bright but inexperienced programmer, and both traits showed). Carl's code was better, quite professional and solid, but his program lacked several important and rather tricky-to-implement fetchpop features (including those I'd coded myself).

Stay or switch? If I switched, I'd be throwing away the coding I'd already done in exchange for a better development base.

A practical motive to switch was the presence of multiple-protocol support. POP3 is the most commonly used of the post-office server protocols, but not the only one. Fetchpop and the other competition didn't do POP2, RPOP, or APOP, and I was already having vague thoughts of perhaps adding IMAP (http://www.imap.org) (Internet Message Access Protocol, the most recently designed and most powerful post-office protocol) just for fun.

But I had a more theoretical reason to think switching might be as good an idea as well, something I learned long before Linux.

3. "Plan to throw one away; you will, anyhow." (Fred Brooks, "The Mythical Man-Month", Chapter 11)

Or, to put it another way, you often don't really understand the problem until after the first time you implement a solution. The second time, maybe you know enough to do it right. So if you want to get it right, be ready to start over *at least* once [JB].

Well (I told myself) the changes to fetchpop had been my first try. So I switched.

After I sent my first set of popclient patches to Carl Harris on 25 June 1996, I found out that he had basically lost interest in popclient some time before. The code was a bit dusty, with minor bugs hanging out. I had many changes to make, and we quickly agreed that the logical thing for me to do was take over the program.

Without my actually noticing, the project had escalated. No longer was I just contemplating minor patches to an existing POP client. I took on maintaining an entire one, and there were ideas bubbling in my head that I knew would probably lead to major changes.

In a software culture that encourages code-sharing, this is a natural way for a project to evolve. I was acting out this principle:

4. If you have the right attitude, interesting problems will find you.

But Carl Harris's attitude was even more important. He understood that

5. When you lose interest in a program, your last duty to it is to hand it off to a competent successor.

Without ever having to discuss it, Carl and I knew we had a common goal of having the best solution out there. The only question for either of us was whether I could establish that I was a safe pair of hands. Once I did that, he acted with grace and dispatch. I hope I will do as well when it comes my turn.

3. The Importance of Having Users

And so I inherited popclient. Just as importantly, I inherited popclient's user base. Users are wonderful things to have, and not just because they demonstrate that you're serving a need, that you've done something right. Properly cultivated, they can become co-developers.

Another strength of the Unix tradition, one that Linux pushes to a happy extreme, is that a lot of users are hackers too. Because source code is available, they can be *effective* hackers. This can be tremendously useful for shortening debugging time. Given a bit of encouragement, your users will diagnose problems, suggest fixes, and help improve the code far more quickly than you could unaided.

6. Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.

The power of this effect is easy to underestimate. In fact, pretty well all of us in the open-source world drastically underestimated how well it would scale up with number of users and against system

complexity, until Linus Torvalds showed us differently.

In fact, I think Linus's cleverest and most consequential hack was not the construction of the Linux kernel itself, but rather his invention of the Linux development model. When I expressed this opinion in his presence once, he smiled and quietly repeated something he has often said: "I'm basically a very lazy person who likes to get credit for things other people actually do." Lazy like a fox. Or, as Robert Heinlein famously wrote of one of his characters, too lazy to fail.

In retrospect, one precedent for the methods and success of Linux can be seen in the development of the GNU Emacs Lisp library and Lisp code archives. In contrast to the cathedral-building style of the Emacs C core and most other GNU tools, the evolution of the Lisp code pool was fluid and very user-driven. Ideas and prototype modes were often rewritten three or four times before reaching a stable final form. And loosely-coupled collaborations enabled by the Internet, a la Linux, were frequent.

Indeed, my own most successful single hack previous to fetchmail was probably Emacs VC (version control) mode, a Linux-like collaboration by email with three other people, only one of whom (Richard Stallman, the author of Emacs and founder of the Free Software Foundation (http://www.fsf.org)) I have met to this day. It was a front-end for SCCS, RCS and later CVS from within Emacs that offered "one-touch" version control operations. It evolved from a tiny, crude sccs.el mode somebody else had written. And the development of VC succeeded because, unlike Emacs itself, Emacs Lisp code could go through release/test/improve generations very quickly.

4. Release Early, Release Often

Early and frequent releases are a critical part of the Linux development model. Most developers (including me) used to believe this was bad policy for larger than trivial projects, because early versions are almost by definition buggy versions and you don't want to wear out the patience of your users.

This belief reinforced the general commitment to a cathedral-building style of development. If the overriding objective was for users to see as few bugs as possible, why then you'd only release a version every six months (or less often), and work like a dog on debugging between releases. The Emacs C core was developed this way. The Lisp library, in effect, was not – because there were active Lisp archives outside the FSF's control, where you could go to find new and development code versions independently of Emacs's release cycle [QR].

The most important of these, the Ohio State elisp archive, anticipated the spirit and many of the features of today's big Linux archives. But few of us really thought very hard about what we were doing, or about what the very existence of that archive suggested about problems in the FSF's cathedral-building development model. I made one serious attempt around 1992 to get a lot of the Ohio code formally merged into the official Emacs Lisp library. I ran into political trouble and was largely unsuccessful.

But by a year later, as Linux became widely visible, it was clear that something different and much healthier was going on there. Linus's open development policy was the very opposite of cathedral-building. Linux's Internet archives were burgeoning, multiple distributions were being floated. And all of this was driven by an unheard-of frequency of core system releases.

Linus was treating his users as co-developers in the most effective possible way:

7. Release early. Release often. And listen to your customers.

Linus's innovation wasn't so much in doing quick-turnaround releases incorporating lots of user feedback (something like this had been Unix-world tradition for a long time), but in scaling it up to a level of intensity that matched the complexity of what he was developing. In those early times (around 1991) it wasn't unknown for him to release a new kernel more than once a *day!* Because he cultivated his base of co-developers and leveraged the Internet for collaboration harder than anyone else, this worked.

But *how* did it work? And was it something I could duplicate, or did it rely on some unique genius of Linus Torvalds?

I didn't think so. Granted, Linus is a damn fine hacker. How many of us could engineer an entire production-quality operating system kernel from scratch?. But Linux didn't represent any awesome conceptual leap forward. Linus is not (or at least, not yet) an innovative genius of design in the way that, say, Richard Stallman or James Gosling (of NeWS and Java) are. Rather, Linus seems to me to be a genius of engineering and implementation, with a sixth sense for avoiding bugs and development dead-ends and a true knack for finding the minimum-effort path from point A to point B. Indeed, the whole design of Linux breathes this quality and mirrors Linus's essentially conservative and simplifying design approach.

So, if rapid releases and leveraging the Internet medium to the hilt were not accidents but integral parts of Linus's engineering-genius insight into the minimum-effort path, what was he maximizing? What was he cranking out of the machinery?

Put that way, the question answers itself. Linus was keeping his hacker/users constantly stimulated and rewarded – stimulated by the prospect of having an ego-satisfying piece of the action, rewarded by the sight of constant (even *daily*) improvement in their work.

Linus was directly aiming to maximize the number of person-hours thrown at debugging and development, even at the possible cost of instability in the code and user-base burnout if any serious bug proved intractable. Linus was behaving as though he believed something like this:

> 8. Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.

Or, less formally, "Given enough eyeballs, all bugs are shallow." I dub this: "Linus's Law".

My original formulation was that every problem "will be transparent to somebody". Linus demurred that the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it. "Somebody finds the problem," he says, "and somebody *else* understands it. And I'll go on record as saying that finding it is the bigger challenge." But the point is that both things tend to happen rapidly.

Here, I think, is the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a dedicated few to develop confidence that you've winkled them all out. Thus the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect.

In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena – or, at least, that they turn shallow pretty quickly when exposed to a thousand eager co-developers pounding on every single new release. Accordingly you release often in order to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door.

And that's it. That's enough. If "Linus's Law" is false, then any system as complex as the Linux kernel, being hacked over by as many hands as the Linux kernel, should at some point have collapsed under the weight of unforseen bad interactions and undiscovered "deep" bugs. If it's true, on the other hand, it is sufficient to explain Linux's relative lack of bugginess and its continuous uptimes spanning months or even years.

Maybe it shouldn't have been such a surprise, at that. Sociologists years ago discovered that the averaged opinion of a mass of equally expert (or equally ignorant) observers is quite a bit more reliable a predictor than that of a single randomly-chosen one of the observers. They called this the "Delphi effect". It appears that what Linus has shown is that this applies even to debugging an operating system – that the Delphi effect can tame development complexity even at the complexity level of an OS kernel.

One special feature of the Linux situation that clearly helps along the Delphi effect is the fact that the contributors for any given project are self-selected. An early respondent pointed out that contributions are received not from a random sample, but from people who are interested enough to use the software, learn about how it works, attempt to find solutions to problems they encounter, and actually produce an apparently reasonable fix. Anyone who passes all these filters is highly likely to have something useful to contribute.

I am indebted to my friend Jeff Dutky <dutky@wam.umd.edu> for pointing out that Linus's Law can be rephrased as "Debugging is parallelizable". Jeff observes that although debugging requires debuggers to communicate with some coordinating developer, it doesn't require significant coordination between debuggers. Thus it doesn't fall prey to the same quadratic complexity and management costs that make adding developers problematic.

In practice, the theoretical loss of efficiency due to duplication of work by debuggers almost never seems to be an issue in the Linux world. One effect of a "release early and often policy" is to minimize such duplication by propagating fed-back fixes quickly [JH].

Brooks (the author of "The Mythical Man-Month") even made an off-hand observation related to Jeff's: "The total cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it. Surprisingly this cost is strongly affected by the number of users. *More users find more bugs*." (my emphasis).

More users find more bugs because adding more users adds more different ways of stressing the program. This effect is amplified when the users are co-developers. Each one approaches the task of bug characterization with a slightly different perceptual set and analytical toolkit, a different angle on the problem. The "Delphi effect" seems to work precisely because of this variation. In the specific context of debugging, the variation also tends to reduce duplication of effort.

So adding more beta-testers may not reduce the complexity of the current "deepest" bug from the *developer's* point of view, but it increases the probability that someone's toolkit will be matched to the problem in such a way that the bug is shallow *to that person*.

Linus coppers his bets, too. In case there *are* serious bugs, Linux kernel version are numbered in such a way that potential users can make a choice either to run the last version designated "stable" or to ride the cutting edge and risk bugs in order to get new features. This tactic is not yet formally imitated by most Linux hackers, but perhaps it should be; the fact that either choice is available makes both more attractive. [HBS]

5. When Is A Rose Not A Rose?

Having studied Linus's behavior and formed a theory about why it was successful, I made a conscious decision to test this theory on my new (admittedly much less complex and ambitious) project.

But the first thing I did was reorganize and simplify popclient a lot. Carl Harris's implementation was very sound, but exhibited a kind of unnecessary complexity common to many C programmers. He treated the code as central and the data structures as support for the code. As a result, the code was beautiful but the data structure design ad-hoc and rather ugly (at least by the high standards of this old LISP hacker).

I had another purpose for rewriting besides improving the code and the data structure design, however. That was to evolve it into something I understood completely. It's no fun to be responsible for fixing bugs in a program you don't understand.

For the first month or so, then, I was simply following out the implications of Carl's basic design. The first serious change I made was to add IMAP support. I did this by reorganizing the protocol machines into a generic driver and three method tables (for POP2, POP3, and IMAP). This and the previous changes illustrate a general principle that's good for programmers to keep in mind, especially in languages like C that don't naturally do dynamic typing:

> 9. Smart data structures and dumb code works a lot better than the other way around.

Brooks, Chapter 9: "Show me your [code] and conceal your [data structures], and I shall continue to be mystified. Show me your [data structures], and I won't usually need your [code]; it'll be obvious."

Actually, he said "flowcharts" and "tables". But allowing for thirty years of terminological/cultural shift, it's almost the same point.

At this point (early September 1996, about six weeks from zero) I started thinking that a name change might be in order – after all, it wasn't just a POP client any more. But I hesitated, because there was as yet nothing genuinely new in the design. My version of popclient had yet to develop an identity of its own.

That changed, radically, when fetchmail learned how to forward fetched mail to the SMTP port. I'll get to that in a moment. But first: I said above that I'd decided to use this project to test my theory about what Linus Torvalds had done right. How (you may well ask) did I do that? In these ways:

- I released early and often (almost never less often than every ten days; during periods of intense development, once a day).
- I grew my beta list by adding to it everyone who contacted me about fetchmail.
- I sent chatty announcements to the beta list whenever I released, encouraging people to participate.
- And I listened to my beta testers, polling them about design decisions and stroking them whenever they sent in patches and feedback.

The payoff from these simple measures was immediate. From the beginning of the project, I got bug reports of a quality most developers would kill for, often with good fixes attached. I got thoughtful criticism, I got fan mail, I got intelligent feature suggestions. Which leads to:

> 10. If you treat your beta-testers as if they're your most valuable resource, they will respond by becoming your most valuable resource.

One interesting measure of fetchmail's success is the sheer size of the project beta list, fetchmail-friends. At the time of last revision (August 2000) it has 249 members and is adding two or three a week.

Actually, as I revise in late May 1997 the list is beginning to lose members from its high of close to 300 for an interesting reason. Several people have asked me to unsubscribe them because fetchmail is working so well for them that they no longer need to see the list traffic! Perhaps this is part of the normal life-cycle of a mature bazaar-style project.

6. Popclient becomes Fetchmail

The real turning point in the project was when Harry Hochheiser sent me his scratch code for forwarding mail to the client machine's SMTP port. I realized almost immediately that a reliable implementation of this feature would make all the other mail delivery modes next to obsolete.

For many weeks I had been tweaking fetchmail rather incrementally while feeling like the interface design was serviceable but grubby – inelegant and with too many exiguous options hanging out all over. The options to dump fetched mail to a mailbox file or standard output particularly bothered me, but I couldn't figure out why.

(If you don't care about the technicalia of Internet mail, the next two paragraphs can be safely skipped.)

What I saw when I thought about SMTP forwarding was that popclient had been trying to do too many things. It had been designed to be both a mail transport agent (MTA) and a local delivery agent (MDA). With SMTP forwarding, it could get out of the MDA business and be a pure MTA, handing off mail to other programs for local delivery just as sendmail does.

Why mess with all the complexity of configuring a mail delivery agent or setting up lock-and-append on a mailbox when port 25 is almost guaranteed to be there on any platform with TCP/IP support in the first place? Especially when this means retrieved mail is guaranteed to look like normal sender-initiated SMTP mail, which is really what we want anyway.

(Back to a higher level...)

Even if you didn't follow the preceding technical jargon, there are several important lessons here. First, this SMTP-forwarding concept was the biggest single payoff I got from consciously trying to emulate Linus's methods. A user gave me this terrific idea – all I had to do was understand the implications.

11. The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is better.

Interestingly enough, you will quickly find that if you are completely and self-deprecatingly truthful about how much you owe other people, the world at large will treat you like you did every bit of the invention yourself and are just being becomingly modest about your innate genius. We can all see how well this worked for Linus!

(When I gave my talk at the Perl conference in August 1997, hacker extraordinaire Larry Wall was in the front row. As I got to the last line above he called out, religious-revival style, "Tell it, tell it, brother!". The whole audience laughed, because they knew this had worked for the inventor of Perl, too.)

After a very few weeks of running the project in the same spirit, I began to get similar praise not just from my users but from other people to whom the word leaked out. I stashed away some of that email; I'll look at it again sometime if I ever start wondering whether my life has been worthwhile :-).

But there are two more fundamental, non-political lessons here that are general to all kinds of design.

12. Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong.

I had been trying to solve the wrong problem by continuing to develop popclient as a combined MTA/MDA with all kinds of funky local delivery modes. Fetchmail's design needed to be rethought from the ground up as a pure MTA, a part of the normal SMTP-speaking Internet mail path.

When you hit a wall in development – when you find yourself hard put to think past the next patch – it's often time to ask not whether you've got the right answer, but whether you're asking the right question. Perhaps the problem needs to be reframed.

Well, I had reframed my problem. Clearly, the right thing to do was (1) hack SMTP forwarding support into the generic driver, (2) make it the default mode, and (3) eventually throw out all the other delivery modes, especially the deliver-to-file and deliver-to-standard-output options.

I hesitated over step 3 for some time, fearing to upset long-time popclient users dependent on the alternate delivery mechanisms. In theory, they could immediately switch to .forward files or their non-sendmail equivalents to get the same effects. In practice the transition might have been messy.

But when I did it, the benefits proved huge. The cruftiest parts of the driver code vanished. Configuration got radically simpler – no more grovelling around for the system MDA and user's mailbox, no more worries about whether the underlying OS supports file locking.

Also, the only way to lose mail vanished. If you specified delivery to a file and the disk got full, your mail got lost. This can't happen with SMTP forwarding because your SMTP listener won't return OK unless the message can be delivered or at least spooled for later delivery.

Also, performance improved (though not so you'd notice it in a single run). Another not insignificant benefit of this change was that the manual page got a lot simpler.

Later, I had to bring delivery via a user-specified local MDA back in order to allow handling of some obscure situations involving dynamic SLIP. But I found a much simpler way to do it.

The moral? Don't hesitate to throw away superannuated features when you can do it without loss of effectiveness. Antoine de Saint-Exupéry (who was an aviator and aircraft designer when he wasn't being the author of classic children's books) said:

> 13. "Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away."

When your code is getting both better and simpler, that is when you *know* it's right. And in the process, the fetchmail design acquired an identity of its own, different from the ancestral popclient.

It was time for the name change. The new design looked much more like a dual of sendmail than the old popclient had; both are MTAs, but where sendmail pushes then delivers, the new popclient pulls then delivers. So, two months off the blocks, I renamed it fetchmail.

There is a more general lesson in this story about how SMTP delivery came to fetchmail. It is not only debugging that is parallelizable; development and (to a perhaps surprising extent) exploration of design space is, too. When your development mode is rapidly iterative, development and enhancement may become special cases of debugging – fixing 'bugs of omission' in the original capabilities or concept of the software.

Even at a higher level of design, it can be very valuable to have the thinking of lots of co-developers random-walking through the design space near your product. Consider the way a puddle of water finds a drain, or better yet how ants find food: exploration essentially by diffusion, followed by exploitation mediated by a scalable communication mechanism. This works very well; as with Harry Hochheiser and me, one of your outriders may well find a huge win nearby that you were just a little too close-focused to see.

7. Fetchmail Grows Up

There I was with a neat and innovative design, code that I knew worked well because I used it every day, and a burgeoning beta list. It gradually dawned on me that I was no longer engaged in a trivial personal

hack that might happen to be useful to few other people. I had my hands on a program every hacker with a Unix box and a SLIP/PPP mail connection really needs.

With the SMTP forwarding feature, it pulled far enough in front of the competition to potentially become a "category killer", one of those classic programs that fills its niche so competently that the alternatives are not just discarded but almost forgotten.

I think you can't really aim or plan for a result like this. You have to get pulled into it by design ideas so powerful that afterward the results just seem inevitable, natural, even foreordained. The only way to try for ideas like that is by having lots of ideas – or by having the engineering judgment to take other peoples' good ideas beyond where the originators thought they could go.

Andy Tanenbaum had the original idea to build a simple native Unix for IBM PCs, for use as a teaching tool (he called it Minix). Linus Torvalds pushed the Minix concept further than Andrew probably thought it could go – and it grew into something wonderful. In the same way (though on a smaller scale), I took some ideas by Carl Harris and Harry Hochheiser and pushed them hard. Neither of us was 'original' in the romantic way people think is genius. But then, most science and engineering and software development isn't done by original genius, hacker mythology to the contrary.

The results were pretty heady stuff all the same – in fact, just the kind of success every hacker lives for! And they meant I would have to set my standards even higher. To make fetchmail as good as I now saw it could be, I'd have to write not just for my own needs, but also include and support features necessary to others but outside my orbit. And do that while keeping the program simple and robust.

The first and overwhelmingly most important feature I wrote after realizing this was multidrop support – the ability to fetch mail from mailboxes that had accumulated all mail for a group of users, and then route each piece of mail to its individual recipients.

I decided to add the multidrop support partly because some users were clamoring for it, but mostly because I thought it would shake bugs out of the single-drop code by forcing me to deal with addressing in full generality. And so it proved. Getting RFC 822 (http://info.internet.isi.edu:80/in-notes/rfc/files/rfc822.txt) address parsing right took me a remarkably long time, not because any individual piece of it is hard but because it involved a pile of interdependent and fussy details.

But multidrop addressing turned out to be an excellent design decision as well. Here's how I knew:

14. Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected.

The unexpected use for multi-drop fetchmail is to run mailing lists with the list kept, and alias expansion done, on the *client* side of the Internet connection. This means someone running a personal machine through an ISP account can manage a mailing list without continuing access to the ISP's alias files.

Another important change demanded by my beta testers was support for 8-bit MIME (Multipurpose Internet Mail Extensions) operation. This was pretty easy to do, because I had been careful to keep the code 8-bit clean. Not because I anticipated the demand for this feature, but rather in obedience to another rule:

15. When writing gateway software of any kind, take pains to disturb the data stream as little as possible – and *never* throw away information unless the recipient forces you to!

Had I not obeyed this rule, 8-bit MIME support would have been difficult and buggy. As it was, all I had to do is read the MIME standard (RFC 1652 (http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1652.txt)) and add a trivial bit of header-generation logic.

Some European users bugged me into adding an option to limit the number of messages retrieved per session (so they can control costs from their expensive phone networks). I resisted this for a long time, and I'm still not entirely happy about it. But if you're writing for the world, you have to listen to your customers – this doesn't change just because they're not paying you in money.

8. A Few More Lessons From Fetchmail

Before we go back to general software-engineering issues, there are a couple more specific lessons from the fetchmail experience to ponder. Nontechnical readers can safely skip this section.

The rc (control) file syntax includes optional 'noise' keywords that are entirely ignored by the parser. The English-like syntax they allow is considerably more readable than the traditional terse keyword-value pairs you get when you strip them all out.

These started out as a late-night experiment when I noticed how much the rc file declarations were beginning to resemble an imperative minilanguage. (This is also why I changed the original popclient

'server' keyword to 'poll').

It seemed to me that trying to make that imperative minilanguage more like English might make it easier to use. Now, although I'm a convinced partisan of the "make it a language" school of design as exemplified by Emacs and HTML and many database engines, I am not normally a big fan of "English-like" syntaxes.

Traditionally programmers have tended to favor control syntaxes that are very precise and compact and have no redundancy at all. This is a cultural legacy from when computing resources were expensive, so parsing stages had to be as cheap and simple as possible. English, with about 50% redundancy, looked like a very inappropriate model then.

This is not my reason for normally avoiding English-like syntaxes; I mention it here only to demolish it. With cheap cycles and core, terseness should not be an end in itself. Nowadays it's more important for a language to be convenient for humans than to be cheap for the computer.

There remain, however, good reasons to be wary. One is the complexity cost of the parsing stage – you don't want to raise that to the point where it's a significant source of bugs and user confusion in itself. Another is that trying to make a language syntax English-like often demands that the "English" it speaks be bent seriously out of shape, so much so that the superficial resemblance to natural language is as confusing as a traditional syntax would have been. (You see this bad effect in a lot of so-called "fourth generation" and commercial database-query languages.)

The fetchmail control syntax seems to avoid these problems because the language domain is extremely restricted. It's nowhere near a general-purpose language; the things it says simply are not very complicated, so there's little potential for confusion in moving mentally between a tiny subset of English and the actual control language. I think there may be a wider lesson here:

16. When your language is nowhere near Turing-complete, syntactic sugar can be your friend.

Another lesson is about security by obscurity. Some fetchmail users asked me to change the software to store passwords encrypted in the rc file, so snoopers wouldn't be able to casually see them.

I didn't do it, because this doesn't actually add protection. Anyone who's acquired permissions to read your rc file will be able to run fetchmail as you anyway – and if it's your password they're after, they'd be able to rip the necessary decoder out of the fetchmail code itself to get it.

All .fetchmailrc password encryption would have done is give a false sense of security to people who don't think very hard. The general rule here is:

17. A security system is only as secure as its secret. Beware of pseudo-secrets.

9. Necessary Preconditions for the Bazaar Style

Early reviewers and test audiences for this paper consistently raised questions about the preconditions for successful bazaar-style development, including both the qualifications of the project leader and the state of code at the time one goes public and starts to try to build a co-developer community.

It's fairly clear that one cannot code from the ground up in bazaar style [IN]. One can test, debug and improve in bazaar style, but it would be very hard to *originate* a project in bazaar mode. Linus didn't try it. I didn't either. Your nascent developer community needs to have something runnable and testable to play with.

When you start community-building, what you need to be able to present is a *plausible promise*. Your program doesn't have to work particularly well. It can be crude, buggy, incomplete, and poorly documented. What it must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future.

Linux and fetchmail both went public with strong, attractive basic designs. Many people thinking about the bazaar model as I have presented it have correctly considered this critical, then jumped from it to the conclusion that a high degree of design intuition and cleverness in the project leader is indispensable.

But Linus got his design from Unix. I got mine initially from the ancestral popclient (though it would later change a great deal, much more proportionately speaking than has Linux). So does the leader/coordinator for a bazaar-style effort really have to have exceptional design talent, or can he get by on leveraging the design talent of others?

I think it is not critical that the coordinator be able to originate designs of exceptional brilliance, but it is absolutely critical that the coordinator be able to *recognize good design ideas from others*.

Both the Linux and fetchmail projects show evidence of this. Linus, while not (as previously discussed) a spectacularly original designer, has displayed a powerful knack for recognizing good design and integrating it into the Linux kernel. And I have already described how the single most powerful design idea in fetchmail (SMTP forwarding) came from somebody else.

Early audiences of this paper complimented me by suggesting that I am prone to undervalue design originality in bazaar projects because I have a lot of it myself, and therefore take it for granted. There may be some truth to this; design (as opposed to coding or debugging) is certainly my strongest skill.

But the problem with being clever and original in software design is that it gets to be a habit – you start reflexively making things cute and complicated when you should be keeping them robust and simple. I have had projects crash on me because I made this mistake, but I managed not to with fetchmail.

So I believe the fetchmail project succeeded partly because I restrained my tendency to be clever; this argues (at least) against design originality being essential for successful bazaar projects. And consider Linux. Suppose Linus Torvalds had been trying to pull off fundamental innovations in operating system design during the development; does it seem at all likely that the resulting kernel would be as stable and successful as what we have?

A certain base level of design and coding skill is required, of course, but I expect almost anybody seriously thinking of launching a bazaar effort will already be above that minimum. The open-source community's internal market in reputation exerts subtle pressure on people not to launch development efforts they're not competent to follow through on. So far this seems to have worked pretty well.

There is another kind of skill not normally associated with software development which I think is as important as design cleverness to bazaar projects – and it may be more important. A bazaar project coordinator or leader must have good people and communications skills.

This should be obvious. In order to build a development community, you need to attract people, interest them in what you're doing, and keep them happy about the amount of work they're doing. Technical sizzle will go a long way towards accomplishing this, but it's far from the whole story. The personality you project matters, too.

It is not a coincidence that Linus is a nice guy who makes people like him and want to help him. It's not a coincidence that I'm an energetic extrovert who enjoys working a crowd and has some of the delivery and instincts of a stand-up comic. To make the bazaar model work, it helps enormously if you have at least a little skill at charming people.

10. The Social Context of Open-Source Software

It is truly written: the best hacks start out as personal solutions to the author's everyday problems, and spread because the problem turns out to be typical for a large class of users. This takes us back to the matter of rule 1, restated in a perhaps more useful way:

> 18. To solve an interesting problem, start by finding a problem that is interesting to you.

So it was with Carl Harris and the ancestral popclient, and so with me and fetchmail. But this has been understood for a long time. The interesting point, the point that the histories of Linux and fetchmail seem to demand we focus on, is the next stage – the evolution of software in the presence of a large and active community of users and co-developers.

In "The Mythical Man-Month", Fred Brooks observed that programmer time is not fungible; adding developers to a late software project makes it later. He argued that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly. This claim has since become known as "Brooks's Law" and is widely regarded as a truism. But if Brooks's Law were the whole picture, Linux would be impossible.

Gerald Weinberg's classic "The Psychology Of Computer Programming" supplied what, in hindsight, we can see as a vital correction to Brooks. In his discussion of "egoless programming", Weinberg observed that in shops where developers are not territorial about their code, and encourage other people to look for bugs and potential improvements in it, improvement happens dramatically faster than elsewhere.

Weinberg's choice of terminology has perhaps prevented his analysis from gaining the acceptance it deserved – one has to smile at the thought of describing Internet hackers as "egoless". But I think his argument looks more compelling today than ever.

The history of Unix should have prepared us for what we're learning from Linux (and what I've verified experimentally on a smaller scale by deliberately copying Linus's methods [EGCS]). That is, that while coding remains an essentially solitary activity, the really great hacks come from harnessing the attention and brainpower of entire communities. The developer who uses only his or her own brain in a closed project is going to fall behind the developer who knows how to create an open, evolutionary context in which feedback exploring the design space, code contributions, bug-spotting, and other improvements come back from hundreds (perhaps thousands) of people.

But the traditional Unix world was prevented from pushing this approach to the ultimate by several factors. One was the legal contraints of various licenses, trade secrets, and commercial interests. Another (in hindsight) was that the Internet wasn't yet good enough.

Before cheap Internet, there were some geographically compact communities where the culture encouraged Weinberg's "egoless" programming, and a developer could easily attract a lot of skilled kibitzers and co-developers. Bell Labs, the MIT AI Lab, UC Berkeley – these became the home of innovations that are legendary and still potent.

Linux was the first project to make a conscious and successful effort to use the entire *world* as its talent pool. I don't think it's a coincidence that the gestation period of Linux coincided with the birth of the World Wide Web, and that Linux left its infancy during the same period in 1993-1994 that saw the takeoff of the ISP industry and the explosion of mainstream interest in the Internet. Linus was the first person who learned how to play by the new rules that pervasive Internet access made possible.

While cheap Internet was a necessary condition for the Linux model to evolve, I think it was not by itself a sufficient condition. Another vital factor was the development of a leadership style and set of cooperative customs that could allow developers to attract co-developers and get maximum leverage out of the medium.

But what is this leadership style and what are these customs? They cannot be based on power relationships – and even if they could be, leadership by coercion would not produce the results we see. Weinberg quotes the autobiography of the 19th-century Russian anarchist Pyotr Alexeyvich Kropotkin's *Memoirs of a Revolutionist* to good effect on this subject:

> Having been brought up in a serf-owner's family, I entered active life, like all young men of my time, with a great deal of confidence in the necessity of commanding, ordering, scolding, punishing and the like. But when, at an early stage, I had to manage serious enterprises and to deal with [free] men, and when each mistake would lead at once to heavy consequences, I began to appreciate the difference between acting on the principle of command and discipline and acting on the principle of common understanding. The former works admirably in a military parade, but it is worth nothing where real life is concerned, and the aim can be achieved only through the severe effort of many converging wills.

The "severe effort of many converging wills" is precisely what a project like Linux requires – and the "principle of command" is effectively impossible to apply among volunteers in the anarchist's paradise we call the Internet. To operate and compete effectively, hackers who want to lead collaborative projects have to learn how to recruit and energize effective communities of interest in the mode vaguely

suggested by Kropotkin's "principle of understanding". They must learn to use Linus's Law.[SP]

Earlier I referred to the "Delphi effect" as a possible explanation for Linus's Law. But more powerful analogies to adaptive systems in biology and economics also irresistably suggest themselves. The Linux world behaves in many respects like a free market or an ecology, a collection of selfish agents attempting to maximize utility which in the process produces a self-correcting spontaneous order more elaborate and efficient than any amount of central planning could have achieved. Here, then, is the place to seek the "principle of understanding".

The "utility function" Linux hackers are maximizing is not classically economic, but is the intangible of their own ego satisfaction and reputation among other hackers. (One may call their motivation "altruistic", but this ignores the fact that altruism is itself a form of ego satisfaction for the altruist). Voluntary cultures that work this way are not actually uncommon; one other in which I have long participated is science fiction fandom, which unlike hackerdom has long explicitly recognized "egoboo" (ego-boosting, or the enhancement of one's reputation among other fans) as the basic drive behind volunteer activity.

Linus, by successfully positioning himself as the gatekeeper of a project in which the development is mostly done by others, and nurturing interest in the project until it became self-sustaining, has shown an acute grasp of Kropotkin's "principle of shared understanding". This quasi-economic view of the Linux world enables us to see how that understanding is applied.

We may view Linus's method as a way to create an efficient market in "egoboo" – to connect the selfishness of individual hackers as firmly as possible to difficult ends that can only be achieved by sustained cooperation. With the fetchmail project I have shown (albeit on a smaller scale) that his methods can be duplicated with good results. Perhaps I have even done it a bit more consciously and systematically than he.

Many people (especially those who politically distrust free markets) would expect a culture of self-directed egoists to be fragmented, territorial, wasteful, secretive, and hostile. But this expectation is clearly falsified by (to give just one example) the stunning variety, quality and depth of Linux documentation. It is a hallowed given that programmers *hate* documenting; how is it, then, that Linux hackers generate so much of it? Evidently Linux's free market in egoboo works better to produce virtuous, other-directed behavior than the massively-funded documentation shops of commercial software producers.

Both the fetchmail and Linux kernel projects show that by properly rewarding the egos of many other

hackers, a strong developer/coordinator can use the Internet to capture the benefits of having lots of co-developers without having a project collapse into a chaotic mess. So to Brooks's Law I counter-propose the following:

> 19: Provided the development coordinator has a medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.

I think the future of open-source software will increasingly belong to people who know how to play Linus's game, people who leave behind the cathedral and embrace the bazaar. This is not to say that individual vision and brilliance will no longer matter; rather, I think that the cutting edge of open-source software will belong to people who start from individual vision and brilliance, then amplify it through the effective construction of voluntary communities of interest.

Perhaps this is not only the future of *open-source* software. No closed-source developer can match the pool of talent the Linux community can bring to bear on a problem. Very few could afford even to hire the more than two hundred (1999: six hundred, 2000: eight hundred) people who have contributed to fetchmail!

Perhaps in the end the open-source culture will triumph not because cooperation is morally right or software "hoarding" is morally wrong (assuming you believe the latter, which neither Linus nor I do), but simply because the closed-source world cannot win an evolutionary arms race with open-source communities that can put orders of magnitude more skilled time into a problem.

11. On Management and the Maginot Line

The original "Cathedral and Bazaar" paper of 1997 ended with the vision above – that of happy networked hordes of programmer/anarchists outcompeting and overwhelming the hierarchical world of conventional closed software.

A good many skeptics weren't convinced, however; and the questions they raise deserve a fair engagement. Most of the objections to the bazaar argument come down to the claim that its proponents have underestimated the productivity-multiplying effect of conventional management.

Traditionally-minded software-development managers often object that the casualness with which project groups form and change and dissolve in the open-source world negates a significant part of the

apparent advantage of numbers that the open-source community has over any single closed-source developer. They would observe that in software development it is really sustained effort over time and the degree to which customers can expect continuing investment in the product that matters, not just how many people have thrown a bone in the pot and left it to simmer.

There is something to this argument, to be sure; in fact, I have developed the idea that expected future service value is the key to the economics of software production in The Magic Cauldron (http://www.tuxedo.org/~esr/writings/magic-cauldron/).

But this argument also has a major hidden problem; its implicit assumption that open-source development cannot deliver such sustained effort. In fact, there have been open-source projects that maintained a coherent direction and an effective maintainer community over quite long periods of time without the kinds of incentive structures or institutional controls that conventional management finds essential. The development of the GNU Emacs editor is an extreme and instructive example; it has absorbed the efforts of hundreds of contributors over fifteen years into a unified architectural vision, despite high turnover and the fact that only one person (its author) has been continuously active during all that time. No closed-source editor has ever matched this longevity record.

This suggests a reason for questioning the advantages of conventionally-managed software development that is independent of the rest of the arguments over cathedral vs. bazaar mode. If it's possible for GNU Emacs to express a consistent architectural vision over fifteen years, or for an operating system like Linux to do the same over eight years of rapidly changing hardware and platform technology; and if (as is indeed the case) there have been many well-architected open-source projects of more than five years duration – then we are entitled to wonder what, if anything, the tremendous overhead of conventionally-managed development is actually buying us.

Whatever it is certainly doesn't include reliable execution by deadline, or on budget, or to all features of the specification; it's a rare 'managed' project that meets even one of these goals, let alone all three. It also does not appear to be ability to adapt to changes in technology and economic context during the project lifetime, either; the open-source community has proven *far* more effective on that score (as one can readily verify, for example, by comparing the thirty-year history of the Internet with the short half-lives of proprietary networking technologies – or the cost of the 16-bit to 32-bit transition in Microsoft Windows with the nearly effortless up-migration of Linux during the same period, not only along the Intel line of development but to more than a dozen other hardware platforms including the 64-bit Alpha as well).

One thing many people think the traditional mode buys you is somebody to hold legally liable and potentially recover compensation from if the project goes wrong. But this is an illusion; most software

licenses are written to disclaim even warranty of merchantability, let alone performance – and cases of successful recovery for software nonperformance are vanishingly rare. Even if they were common, feeling comforted by having somebody to sue would be missing the point. You didn't want to be in a lawsuit; you wanted working software.

So what is all that management overhead buying?

In order to understand that, we need to understand what software development managers believe they do. A woman I know who seems to be very good at this job says software project management has five functions:

- To *define goals* and keep everybody pointed in the same direction.
- To *monitor* and make sure crucial details don't get skipped.
- To *motivate* people to do boring but necessary drudgework.
- To *organize* the deployment of people for best productivity.
- To *marshal resources* needed to sustain the project.

Apparently worthy goals, all of these; but under the open-source model, and in its surrounding social context, they can begin to seem strangely irrelevant. We'll take them in reverse order.

My friend reports that a lot of *resource marshalling* is basically defensive; once you have your people and machines and office space, you have to defend them from peer managers competing for the same resources, and higher-ups trying to allocate the most efficient use of a limited pool.

But open-source developers are volunteers, self-selected for both interest and ability to contribute to the projects they work on (and this remains generally true even when they are being paid a salary to hack open source.) The volunteer ethos tends to take care of the 'attack' side of resource-marshalling automatically; people bring their own resources to the table. And there is little or no need for a manager to 'play defense' in the conventional sense.

Anyway, in a world of cheap PCs and fast Internet links, we find pretty consistently that the only really limiting resource is skilled attention. Open-source projects, when they founder, essentially never do so for want of machines or links or office space; they die only when the developers themselves lose interest.

That being the case, it's doubly important that open-source hackers *organize themselves* for maximum productivity by self-selection – and the social milieu selects ruthlessly for competence. My friend, familiar with both the open-source world and large closed projects, believes that open source has been successful partly because its culture only accepts the most talented 5% or so of the programming population. She spends most of her time organizing the deployment of the other 95%, and has thus observed first-hand the well-known variance of a factor of one hundred in productivity between the most able programmers and the merely competent.

The size of that variance has always raised an awkward question: would individual projects, and the field as a whole, be better off without more than 50% of the least able in it? Thoughtful managers have understood for a long time that if conventional software management's only function were to convert the least able from a net loss to a marginal win, the game might not be worth the candle.

The success of the open-source community sharpens this question considerably, by providing hard evidence that it is often cheaper and more effective to recruit self-selected volunteers from the Internet than it is to manage buildings full of people who would rather be doing something else.

Which brings us neatly to the question of *motivation*. An equivalent and often-heard way to state my friend's point is that traditional development management is a necessary compensation for poorly motivated programmers who would not otherwise turn out good work.

This answer usually travels with a claim that the open-source community can only be relied on to do work that is 'sexy' or technically sweet; anything else will be left undone (or done only poorly) unless it's churned out by money-motivated cubicle peons with managers cracking whips over them. I address the psychological and social reasons for being skeptical of this claim in "Homesteading the Noosphere". For present purposes, however, I think it's more interesting to point out the implications of accepting it as true.

If the conventional, closed-source, heavily-managed style of software development is really defended only by a sort of Maginot line of problems conducive to boredom, then it's going to remain viable in each individual application area for only so long as nobody finds those problems really interesting and nobody else finds any way to route around them. Because the moment there is open-source competition for a 'boring' piece of software, customers are going to know that it was finally tackled by someone who chose that problem to solve because of a fascination with the problem itself – which, in software as in other kinds of creative work, is a far more effective motivator than money alone.

Having a conventional management structure solely in order to motivate, then, is probably good tactics but bad strategy; a short-term win, but in the longer term a surer loss.

So far, conventional development management looks like a bad bet now against open source on two points (resource marshalling, organization), and like it's living on borrowed time with respect to a third (motivation). And the poor beleaguered conventional manager is not going to get any succour from the *monitoring* issue; the strongest argument the open-source community has is that decentralized peer review trumps all the conventional methods for trying to ensure that details don't get slipped.

Can we save *defining goals* as a justification for the overhead of conventional software project management? Perhaps; but to do so, we'll need good reason to believe that management committees and corporate roadmaps are more successful at defining worthy and widely-shared goals than the project leaders and tribal elders who fill the analogous role in the open-source world.

That is on the face of it a pretty hard case to make. And it's not so much the open-source side of the balance (the longevity of Emacs, or Linus Torvalds's ability to mobilize hordes of developers with talk of "world domination") that makes it tough. Rather, it's the demonstrated awfulness of conventional mechanisms for defining the goals of software projects.

One of the best-known folk theorems of software engineering is that 60% to 75% of conventional software projects either are never completed or are rejected by their intended users. If that range is anywhere near true (and I've never met a manager of any experience who disputes it) then more projects than not are being aimed at goals that are either (a) not realistically attainable, or (b) just plain wrong.

This, more than any other problem, is the reason that in today's software engineering world the very phrase "management committee" is likely to send chills down the hearer's spine – even (or perhaps especially) if the hearer is a manager. The days when only programmers griped about this pattern are long past; 'Dilbert' cartoons hang over *executives'* desks now.

Our reply, then, to the traditional software development manager, is simple – if the open-source community has really underestimated the value of conventional management, *why do so many of you display contempt for your own process?*

Once again the existence of the open-source community sharpens this question considerably – because we have *fun* doing what we do. Our creative play has been racking up technical, market-share, and mind-share successes at an astounding rate. We're proving not only that we can do better software, but that *joy is an asset*.

Two and a half years after the first version of this essay, the most radical thought I can offer to close with is no longer a vision of an open-source-dominated software world; that, after all, looks plausible to a lot of sober people in suits these days.

Rather, I want to suggest what may be a wider lesson about software, (and probably about every kind of creative or professional work). Human beings generally take pleasure in a task when it falls in a sort of optimal-challenge zone; not so easy as to be boring, not too hard to achieve. A happy programmer is one who is neither underutilized nor weighed down with ill-formulated goals and stressful process friction. *Enjoyment predicts efficiency.*.

Relating to your own work process with fear and loathing (even in the displaced, ironic way suggested by hanging up Dilbert cartoons) should therefore be regarded in itself as a sign that the process has failed. Joy, humor, and playfulness are indeed assets; it was not mainly for the alliteration that I wrote of "happy hordes" above, and it is no mere joke that the Linux mascot is a cuddly, neotenous penguin.

It may well turn out that one of the most important effects of open source's success will be to teach us that play is the most economically efficient mode of creative work.

12. Acknowledgements

This paper was improved by conversations with a large number of people who helped debug it. Particular thanks to Jeff Dutky <dutky@wam.umd.edu>, who suggested the "debugging is parallelizable" formulation, and helped develop the analysis that proceeds from it. Also to Nancy Lebovitz <nancyl@universe.digex.net> for her suggestion that I emulate Weinberg by quoting Kropotkin. Perceptive criticisms also came from Joan Eslinger <wombat@kilimanjaro.engr.sgi.com> and Marty Franz <marty@net-link.net> of the General Technics list. Glen Vandenburg <glv@vanderburg.org> pointeed out the importance of self-selection in contributor populations and suggested the fruitful idea that much development rectifies 'bugs of omission'; Daniel Upper <upper@peak.org> suggested the natural analogies for this. I'm grateful to the members of PLUG, the Philadelphia Linux User's group, for providing the first test audience for the first public version of this paper. Paula Matuszek <matusp00@mh.us.sbphrd.com> enlightened me about the practice of software management. Phil Hudson <phil.hudson@iname.com> reminded me that the social organization of the hacker culture mirrors the organization of its software, and vice-versa. Finally, Linus Torvalds's comments were helpful and his early endorsement very encouraging.

13. For Further Reading

I quoted several bits from Frederick P. Brooks's classic *The Mythical Man-Month* because, in many respects, his insights have yet to be improved upon. I heartily recommend the 25th Anniversary edition from Addison-Wesley (ISBN 0-201-83595-9), which adds his 1986 *"No Silver Bullet"* paper.

The new edition is wrapped up by an invaluable 20-years-later retrospective in which Brooks forthrightly admits to the few judgements in the original text which have not stood the test of time. I first read the retrospective after the first public version of this paper was substantially complete, and was surprised to discover that Brooks attributes bazaar-like practices to Microsoft! (In fact, however, this attribution turned out to be mistaken. In 1998 we learned from the Halloween Documents (http://www.opensource.org/halloween/) that Microsoft's internal developer community is heavily balkanized, with the kind of general source access needed to support a bazaar not even truly possible.)

Gerald M. Weinberg's *The Psychology Of Computer Programming* (New York, Van Nostrand Reinhold 1971) introduced the rather unfortunately-labeled concept of "egoless programming". While he was nowhere near the first person to realize the futility of the "principle of command", he was probably the first to recognize and argue the point in particular connection with software development.

Richard P. Gabriel, contemplating the Unix culture of the pre-Linux era, reluctantly argued for the superiority of a primitive bazaar-like model in his 1989 paper *Lisp: Good News, Bad News, and How To Win Big*. Though dated in some respects, this essay is still rightly celebrated among Lisp fans (including me). A correspondent reminded me that the section titled "Worse Is Better" reads almost as an anticipation of Linux. The paper is accessible on the World Wide Web at http://www.naggum.no/worse-is-better.html.

De Marco and Lister's *Peopleware: Productive Projects and Teams* (New York; Dorset House, 1987; ISBN 0-932633-05-6) is an underappreciated gem which I was delighted to see Fred Brooks cite in his retrospective. While little of what the authors have to say is directly applicable to the Linux or open-source communities, the authors' insight into the conditions necessary for creative work is acute and worthwhile for anyone attempting to import some of the bazaar model's virtues into a commercial context.

Finally, I must admit that I very nearly called this paper "The Cathedral and the Agora", the latter term being the Greek for an open market or public meeting place. The seminal "agoric systems" papers by Mark Miller and Eric Drexler, by describing the emergent properties of market-like computational ecologies, helped prepare me to think clearly about analogous phenomena in the open-source culture

when Linux rubbed my nose in them five years later. These papers are available on the Web at http://www.agorics.com/agorpapers.html.

14. Epilog: Netscape Embraces the Bazaar

It's a strange feeling to realize you're helping make history....

On January 22 1998, approximately seven months after I first published "The Cathedral and the Bazaar", Netscape Communications, Inc. announced plans to give away the source for Netscape Communicator (http://www.netscape.com/newsref/pr/newsrelease558.html). I had had no clue this was going to happen before the day of the announcement.

Eric Hahn, Executive Vice President and Chief Technology Officer at Netscape, emailed me shortly afterwards as follows: "On behalf of everyone at Netscape, I want to thank you for helping us get to this point in the first place. Your thinking and writings were fundamental inspirations to our decision."

The following week I flew out to Silicon Valley at Netscape's invitation for a day-long strategy conference (on Feb 4 1998) with some of their top executives and technical people. We designed Netscape's source-release strategy and license together.

A few days later I wrote the following:

> Netscape is about to provide us with a large-scale, real-world test of the bazaar model in the commercial world. The open-source culture now faces a danger; if Netscape's execution doesn't work, the open-source concept may be so discredited that the commercial world won't touch it again for another decade.
>
> On the other hand, this is also a spectacular opportunity. Initial reaction to the move on Wall Street and elsewhere has been cautiously positive. We're being given a chance to prove ourselves, too. If Netscape regains substantial market share through this move, it just may set off a long-overdue revolution in the software industry.
>
> The next year should be a very instructive and interesting time.

And indeed it was. As I write in mid-1999, the development of what was later named 'Mozilla' has been only a qualified success. It achieved Netscape's original goal, which was to deny Microsoft a monopoly lock on the browser market. It has also achieved some dramatic successes (notably the release of the next-generation Gecko rendering engine).

However, it has not yet garnered the massive development effort from outside Netscape that the Mozilla founders had originally hoped for. The problem here seems to be that for a long time the Mozilla distribution actually broke one of the basic rules of the bazaar model; they didn't ship something potential contributors could easily run and see working. (Until more than a year after release, building Mozilla from source required a license for the proprietary Motif library.)

Most negatively (from the point of view of the outside world) the Mozilla group has yet to ship a production-quality browser – and one of the project's principals caused a bit of a sensation by resigning, complaining of poor management and missed opportunities. "Open source," he correctly observed, "is not magic pixie dust."

And indeed it is not. The long-term prognosis for Mozilla looks dramatically better now (in August 2000) than it did at the time of Jamie Zawinski's resignation letter – but he was right to point out that going open will not necessarily save an existing project that suffers from ill-defined goals or spaghetti code or any of the software engineering's other chronic ills. Mozilla has managed to provide an example simultaneously of how open source can succeed and how it could fail.

In the mean time, however, the open-source idea has scored successes and found backers elsewhere. 1998 and late 1999 saw a tremendous explosion of interest in the open-source development model, a trend both driven by and driving the continuing success of the Linux operating system. The trend Mozilla touched off is continuing at an accelerating rate.

15. Endnotes

[JB] In *Programing Pearls*, the noted computer-science aphorist Jon Bentley comments on Brooks's observation with "If you plan to throw one away, you will throw away two.". He is almost certainly right. The point of Brooks's observation, and Bentley's, isn't merely that you should expect first attempt to be wrong, it's that starting over with the right idea is usually more effective than trying to salvage a mess.

[QR] Examples of successful open-source, bazaar development predating the Internet explosion and unrelated to the Unix and Internet traditions have existed. The development of the info-Zip

(http://www.cdrom.com/pub/infozip/) compression utility during 1990-1992, primarily for DOS machines, was one such. Another was the RBBS bulletin board system (again for DOS), which began in 1983 and developed a sufficiently strong community that there have been fairly regular releases up to the present (mid-1999) despite the huge technical advantages of Internet mail and file-sharing over local BBSs. While the info-Zip community relied to some extent on Internet mail, the RBBS developer culture was actually able to base a substantial on-line community on RBBS that was completely independent of the TCP/IP infrastructure.

[JH] John Hasler has suggested an interesting explanation for the fact that duplication of effort doesn't seem to be a net drag on open-source development. He proposes what I'll dub "Hasler's Law": the costs of duplicated work tend to scale sub-qadratically with team size – that is, more slowly than the planning and management overhead that would be needed to eliminate them.

This claim actually does not contradict Brooks's Law. It may be the case that total complexity overhead and vulnerability to bugs scales with the square of team size, but that the costs from *duplicated* work are nevertheless a special case that scales more slowly. It's not hard to develop plausible reasons for this, starting with the undoubted fact that it is much easier to agree on functional boundaries between different developers' code that will prevent duplication of effort than it is to prevent the kinds of unplanned bad interactions across the whole system that underly most bugs.

The combination of Linus's Law and Hasler's Law suggests that there are actually three critical size regimes in software projects. On small projects (I would say one to at most three developers) no management structure more elaborate than picking a lead programmer is needed. And there is some intermediate range above that in which the cost of traditional management is relatively low, so its benefits from avoiding duplication of effort, bug-tracking, and pushing to see that details are not overlooked actually net out positive.

Above that, however, the combination of Linus's Law and Hasler's Law suggests there is a large-project range in which the costs and problems of traditional management rise much faster than the expected cost from duplication of effort. Not the least of these costs is a structural inability to harness the many-eyeballs effect, which (as we've seen) seems to do a much better job than traditional management at making sure bugs and details are not overlooked. Thus, in the large-project case, the combination of these laws effectively drives the net payoff of traditional management to zero.

[HBS] The split between Linux's experimental and stable versions has another function related to, but distinct from, hedging risk. The split attacks another problem: the deadliness of deadlines. When programmers are held both to an immutable feature list and a fixed drop-dead date, quality goes out the window and there is likely a colossal mess in the making. I am indebted to Marco Iansiti and Alan

MacCormack of the Harvard Business School for pointing me at evidence that relaxing either one of these constraints can make scheduling workable.

One way to do this is to fix the deadline but leave the feature list flexible, allowing features to drop off if not completed by deadline. This is essentially the strategy of the "stable" kernel branch; Alan Cox (the stable-kernel maintainer) puts out releases at fairly regular intervals, but makes no guarantees about when particular bugs will be fixed or features back-ported from the experimental branch.

The other way to do this is to set a desired feature list and deliver only when it is done. This is essentially the strategy of the "experimental" kernel branch. De Marco and Lister cited research showing that this scheduling policy ("wake me up when it's done") produces not only the highest quality but, on average, shorter delivery times than either "realistic" or "aggressive" scheduling.

I have come to suspect (as of early 2000) that in earlier versions of this paper I severely underestimated the importance of the "wake me up when it's done" anti-dealine policy to the open-source community's productivity and quality. General experience with the rushed GNOME 1.0 in 1999 suggests that pressure for a premature release can neutralize many of the quality benefits open source normally confers.

It may well turn out to be that the process transparency of open source is one of three coequal drivers of its quality, along with "wake me up when it's done" scheduling and developer self-selection.

[IN] An issue related to whether one can start projects from zero in the bazaar style is whether the bazaar style is capable of supporting truly innovative work. Some claim that, lacking strong leadership, the bazaar can only handle the cloning and improvement of ideas already present at the engineering state of the art, but is unable to push the state of the art. This argument was perhaps most infamously made by the Halloween Documents (http://www.opensource.org/halloween/), two embarrassing internal Microsoft memoranda written about the open-source phenomenon. The authors compared Linux's development of a Unix-like operating system to "chasing taillights", and opined "(once a project has achieved "parity" with the state-of-the-art), the level of management necessary to push towards new frontiers becomes massive."

There are serious errors of fact implied in this argument. One is exposed when the Halloween authors themseselves later observe that "often [...] new research ideas are first implemented and available on Linux before they are available / incorporated into other platforms."

If we read "open source" for "Linux", we see that this is far from a new phenomenon. Historically, the open-source community did not invent Emacs or the World Wide Web or the Internet itself by chasing taillights or being massively managed – and in the present, there is so much innovative work going on in

open source that one is spoiled for choice. The GNOME project (to pick one of many) is pushing the state of the art in GUIs and object technology hard enough to have attracted considerable notice in the computer trade press well outside the Linux community. Other examples are legion, as a visit to Freshmeat (http://freshmeat.net/) on any given day will quickly prove.

But there is a more fundamental error in the implicit assumption that the *cathedral model* (or the bazaar model, or any other kind of management structure) can somehow make innovation happen reliably. This is nonsense. Gangs don't have breakthrough insights – even volunteer groups of bazaar anarchists are usually incapable of genuine originality, let alone corporate committees of people with a survival stake in some status quo ante. *Insight comes from individuals*. The most their surrounding social machinery can ever hope to do is to be *responsive* to breakthrough insights – to nourish and reward and rigorously test them instead of squashing them.

Some will characterize this as a romantic view, a reversion to outmoded lone-inventor stereotypes. Not so; I am not asserting that groups are incapable of *developing* breakthrough insights once they have been hatched; indeed, we learn from the peer-review process that such development groups are essential to producing a high-quality result. Rather I am pointing out that every such group development starts from – is necessarily sparked by – one good idea in one person's head. Cathedrals and bazaars and other social structures can catch that lightning and refine it, but they cannot make it on demand.

Therefore the root problem of innovation (in software, or anywhere else) is indeed how not to squash it – but, even more fundamentally, it is *how to grow lots of people who can have insights in the first place*.

To suppose that cathedral-style development could manage this trick but the low entry barriers and process fluidity of the bazaar cannot would be absurd. If what it takes is one person with one good idea, then a social milieu in which one person can rapidly attract the cooperation of hundreds or thousands of others with that good idea is going inevitably to out-innovate any in which the person has to do a political sales job to a hierarchy before he can work on his idea without risk of getting fired.

And, indeed, if we look at the history of software innovation by organizations using the cathedral model, we quickly find it is rather rare. Large corporations rely on university research for new ideas (thus the Halloween Documents authors' unease about Linux's facility at coopting that research more rapidly). Or they buy out small companies built around some innovator's brain. In neither case is the innovation native to the cathedral culture; indeed, many innovations so imported end up being quietly suffocated under the "massive level of management" the Halloween Documents' authors so extol.

That, however, is a negative point. The reader would be better served by a positive one. I suggest, as an experiment, the following;

- Pick a criterion for originality that you believe you can apply consistently. If your definition is "I know it when I see it", that's not a problem for purposes of this test.
- Pick any closed-source operating system competing with Linux, and a best source for accounts of current development work on it.
- Watch that source and Freshmeat for one month. Every day, count the number of release announcements on Freshmeat that you consider 'original' work. Apply the same definition of 'original' to announcements for that other OS and count them.
- Thirty days later, total up both figures.

The day I wrote this, Freshmeat carried twenty-two release announcements, of which three appear they might push state of the art in some respect, This was a slow day for Freshmeat, but I will be astonished if any reader reports as many as three likely innovations *a month* in any closed-source channel.

[EGCS] We how have history on a project that, in several ways, may provide a more indicative test of the bazaar premise than fetchmail; EGCS (http://egcs.cygnus.com/), the Experimental GNU Compiler System.

This project was announced in mid-August of 1997 as a conscious attempt to apply the ideas in the early public versions of "The Cathedral and the Bazaar". The project founders felt that the development of GCC, the Gnu C Compiler, had been stagnating. For about twenty months afterwards, GCC and EGCS continued as parallel products – both drawing from the same Internet developer population, both starting from the same GCC source base, both using pretty much the same Unix toolsets and development environment. The projects differed only in that EGCS consciously tried to apply the bazaar tactics I have previously described, while GCC retained a more cathedral-like organization with a closed developer group and infrequent releases.

This was about as close to a controlled experiment as one could ask for, and the results were dramatic. Within months, the EGCS versions had pulled substantially ahead in features; better optimization, better support for FORTRAN and C++. Many people found the EGCS development snapshots to be more reliable than the most recent stable version of GCC, and major Linux distributions began to switch to EGCS.

In April of 1999, the Free Software Foundation (the official sponsors of GCC) dissolved the original GCC development group and officially handed control of the project to the the EGCS steering team.

[SP] Of course, Kropotkin's critique and Linus's Law raise some wider issues about the cybernetics of social organizations. Another folk theorem of software engineering suggests one of them; Conway's Law – commonly stated as "If you have four groups working on a compiler, you'll get a 4-pass compiler". The original statement was more general: "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." We might put it more succinctly as "The means determine the ends", or even "Process becomes product".

It is accordingly worth noting that in the open-source community organizational form and function match on many levels. The network is everything and everywhere: not just the Internet, but the people doing the work form a distributed, loosely coupled, peer-to-peer network which provides multiple redundancy and degrades very gracefully. In both networks, each node is important only to the extent that other nodes want to cooperate with it.

The peer-to-peer part is essential to the community's astonishing productivity. The point Kropotkin was trying to make about power relationships is developed further by the 'SNAFU Principle': "True communication is possible only between equals, because inferiors are more consistently rewarded for telling their superiors pleasant lies than for telling the truth." Creative teamwork utterly depends on true communication and is thus very seriously hindered by the presence of power relationships. The open-source community, effectively free of such power relationships, is teaching us by contrast how dreadfully much they cost in bugs, in lowered productivity, and in lost opportunities.

Further, the SNAFU principle predicts in authoritarian organizations a progressive disconnect between decision-makers and reality, as more and more of the input to those who decide tends to become pleasant lies. The way this plays out in conventional software development is easy to see; there are strong incentives for the inferiors to hide, ignore, and minimize problems. When this process becomes product, software is a disaster.