**일찍, 그리고 자주 릴리즈 하라**

Early and frequent releases are a critical part of the Linux development model.

초기 및 빈번한 릴리스는 Linux 개발 모델의 중요한 부분입니다.

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.

초기 버전이 버그 버전이었고 사용자의 인내심을 잃어 버리기를 원치 않았기 때문에, 대부분의 개발자 (나 포함)는 사소한 프로젝트보다 큰 정책에 대한 나쁜 정책이라고 생각했습니다.

This belief reinforced the general commitment to a cathedral-building style of development.

이 신념은 대성당 건축 스타일 개발에 대한 일반적인 의지를 강화시켰다.

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.

사용자가 가능한 한 적은 수의 버그 만 볼 수있게하려는 최우선 목표가 있다면 왜 6 개월마다 (또는 덜 자주) 버전을 릴리스하고 릴리스 간 디버깅시에는 개처럼 작동합니다.

The Emacs C core was developed this way.

이맥스 C 코어는 이런 식으로 개발되었다.

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.

사실상 Lisp 라이브러리는 FSF의 컨트롤 밖에 있는 Lisp 아카이브가 있었기 때문에 Emacs의 릴리스 사이클과 독립적으로 새로운 개발 코드 버전을 찾을 수 있었습니다.

The most important of these, the Ohio State Emacs Lisp archive, anticipated the spirit and many of the features of today's big Linux archives.

오하이오 주 Emacs Lisp 아카이브 중 가장 중요한 점은 오늘날의 큰 리눅스 아카이브의 정신과 많은 기능을 예상한다는 것입니다.

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.

그러나 우리 중 일부는 실제로 우리가 하고 있는 일에 대해 매우 열심히 생각하거나 FSF 대성당 건설 개발 모델의 문제점에 대해 그 아카이브의 존재가 제안한 바를 거의 생각하지 않았습니다.

I made one serious attempt around 1992 to get a lot of the Ohio code formally merged into the official Emacs Lisp library.

나는 오하이오 코드를 정식으로 공식 Emacs Lisp 라이브러리에 병합하기 위해 1992 년 경에 하나의 진지한 시도를했다.

I ran into political trouble and was largely unsuccessful.

나는 정치적인 문제로 달려 들고 거의 실패했다.

But by a year later, as Linux became widely visible, it was clear that something different and much healthier was going on there.

그러나 1 년 후, 리눅스가 널리 보급됨에 따라, 뭔가 다르고 훨씬 건강해진 것이 분명해졌습니다.

Linus's open development policy was the very opposite of cathedral-building.

리누스의 공개 개발 정책은 대성당 건설과는 정반대였다.

Linux's Internet archives were burgeoning, multiple distributions were being floated.

리눅스의 인터넷 아카이브가 급성장하고 있었고, 여러 배포판이 떠돌 았습니다.

And all of this was driven by an unheard-of frequency of core system releases.

그리고이 모든 것은 핵심 시스템 릴리스의 전례없는 빈도로 인해 이루어졌습니다.

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

Linus는 가장 효과적인 방법으로 공동 개발자로서 사용자를 대하고있었습니다.

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

7. 일찍 석방하십시오. 자주 출시하십시오. 그리고 고객의 말을 들어보십시오.

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.

리누스의 혁신은 사용자 의견 (많은 유닉스 세계의 오랜 전통 이었음)이 많이 포함 된 빠른 전환 릴리스에서 그다지 많은 것이 아니 었습니다.

In those early times (around 1991) it wasn't unknown for him to release a new kernel more than once a day!

그는 발전하고 있었다. 초기 (1991 년경)에는 하루에 한 번 이상 새 커널을 출시하는 것이 알려지지 않았습니다.

Because he cultivated his base of codevelopers and leveraged the Internet for collaboration harder than anyone else, this worked.

그는 코드 랩터를 기반으로 다른 누구보다도 열심히 협력하기 위해 인터넷을 활용했기 때문에이 방법이 효과적이었습니다.

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

그러나 어떻게 작동 했습니까? 리누스 토발즈 (Linus Torvalds)의 독특한 천재성에 의존했거나 복제 할 수있는 것이 었나요?

I didn't think so.

나는 그렇게 생각하지 않았다.

Granted, Linus is a damn fine hacker.

허락하신다면, 리누스는 괜찮은 해커입니다.

How many of us could engineer an entire productionquality operating system kernel from scratch?

얼마나 많은 사람들이 전체 productionquality 운영 체제 커널을 처음부터 개발할 수 있습니까?

But Linux didn't represent any awesome conceptual leap forward.

그러나 Linux는 개념적으로 도약을 이루지 못했습니다.

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.

Linus는 Richard Stallman이나 James Gosling (NeWS와 Java)과 같은 혁신적인 디자인의 천재는 아닙니다 (또는 적어도 아직은 아닙니다).

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.

오히려 리누스는 버그와 개발의 한계를 피할 수있는 여섯 번째 감각과 점 A에서 점 B까지 최소 노력 경로를 찾는 진정한 요령으로 엔지니어링과 구현의 천재라고 생각합니다.

Indeed, the whole design of Linux breathes this quality and mirrors Linus's essentially conservative and simplifying design approach.

실제로, Linux는 이러한 품질을 호흡하며 리누스가 근본적으로 보수적이고 단순한 설계 접근 방식을 반영합니다.

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?

따라서 급속한 출시와 인터넷 매체를 활용하여 실수를 저지른 것이 아니라 최소 노력 경로에 대한 리누스의 엔지니어링 - 천재 통찰력의 핵심 부분이었던 경우, 그는 무엇을 극대화 했습니까? 그가 기계에서 무엇을 꺼내 왔습니까?

Put that way, the question answers itself.

그런 식으로하면 질문 자체가 답을합니다.

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.

Linus는 자신의 해커 / 사용자가 자아 - 만족할만한 행동을 취할 것이라는 전망에 끊임없이 자극되고 보상을 받으면서 일을 꾸준히 (심지어 매일) 개선함으로써 보람을 느끼게했습니다.

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.

Linus는 심각한 버그가 다루기 힘든 것으로 판명되면 코드 및 사용자 기반 소모로 인해 발생할 수있는 불안정성 비용으로도 디버깅 및 개발시 인력 수를 최대화하는 것을 직접 목표로 삼았습니다.

Linus was behaving as though he believed something like this:

리누스는 마치 이런 말을 믿는 것처럼 행동했습니다.

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

8. 충분히 큰 베타 테스터와 공동 개발자 기반이 주어지면, 거의 모든 문제는 신속하게 특징 지어 질 것이고, 수정은 누군가에게 명백해질 것입니다.

Or, less formally, ``Given enough eyeballs, all bugs are shallow.''

또는 공식적으로 덜 공식적으로, "충분한 눈알이 주어지면 모든 버그는 얕습니다.

I dub this: ``Linus's Law''.

"나는 이것을 "리누스의 법칙"이라고 부릅니다.

My original formulation was that every problem ``will be transparent to somebody''.

저의 원래 공식은 모든 문제가 "누군가에게 투명해질 것"이었습니다.

Linus demurred that the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it.

리누스는 문제를 이해하고 고치는 사람이 반드시 그 사람을 처음으로 특성화 한 사람 일 필요는 없으며 대개는 그렇지 않다고 주장했습니다.

``Somebody finds the problem,'' he says, ``and somebody else understands it.

``누군가가 문제를 발견하고``누군가는 그것을 이해한다.

And I'll go on record as saying that finding it is the bigger challenge.''

그리고 나는 그것을 발견하는 것이 더 큰 도전이라고 말하면서 기록에 남을 것입니다.

That correction is important; we'll see how in the next section, when we examine the practice of debugging in more detail.

" 다음 섹션에서 디버깅 실습을 자세히 살펴볼 때 어떻게되는지 살펴 보겠습니다.

But the key point is that both parts of the process (finding and fixing) tend to happen rapidly.

그러나 요점은 프로세스의 두 부분 (발견 및 수정)이 빠르게 발생하는 경향이 있다는 것입니다.

In Linus's Law, I think, lies the core difference underlying the cathedral-builder and bazaar styles.

Linus의 법칙에 따르면, 대성당 건축가와 바자 스타일의 근본적인 차이가 거짓말이라고 생각합니다.

In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena.

대성당 건축가의 프로그래밍 관점에서 볼 때, 버그와 개발 문제는 까다 롭고 교활하며 심오한 현상입니다.

It takes months of scrutiny by a dedicated few to develop confidence that you've winkled them all out.

헌신적 인 소수의 사람들이 당신을 완전히 깜짝 놀라게했다는 확신을 갖기 위해 수개월의 조사가 필요합니다.

Thus the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect.

따라서 긴 릴리스 간격과 오랫동안 기다려온 릴리스가 완벽하지 않을 때 필연적 인 실망.

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.

한편, 바자 뷰에서는 버그가 일반적으로 얕은 현상이라고 생각하거나 적어도 매번 새로운 릴리스가 나올 때마다 열심히 일하는 수천 명의 공동 개발자에게 노출 될 때 매우 빠르게 변경됩니다.

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.

따라서 더 많은 수정을하기 위해 자주 석방합니다. 가끔씩 문을 두드리면 유익한 부작용으로 잃을 일이 줄어 듭니다.

And that's it. That's enough.

그리고 그게 다야. 충분 해.

If ``Linus's Law'' is false, then any system as complex as the Linux kernel, being hacked over by as many hands as the that kernel was, should at some point have collapsed under the weight of unforseen bad interactions and undiscovered ``deep'' bugs.

만약``리누스의 법칙 ''이 거짓이라면, 리눅스 커널만큼 복잡한 시스템은 커널과 같이 많은 손으로 해킹 당할 수 있습니다. 예기치 않은 잘못된 상호 작용과 알려지지 않은` 깊은 "버그.

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.

반면에 사실이라면, 리눅스의 상대적으로 부족한 버그와 몇 달 또는 몇 년에 걸친 지속적인 가동 시간을 설명하는 것만으로도 충분합니다.

Maybe it shouldn't have been such a surprise, at that.

어쩌면 그렇게 놀라서는 안될 것입니다.

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 the opinion of a single randomly-chosen one of the observers.

사회 학자들은 수년 전에 똑같이 전문가 (또는 똑같이 무식한) 관측자의 평균적인 의견이 관찰자 중 무작위로 선택된 사람의 의견보다는 훨씬 더 신뢰할만한 예측 자임을 발견했다.

They called this the Delphi effect.

그들은 이것을 델파이 효과라고 불렀습니다.

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.

리누스가 보여 주었던 것은 운영체제의 디버깅에도 적용된다는 것입니다. 델파이 효과는 OS 커널의 복잡성 수준에서도 개발 복잡성을 무색하게합니다.

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.

델파이 효과에 명확하게 도움이되는 리눅스 상황의 한 가지 특별한 특징은 주어진 프로젝트의 참여자가 스스로 선택된다는 사실입니다.

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.

초기 응답자는 샘플을 무작위로 추출한 것이 아니라 소프트웨어 사용에 대해 충분히 관심이있는 사람들로부터 어떻게 작동하는지 배우고, 발생하는 문제에 대한 해결책을 찾으려고 시도하며, 실제로는 명백하게 합당한 해결책을 제시한다고 지적했습니다.

Anyone who passes all these filters is highly likely to have something useful to contribute.

이러한 모든 필터를 통과 한 사람은 기여할 ​​수있는 무언가를 가질 가능성이 큽니다.

Linus's Law can be rephrased as ``Debugging is parallelizable''.

리누스의 법칙은``디버깅이 병렬화 될 수있다 ''라는 말을 다시 할 수 있습니다.

Although debugging requires debuggers to communicate with some coordinating developer, it doesn't require significant coordination between debuggers.

디버깅은 일부 조정 개발자와 통신하기 위해 디버거가 필요하지만 디버거 간에는 크게 조정할 필요가 없습니다.

Thus it doesn't fall prey to the same quadratic complexity and management costs that make adding developers problematic.

따라서 개발자를 추가하는 데 문제가되는 동일한 2 차 복잡성 및 관리 비용에 사로 잡히지 않습니다.

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.

실제로, 디버거에 의한 작업의 중복으로 인한 이론적 인 효율성 저하는 Linux 세계에서 결코 문제가되지 않는 것으로 보입니다.

One effect of a ``release early and often'' policy is to minimize such duplication by propagating fed-back fixes quickly.

"조기 및 자주 발표"정책의 한 가지 효과는 피드백 된 수정 사항을 신속하게 전파함으로써 이러한 중복을 최소화하는 것입니다.

Brooks (the author of The Mythical Man-Month) even made an off-hand observation related to this: ``The total cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it.

Brooks (The Mythical Man-Month의 저자)는 이와 관련하여 "널리 사용되는 프로그램을 유지 관리하는 데 드는 총 비용은 일반적으로 개발 비용의 40 % 이상입니다.

Surprisingly this cost is strongly affected by the number of users.

놀랍게도이 비용은 사용자 수에 크게 영향을받습니다.

More users find more bugs.”

더 많은 사용자가 더 많은 버그를 발견합니다.”

More users find more bugs because adding more users adds more different ways of stressing the program.

더 많은 사용자가 더 많은 프로그램을 강조하는 다른 방법을 추가하기 때문에 더 많은 사용자가 더 많은 버그를 찾습니다.

This effect is amplified when the users are co-developers.

이 효과는 사용자가 공동 개발자 일 때 증폭됩니다.

Each one approaches the task of bug characterization with a slightly different perceptual set and analytical toolkit, a different angle on the problem.

각각은 약간 다른 지각 세트와 분석 툴킷을 사용하여 버그 특성화 작업에 접근합니다. 문제에 대한 다른 각도입니다.

The ``Delphi effect'' seems to work precisely because of this variation.

"델파이 효과"는 이러한 차이 때문에 정확하게 작동하는 것으로 보인다.

In the specific context of debugging, the variation also tends to reduce duplication of effort.

디버깅의 특정 컨텍스트에서 변형은 또한 노력의 중복을 줄이는 경향이 있습니다.

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.

더 많은 베타 테스터를 추가하는 것은 개발자의 관점에서 현재의 "가장 깊은"버그의 복잡성을 줄이지는 못하지만 누군가의 툴킷이 버그가 얕은 방식으로 문제와 일치 할 가능성을 높입니다 그 사람에게.

Linus coppers his bets, too.

리누스도 그의 내기를 망칩니다.

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.

심각한 버그가있는 경우 Linux 커널 버전 번호가 매겨져 잠재적 인 사용자가 새로운 기능을 얻으려면 최후의 버전을 "안정적"으로 실행하거나 최첨단 및 위험 버그를 타기 위해 선택할 수 있습니다.

This tactic is not yet systematically imitated by most Linux hackers, but perhaps it should be;

이 전략은 아직 대부분의 리눅스 해커들에 의해 체계적으로 모방되지는 않았지만 아마도 그렇게되어야 할 것이다.

the fact that either choice is available makes both more attractive.

두 선택 중 하나를 사용할 수 있다는 사실은 더 매력적입니다.