SelfLinux-0.13.1

Die Kathedrale und der Basar

Autor: Eric S. Raymond (*esr@thyrsus.com*)
Formatierung: Matthias Hagedorn (*matthias.hagedorn@selflinux.org*)
Lizenz: OPL

Aus dem Amerikanischen von Reinhard Gantar

Diese Übersetzung basiert auf der Fassung vom 8. August 1999 Zum Original.

Ich untersuche das erfolgreiche Open Source-Projekt fetchmail, das ich willkürlich als Test für einige überraschende Theorien über Software-Entwicklung herausgegriffen habe, und die durch die Geschichte von Linux nahe gelegt werden. Ich erörtere diese Theorien unter dem Blickwinkel zweier grundsätzlich verschiedener Entwicklungsarten. Das eine Modell ist das der "Kathedrale", das in der kommerziell orientierten Software-Welt überwiegt. Das zweite ist im Gegensatz dazu das des "Basars" und der Linux-Welt. Ich werde hier zeigen, dass diese Modelle auf jeweils entgegengesetzten Annahmen über die Natur des Debuggings von Software beruhen. Es folgt die These, dass jeglicher Bug (Fehler) schnell gefunden wird, wenn sich nur genug Entwickler damit befassen - "Given enough eyeballs, all bugs are shallow" - was auf den in der Geschichte von Linux gemachten Erfahrungen beruht. Ich zeige Analogien zu anderen selbst-korrigierenden Systemen von egoistischen Vertretern und erforsche vor dem Abschluss noch einige Implikationen dieser Einsichten für die Zukunft der Software.

Inhaltsverzeichnis

1 Die Kathedrale und der Basar

2 Post muss immer ankommen

3 Von der Wichtigkeit, Benutzer zu haben

4 Früh freigeben, oft freigeben

5 Wann ist eine Rose keine Rose?

6 Aus popclient wird fetchmail

7 Fetchmail wird erwachsen

8 Was wir von fetchmail sonst noch lernen können

9 Voraussetzungen für den Basar-Stil

10 Der soziale Kontext der Open Source-Software

11 Über Management und die Maginotlinie

12 Danksagung

13 Weiterführende Literatur

14 Epilog: Netscape geht auf den Basar

15 Fußnoten

16 Version and Change History

1 Die Kathedrale und der Basar

Linux ist subversiv. Wer hätte auch vor nur fünf Jahren (1991) gedacht, dass sich ein Betriebssystem der Spitzenklasse wie durch Zauberei materialisieren könnte, geschaffen von Tausenden über den ganzen Planeten verstreuten Nebenerwerbs-Hackern, die durch die eng verwobenen Stränge des Internets verbunden sind?

Ich sicher nicht. Zu dem Zeitpunkt, als Linux 1993 auf meinem Radarschirm auftauchte, hatte ich bereits zehn Jahre in der Unix- und Open Source-Entwicklung verbracht. Mitte der Achtziger war ich einer der ersten Beitragenden zu GNU. Ich hatte bereits umfangreiche Open Source-Software im Internet veröffentlicht, die ich selbst entwickelt oder mitentwickelt hatte (nethack, Emacs VC (Version Control) und GUD modes, xlife und andere) und die heute noch viel verwendet wird. Ich dachte, ich wüsste, wie es gemacht wird.

Dann stellte Linux alles in Frage, was ich zu wissen glaubte. Ich hatte das Unix-Evangelium der kleinen Tools, des rapid prototyping und der inkrementellen Verbesserung seit der ersten Stunde verbreitet. Ich glaubte aber auch, dass es eine bestimmte kritische Komplexitätsstufe gebe, ab der ein zentralisierterer Ansatz mit sehr genauer Vorausplanung erforderlich wird. Ich glaubte, dass die wichtigste Software (Betriebssysteme und wirklich umfangreiche Tools wie Emacs) so gebaut werden müssten wie Kathedralen, sorgsam gemeißelt von einzelnen Druiden oder kleinen Teams von Hohepriestern, die in totaler Abgeschiedenheit wirkten und keine unfertigen Beta-Freigaben veröffentlichen dürften.

Linus Torvalds Entwicklungsstil auf der anderen Seite - mit seinen frühen und häufigen Freigaben, seinem Delegieren von allem, was nur irgendwie möglich ist, und der an Promiskuität grenzenden Offenheit - war eine echte Überraschung. Es handelte sich nicht gerade um eine stille und ehrfurchtsvolle Tätigkeit, wie der Bau einer Kathedrale eine ist -- stattdessen schien die Linux-Gemeinde ein großer, wild durcheinander plappernder Basar von verschiedenen Zielsetzungen und Ansätzen zu sein (alles sehr treffend durch die Linux-Archivsites repräsentiert, die Beiträge von *jedem* nehmen), der ein kohärentes und stabiles System wohl nur durch eine Reihe von Wundern hervorbringen konnte.

Die Tatsache, dass der Basar zu funktionieren schien, und zwar sehr gut zu funktionieren schien, war ein ausgesprochener Schock. Während ich lernte, mich in dieser neuen Umgebung zurechtzufinden, arbeitete ich nicht nur angestrengt an eigenen Projekten, sondern versuchte auch zu verstehen, warum die Linux-Welt sich nicht nur nicht einfach in völliger Konfusion auflöste, sondern an Durchschlagskraft immer weiter zulegte und eine Produktivität ausbildete, die für die Erbauer einer Software-Kathedrale kaum vorstellbar gewesen ist.

Mitte 1996 dachte ich, dass mir ein genaueres Verständnis dämmerte. Durch Zufall bekam ich eine ausgezeichnete Gelegenheit, meine Theorie zu testen, und zwar in Form eines Open Source-Projekts, das ich bewusst im Basar-Stil abwickeln konnte. Das tat ich dann auch -- und es wurde ein bedeutender Erfolg.

Dies ist die Geschichte dieses Projekts. Ich verwende es, um einige Aphorismen über effektive Open Source-Entwicklung vorzustellen. Nicht alle davon erfuhr ich als erstes in der Linux-Welt, ich werde aber auf Beispiele aus der Linux-Welt zurückgreifen, um bestimmte Punkte zu illustrieren. Wenn ich damit richtig liege, werden sie helfen zu verstehen, warum gerade die Linux-Gemeinde zu so einem steten Quell guter Software geworden ist -- und vielleicht auch, wie Sie selbst produktiver werden können.

2 Post muss immer ankommen

Seit 1993 kümmere ich mich um die technische Seite eines kleinen Gratis-ISP namens >Chester County InterLink (CCIL) in West Chester in Pennsylvania (ich bin Mitbegründer von CCIL und schrieb unsere einzigartige Multiuser BBS-Software -- man kann sie sich durch einen `telnet` zu locke.ccil.org telnet://locke.ccil.org/ ansehen. Heute werden fast dreitausend Benutzer auf dreißig Leitungen unterstützt). Der Job gestattete mir nicht nur den Zugriff auf CCILs 56K-Leitung , sondern machte ihn praktisch unbedingt notwendig!

Dementsprechend gewöhnte ich mich an einen ununterbrochenen Zugriff auf Internet E-Mail. Aus komplizierten Gründen war es sehr schwierig, SLIP (Serial Line Internet Protocol) für die Verbindung zwischen meiner Maschine zu Hause (snark.thyrsus.com) und CCIL tauglich zu machen. Als ich endlich Erfolg damit hatte, fand ich das periodische Telnetten zu locke.ccil.org bald lästig. Was ich wollte, war eine sofortige Übermittlung meiner elektronischen Post zu meiner snark-Maschine und eine Benachrichtigung des Eintreffens, um sie gleich mit meiner lokalen Software bearbeiten zu können.

Ein schlichtes Weiterreichen durch sendmail hätte nicht funktioniert, da meine persönliche Maschine nicht immer am Netzwerk angeschlossen ist und keine statische IP-Adresse hat. Was ich brauchte, war ein Programm, das über meine SLIP-Verbindung die E-Mail abholte und lokal verfügbar machte. Ich wusste, dass es derartige Software gab, und dass die meiste davon ein einfaches Anwendungsprotokoll namens POP (Post Office Protocol) verwendete. Es gab natürlich bereits einen POP3-Server als Teil von Locke's Betriebssystem BSD/OS.

Ich brauchte also einen POP3-Klienten. So ging ich hinaus ins Netz und fand einen. Tatsächlich fand ich sogar drei oder vier. Einige Zeit lang verwendete ich pop-perl, vermisste aber ein für mich sehr plausibles Leistungsmerkmal -- die Fähigkeit, die Adressen in abgeholter E-Mail umzuhacken, so dass auch die Antworten darauf richtig weitergereicht würden.

Das Problem war folgendes. Nehmen wir an, jemand namens **joe** mit einem Mail-Konto bei **locke** würde mir eine E-Mail schicken. Wenn ich dann die Mail zu mir auf meine snark-Maschine hole und dann versuche, darauf zu antworten, dann würde mein Mailer diese Antwort an einen nicht-existenten **joe** auf snark schicken. Ein händisches Ergänzen von @ccil.org wird schnell zu einer ernsthaften Plage.

Das war ganz offensichtlich eine Mühe, die sich der Computer machen sollte, nicht ich. Aber keiner der existierenden POP-Klienten konnte das tun. Und das bringt uns zur ersten Lektion:

> *1. Jede gute Software beginnt mit den persönlichen Sehnsüchten eines Entwicklers.*

Das hätte vielleicht jedem sofort einleuchten sollen (schließlich gibt es das Sprichwort **Not macht erfinderisch** schon seit geraumer Zeit), aber viel zu oft haben Softwareentwickler ihre Tage mit der Arbeit an Programmen verbringen müssen, die sie weder gebraucht noch geliebt haben. Aber nicht in der Linux-Welt -- was vielleicht die überdurchschnittliche Qualität der von der Linux-Gemeinde geschaffenen Software erklärt.

Setzte ich mich also gleich hin und begann im Schaffensrausch einen brandneuen POP3-Klienten zu codieren, der mit den bereits bestehenden konkurrierte? Das kam natürlich nicht in Frage und war auch nicht nötig. Ich erforschte sorgfältig die POP-Utilities, die ich zur Hand hatte und stellte mir die Frage: **Welcher davon kommt meinen Anforderungen am nächsten?** Denn was gilt, ist dies:

> *2. Gute Programmierer wissen, welchen Code sie schreiben sollen. Großartige Programmierer wissen, welchen Code sie umschreiben (und recyceln) können.*

Obwohl ich nicht behaupte, ein großartiger Programmierer zu sein, bemühe ich mich immer, einen zu imitieren.

Ein wichtiger Charakterzug großartiger Programmierer ist konstruktive Faulheit. Sie alle wissen, dass man sehr gute Noten nicht durch sehr großen Aufwand, sondern durch sehr gute Resultate bekommt -- und dass es fast immer einfacher ist, bei einer guten Teillösung anzufangen als ganz von vorne.

Linus Torvalds zum Beispiel versuchte erst gar nicht, Linux ohne grundlegenden Code auf der grüne Wiese zu erstellen. Stattdessen borgte er Code und Ideen von Minix, einem Unix-ähnlichen Betriebssystem für PC-Clones. Schließlich war aller Minix-Code durch neu entwickelten Code ersetzt oder komplett umgeschrieben -- aber solange er präsent war, lieferte er ein Gerüst für den Säugling, der schließlich zu Linux wurde.

Nach der gleichen Philosophie suchte ich nach einem bestehenden POP-Utility, das ausreichend gekonnt geschrieben war und als Grundlage für meine Weiterentwicklung dienen konnte.

Die Tradition der Weitergabe von Software der Unix-Welt war dem Recyceln von Code gegenüber immer sehr wohlwollend und aufgeschlossen eingestellt (was auch der Grund dafür ist, dass das GNU-Projekt Unix als sein Basis-Betriebssystem gewählt hat - und das trotz starker Vorbehalte gegenüber Unix selbst). Die Linux-Welt hat diese Tradition bis fast an die Grenzen des technisch Möglichen weitergeführt und stellt Terabytes an offen gelegtem Sourcecode zur Verfügung. Daher ist es in der Linux-Welt am wahrscheinlichsten, bei der Arbeit an eigenen Projekten auf ausreichend guten Source-Code zu stoßen, als irgendwo sonst.

Bei meinem Projekt funktionierte es. Zusammen mit den schon vorher gefundenen Programmen hatte ich nach meiner zweiten Suche insgesamt neun Kandidaten -- `fetchpop`, `PopTart`, `get-mail`, `gwpop`, `pimp`, `pop-perl`, `popc`, `popmail` und `upop`. Als erstes entschied ich mich für `fetchpop` von *Seung-Hong Oh*. Ich fügte meinen Code zum automatischen Umschreiben von Headern ein und nahm eine Reihe anderer Verbesserungen vor, die der Autor in seine Release 1.9 aufnahm.

Ein paar Wochen später aber stolperte ich über den Code für popclient von *Carl Harris* und fand heraus, dass ich ein Problem hatte. Obwohl `fetchpop` ein paar gute und originelle Ideen implementierte (wie etwa seinen Daemon-Mode), konnte es nur POP3 bedienen und war auch sehr laienhaft geschrieben (*Seung-Hong* war damals ein brillanter, aber unerfahrener Programmierer und beides konnte man an `fetchpop` sehr gut erkennen). *Carls* Code war besser, sehr professionell und solide, aber dem Programm fehlten einige wichtige und knifflig zu implementierende Leistungsmerkmale, die `fetchpop` sehr wohl hatte (darunter auch die von mir geschriebenen).

Würde es sich lohnen zu wechseln? Wenn ich wechselte, müsste ich meinen schon geschriebenen Code verwerfen und würde dafür eine bessere Ausgangsbasis gewinnen.

Ein praktischer Anreiz dafür war die Unterstützung mehrerer Protokolle. POP3 ist das üblichste Protokoll für Post Office Server, aber nicht das einzige. `Fetchpop` und seine Mitbewerber konnten kein POP2, RPOP oder APOP, und ich hatte bereits vage Pläne für einen Zusatz für IMAP (Internet Message Access Protocol, das neueste und mächtigste Post Office-Protokoll), das ich nur so zum Spaß implementieren wollte.

Es gab aber noch einen - theoretischeren - Anreiz, einen Wechsel zu `popclient` zu erwägen. Das hatte mit etwas zu tun, das ich schon lange vor Linux gelernt hatte.

> *3. "Plane eines für den Papierkorb; auf die eine oder andere Art machst du das sowieso." (Frederick P. Brooks, "Vom Mythos des Mann-Monats", Kapitel 11, in Addison-Wesleys Übersetzung von Arne Schäpers und Armin Hopp).*

Anders gesagt: oft versteht man das Problem gar nicht richtig, bevor man nicht die erste Implementation einer Lösung wenigstens halbwegs vollendet hat. Beim zweiten Mal hat man aber vielleicht schon genug gelernt, um es dann richtig zu machen. Wenn man also eine gute Implementation wünscht, sollte man sich darauf gefasst machen, wenigstens einmal ganz von vorne anzufangen [JB].

Nun, so sagte ich mir, meine Verbesserungen an `fetchpop` waren mein erster Versuch. Auf zu `popclient`.

Nachdem ich am 25. Juni 1996 meine ersten Patches für `popclient` an *Carl Harris* geschickt hatte, fand ich heraus, dass er schon einige Zeit vorher jedes Interesse an seinem Programm verloren hatte. Der Code war ein wenig verstaubt und enthielt kleinere Fehler. Ich musste viele Änderungen machen und wir kamen schnell überein, dass es das logischste wäre, wenn ich das Programm einfach übernehmen würde.

Ohne dass ich es wirklich gemerkt hatte, war das Projekt eskaliert. Ich dachte nicht mehr einfach über kleine Patches für einen bestehenden POP-Klienten nach. Ich übernahm die Wartung eines ganzen Programmpakets und in meinem Kopf nahmen Ideen Formen an, von denen ich wusste, dass sie zu weitreichenden Umstellungen führen würden.

In einer Software-Kultur, die zur gemeinsamen Nutzung von Code ermuntert, ist das der natürliche Weg, auf dem sich Projekte weiterentwickeln. Ich tat nichts anderes, als folgende Regel zu leben:

> *4. Mit der richtigen Einstellung werden interessante Probleme dich finden.*

Aber *Carl Harris* Haltung war sogar noch wichtiger. Er verstand, dass

> *5. Sobald man das Interessen an seinem Programm verliert, ist es die letzte Pflicht, es einem kompetenten Nachfolger zu überlassen.*

Ohne es jemals diskutiert haben zu müssen, wussten *Carl* und ich, dass wir eine gemeinsame Vorstellung von der besten Lösung hatten. Die einzige Frage war für jeden von uns beiden, ob ich meine Qualifikation dafür beweisen könnte. Sobald ich das getan hatte, handelte er großzügig und gelassen. Ich hoffe, dass ich es auch so gut machen werde, sobald die Zeit gekommen ist.

3 Von der Wichtigkeit, Benutzer zu haben

Und so erbte ich `popclient`. Ebenso wichtig war, dass ich `popclient`s Benutzerstamm erbte. Benutzer sind etwas wunderbares, und nicht nur, weil sie deutlich vor Augen führen, dass man einem Bedarf nachkommt und etwas richtig gemacht hat. Gut kultiviert, können sie zu Mit-Entwicklern werden.

Eine weitere Stärke der Unix-Tradition, die von Linux unbekümmert bis auf die Spitze getrieben wird, ist, dass viele Benutzer gleichzeitig auch Hacker sind. Da der Quellcode verfügbar ist, können sie zu sehr effektiven Hackern werden. Das fördert das prompte Entfernen von Bugs sehr. Mit ein bisschen Ermunterung werden Ihre Anwender Probleme diagnostizieren, entsprechende Änderungen vorschlagen und bei der Verbesserung des Codes in einer Weise mitwirken, die Sie alleine nie zustande bringen könnten.

> *6. Die Anwender als Mit-Entwickler zu sehen ist der Weg zu schnellen Verbesserungen und Fehlerbehebungen, der die geringsten Umstände macht.*

Die Durchschlagskraft dieser Erscheinung unterschätzt man leicht. Tatsächlich ist es so, dass so gut wie alle von uns in der Open Source-Welt drastisch unterschätzt haben, wie gut diese Kraft mit der Anzahl der Anwender und gegen die Systemkomplexität skaliert, bis *Linus Torvalds* uns darauf hingewiesen und es demonstriert hat.

Und tatsächlich denke ich, dass Linus' cleverster und ausschlaggebendster Hack nicht der Linux-Kernel selbst war, sondern die Erfindung des Linux-Entwicklungsmodells. Als ich diese Meinung einmal in seiner Gegenwart äußerte, grinste er und wiederholte etwas, das er schon oft gesagt hatte:

> Ich bin ein sehr fauler Mensch, der sich gerne mit fremden Federn schmückt und anderer Leute Lorbeeren erntet.

Ein echtes Faultier; oder, wie der Science Fiction-Autor Robert Heinlein es einmal für eine seiner Figuren formulierte:

> zu faul, um zu versagen.

Im Rückblick fällt einem ein Vorläufer der Methode und des Erfolges von Linux ein -- die Entwicklung der GNU Emacs Lisp-Bibliothek und Lisp Code-Archive. Im Gegensatz zum Stil der Kathedrale, in dem der Emacs C-Kern und die meisten anderen FSF-Tools entwickelt wurden, war die Evolution des Lisp-Codepools sehr fließend und hatte die Anwender als ihre treibende Kraft. Ideen und Prototypen wurden oft drei- oder viermal umgeschrieben, bevor sie ihre endgültige Form bekamen. Und lose verbundene Zusammenarbeit über das Internet - a la Linux - gab es sehr oft.

Tatsächlich war mein erfolgreichster einzelner Hack vor `fetchmail` wahrscheinlich der Emacs VC mode, eine Linux-ähnliche Zusammenarbeit über E-Mail mit drei anderen Leuten, von denen ich nur einen (Richard Stallman, Autor von Emacs und Gründer der FSF) bis heute getroffen habe. Es war ein Front End für SCCS, RCS und später CVS für Emacs, das Operationen zur Versionskontrolle **auf Knopfdruck** ermöglichte. Es entwickelte sich aus einem winzigen, groben sccs.el-Mode, den jemand anderer geschrieben hatte. Die Entwicklung von VC wurde ein Erfolg, weil - anders als Emacs selbst - der Emacs Lisp-Code die einzelnen Generationen der Release/Test/Verbesserung sehr schnell durchlief.

4 Früh freigeben, oft freigeben

Frühe und häufige Freigaben sind ein wichtiger Bestandteil des Linux-Entwicklungsmodells. Die meisten Entwickler (darunter auch ich) glaubten früher, dass das eine schlechte Politik für nicht-triviale Projekte wäre, weil frühe Versionen fast per Definition viele Bugs haben und man ja nicht die Geduld seiner Anwender überstrapazieren will.

Diese Auffassung verstärkte das allgemeine Festhalten an einem kathedralenartigen Stil bei der Entwicklung. Wenn die erste Zielsetzung war, dass Benutzer so wenige Fehler wie möglich sehen sollten, warum dann nur alle sechs Monate (oder noch weniger häufig) eine Release durchführen und wie ein Pferd am Debugging schuften? Der Emacs C-Kern wurde in dieser Weise entwickelt. Bei der Lisp-Bibliothek war das anders -- da es beliebte Lisp-Archive außerhalb der Kontrolle der FSF gab, konnte jeder neuesten und noch in Entwicklung befindlichen Code finden, der von Emacs' Freigabezyklus unabhängig war [QR].

Das wichtigste dieser Archive, das Ohio State elisp Archive, nahm die Philosophie und viele der Merkmale der heutigen großen Linux-Archive vorweg. Aber wenige von uns dachten sehr angestrengt darüber nach, was dort eigentlich vorging oder was die bloße Existenz dieser Archive für die Probleme der FSF mit dem Kathedralenmodell bedeuten könnte. Ich unternahm um 1992 herum einen ernsthaften Versuch, einen Großteil des Ohio-Codes formal mit der offiziellen Emacs Lisp-Bibliothek zu verschmelzen. Ich bekam politische Probleme und war nicht sehr erfolgreich.

Aber nur ein Jahr später, als Linux bereits einige Breitenwirkung entfaltet hatte, war klar, dass dort etwas anderes und viel gesünderes vorging. *Linus'* Politik der für alle offenen Entwicklung war das exakte Gegenteil des Kathedralen-Stils. Die Sunsite- (heute Metalab) und tsx-11-Archive boomten, mehrere Distributionen wurden veröffentlicht. Und all das wurde durch eine unerhörte Frequenz von Kernsystemfreigaben angetrieben.

Linus behandelte Anwender als Mit-Entwickler, und das in der effektivsten nur möglichen Weise.

> *7. Früh freigeben. Oft freigeben. Seinen Anwendern zuhören.*

Linus' Innovation war nicht so sehr, dass er es so machte (das war ja schon seit langer Zeit so die Tradition in der Welt von Unix), sondern bestand darin, es bis zu einer Intensität hin zu betreiben, die der Komplexität seines Unternehmens angemessen war. In jenen frühen Tagen (um 1991) war es für ihn nicht ungewöhnlich, mehr als einen neuen Kernel pro Tag freizugeben. Da er seinen Stamm von Mit-Entwicklern gut kultiviert hatte und das Internet weit ausgiebiger für die Zusammenarbeit nutzte als irgendjemand sonst, funktionierte es auch.

Aber wie funktionierte es? Und war es eine Methode, die ich nachahmen konnte oder beruhte sie auf einem einzigartigen Talent von *Linus Torvalds*?

Das glaubte ich nicht. Sicher, *Linus* ist ein verdammt guter Hacker (wie viele von uns könnten einen ganzen Betriebssystem-Kernel mit Produktqualität bauen?). Aber an Linux war nichts, was einen atemberaubenden Quantensprung dargestellt hätte. *Linus* ist kein (oder wenigstens noch kein) genialer Innovator wie zum Beispiel Richard Stallman oder James Gosling (NeWS und Java) welche sind. Stattdessen erscheint er mir als ein eminent begabter Ingenieur, der mit einem sechsten Sinn für das Vermeiden von Programmierfehlern und Entwicklungssackgassen und mit einem wirklichen Talent für das Auffinden des Wegs des geringsten Aufwandes ausgestattet ist. Das ganze Design von Linux atmet diese Qualitäten und spiegelt *Linus*' grundsätzlich konservativen und vereinfachenden Entwicklungsansatz wider.

Wenn also rasch aufeinander folgende Releases und die kompromisslose Nutzung des Internets nicht zufällig, sondern integraler Bestandteil von *Linus*' Ingenieurstalent und seiner profunden Einsicht in den Weg des geringsten Aufwandes waren, was maximierte er dann? Was war sein Beitrag zum Code schaffenden Ameisenhaufen?

In dieser Weise gestellt, beantwortet sich die Frage von selbst. *Linus* stimulierte und belohnte seine Hacker/Anwender ununterbrochen -- stimulierte durch die Aussicht auf Ego-pflegende Urheberschaft und Anteil an der Bewegung, belohnte durch Vorzeigen ständiger (sogar täglicher) Verbesserung des großen Werks.

Linus zielte direkt auf die Maximierung der Anzahl der Mannstunden, die für das Debugging und die Entwicklung aufgewendet wurden, und ließ es sogar darauf ankommen, dass sich möglicherweise Instabilitäten in den Code schleichen und die Benutzerbasis ausgebrannt werden konnte, falls sich irgendein Fehler als unbehebbar herausstellen sollte. *Linus* verhielt sich, als würde er ungefähr an folgendes glauben:

> *8. Wenn man einen ausreichend großen Stamm an Betatestern und Mit-Entwicklern hat, wird jedes Problem schnell identifiziert und die Lösung jedem offensichtlich sein.*

Oder, salopper ausgedrückt:

> **Alle Bugs sind trivial, wenn man nur genügend Entwickler hat** ("Given enough eyeballs, all bugs are shallow"). Ich nenne das ***Linus*' Gesetz**.

Meine ursprüngliche Formulierung war, dass jedes Problem wenigstens für **irgend jemanden offensichtlich sein wird**. *Linus* wendete ein, dass die Person, die das Problem versteht und behebt, nicht notwendigerweise oder nicht einmal üblicherweise dieselbe ist, die das Problem als erstes charakterisiert hat. **Jemand entdeckt das Problem**, so *Linus*, **und jemand anderer versteht es. Und ich unterschreibe jederzeit, dass das Auffinden der schwierigere Teil davon ist.** Der springende Punkt hier ist aber, dass beides in der Regel sehr schnell passiert.

Hier haben wir, so denke ich, den grundlegenden Unterschied zwischen den Erbauern einer Kathedrale und dem Stil des Basars. Nach Auffassung der Erbauer der Kathedrale sind Programmierfehler und Entwicklungsprobleme knifflige, tief gehende und heimtückische Erscheinungen. Es dauert Monate der Analyse durch eine entschlossene Elite, um Zuversicht in die Fehlerfreiheit des Codes zu bilden. Daher die langen Intervalle zwischen den Freigaben und die langen Gesichter, wenn eine lang erwartete Release nicht fehlerfrei ist.

Die Auffassung hinter dem Basarstil ist eine ganz andere. Man geht davon aus, dass Bugs ein sehr triviales Phänomen sind -- oder wenigstens eines, das sehr schnell trivial werden kann, wenn es tausend begeisterten Mit-Entwicklern ausgesetzt wird, die nach jeder neuen Release darauf herum trampeln. Dementsprechend führt man sehr oft Freigaben durch, um so zu mehr Korrekturen zu kommen, und ein nützlicher Nebeneffekt ist, dass man weniger zu verlieren hat, sollte gelegentlich eine besonders verunglückte zur Tür hinaus gelangen.

Das ist alles. Das ist genug. Wenn ***Linus*' Gesetz** falsch ist, dann hätte jedes System, das so komplex ist wie der Linux-Kernel und das von so vielen Händen überarbeitet wird, ab irgendeinem Zeitpunkt unter dem Gewicht unvorhergesehener und schädlicher Interaktionen zusammenbrechen müssen -- und was anderes sind unentdeckte, **tiefe** Bugs? Wenn *Linus*' Gesetz aber stichhaltig ist, dann reicht es aus, um die relative Fehlerfreiheit von Linux zu erklären und auch, dass es monatelang oder sogar jahrelang ununterbrochen laufen kann.

Vielleicht sollte das ja gar keine so große Überraschung sein. Soziologen haben schon vor Jahren herausgefunden, dass die gemittelte Meinung einer Masse von gleich kompetenten (oder gleich dummen) Beobachtern etwas zuverlässigere Vorhersagen macht als die eines einzelnen willkürlich herausgepickten Beobachters. Sie nennen das den **Delphi-Effekt**. Es scheint, dass es das ist, was *Linus* gezeigt hat: Diese Erscheinung lässt sich sogar auf das Debuggen eines Betriebssystems anwenden -- der Delphi-Effekt kann sogar die Komplexität eines Entwicklungsprojektes zähmen, sogar wenn sie so hoch ist wie die für einen OS-Kernel.

Ein spezielles und dem Delphi-Effekt sehr förderliches Merkmal der Situation, in der sich Linux befindet, ist der Umstand, dass die Teilnehmer an einem Projekt sich praktisch selbst auswählen. Einer der Teilnehmer der ersten

Stunde wies darauf hin, dass die Beiträge nicht einfach von irgendwelchen Freiwilligen kommen, sondern von Menschen, die ausreichend an der Software interessiert sind, um ihre Funktionsweise und Aufbau zu erforschen, Lösungen für vorgefundene Probleme zu finden und dann tatsächlich eine halbwegs brauchbare Verbesserung zu entwickeln. Jeder, der alle diese Filter durchlaufen hat, wird sehr wahrscheinlich fähig sein, einen echten Nutzen zu stiften.

Meinem Freund *Jeff Dutky* verdanke ich den Hinweis, dass *Linus*' Gesetz auch zu **Debugging lässt sich parallelisieren** umformuliert werden kann. *Jeffs* Beobachtung ist, dass obwohl Debugging vom Debuggenden wenigstens ein bisschen Kommunikation mit einem koordinierenden Entwickler erfordert, es doch keine bedeutende Koordination der Debuggenden untereinander notwendig macht. Daher ist der Gesamtaufwand nicht anfällig für das quadratische Wachstum der Komplexität und Managementkosten, die das Mehr an Entwicklern problematisch macht.

In der Praxis ist der theoretisch mögliche, doppelte Aufwand durch einander überlappende Arbeit von debuggenden Entwicklern in der Linux-Welt fast gar kein Thema. Ein Effekt des **früh freigeben, oft freigeben** ist die rasche Weitergabe und das rasche Rückspeisen von Verbesserungen, was solche Überlappungen minimiert.

Mehr Anwender finden mehr Bugs, weil das Mehr an Benutzern auch ein Mehr an Möglichkeiten bedeutet, das Programm zu verwenden und zu belasten. Dieser Effekt wird noch verstärkt, wenn die Benutzer gleichzeitig Mit-Entwickler sind. Jeder hat individuelle Ansätze, Auffassungen und analytische Werkzeuge, wenn es um die Betrachtung eines Problems geht. Der **Delphi-Effekt** scheint wegen eben dieser Vielfältigkeit gut zu funktionieren. Im speziellen Kontext des Debuggens vermindert so eine breite Palette auch noch das Duplizieren von Aufwand.

Mit anderen Worten, ein Mehr an Betatestern mag die Kniffeligkeit des augenblicklich **tiefstsitzenden** Bugs vom Standpunkt des Entwicklers aus nicht reduzieren, aber das Mehr erhöht die Wahrscheinlichkeit, dass irgend jemandes Methoden an das Problem dermaßen gut angepasst sind, um dieser Person trivial zu erscheinen.

Aber auch *Linus* will nicht alles auf eine Karte setzen. Für den Fall, dass es tatsächlich schwer zu behebenden Fehlern gibt, sind die Versionen des Linux-Kernels so nummeriert, dass potentielle Benutzer die Wahl haben, entweder die letzte offiziell **stabile** Version zu verwenden, oder aber an die vorderste Front der Entwicklung zu gehen und für neue Features das Risiko von Fehlern einzugehen. Diese Taktik wird von den meisten Linux-Hackern offiziell noch nicht imitiert -- vielleicht sollten sie das aber tun, denn eine Auswahl zu haben, macht jede der beiden Möglichkeiten für sich attraktiver.

5 Wann ist eine Rose keine Rose?

Nach dem Befassen mit *Linus*' Verhalten und dem Entwickeln einer Theorie darüber, warum es so erfolgreich ist, traf ich eine bewusste Entscheidung, um diese Theorie an meinem neuen (und zugegebenermaßen weniger komplexen und weniger ehrgeizigen) Projekt zu testen.

Das erste, was ich aber machte, war, `popclient` zu vereinfachen und zu reorganisieren. *Carl Harris*' Implementation war robust und solide, zeigte aber eine Art von unnötiger Kompliziertheit, die für viele C-Programmierer charakteristisch ist. Für ihn war der Code das Zentrale, und die Datenstrukturen waren die Unterstützung für den Code. Das führte zu sehr schön gestaltetem Code, aber zu sehr improvisierten und unansehnlichen Datenstrukturen (wenigstens nach den hohen Standards dieses alten LISP-Hackers).

Neben dieser Verbesserung des Designs hatte ich aber noch einen anderen Nutzen im Auge, als ich den Code umschrieb. Die erste tief greifende Umstellung war, dass ich Unterstützung für IMAP einbaute. Ich schrieb die Protokoll-Automaten in einen generischen Treiber plus drei Tabellen mit Methoden um (jeweils eine für POP2, POP3 und IMAP). Diese und die vorhergehenden Änderungen illustrieren ein allgemeines Prinzip, das Programmierer im Auge behalten sollten, speziell bei Sprachen, die dynamisches Typisieren nicht von Haus aus unterstützen:

> *9. Smarte Datenstrukturen und dummer Code funktionieren viel besser als umgekehrt.*

Brooks, Kapitel 9: **Also zeige mir Deinen [Code], aber verhülle Deine [Datenstrukturen], und ich werde auf ewig im Dunkeln tappen. Zeige mir aber Deine [Datenstrukturen] und ich werde Deinen [Code] gar nicht brauchen, denn ich weiß, wie er aussieht.**

Eigentlich redete er von **Flussdiagrammen** und **Tabellen**, aber unter Beachtung der dreißig Jahre, die seither vergangen sind, ist die kleine Anpassung zulässig.

Zu diesem Zeitpunkt (Anfang September 1996, sechs Wochen nach der Stunde 0 des Projekts), dachte ich, dass eine Namensänderung angebracht sei -- immerhin war das Programm kein reiner POP-Klient mehr. Ich zögerte aber, weil es noch keine wirklich neuen Erweiterungen im Design gab. Meine Fassung von `popclient` musste erst eine eigene Identität entwickeln.

Das änderte sich aber, und zwar radikal, als `fetchmail` lernte, wie man abgeholte Mail an das SMTP-Port weiterreicht. Ich werde das gleich näher erläutern, vorher aber noch Folgendes: Ich habe meine Entscheidung schon erwähnt, dieses Projekt als Test für meine Hypothese zu verwenden, die ich mir über *Linus Torvalds*' Erfolg gebildet hatte. Was genau bedeutet das? Die Frage ist berechtigt, hier also, was ich tat:

1. Ich veröffentlichte früh und häufig Freigaben (fast immer wenigstens alle zehn Tage; während der Zeiten intensiver Entwicklung eine pro Tag).
2. Ich fügte meiner wachsenden Beta-Liste jeden hinzu, der mich zu `fetchmail` kontaktierte.
3. Ich verschickte ausführliche und in lockerem Tonfall gehaltene Ankündigungen, wann immer ich eine Freigabe machte, und ermunterte meine Leute, am Prozess teilzuhaben.
4. Ich hörte auf meine Betatester und befragte sie regelmäßig zu Design-Entscheidungen und lobte sie wann immer sie Patches oder Anregungen lieferten.

Diese simplen Maßnahmen zahlten sich unmittelbar aus. Vom Anfang des Projekts an bekam ich Bugreports von einer Qualität, für die die meisten Entwickler alles und ihren linken Arm hergeben würden; oft waren auch gute Korrekturen beigelegt. Ich bekam konstruktive Kritik zu hören, Fanpost und wohldurchdachte Anregungen für neue Features. Das führt uns zu:

> *10. Wenn man seine Betatester wie die wertvollste Ressource behandelt, werden sie als Reaktion darauf zur*

wertvollsten Ressource werden.

Eine interessante Maßnahme hinter `fetchmail`s Erfolg ist die schiere Größe der Beta-Liste, die `fetchmail`-friends. Im Augenblick der Abfassung dieses Textes enthält sie 249 Mitglieder und wächst jede Woche um zwei oder drei.

Tatsächlich ist die Liste zum Zeitpunkt der Überarbeitung im Mai 1997 von einem Zenit von 300 wieder geschrumpft, und das aus einem interessanten Grund. Mehrere Leute haben mich gebeten, sie von der Liste zu nehmen, weil `fetchmail` so gut funktioniert, dass es für sie keine Veranlassung mehr gibt, am Gedankenaustausch teilzunehmen! Dies ist vielleicht ein für reife Basar-artige Projekte normaler Lebensabschnitt.

6 Aus popclient wird fetchmail

Der wirkliche Wendepunkt im Projekt kam, als *Harry Hochheiser* mir seinen von `fetchmail` unabhängigen Code für das Weiterreichen von Mail an das SMTP-Port der Klientenmaschine schickte. Ich erkannte fast augenblicklich, dass eine zuverlässige Implementation dieses Leistungsmerkmals alle anderen Modi der Zustellung fast überflüssig machen würde.

Viele Wochen lang hatte ich an `fetchmail` und an einer kleinen Verbesserung nach der anderen gebastelt und hatte dabei den Eindruck, dass das Schnittstellen-Design zwar leicht zu handhaben, aber nicht ganz sauber durchdacht sei -- ohne Eleganz und mit zu vielen geringfügigen Optionen, die überall heraushängten. Die Optionen für das Umleiten von abgeholter Mail in einer Mailbox-Datei oder zur Standardausgabe war mir ein besonderer Dorn im Auge, ich konnte aber nicht herausfinden, warum.

Als ich mir die Sache mit dem SMTP-Forwarding überlegte, erkannte ich das Problem. `Popclient` versuchte zu viele Dinge auf einmal. Er wurde entworfen und gebaut, um sowohl als Mail Transport Agent (MTA) als auch als Local Mail Delivery Agent (MDA) aufzutreten. SMTP-Forwarding würde ihn aus dem MDA-Geschäft hinausbugsieren und ihn zu einem reinen MTA machen, der Mail für die lokale Zustellung an andere Programme weiterreicht -- so wie sendmail das tut.

Warum eigentlich, so meine selbstgestellte Frage, sollte ich mich mit der ganzen Komplexität der Konfiguration eines Mail Delivery Agent plagen oder lock-and-append für eine Mailbox aufbauen, wenn doch Port 25 fast garantiert auf jeder Plattform existiert, die TCP/IP unterstützt? Speziell wenn dies bedeutete, dass die abgeholte Mail garantiert wie gewöhnliche sender-initiated SMTP-Mail aussehen würde - was eigentlich das ist, was wir wirklich wollen.

Hier kann man einige Lektionen lernen. Die erste: Diese Idee mit dem Weiterreichen von SMTP-Mail war die lohnendste Einzelleistung, in deren Genuss ich durch bewusstes Nachahmen von *Linus*' Methoden kam. Diese glänzende Idee kam von einem Benutzer -- alles, was ich tun musste war, deren Auswirkungen zu verstehen.

> *11. Das zweitbeste nach eigenen guten Ideen ist das Erkennen von guten Ideen von Benutzern. Manchmal ist letzteres sogar das bessere.*

Paradoxerweise werden Sie schnell herausfinden, dass, wenn Sie total selbstlos die ganze Wahrheit darüber offenbaren, wieviel Sie anderen Menschen verdanken, Sie die ganze Welt behandeln wird, als hätten Sie jede einzelne Erfindung höchstpersönlich gemacht und würden einfach nur die natürliche Bescheidenheit des wahren Genies zeigen. Wir haben alle gesehen, wie gut das für *Linus* funktionierte!

(Als ich auf der perl-Konferenz im August 1997 meine Rede hielt, saß Larry Wall in der ersten Reihe. Als ich zur eben aufgeschriebenen Zeile kam, ereiferte er sich im gespielten Tonfall eines Predigers:

> Verkünde es, Bruder, verkünde es!.

Das ganze Publikum lachte, denn jeder wusste, dass sie auch für den Erfinder von perl funktioniert hatte.)

Nach einigen wenigen Wochen in diesem Geiste begann ich in den Genuss von ähnlichem Lob zu kommen -- nicht nur von Benutzern, sondern auch von Leuten, die von meinem Projekt erfahren hatten. Ich habe einige von den E-Mails aufgehoben; eines Tages werfe ich vielleicht noch einen Blick darauf, falls ich mich fragen sollte, ob mein Leben einen Sinn gehabt hat :-).

Es gibt hier aber zwei fundamentalere, nicht-politische Lektionen zu lernen, die für alle Arten von Design gelten.

> *12. Oft stammen die hervorragendsten und innovativsten Lösungen aus der Erkenntnis, dass die ganze*

Vorstellung vom Problem falsch war.

Ich hatte versucht, das falsche Problem zu lösen, als ich fortfuhr, `popclient` als einen kombinierten MTA/MDA zu entwickeln, der alle möglichen Modi der lokalen Zustellung unterstützte. `Fetchmail`s Design musste von Grund auf neu überdacht und als reiner MTA gesehen werden, und als Teil des üblichen SMTP-sprechenden Internet-Postweges.

Nun, so kam ich zu einer Neufassung meines Problems. Klar war, dass (1) Unterstützung für SMTP-Forwarding in den generischen Treiber dazugehackt werden musste, (2) dies der Standardmodus werden musste, und (3) schließlich alle anderen Modi der Zustellung hinaus gehörten, besonders die Möglichkeit, Mail in eine Datei oder zur Standardausgabe umzuleiten.

Bei Schritt 3 zögerte ich für einige Zeit. Ich hatte Angst, alte `popclient`-Benutzer zu vergrämen, die von alternativen Zustellungsmechanismen abhängig waren. Theoretisch hätten sie sofort zu .forward-Dateien oder deren Non-sendmail-Äquivalenten wechseln können, praktisch aber war dieser Wechsel ein Alptraum.

Als ich mich dazu entschied, stellten sich die Vorzüge als riesig heraus. Die haarigsten Teile des Treiber-Codes verschwanden einfach. Die Konfiguration wurde radikal leichter -- kein Stöbern mehr nach der System-MDA und der Mailbox des Benutzers, keine Sorgen mehr darüber, ob das zugrunde liegende Betriebssystem das file locking unterstützt.

Auch verschwand die einzige Möglichkeit, Mail zu verlieren. Wenn man Zustellung in eine Datei spezifizierte und die Disk voll war, verschwand die Mail. Das konnte bei SMTP-Forwarding nicht mehr passieren, da der SMTP-Listener einfach kein OK gab, solange die Nachricht nicht zugestellt werden oder wenigstens für spätere Zustellung gespoolt werden konnte.

Auch der Durchsatz erhöhte sich (obwohl man das in einem einzigen Durchlauf wahrscheinlich nicht bemerken könnte). Ein weiterer, nicht unbedeutender Nutzen dieser Änderung war, dass die Manpage wesentlich simpler wurde.

Später musste ich einen benutzerspezifizierten lokalen MDA wieder zum Funktionieren bringen, um einige obskure Situationen abzudecken, die Dynamic SLIP involvierten. Ich fand aber eine viel schlichtere Methode dafür.

Die Moral der Geschichte? Zögern Sie nicht, überholte Features aufzugeben, wenn die Effektivität davon unbeeinflusst bleibt. Antoine de Saint-Exupéry (der Pilot und Flugzeugdesigner war, wenn er nicht mit dem Schreiben von klassisch gewordenen Kinderbüchern beschäftigt war) sagte einmal:

> *13. "Perfektion (im Design) ist nicht erreicht, wenn es nichts mehr hinzuzufügen gibt, sondern wenn es nichts mehr wegzunehmen gibt."*

Wenn Ihr Code sowohl besser als auch einfacher wird, dann wissen Sie, dass Sie es richtig gemacht haben. Dadurch bekam `fetchmail` seine eigene Identität und löste sich von seinem Vorfahren `popclient`.

Es wurde Zeit für eine Namensänderung. Das neue Design sah mehr nach einer Entsprechung von `sendmail` aus als der alte `popclient`; beide sind MTAs, aber so wie `sendmail` die Zustellungen fortschickt, so fängt sie der neue `popclient` ein. Daher taufte ich ihn in `fetchmail` um.

Es gibt hier aber eine noch allgemeiner anwendbare Lektion darüber, wie SMTP-Zustellung ein `fetchmail`-Feature wurde. Es ist die, dass nicht nur Debugging parallelisierbar ist -- auch die Entwicklung ist es und (in einem vielleicht überraschenden Ausmaß) auch die Erforschung des Designraumes. Wenn der Entwicklungsmodus mit sehr kurzen Iterationszyklen arbeitet, werden Entwicklung und Erweiterung zu Spezialfällen des Debuggings -- zu einer Beseitigung von **Fehlern, die Auslassungen sind** - Auslassungen im ursprünglichen Konzept der Software.

Sogar auf höheren Ebenen des Entwurfs kann es sehr wertvoll sein, das Denken vieler Mit-Entwickler in den Design-Raum eines Produkts ausschwärmen zu lassen. Stellen Sie sich dazu vor, wie eine Wasserpfütze einen Abfluss findet, oder noch besser, wie Ameisen Essen finden: Erforschung durch Einsickern, gefolgt von einer Diskussion, die über skalierbare Kommunikationsmittel geführt wird. Das funktioniert sehr gut; wie bei *Harry Hochheiser* und mir, wird einer Ihrer Späher einen riesigen Gewinn in Ihrer Reichweite entdecken, den Sie aufgrund eines vernagelten Blickfeldes nicht gesehen hatten.

7 Fetchmail wird erwachsen

Da war ich also mit einem eleganten und innovativen Design, einem Code, von dem ich wusste, dass er gut funktionierte, da ich ihn jeden Tag verwendete, und einer lebhaft wachsenden Liste von Betatestern. Nach und nach dämmerte mir, dass ich nicht mehr mit einem trivialen persönlichen Hack befasst war, der einigen Leuten nützlich sein konnte. Ich hatte meine Finger in einem Programm, dass jeder Hacker mit einer Unix-Maschine und einer SLIP/PPP-Mailverbindung wirklich braucht.

Durch das Leistungsmerkmal des SMTP-Forwarding zog `fetchmail` der Konkurrenz soweit davon, dass es realistisch wurde, in ihm einen baldigen **Kategorienkiller** zu sehen, eines jener klassischen Programme, die eine Nische so kompetent ausfüllen, dass Alternativen nicht einfach nur weggeworfen, sondern fast vergessen werden.

Ich denke, man kann ein solches Ergebnis nicht wirklich planen oder anstreben. Man muss von Designideen hineingezogen werden, die so mächtig sind, dass sie im Rückblick wie offensichtlich, natürlich oder sogar gottgewollt wirken. Die einzige Möglichkeit, so eine Idee zu finden, ist, eine Menge solcher Ideen zu haben -- oder das technische Urteilsvermögen, um die guten Ideen anderer Leute jenseits deren Vorstellungen weiterzuentwickeln.

Andy Tanenbaum hatte die ursprüngliche Idee, ein einfaches Unix für den IBM PC zu schaffen, der eigentlich ein Lehrbehelf war. *Linus Torvalds* hievte das Minix-Konzept weiter als *Andrew* es wahrscheinlich für möglich gehalten hätte -- und es wurde etwas ganz Wunderbares daraus. In der selben Weise (nur in kleinerem Maßstab) übernahm ich einige Ideen von *Carl Harris* und *Harry Hochheiser* und hetzte sie zu Höchstleistungen. Keiner von uns war **originell** in dem romantischen Sinne, in dem sich die Leute ein Genie vorstellen. Es ist aber so, dass die Wissenschaften, Ingenieurskünste und Softwareentwicklungen nicht von Genies weitergebracht werden, was immer anderes die Hackermythologie auch behauptet.

Die Resultate konnten sich trotzdem sehen lassen -- tatsächlich war es genau die Sorte Erfolg, für die jeder Hacker lebt! Und das bedeutete, dass ich meine Latte noch höher legen musste, um `fetchmail` so gut zu machen, wie ich nun sah, dass es werden konnte. Ich musste nicht nur für den eigenen Bedarf schreiben, sondern auch Leistungsmerkmale unterstützen, die für Menschen außerhalb meines Orbits wichtig wären. Und simpel und robust sollte mein Programm auch noch sein.

Das erste und mit Abstand wichtigste Leistungsmerkmal, das ich nach dieser Erkenntnis schrieb, war Multidrop-Unterstützung -- die Fähigkeit, Mail aus Mailboxes zu holen, die alle Mail für eine ganze Gruppe von Benutzern angesammelt hatte, und dann jede einzelne davon an die jeweiligen Empfänger zuzustellen.

Ich entschied mich für den Einbau von Multidrop-Unterstützung zum Teil deswegen, weil Benutzer es sehr nachdrücklich forderten, aber der Hauptgrund dafür war meine Überlegung, dass es einige Bugs aus dem Single Drop-Code herausrütteln würde, da es mich zwang, Adressierung in allgemeinster Form zu behandeln. Und genau so war es auch. Das Parsing RFC 822 konform zu bekommen kostete mich erstaunlich viel Zeit, die aber nicht für ein individuelles, besonders kniffliges Stück Code draufging, sondern weil es eine Menge Details gab, die alle voneinander abhängig waren und peinlich genaue Implementation erforderten.

Die Verbesserung lohnte sich aber; die Multidrop-Adressierung stellte sich als eine exzellente Design-Entscheidung heraus. Ich wusste danach auch, warum:

> *14. Jedes Tool sollte in der erwarteten Weise nützlich sein, aber wirklich großartige Tools bieten darüber hinaus unerwarteten Nutzen.*

Der unerwartete Nutzen von `fetchmail` mit Multidrop ist der, dass man damit eine Mailingliste mit Alias-Expansion betreiben kann -- und das von der Klientenseite der SLIP/PPP-Verbindung aus. Das bedeutet,

dass jemand mit einem ISP-Konto eine Mailingliste ohne Zugriff auf die Alias-Dateien des ISP (Anmerkung: Internet Service Provider) unterhalten kann.

Eine weitere bedeutende Änderung, die von Betatestern gefordert wurde, war die Unterstützung von 8 bit MIME-Betrieb. Das war relativ einfach, da ich mir Mühe gegeben hatte, den Code 8-bit-clean zu halten. Nicht dass ich die Nachfrage für dieses Feature vorhergesehen hätte, aber ich befolgte einfach eine andere Regel:

> *15. Beim Entwickeln von Gateway-Software jeglicher Art ist jeder Aufwand gerechtfertigt, um den Datenstrom so wenig wie möglich zu beeinflussen -- und man darf Information niemals wegwerfen, außer der Empfänger verlangt es so!*

Wenn ich diese Regel nicht befolgt hätte, wäre 8-bit MIME-Unterstützung schwierig und voller Fehler geworden. So aber war alles, was ich zu tun hatte, die RFC 1652 zu lesen und ein triviales Schnipsel von Header-generierender Logik einzubauen.

Einige europäische Benutzer nervten mich solange, bis ich eine Option einbaute, die die Anzahl der Nachrichten pro Verbindung begrenzte (so dass sie die Kosten für ihre teuren Telefonleitungen steuern konnten). Ich zierte mich lange Zeit und bin noch immer nicht ganz glücklich damit, aber wenn man für die ganze Welt Software schreibt, muss man auf seine Kunden hören -- das ändert sich auch dann nicht, wenn sie einem dafür nichts bezahlen.

8 Was wir von fetchmail sonst noch lernen können

Bevor wir zu den allgemeinen Belangen der Software-Entwicklung zurückkehren, wollen wir noch einige spezifische Lektionen der `fetchmail`-Erfahrung untersuchen.

Die rc-Syntax beinhaltet die optionalen **noise**-Schlüsselwörter, die vom Parser einfach ignoriert werden. Sie gestatten eine an Englisch angelehnte Syntax und sind bei weitem lesbarer als die traditionell knappen Schlüssel/Wert-Paare, die man erhält, wenn man diese **überflüssigen** Schlüsselwörter auslässt.

Diese Schlüsselwörter begannen ihre Karriere als spät nächtliche Experimente, als ich bemerkte, wie sehr die rc-Deklarationen bereits einer imperativen Mini-Programmiersprache ähnelten. (Aus diesem Grund änderte ich auch das ursprüngliche `popclient`-Schlüsselwort `server` zu `poll`).

Mir schien es, als würde eine Anpassung dieser imperativen Mini-Programmiersprache an gewöhnliches Englisch die Benutzung vereinfachen. Obwohl ich ein überzeugter Partisan für die **Mach eine Sprache daraus**-Schule des Designs bin (Beispiele dafür sind Emacs, HTML und viele Datenbankmaschinen), zweifle ich normalerweise an **Englisch-ähnlichen** Syntaxen.

Traditionellerweise haben Programmierer eine Tendenz, Steuerungssyntaxen zu bevorzugen, die sehr präzise und kompakt sind und keine Redundanzen beinhalten. Das ist ein kulturelles Erbe aus jener Zeit, als Computerressourcen teuer waren und die einzelnen Phasen des Parsing so einfach und billig wie möglich sein mussten. Englisch auf der anderen Seite, mit seiner 50-prozentigen Redundanz, sah damals nach einem sehr unzulänglichen Modell aus.

Das ist jedoch nicht der Grund, aus dem ich eine der englischen Sprache nahe Syntax normalerweise vermeide. Ich erwähne dieses Argument hier nur, um es zu entkräften. Mit den heutigen billigen CPU-Zyklen und Speicherbits sollte Kompaktheit kein Selbstzweck sein. Heute ist es für eine Programmiersprache wichtiger, für die Menschen bequem in der Handhabung zu sein, als billig für den Computer.

Es bleiben aber andere gute Gründe, um vorsichtig damit zu sein. Einer davon sind die Kosten für die Komplexität des Parsers -- man will die Komplexität ja nicht so weit aufblähen, dass sie eine bedeutende Quelle für Bugs und Verwirrung unter den Anwendern wird. Ein weiterer ist der, dass eine englisch-ähnliche Syntax von ihrem gesprochenen **Englisch** oft verlangt, dass es in eine groteske Form gepresst wird, die dann mit einer ethnischen Sprache nur mehr oberflächlich zu tun hat und so verwirrend ist wie eine traditionelle Syntax einer Programmiersprache (was man oft bei so genannten **Fourth Generation Languages** und kommerziellen Datenbankabfragesprachen beobachten kann).

Die Steuerungssyntax von `fetchmail` scheint diese Probleme zu umgehen, da die Sprache extrem eingeschränkt ist. Sie kommt nicht einmal in die Nähe einer Allzweck-Sprache, und die Dinge, die sich damit ausdrücken lassen, sind nicht sehr kompliziert. Das Potential für Verwirrung bei der Unterscheidung zwischen einer kleinen Teilmenge von Englisch und der eigentlichen Steuerungssprache ist daher gering. Ich glaube, es gibt hier eine breiter anwendbare Lektion zu lernen:

> *16. Wenn Ihre Programmiersprache in keiner Weise Turing-vollständig ist, können Sie sich mit syntaktischer Glasur anfreunden.*

Eine weitere Lehre betrifft Sicherheit durch Unsichtbarkeit (**security by obscurity**). Einige `fetchmail`-Anwender baten mich, die Software so zu ändern, dass Passwörter in der rc-Datei verschlüsselt werden, so dass Spione sie nicht so leicht entdecken könnten.

Ich folgte dieser Bitte nicht, denn es führt keineswegs zu einem Schutz. Jeder, der das entsprechende Privileg erhalten hat, Ihre rc-Datei zu lesen, wird `fetchmail` genau wie Sie aufrufen und verwenden können -- und

wenn es Ihr Passwort ist, hinter dem er her ist, wird er in der Lage sein, den Dekoder aus dem `fetchmail`-Quellcode selbst herauszulesen.

Alles, was verschlüsselte `.fetchmailrc`-Passwörter für Sie tun könnten, ist, Ihnen ein trügerisches Gefühl der Sicherheit zu vermitteln, wenn Sie nicht sehr angestrengt darüber nachdenken. Die allgemeine Regel hier ist:

> *17. Ein Sicherheitssystem ist nur so sicher wie seine Geheimnisse. Hüten Sie sich vor Pseudo-Geheimnissen.*

9 Voraussetzungen für den Basar-Stil

Frühe Kritiker und das Testpublikum für dieses Dokument fragten sehr oft nach den Voraussetzungen für eine erfolgreiche Basar-geführte Entwicklung, darunter sowohl Fragen nach der Qualifikation für den Projektleiter und den Zustand des Codes zum Zeitpunkt der Veröffentlichung für die Gemeinde der Mit-Entwickler.

Es sollte klar sein, dass man nicht von Null an im Stile des Basars entwickeln kann [IN]. Man kann am Basar testen, debuggen und verbessern, aber es wäre sehr schwer, ein Projekt im Basar-Modus zu beginnen. *Linus* versuchte das gar nicht erst, und ich auch nicht. Ihre keimende Entwicklergemeinde muss etwas Lauffähiges und Testbares haben, um damit spielen zu können.

Wenn man mit dem Aufbau der Gemeinde beginnt, benötigt man ein herzeigbares, überzeugendes Versprechen. Ihr Programm braucht nicht besonders gut zu funktionieren. Es kann sehr ungeschliffen, von Bugs geplagt, unvollständig und spärlich dokumentiert sein. Was es aber nicht verfehlen darf, ist, (a) zu laufen, und (b) potentielle Mit-Entwickler davon zu überzeugen, dass es sich in absehbarer Zukunft zu etwas wirklich ordentlichem entwickeln lässt.

Linux und `fetchmail` gingen mit starken und attraktiven Designs an die Öffentlichkeit. Viele Menschen haben hier vom Basarmodell, wie ich es präsentierte, die korrekte Auffassung, dass dies das allerwichtigste ist und zogen daraus den Schluss, dass für den Projektleiter ein hohes Maß an Intuition für gutes Design und viel Cleverness unentbehrlich sind.

Linus aber holte sich sein Design von Unix. Ich erbte meines vom Vorfahren `popclient` (obwohl es sich später noch ganz schön ändern sollte, und zwar überproportional im Vergleich zu Linux). Muss der Leiter/Koordinator eines Entwicklungsbasars wirklich über herausragendes Talent für Design verfügen oder kann er damit durchkommen, das Designtalent anderer zu nutzen?

Ich denke, dass es für den Koordinator nicht lebenswichtig ist, ein Design von atemberaubender Brillanz zu stiften, dass es aber für ihn lebenswichtig ist, ein gutes Design anderer als solches zu erkennen.

Sowohl das Linux- als auch das `fetchmail`-Projekt liefern Indizien dafür. *Linus*, der kein besonders origineller Designer ist (wie vorher schon erklärt), zeigte doch ein eminentes Talent für das Erkennen guter Designs und für die Integration in den Linux-Kernel. Und ich habe auch schon beschrieben, wie die mächtigste, einzelne Design-Idee hinter `fetchmail` (SMTP Forwarding) Eingang in meine Software gefunden hat.

Das erste Publikum dieses Dokuments machte mir durch den Hinweis ein Kompliment, dass ich für das Unterschätzen von Originalität beim Designen für Basar-Projekte anfällig wäre, da ich selbst einiges davon besäße und daher als selbstverständlich nähme. Das könnte stimmen; Design (im Gegensatz zu Coding und Debugging) ist sicherlich meine größte Stärke.

Das Problem mit der Cleverness und der Originalität beim Software-Design ist aber, dass beides zur Gewohnheit wird -- man beginnt, die Dinge auszuschmücken und zu verkomplizieren, obwohl man sie doch eigentlich robust und simpel halten sollte. Mir sind schon ganze Projekte um die Ohren geflogen, weil ich eben diesen Fehler machte, aber bei `fetchmail` schaffte ich es, mich zurückzuhalten.

Ich glaube also, dass das `fetchmail`-Projekt zum Teil deswegen erfolgreich war, weil ich meine Tendenz, clever zu sein, unter Kontrolle hielt, was (wenn schon sonst nichts) ein Einwand gegen die Wichtigkeit von Design-Originalität für Basar-Erfolge ist. Denken Sie nur an Linux: nehmen wir an, *Linus Torvalds* hätte versucht, sämtliche grundlegenden Innovationen im Design von Betriebssystemen während der Entwicklung selbst zu erfinden; erscheint es dann als wahrscheinlich, dass der resultierende Kernel so stabil und erfolgreich wäre wie der, den wir haben?

Ein gewisses Minimum an Kompetenz für Design und Codierung ist natürlich notwendig, aber ich erwarte, dass fast jeder automatisch über mehr als dieses Minimum verfügt, der ernsthaft über die Gründung eines Basar-Projektes nachdenkt. Der interne Reputations-Markt der Open Source-Gemeinde übt einen subtilen Druck auf die Leute aus, und verhindert, dass sie den Keim für ein Unternehmen legen, dem sie nicht gewachsen sind. Bisher hat das allem Anschein nach tadellos funktioniert.

Es gibt aber noch eine weitere Kompetenz, die normalerweise nicht mit der Entwicklung von Software in Verbindung gebracht wird, die aber für Basar-Projekte ebenso wichtig ist wie Findigkeit beim Design -- oder sogar noch wichtiger. Der Koordinator oder Leiter eines Basar-Projekts muss mit Leuten umgehen und kommunizieren können.

Das sollte einleuchten. Um eine Entwicklergemeinde aufzubauen, muss man Menschen begeistern und für seine Sache interessieren können und bewirken, dass sie mit dem eigenen Anteil am Aufwand zufrieden sind. Technischer Glamour wird das meiste der Beinarbeit auf diesem Weg erledigen, aber er alleine ist bei weitem nicht die ganze Geschichte. Der persönliche Eindruck, den man vermittelt, ist ebenfalls ausschlaggebend.

Es ist kein Zufall, dass *Linus* ein netter Kerl ist, der bewirkt, dass die Leute ihn mögen und ihn unterstützen wollen. Es ist kein Zufall, dass ich ein extrovertierter und energischer Mensch bin, der das Arbeiten in der Gruppe sehr genießt und einiges von der Selbstdarstellung und dem Instinkt eines Kabarettisten hat. Um das Basarmodell zum Funktionieren zu bringen, hilft es enorm, wenn man wenigstens ein bisschen Charme hat, um die Menschen für sich einzunehmen.

10 Der soziale Kontext der Open Source-Software

So steht es geschrieben, und wahr ist es: die besten Hacks beginnen als Lösungen für die persönlichen und alltäglichen technischen Probleme des Autors und verbreiten sich, weil sich das Problem als typisch für eine umfangreiche Klasse von Benutzern herausstellt. Das bringt uns zum Kern von Regel 18, die sich vielleicht etwas nützlicher so formulieren lässt:

> *18. Um ein interessantes Problem zu lösen, fängt man mit einem Problem an, das einen selbst interessiert.*

So war es bei *Carl Harris* und seinem Ahnen `popclient`, und so war es bei mir und `fetchmail`. Diese Erkenntnis ist nicht neu. Der interessante Punkt hier, der Punkt, vom dem die Geschichte von Linux und `fetchmail` verlangt, dass wir unsere ganze Aufmerksamkeit auf ihn richten, ist die nächste Phase -- die Evolution von Software in Gegenwart einer großen, aktiven Gemeinde von Anwendern und Mit-Entwicklern. Im **Vom Mythos des Mann-Monats** beobachtet *Fred Brooks*, dass Programmiererzeit nicht **gut stapelbar** ist -- mehr Entwickler in ein verschlepptes Software-Projekt zu werfen verschleppt es nur noch mehr. Er behauptet, dass die Komplexitäts- und Kommunikationskosten proportional zum Quadrat der Anzahl der Entwickler wachsen, während die vollendete Arbeit nur linear dazu wächst. Diese Feststellung wurde unter **Brooks' Gesetz** bekannt und oft als Binsenweisheit betrachtet. Wenn aber Brooks' Gesetz nichts hinzuzufügen wäre, dann wäre Linux unmöglich.

Gerald Weinbergs Klassiker **The Psychology Of Computer Programming** (A. d. Ü.: auf Deutsch nicht erhältlich, in Englisch bei DORSET HOUSE PUBLISHING CO., INC.) lieferte, was wir aus heutiger Perspektive als eine wichtige Korrektur von Brooks Gesetz sehen können. In seiner Erörterung des **Programmieren ohne Ego** stellt *Weinberg* fest, dass in Labors Verbesserungen dramatisch schneller vonstatten gehen, in denen die Entwickler für ihren Code nicht das Verhalten eines Revierbesitzers an den Tag legen und andere Leute dazu anhalten, nach Bugs und möglichen Verbesserungen zu suchen.

Vielleicht hat es *Weinbergs* Wahl der Terminologie verhindert, dass diese Analyse die verdiente Akzeptanz erhielt -- man muss unwillkürlich lächeln bei dem Gedanken, Internet-Hacker als **ohne Ego** zu bezeichnen. Aber ich denke, dass dieses Argument heute überzeugender wirkt denn je.

Die Geschichte von Unix hätte uns darauf vorbereiten sollen, was wir gerade von Linux lernen (und bereits experimentell in kleinerem Maßstab nachweisen konnten, indem wir *Linus*' Methoden willkürlich nachahmten [EGCS]). Konkret heißt das: Während das Codieren eine grundsätzlich autistische Aktivität bleibt, entstehen die wirklich coolen Hacks durch die Organisation der Aufmerksamkeit und Gehirnpower ganzer Gemeinden. Der Entwickler, der nur sein eigenes Gehirn in einem geschlossenen Projekt verwendet, wird gegenüber dem Entwickler zurückfallen, der weiß, wie man einen offenen, evolutionären Kontext schafft, in dem Feedback den Design-Raum erforscht und Code-Schnipsel, Bug Reports und andere Verbesserungen von hunderten (oder sogar tausenden) von Teilnehmern kommen.

Die traditionelle Unix Welt verhinderte ein solches Vorgehen aufgrund verschiedener Faktoren. Einer davon waren die juristischen Einschränkungen der verschiedenen Lizenzen, Firmengeheimnisse und kommerziellen Interessen. Ein anderer war (rückblickend), dass das Internet einfach noch nicht gut genug war.

Vor dem billigen Internet gab es einige geographisch kompakte Gemeinschaften, deren Kultur zu *Weinbergs* **Programmieren ohne Ego** ermunterte, und ein Entwickler konnte leicht viele kompetente Kiebitze und Mit-Entwickler anziehen. Bell Labs, das MIT AI Lab, UC Berkeley -- sie alle wurden zur Heimat von inzwischen Legende gewordenen Innovationen, von denen noch immer enorme Kraft ausgeht.

Linux war das erste Projekt, das eine bewusste und erfolgreiche Anstrengung unternahm, die ganze Welt als seinen Pool von Talenten zu verwenden. Ich glaube nicht, dass es Zufall ist, dass die Zeit des Reifens von Linux mit der Geburt des World Wide Web zusammenfällt. Das war 1993-1994, jene Zeit, zu der die ISP-Industrie

abhob, das Interesse der etablierten Medien am Internet explodierte und Linux aus der Gehschule herauswuchs. *Linus* war der erste, der lernte, wie man nach den neuen Regeln spielt, die das alles durchdringende Internet ermöglichte.

Während ein billiges Internet eine Voraussetzung für die Evolution des Linux-Modells war, denke ich nicht, dass es die einzige war. Ein weiterer wichtiger Faktor war die Entwicklung eines Führungsstils und eines Repertoires von Sitten bei der Zusammenarbeit, die es den Entwicklern gestatteten, Mit-Entwickler anzuziehen und das Maximum aus dem Medium herauszuholen.

Aber was war dieser Führungsstil und was waren die Sitten? Auf Machtverhältnissen konnten sie nicht beruhen -- und sogar wenn es so wäre, Führung durch Zwang könnte das beobachtete Resultat niemals hervorbringen. *Weinberg* zitiert die Autobiographie **Memoirs of a Revolutionist** des russischen Anarchisten Pyotr Alexeyvich Kropotkin (19. Jahrhundert), um dieses Thema sehr gut zu beleuchten:

> "Aus einer Familie stammend, die Leibeigene besaß, begann ich mein Leben als Erwachsener wie alle jungen Männer meiner Zeit mit dem Glauben an die Notwendigkeit des Befehlens, Bestrafens, Scheltens und dergleichen. Als ich aber, noch sehr jung, ernsthafte Unternehmen leiten musste und es mit [freien] Menschen zu tun bekam, deren Fehler schwerwiegende Konsequenzen hatten, begann ich, den Unterschied zwischen dem Handeln nach dem Prinzip des Befehlens und der Disziplin und dem Handeln nach dem Prinzip der Übereinkunft zu würdigen. Ersteres funktioniert vortrefflich in der militärischen Parade, ist aber im wirklichen Leben nichts wert, denn ein Ziel kann nur durch die ernst gemeinte Anstrengung übereinstimmender Willen erreicht werden."

Die **ernst gemeinte Anstrengung übereinstimmender Willen** ist genau das, was ein Projekt wie Linux erfordert -- und das **Prinzip des Befehlens** ist auf die Freiwilligen unmöglich anzuwenden, die wir im Anarchistenparadies Internet vorfinden. Um effektiv zu kooperieren und zu wetteifern, müssen Hacker, die ein kollaboratives Projekt leiten wollen, lernen, wie man effektive Gemeinden im Sinne von Kropotkins **Prinzip der Übereinkunft** rekrutiert und begeistert. Sie müssen lernen, *Linus* Gesetz anzuwenden. [SP]

Ich habe vorher den **Delphi-Effekt** als mögliche Erklärung für *Linus*' Gesetz erwähnt. Es empfehlen sich aber auch kraftvollere Analogien der adaptiven Systeme, wie sie Biologen und Ökonomen kennen, und das viel eindringlicher. Die Linux-Welt verhält sich in vielen Aspekten wie ein freier Markt oder eine Ökologie, eine Sammlung von selbstsüchtig Agierenden, die versuchen, ihren eigenen Nutzen zu maximieren und dabei von selbst eine selbstkorrigierende Ordnung schaffen, die wesentlich raffinierter und effizienter ist als jede zentrale Planung. Hier ist dann also das **Prinzip der Übereinkunft** zu suchen.

Die **Nützlichkeitsfunktion**, die Linux-Hacker zu maximieren trachten, ist nicht in klassischem Sinne ökonomischer Natur, sondern die - etwas unkonkret - Pflege ihres jeweiligen Egos und ihrer Reputation unter Hackerkollegen. (Man mag diese Motivation **altruistisch** nennen, aber dabei vergisst man dann den Umstand, dass Altruismus selbst eine Form von Ego-Pflege für den Altruisten ist.) Tatsächlich sind Kulturen von Freiwilligen, die in dieser Weise funktionieren, nicht ungewöhnlich; eine, an der ich lange teilgenommen habe, war die der Science Fiction-Fans, die der Hackerkultur aber insofern unähnlich ist, als dass sie **egoboo** (die Vermehrung der eigenen Reputation unter anderen Fans) ausdrücklich als den grundlegenden Antrieb hinter freiwilligen Aktivitäten anerkennt.

Linus positionierte sich als Schrankenwärter eines Projekts, dessen Entwicklung vorwiegend durch andere getrieben wird und hegte und pflegte es, bis dieses Projekt auf eigenen Beinen stehen konnte. Dadurch hat er einen eminenten Scharfsinn für Kropotkins **Prinzip der Übereinkunft** gezeigt. Diese quasi-ökonomische Auffassung von der Linux-Welt ermöglicht es uns zu sehen, wie diese Übereinkunft angewendet wird.

Wir könnten *Linus*' Methode als einen Weg ansehen, um einen effizienten Markt für **egoboo** zu erzeugen -- um die Selbstsucht individueller Hacker so straff wie möglich zu vernetzen und sie vor einen sehr sperrigen Karren zu spannen, der alleine durch nachhaltige Kooperation in Bewegung gehalten werden kann. Mit dem

`fetchmail`-Projekt habe ich gezeigt (zugegebenermaßen in viel kleineren Dimensionen), dass seine Methoden mit gutem Erfolg nachgeahmt werden können. Vielleicht habe ich es sogar etwas bewusster und systematischer getan als er.

Viele Leute (besonders jene, die freien Märkten aus ideologischen Gründen misstrauen) würden von einer Kultur von Egoisten erwarten, dass sie fragmentiert, in Parzellen zergliedert, verschwenderisch, geheimnistuerisch und feindselig ist. Diese Erwartung wird aber durch viele Beispiele klar widerlegt; eines davon ist die erstaunliche Vielfalt, Qualität und Tiefe der Linux-Dokumentation. Es ist eine oft strapazierte Binsenweisheit, dass Programmierer das Dokumentieren hassen. Wie kommt es dann, dass Linux-Hacker so viel davon hervorbringen? Offensichtlich funktioniert Linux' freier Markt für Egoboo besser zur Erzeugung von bravem, rücksichtsvollem Benehmen als die Moneten verbrennenden Dokumentationsfabriken der kommerziellen Softwareproduzenten.

Sowohl das `fetchmail`- als auch das Linux-Kernel-Projekt zeigen, dass durch angemessene Pflege der Egos vieler anderer Hacker ein starker Entwickler/Koordinator das Internet verwenden kann, um in den Genuss vieler Mit-Entwickler zu kommen, ohne das Projekt unter seiner eigenen Masse zusammenbrechen zu sehen. Als Gegengewicht zu Brooks Gesetz stelle ich Folgendes fest:

> *19. Unter der Voraussetzung, dass der Entwicklungskoordinator ein Medium zur Verfügung hat, dass wenigstens so gut ist wie das Internet, und dieser Koordinator weiß, wie man ohne Zwang führt, werden viele Köpfe zwangsläufig besser arbeiten als nur einer.*

Ich glaube, dass die Zukunft von Open Source-Software zunehmend Leuten gehören wird, die wissen, wie man *Linus*' Spiel spielt -- Leuten, die die Kathedrale hinter sich lassen und sich für den Basar entscheiden. Das bedeutet nicht, dass individuelle Weitsicht und individuelle Brillanz nicht mehr zählen werden. Ich denke, dass die vorderste Front der Open Source-Software von Leuten geschaffen werden wird, deren individuelle Weitsicht und Brillanz dann durch die effektive Konstruktion von Gemeinden von Freiwilligen mit ähnlichen Interessen verstärkt wird.

Und vielleicht ist das nicht nur die Zukunft der Open Source-Software. Kein Entwickler von nicht-öffentlicher (**Closed Source**) Software kann mit dem Talentpool der Linux-Gemeinde mithalten, wenn es um das Bearbeiten einer technischen Problemstellung geht. Sehr wenige könnten es sich leisten, auch nur die zweihundert (1999: sechshundert) Leute anzuheuern, die zu `fetchmail` beigetragen haben!

Vielleicht wird die Open Source-Kultur schließlich nicht aus dem Grund triumphieren, dass Kooperation moralisch richtig oder das **Horten** von Software moralisch verwerflich ist (was unterstellt, dass Sie letzteres glauben, was weder *Linus* noch ich tun), sondern einfach, weil die Welt der nicht-öffentlichen Software in einem evolutionären Wettrüsten mit den Open Source-Gemeinden nicht gewinnen kann, die ein Vielfaches an hochqualifizierter Entwicklerzeit in eine Problemstellung investiert.

11 Über Management und die Maginotlinie

Die ursprüngliche Fassung von **Cathedral and Bazaar** endete mit der oben beschriebenen Vision -- der von fröhlichen vernetzten Horden von Programmierern/Anarchisten, denen die hierarchische Welt konventioneller Software in keiner Weise gewachsen ist.

Nicht wenige Skeptiker überzeugte das aber gar nicht, und ihre Zweifel verdienen eine faire Erörterung. Die meisten der Einwände gegen das Basar-Argument lassen sich so zusammenfassen, dass Open Source-Verfechter den die Produktivität multiplizierenden Effekt konventioneller Management-Verfahren unterschätzen.

Traditionell orientierte Manager von Projekten der Software-Entwicklung wenden oft ein, dass die Lässigkeit, mit der sich Projektgruppen in der Open Source-Welt bilden, verändern und wieder auflösen, einen bedeutenden Teil der Vorzüge einer großen Zahl von Entwicklern zunichte mache. Ihre Feststellung hier ist, dass in der Software-Entwicklung nicht die Anzahl der auf einen Haufen geworfenen Leute zählt, sondern das Ausmaß der nachhaltigen Investitionen in ein Produkt, die Kunden erwarten können.

An diesem Einwand ist etwas dran, so viel ist sicher; tatsächlich habe ich in Der verzauberte Kessel die Idee entwickelt, dass in Zukunft der Wert der Dienstleistung der Schlüssel zur Ökonomie in der Softwareproduktion ist.

Dieses Argument hat aber auch ein ernstes verstecktes Problem; es unterstellt, dass Open Source-Entwicklung diese nachhaltigen Investitionen nicht liefern kann. Tatsächlich gab und gibt es Open Source-Projekte, die eine bestimmte Richtung und durchschlagskräftige Gemeinden von Teilnehmern treu verfolgt haben, und das über sehr lange Zeiträume hinweg und ohne institutionelle Aufsicht, die konventionelles Management so bedeutend findet. Die Entwicklung des Editors GNU Emacs ist ein extremes und lehrreiches Beispiel dafür; sie hat über fünfzehn Jahre hinweg die Zuwendung und Mühe von hunderten von Beitragenden in eine vereinigte architektonische Vision absorbiert, und das trotz hoher Fluktuation und trotz der Tatsache, dass nur eine Person (der Autor von GNU) während der ganzen Zeit ununterbrochen aktiv war. Kein Closed Source-Editor hat jemals solche Langlebigkeit erreicht.

Das alles legt es nahe, die Vorzüge von konventionell gemanageter Software-Entwicklung anzuzweifeln, die vom Rest der Einwände gegen das Kathedralen- vs. Basar-Modell unabhängig sind. Wenn es für GNU Emacs möglich ist, fünfzehn Jahre lang eine gerade Linie in der Architektur zu verfolgen, oder für acht Jahre im Falle eines Betriebssystems wie Linux, und wenn (was tatsächlich der Fall ist) es sehr viele gut entworfener Open Source-Projekte gibt, die länger als fünf Jahre am Leben waren -- dann sind wir berechtigt, uns zu fragen, was wir denn, wenn überhaupt irgendetwas, für die Kosten für konventionelles Management eigentlich kaufen.

Was auch immer es ist, es hat sicher nichts mit der Lieferung von zuverlässiger Software zu einem versprochenen Termin zu tun, oder mit dem Einhalten des Budgets, oder mit allen in der Spezifikation geforderten Leistungsmerkmalen; es kommt ausgesprochen selten vor, dass ein **gemanagetes** Projekt auch nur eines dieser Ziele erreicht, von allen dreien gar nicht zu reden. Es scheint auch kein besonderes Talent von konventionellem Management zu sein, während des Lebenszyklus eines Projekts flexibel auf Veränderungen im technologischen oder ökonomischen Kontext zu reagieren. Die Open Source-Gemeinde hat sich bei dieser Wertung als bei weitem effektiver gezeigt. Davon kann man sich sehr leicht selbst überzeugen, beispielsweise durch den Vergleich der dreißigjährigen Geschichte des Internets mit den kurzen Halbwertszeiten proprietärer Netzwerktechnologien, oder den Kosten für die Migration von Microsoft Windows von 16-bit auf 32-bit auf der einen Seite und die fast kostenlose Migration von Linux im selben Zeitraum, die nicht nur auf Intel-Prozessoren beschränkt war, sondern auf mehr als ein Dutzend anderer Hardware-Plattformen ausgedehnt wurde, darunter den 64-bit Alpha.

Eines, was sich die Menschen vom traditionellen Modus der Software-Entwicklung versprechen, ist die Möglichkeit, jemanden nach einem schief gegangen Projekt auf Schadenersatz zu verklagen. Das ist aber eine

Illusion; die meisten Software-Lizenzen sind so abgefasst, dass sie sogar jede Verantwortung für Verkäuflichkeit, ganz zu schweigen von Funktionstauglichkeit, ablehnen -- und die Fälle, in denen der Schaden durch nichtfunktionierende Software erfolgreich eingeklagt werden konnte, sind verschwindend gering. Sogar wenn das üblich wäre, ginge dieser Trost, jemanden belangen zu können, am Thema vorbei. Sie wollen keinen Prozess, Sie wollen funktionierende Software.

Was erhalten wir also durch den Overhead des Managements?

Um das zu verstehen, müssen wir uns zunächst mit dem vertraut machen, was Software-Entwicklungsleiter glauben zu tun. Eine Bekannte, die sehr gut in diesem Job zu sein scheint, erklärt, dass Software-Projektmanagement fünf Funktionen erfüllt:

1. Es definiert Ziele und sorgt dafür, dass alle Teilnehmer am selben Strang ziehen.
2. Es überwacht den Fortschritt und stellt sicher, dass wichtige Details nicht einfach unter den Tisch fallen.
3. Es motiviert die Leute dazu, auch stumpfsinnige, aber notwendige Plackerei zu machen.
4. Es organisiert den Aufwand der Leute zu maximaler Produktivität.
5. Es beschafft Ressourcen, die für den Fortschritt des Projekts notwendig sind.

Anscheinend sind das alles erstrebenswerte Ziele, aber beim Open Source-Modell und seinem umgebenden sozialen Kontext können sie alle sehr schnell eine eigentümliche Bedeutungslosigkeit bekommen. Gehen wir die Ziele in umgekehrter Reihenfolge durch.

Meine Bekannte berichtet, dass vieles von der Beschaffung von Ressourcen prinzipiell defensiv ist; sobald man seine Leute, Geräte und Büroräumlichkeiten beisammen hat, muss man sie gegen gleichgestellte Manager verteidigen, die um die selben Ressourcen wetteifern, und gegen Vorgesetzte, die den größtmöglichen Nutzen aus einem begrenzten Pool herausholen wollen.

Open Source-Entwickler aber sind Freiwillige, die nach Interesse und Fähigkeit selbst ernannt sind, um zu einem Projekt beizutragen (das bleibt sogar dann wahr, wenn sie für das Open Source-Hacken ein Gehalt bezahlt bekommen). Der Ethos der Freiwilligen hat die Tendenz, die gesamte **räuberische** Seite der Beschaffung von Ressourcen automatisch zu entschärfen: die Leute stiften ihre eigenen Ressourcen. Und es gibt wenig oder keinen Bedarf nach einem Manager, der den **Beschützer** im konventionellen Sinne spielt.

In einer Welt der billigen PCs und schnellen Internet-Verbindungen stellen wir praktisch überall fest, dass die einzige begrenzte Ressource kompetente Aufmerksamkeit ist. Open Source-Projekte, die im Sand verlaufen, scheitern nicht an einem Mangel von Maschinen oder Anschlüssen oder Büroräumlichkeiten, sondern daran, dass die Entwickler das Interesse daran verlieren.

Aus diesem Grund ist es doppelt wichtig, dass Open Source-Hacker sich selbst organisieren, um durch Selbsternennung das Maximum an Produktivität zu liefern -- und das soziale Milieu wählt gnadenlos nur die höchste Kompetenz. Meine Bekannte, die sowohl mit der Open Source-Welt als auch mit umfangreichen Projekten unter Ausschluss der Öffentlichkeit vertraut ist, glaubt, dass Open Source teilweise wegen seiner Auswahlkriterien so erfolgreich war, die nur 5 Prozent der programmierenden Bevölkerung zulässt. Sie verbringt die meiste Zeit mir der Organisation der restlichen 95 Prozent und kennt daher den Unterschied zwischen den fähigsten Programmierern und den gerade noch kompetenten aus erster Hand; die Produktivität verhält sich ungefähr 1:100.

Dieser bedeutende Unterschied hat immer eine verlegene Frage hervorgerufen: wären individuelle Projekte und das gesamte Feld besser dran, wenn nicht mehr als 50 Prozent der weniger Fähigen daran teilnehmen würden? Besonnene Manager verstehen seit langem, dass das ganze Spiel keinen Wert mehr hätte, wenn es die einzige Funktion des konventionellen Managements wäre, die weniger Fähigen von einem Nettoverlust zu einem marginalen Gewinn zu machen.

Der Erfolg der Open Source-Gemeinde verschärft diese Frage bedeutend. Sie liefert harte Beweise dafür, dass es oft billiger und effektiver ist, selbst ernannte Freiwillige über das Internet anzuheuern als ganze Gebäude voller Leute zu managen, die lieber etwas anderes täten.

Das bringt uns nahtlos zur Frage der Motivation. Eine äquivalente und oft gehörte Umformulierung der betreffenden Aussage meiner Bekannten ist, dass traditionelles Entwicklermanagement eine notwendige Kompensation für schlecht motivierte Programmierer ist, die andernfalls keine gute Arbeit liefern würden.

Diese Antwort tritt meistens zusammen mit der Behauptung auf, dass die Open Source-Gemeinde sich nur dort auf das Erbringen von Aufwand verlassen kann, wo er sexy ist oder technischen Glamour ausstrahlt; alles andere würde ungetan oder schlecht gemacht bleiben, wenn nicht geld-motivierte Großraumbürobewohner unter der Knute von Managern zu Hilfe kommen würden. Ich befasse mich mit den psychologischen und sozialen Gründen für Zweifel an dieser Behauptung in **Homesteading the Noosphere**. Im Augenblick möchte ich aber auf die interessanten Implikationen hinweisen, wenn man unterstellt, dass diese Behauptung wahr ist.

Wenn der konventionelle, völlig durchgemanagete und nicht-öffentliche (**Closed Source**) Stil der Software-Entwicklung wirklich nur durch eine Art Maginot-Linie von Problemen verteidigt wird, die Langeweile hervorrufen, dann wird jedes einzelne Feld von Anwendungen nur solange davor sicher sein, solange diese Problemstellungen niemand interessant findet und niemand einen Weg findet, sie zu umgehen. Ab dem Zeitpunkt, ab dem es einen Wettbewerb um ein **langweiliges** Stück Software gibt, erfahren dann auch die Kunden, dass sich endlich jemand gefunden hat, der dieses Problem interessant genug fand, um sich darum zu kümmern -- was bei Software wie bei jeder anderen kreativen Tätigkeit ein bei weitem machtvollerer Motivator ist als Geld allein.

Eine konventionelle Managementstruktur zu haben, um zu motivieren, ist so gesehen gute Taktik, aber schlechte Strategie; ein kurzfristiger Gewinn, aber langfristig ein sicherer Verlust.

Bis jetzt sieht das konventionelle Entwicklungsmanagement in zwei Punkten wie eine schlechte Wette gegen Open Source aus (Beschaffung und Organisation) und bei einem dritten (Motivation) lebt es von geborgtem Glück. Der arme, unter Druck stehende konventionelle Manager wird keine Aufgaben bei Überwachung vorfinden, auf dem er Punkte wettmachen kann; das stärkste Argument für Open Source ist die dezentralisierte, unentwegte Kritik und Verfeinerung durch Gleichgesinnte (**peer review**), die allen konventionellen Methoden der Qualitätssicherung weit überlegen ist.

Bleibt uns wenigstens die Definition von Zielen als Rechtfertigung für den Overhead des konventionellen Software-Projektmanagements? Vielleicht, aber das verlangt gute Gründe für den Glauben daran, dass Management-Komitees und Firmenerlässe beim Definieren von lohnenden und allgemein anerkannten Zielen erfolgreicher sind als die Projektleiter und Stammesältesten, die eine analoge Rolle für die Open Source-Welt ausfüllen.

Diesen Einwand zu einem überzeugenden zu machen wird schwierig sein, und es ist nicht so sehr die Open Source-Seite dieser Gleichung (Langlebigkeit von Emacs und *Linus Torvalds*' Fähigkeit, ganze Horden von Entwicklern durch Reden über **Weltherrschaft** zu mobilisieren), die das so erschwert. Stattdessen ist es die Erbärmlichkeit der konventionellen Mechanismen zur Definition von Zielen für Software-Projekte.

Eines der bekanntesten Volkstheoreme des Software-Ingenieurwesens ist, dass 60 bis 75 Prozent aller konventionellen Software-Projekte entweder nie fertig oder von den vorgesehenen Anwendern nicht angenommen werden. Wenn diese Zahlen auch nur ungefähr stimmen (und ich habe niemals einen erfahrenen Manager getroffen, der sie abgestritten hätte), dann ist mehr als die Hälfte aller Projekte entweder (a) nicht realistisch spezifiziert oder (b) an den Anwendern vorbeispezifiziert.

Das ist mehr als alles andere der Grund dafür, dass in der Welt des heutigen Software-Ingenieurwesens schon alleine die Formel **Management-Komitee** dem Hörer die Nackenhaare aufstellt -- sogar (oder vielleicht

besonders) wenn der Hörer selbst ein Manager ist. Die Tage, in denen ausschließlich Programmierer dagegen Allergiereaktionen zeigten, sind lange vorbei; Dilbert-Comics hängen heute auch über den Schreibtischen der Chefs.

Unsere Stellungnahme gegenüber dem traditionellen Manager von Software-Entwicklung ist daher simpel: wenn die Open Source-Gemeinde den Wert des konventionellen Managements unterschätzt, warum begegnen dann so viele von euch euren eigenen Verfahren mit so viel Geringschätzung?

Wieder einmal verschärft die Existenz der Open Source-Gemeinde diese Frage erheblich -- weil wir Spaß an dem haben, was wir tun. Unsere kreative Spielerei hat Erfolge nach Kriterien der Technologie, Marktanteile und Beachtung gebracht, deren Häufigkeit das Erstaunliche ist. Wir beweisen nicht nur, dass wir die bessere Software machen können, sondern auch, dass Spaß Kapital ist.

Zweieinhalb Jahre nach der ersten Version dieses Aufsatzes ist der radikalste Gedanke, den ich als Schlusswort anbieten kann, nicht mehr länger eine Vision einer Open Source-dominierten Software-Welt, die heute sogar vielen nüchternen Menschen in Schlips und Kragen plausibel scheint.

Stattdessen weise ich auf eine allgemeinere Lektion zum Thema Software hin (die sich vielleicht auf jede Form von kreativer oder professioneller Arbeit ausdehnen lässt). Die Menschen haben in der Regel ihre Freude an einer Aufgabe, die in irgendeiner Weise in die Zone einer optimalen Herausforderung fällt, die also weder so leicht ist um zu langweilen noch so schwierig um zu überfordern. Ein glücklicher Programmierer ist einer, der weder unterfordert noch von schlecht formulierten Zielsetzungen und dem Stress bürokratischer Reibungsverluste geplagt ist. **Der Spaß kommt mit der Effizienz**.

Von den Umständen und Methoden der eigenen Arbeit angewidert zu sein (sogar wenn sich nur milder Ekel durch Aufhängen von Dilbert-Comics zeigt) sollte daher als Zeichen dafür gewertet werden, dass der Prozess selbst versagt hat. Freude, Humor und Verspieltheit sind ein wertvolles Gut und das Schlagwort von den **fröhlichen Horden** habe ich nicht nur wegen seiner Griffigkeit verwendet. Auch ist es mehr als nur ein Witz, dass für Linux' Maskottchen ein kuscheliger, kindlicher Pinguin ausgesucht wurde.

Es könnte sich ohne weiteres herausstellen, dass die wichtigste Auswirkung des Erfolges der Open Source die Einsicht ist, dass es keinen ökonomisch effektiveren Modus kreativer Arbeit gibt als das Spielen.

12 Danksagung

Dieses Dokument gewann durch Gespräche mit einer großen Zahl von Leuten, die alle mithalfen, es zu debuggen. Mein spezieller Dank gilt *Jeff Dutky*, der die Formulierung **Debugging ist parallelisierbar** beisteuerte und bei der Entwicklung der darauf folgenden Analyse half. Auch *Nancy Lebovitz* gilt mein besonderer Dank. Sie schlug vor, es *Weinberg* gleichzutun und *Kropotkin* zu zitieren. Konstruktive Kritik kam auch von *Joan Eslinger* und *Marty Franz* von der Mailingliste General Technics. *Glen Vandenburg* wies auf die Wichtigkeit der Selbsternennung unter Beitragenden hin und stiftete die fruchtbare Idee, dass viel Entwicklungsarbeit **Bugs der Auslassung** behebt; *Daniel Upper* schlug die natürlichen Analogien dafür vor. Ich bin den Mitgliedern von PLUG, der Philadelphia Linux Users Group sehr dankbar für ihr Auftreten als erstes Testpublikum für die erste öffentliche Version dieses Papiers. *Paula Matuszek* erleuchtete mich zum Thema Praxis des Software-Managements. *Phil Hudson* erinnerte mich daran, dass die soziale Organisation der Hacker-Kultur die Organisation ihrer Software widerspiegelt und umgekehrt. Schließlich waren *Linus Torvalds*' Kommentare sehr hilfreich und seine frühe Unterstützung eine große Ermutigung.

13 Weiterführende Literatur

Ich zitierte mehrere Passagen aus *Frederick P.* Brooks' Klassiker **Vom Mythos des Mann-Monats**, da seinen Einsichten in vielerlei Hinsicht noch nicht wirklich nachgekommen wurde. Ich möchte Ihnen die Ausgabe zum 25. Jahresjubiläum ganz herzlich empfehlen, die bei Addison-Wesley (ISBN 0-201-83595-9) erschienen ist und auch sein 1986 geschriebenes Papier **No Silver Bullet** enthält. (Anm. d. Ü: auf Deutsch: **Vom Mythos des Mann-Monats**, Addison-Wesley, ISBN 3-925118-09-8)

Die neue Ausgabe wird durch eine unschätzbare Rückschau nach 20 Jahren ergänzt, in der sich Brooks unverblümt zu den wenigen Urteilen im ursprünglichen Text bekennt, die sich im Laufe der Zeit als nicht stichhaltig erwiesen haben. Ich las diese Rückschau zunächst nachdem die erste öffentliche Version im Großen und Ganzen vollendet war und wurde durch die Entdeckung überrascht, dass Brooks Basar-ähnliche Praktiken Microsoft zuschreibt! (Tatsächlich stellte sich aber heraus, dass dieses Urteil nicht richtig ist. 1998 erfuhren wir durch die Halloween Documente, dass Microsofts interne Entwicklergemeinde massiv balkanisiert ist und jene Freizügigkeit im allgemeinen Zugriff auf Quellcode nicht einmal wirklich möglich ist.)

Gerald M. Weinbergs **The Psychology Of Computer Programming** (New York, Van Nostrand Reinhold 1971) führte das etwas unglücklich getaufte Konzept des **Programmieren ohne Ego** ein. Während er nicht einmal annähernd unter den ersten war, die die Zwecklosigkeit des **Prinzip des Befehlens** erkannte, war er wahrscheinlich der erste, der diesen Punkt in Verbindung mit Software-Entwicklung bemerkte und diskutierte.

Richard P. Gabriel machte sich Gedanken über die Unix-Kultur der Prä-Linux-Epoche und gestand in seinem 1989 erschienen Papier Lisp: Good News, Bad News, and How To Win Big einem primitiven Basar-ähnlichen Modell Überlegenheit zu. Obwohl er in vielen Aspekten schon etwas veraltet wirkt, genießt sein Aufsatz noch heute zu Recht hohes Ansehen unter Lisp-Fans (mich eingeschlossen). Ein Korrespondent machte mich darauf aufmerksam, dass sich der Abschnitt mit dem Titel **Worse Is Better** fast wie eine Vorwegnahme von Linux liest.

De Marco und *Listers* **Peopleware: Productive Projects and Teams** (New York; Dorset House, 1987; ISBN 0-932633-05-6) ist ein wenig gewürdigtes Juwel, das ich zu meiner großen Freude in *Fred Brooks* Rückblick zitiert gesehen habe. Obwohl nur wenig von dem, was die Autoren zu sagen haben, direkt auf die Linux- oder Open Source-Gemeinden anwendbar ist, sind die Einsichten der Autoren in die Voraussetzungen kreativen Schaffens stichhaltig und haben ihren Wert für jeden, der versucht, einige der Tugenden des Basar-Modells in einen kommerziellen Kontext zu importieren.

Schließlich muss ich gestehen, dass ich diesen Aufsatz beinahe **The Cathedral and The Agora** (**Die Kathedrale und der Marktplatz**, A. d. Ü.) betitelt hätte. Die klassischen Papiere über **agorischen Systeme** von *Mark Miller* und *Eric Drexler*, beschreiben die gerade erkennbar werdenden Eigenschaften von Markt-ähnlichen Rechner-Ökologien und halfen mir dabei, klare Gedanken über analoge Phänomene in der Open Source-Kultur fassen, auf die mich Linux fünf Jahre zuvor mit der Nase gestoßen hatte. Diese Aufsätze gibt es im Web unter http://www.agorics.com/Library/agoricpapers.html.

14 Epilog: Netscape geht auf den Basar

Es ist ein komisches Gefühl zu merken, dass man dabei mithilft, Geschichte zu machen...

Am 22. Januar 1998, ungefähr sieben Monate, nachdem ich **Die Kathedrale und der Basar** erstmalig publiziert hatte, gab Netscape Communications, Inc. ihre Pläne bekannt, den Quellcode für den Netscape Communicator zu veröffentlichen. Ich hatte noch am Tag vor dieser Verlautbarung keine Ahnung davon, dass dies passieren würde.

Eric Hahn, der Executive Vice President und Chief Technology Officer bei Netscape, schrieb mir kurz danach eine E-Mail:

> "Im Namen aller Netscape-Mitarbeiter möchte ich Ihnen dafür danken, dass Sie uns so weit gebracht haben. Ihr Denken und Schreiben waren die grundlegenden Inspirationen für unsere Entscheidung."

In der Woche darauf folgte ich Netscapes Einladung und flog nach Silicon Valley, um zusammen mit ihren Top Executives und technischen Leuten an einer eintägigen Strategiekonferenz teilzunehmen (das war am 4. Februar 1998). Wir entwarfen Netscapes Strategie zur Freigabe des Quellcodes und Lizenzierung.

Ein paar Tage später schrieb ich Folgendes:

> "Netscape steht kurz davor, uns einen riesigen Praxistest für das Basar-Modell in der kommerziellen Welt zu ermöglichen. Die Open Source-Kultur sieht sich nun einer Gefahr gegenüber; wenn Netscapes Plan scheitert, kann das Open Source-Konzept dermaßen in Misskredit geraten, dass sich die kommerzielle Welt für ein Jahrzehnt lang nicht mehr darauf einlassen wird.
>
> Auf der anderen Seite ist es eine großartige Gelegenheit. Die erste Reaktion auf den Schachzug in der Wall Street und anderswo war vorsichtiger Optimismus. Wir haben auch die Chance, etwas zu beweisen. Wenn Netscape dadurch bedeutende Marktanteile zurückgewinnen kann, könnte es eine längst fällige Revolution in der Software-Industrie auslösen.
>
> Das nächste Jahr sollte eine sehr lehrreiche und interessante Zeit werden."

Und das war es dann auch. Ich schreibe dies zur Jahresmitte 1999. Die Entwicklung dessen, was später Mozilla genannt wurde, war nur ein teilweiser Erfolg. Das ursprüngliche Ziel Netscapes wurde erreicht -- Microsoft ein Monopol am Browser-Markt zu vereiteln. Der Schachzug bewirkte auch einige dramatische Erfolge (vor allem die Freigabe der Rendering Engine der nächsten Generation - Gecko).

Das Unternehmen war aber noch nicht in der Lage, außerhalb Netscapes jene massive Entwicklungskapazität zu sammeln, die sich die Gründer von Mozilla ursprünglich erhofft hatten. Das Problem dabei scheint zu sein, dass die Mozilla-Distribution lange Zeit eine der grundlegenden Regeln des Basar-Modells brachen; sie lieferten nichts aus, was potentielle Teilnehmer leicht laufen lassen und arbeiten sehen konnten. (Bis mehr als ein Jahr nach der Release erforderte das Kompilieren von Mozilla eine Lizenz für eine proprietäre Motif-Bibliothek.)

Am negativsten wirkte sich aus (vom Standpunkt der Welt außerhalb Netscapes), dass die Mozilla Group noch keinen Browser von professioneller Qualität geliefert hat -- und eine der Vorstände des Projekts sorgte für eine kleine Sensation, als er zurücktrat und sich über schlechtes Management und verpasste Gelegenheiten beschwerte. **Open Source**, so seine korrekte Beobachtung, **ist kein Zauberstab**.

Das stimmt natürlich. Die langfristige Prognose für Mozilla sieht heute (August 1999) viel besser aus als zur Zeit von *Jamie Zawinskis* Rücktrittserklärung -- er lag aber ganz richtig mit dem Hinweis, dass die Veröffentlichung des Quellcodes nicht notwendigerweise ein existierendes Projekt retten kann, das durch schlecht definierte Ziele,

Spaghetticode und anderen chronischen Krankheiten der Software-Entwicklung geplagt ist. Mozilla hat es geschafft, uns gleichzeitig ein Beispiel dafür zu liefern, unter welchen Umständen Open Source Erfolg haben kann und unter welchen Umständen die Methode scheitern wird.

In der Zwischenzeit hat die Open Source-Idee aber weitere Erfolge landen und weitere Unterstützung finden können. 1998 und Ende 1999 erlebten wir eine gewaltige Explosion des Interesses an der Open Source-Entwicklungsmethode -- ein Trend, der sowohl den anhaltenden Erfolg von Linux antreibt als auch davon in Schwung gehalten wird. Die von Mozilla in Gang gesetzte Bewegung beschleunigt sich noch weiter.

15 Fußnoten

[JB] In Programing Pearls, kommentiert *Jon Bentley*, der berühmte Aphorist der Informatik, Brooks' Beobachtungen mit **Wenn Sie planen, eine Version wegzuschmeißen, werden Sie zwei wegschmeißen**. Er liegt damit ziemlich sicher richtig. Der springende Punkt hinter Brooks' Feststellung und der hinter Bentleys' ist nicht bloß, dass man vom ersten Versuch erwarten sollte, dass er fehlschlägt, sondern dass es effektiver ist, noch einmal von vorne anzufangen, als zu versuchen, einen Sauhaufen auszumisten.

[QR] Es gibt Beispiele erfolgreicher Open Source-Projekte im Basar-Stil, die noch vor der Internet-Explosion stattfanden und mit Unix in keinem Bezug stehen. Die Entwicklung des DOS-Packprogramms info-Zip während der Jahre 1990-1992 ist ein solches. Ein weiteres war das Bulletin Board System RBBS (auch für DOS), das 1983 begann und eine so starke Gemeinde entwickelte, dass es bis heute (Mitte 1999) sehr regelmäßige Freigaben gibt, und das trotz der gewaltigen technischen Vorteile von Internet-Mail und gemeinsamer Nutzung von Dateien über lokale BBSs. Während die Info-Zip-Gemeinde bis zu einem gewissen Grad Internet-Mail nutzte, basierte die Entwicklerkultur der umfangreichen RBBS-Gemeinde auf RBBS, das von der TCP/IP-Infrastruktur völlig unabhängig war.

[JH] *John Hasler* regte eine interessante Erklärung für den Umstand an, dass es bei Open Source-Projekten verhältnismäßig wenig vergebliche Mühe durch überlappende Anstrengungen gibt. Ich nenne seine Vorstellung **Haslers Gesetz**: Die Kosten für doppelt gemachte Arbeit tendieren dazu, weniger als mit dem Quadrat der Team-Größe zu wachsen. In anderen Worten, sie werden immer geringer sein als die Planungs- und Management-Unkosten, die gebraucht würden, um sie zu eliminieren.

Diese Behauptung widerspricht Brooks' Gesetz nicht. Es mag sein, dass die Unkosten für die gesamte Komplexität und die Anfälligkeit für Bugs mit dem Quadrat der Team-Größe wachsen, aber die Kosten für doppelt vorgenommene Arbeiten sind nichtsdestoweniger ein Spezialfall, der eine weniger dramatische Wachstumscharakteristik hat. Es ist nicht schwierig, plausible Erklärungen dafür zu finden. Bedenken Sie nur, dass es viel einfacher ist, sich auf die Abgrenzung der Funktionalität der jeweiligen Module verschiedener Entwickler zu einigen (was doppelt investierten Aufwand verhindert), als die ungeplanten Interaktionen über das ganze System hinweg auszuschließen, das den meisten Bugs zugrunde liegt.

Die Kombination von *Linus*' Gesetz und *Haslers* Gesetz legt drei ausschlaggebende Umstände von Software-Projekten nahe. Bei kleinen Projekten (bis zu drei Entwickler, würde ich sagen) ist keine aufwendigere Managementstruktur notwendig, als einen Entwickler als Chef-Programmierer zu haben. Dann gibt es einen mittleren Bereich der Projektgröße, bei dem die Kosten für traditionelles Management relativ gering sind, so dass die Vorzüge aus der Vermeidung von überlappendem Aufwand, Bug-Tracking und dem unentwegten Kampf gegen übersehene Details unter dem Strich einen Gewinn darstellen.

Für jenseits dieser Projektgröße aber sagt die Kombination aus *Linus*' Gesetz und Haslers Gesetz voraus, dass die Kosten und Probleme des traditionellen Managements viel schneller anwachsen als die zu erwartenden Kosten für doppelt erbrachten Aufwand. Sicher nicht die geringsten dieser Kosten entsteht durch die immanente Unfähigkeit, die Zuwendung sehr vieler Entwickler in effektiver Weise vor den Wagen zu spannen, was aber (wie wir gesehen haben) eine bessere Methode als das traditionelle Management ist, um zu garantieren, dass keine Bugs und Details übersehen werden. Daher treibt die Kombination der beiden Gesetze den Nettogewinn aus traditionellem Management praktisch gegen Null, wenn es um große Projekte geht.

[IN] Ob jemand Projekte von Beginn an als Basar abwickeln kann, hängt von der Frage ab, ob der Basar wahrlich innovative Arbeit ermöglicht. Einige behaupten, dass der Basar aus Mangel an starker Führung nur das Klonen und Verbessern von Ideen handhaben kann, die bereits allgemeiner Standard der Ingenieurskunst sind, aber nicht in der Lage ist, darüber hinaus etwas zum Fortschritt beizutragen. Dieses Argument wurde vielleicht am prominentesten in den berüchtigten Halloween Dokumenten breitgetreten, zwei Microsoft-internen

Memoranden über das Open Source-Phänomen. Die Autoren verglichen Linux' Entwicklung mit der **Jagd von Rücklichtern** (**chasing taillights**) und drückten die Meinung aus, dass **sobald ein Projekt Gleichstand mit dem State Of The Art erreicht habe, der Aufwand an Management, der für das Erreichen neuer Fronten notwendig ist, zu hoch wird** (**once a project has achieved "parity" with the state-of-the-art, the level of management necessary to push towards new frontiers becomes massive**).

Dabei macht man aber grobe Fehler, die Implikationen dieser Überlegung widersprechen den Tatsachen. Einer wird deutlich, wenn die Autoren der Halloween Dokumente später selbst feststellen, dass **neuartige Ideen der Grundlagenforschung als erstes unter Linux implementiert werden und verfügbar sind** (**often [...] new research ideas are first implemented and available on Linux before they are available / incorporated into other platforms**).

Wenn wir **Open Source** für **Linux** einsetzen, sehen wir, dass dies weit von einer neuen Erscheinung entfernt ist. Historisch gesprochen: die Open Source-Gemeinde erfand Emacs, das World Wide Web oder das Internet nicht durch die Jagd nach Rücklichtern oder durch erheblichen Managementaufwand -- und gegenwärtig geht in der Open Source-Welt soviel an innovativer Arbeit vor sich, dass man von der Auswahl regelrecht verwöhnt wird. Im GNOME-Projekt (um willkürlich eines herauszugreifen) begegnen wir zum Beispiel erheblichen Fortschritten auf den Gebieten der graphischen Benutzeroberflächen und Objekttechnologie, die bedeutend genug sind, um die Aufmerksamkeit der Fachpresse auch außerhalb der Linux-Gemeinde auf sich zu ziehen. Es gibt Berge von anderen Beispielen, wovon sich jeder durch einen Besuch bei Freshmeat selbst überzeugen kann.

Es gibt aber einen noch fundamentaleren Fehler in der impliziten Annahme, dass das Modell der Kathedrale (oder das des Basars oder irgendeine andere Form von Management-Struktur) Innovation zuverlässig erzeugen kann. Das ist Nonsense. Bandenbildung verhilft nicht automatisch zu bahnbrechenden Geistesblitzen -- nicht einmal die Gruppen von freiwilligen Basaranarchisten sind für gewöhnlich zu echter Originalität in der Lage, gar nicht zu reden von Kommittees im Bauch von Firmensauriern, deren Überleben von Status Quo Ante abhängt. Einsichten stammen von Individuen. Das Beste, auf das ihre umgebende soziale Maschinerie jemals hoffen kann, ist, auf bahnbrechende Einfälle fördernd zu reagieren -- sie zu belohnen und zu testen, anstatt sie zu zermalmen.

Einige werden das als romantische Ansicht sehen, einen weiteren Aufguss des überholten Cliches vom einsamen Erfinder. Das ist es aber nicht; ich unterstelle hier nicht, dass Gruppen unfähig sind, bahnbrechende Einsichten weiterzuentwickeln, sobald es sie gibt. Es ist ja so, dass gerade der Prozess der Kritik durch andere Teilnehmer (peer review) solche Gruppen in die Lage versetzt, Resultate von höchster Güte hervorzubringen. Stattdessen weise ich darauf hin, dass jede solche Gruppe ihren Anfang -- den notwendigen Funken -- durch eine gute Idee in jemandes Kopf erhält. Kathedralen und Basare und andere soziale Strukturen können diesen Geistesblitz aufnehmen und verfeinern, aber nicht auf Kommando erzeugen.

Daher ist die Wurzel der Problemstellung der Innovation (in der Software und auch überall sonst) mehr die Frage, wie man vermeiden kann, den Funken auszulöschen -- oder, noch fundamentaler, wie man möglichst viele Menschen heranzieht, die zu Einsichten und Geistesblitzen fähig sind.

Einfach anzunehmen, die Software-Entwicklung im Stil der Kathedrale könnte diesen Trick zustande bringen, aber die geringen Hürden und Flexibilität des Basars könnte es nicht, das ist absurd. Wenn es nur einer Person mit einer guten Idee bedarf, um in einem entsprechend gewogenem sozialen Milieu rasch eine Kooperation von hunderten oder tausenden von Teilnehmern zu etablieren, dann wird diese Person jeder anderen mit dieser Idee davonziehen, die sie erst einmal an eine Hierarchie politisch verkaufen muss, um daran arbeiten zu können, ohne gefeuert zu werden.

Und, mehr noch: Wenn man die Geschichte der Software-Innovationen betrachtet, die von Organisationen hervorgebracht wurden, die nach dem Modell der Kathedrale arbeiteten, finden wir schnell heraus, dass dergleichen sehr selten geschieht. Große Firmen sind bei neuen Ideen von der Grundlagenforschung der Universitäten abhängig (daher die Nervosität der Autoren der Halloween Dokumente darüber, dass Linux die

Gabe hat, dieser Grundlagenforschung schneller zu entsprechen). Oder sie schlucken kleine Firmen, die um das Gehirn eines Innovators herum aufgebaut sind. In keinem der beiden Fälle ist die Innovation auf dem Nährboden der Kathedralenkultur entstanden -- viele derart importierten Innovationen enden damit, dass sie unter dem Druck des **massiven Managements** (wie das die Autoren der Halloween Dokumente huldigend nennen) still erstickt werden.

Das ist aber ein Negativ-Beispiel. Der Leser wäre durch ein positives besser bedient. Ich schlage daher folgendes Experiment vor:

1. Greifen Sie irgendein Kriterium für Originalität heraus, von dem Sie annehmen, dass es durchgehend angewendet werden kann. Wenn Ihre Definition lautet **Ich erkenne es als solches, wenn ich es sehe**, dann ist das für diesen Test auch kein Problem.
2. Greifen Sie irgendein Closed Source-Betriebssystem heraus, das mit Linux konkurriert und die beste Quelle für Berichte über gegenwärtige Entwicklungsarbeit an ihm.
3. Beobachten Sie diese Quelle und Freshmeat für einen Monat. Zählen Sie jeden Tag die Anzahl der Verlautbarungen von Freigaben auf Freshmeat, die Sie für eine Innovation halten. Wenden Sie die gleiche Definition von **Innovation** auf das andere Betriebssystem an und zählen Sie mit.
4. Nach dreißig Tagen addieren Sie die jeweiligen Zahlen.

Am Tag als ich dies schrieb, gab es bei Freshmeat zweiundzwanzig Ankündigungen von Freigaben, von denen drei den State Of The Art in gewisser Hinsicht weiterzubringen scheinen. Das war ein schwacher Tag bei Freshmeat, aber ich wäre erstaunt, wenn es irgendeinem Leser möglich sein sollte, auf irgendeinem Closed Source-Kanal auch auf drei wahrscheinliche Innovationen pro Monat zu kommen.

[EGCS] Wir können heute auf die längere Geschichte eines Projekts zurückblicken, das einen aussagekräftigeren Test der Behauptungen über den Basar zulässt als `fetchmail`; die Rede ist von EGCS, dem Experimental GNU Compiler System.

Dieses Projekt wurde Mitte August 1997 angekündigt und war ein bewusster Versuch, die Ideen der frühen Versionen von **Die Kathedrale und der Basar** anzuwenden. Die Gründer des Projekts standen unter dem Eindruck, dass die Entwicklung von GCC, des Gnu C Compilers, seit geraumer Zeit stagnierte. Für die zwanzig darauf folgenden Monate führte man GCC und EGCS als parallele Produkte weiter -- beide schöpften aus dem selben Pool der Internet-Entwicklerpopulation, beide starteten vom selben GCC-Codestamm und beide verwendeten im Großen und Ganzen die selben Unix-Tools und -Entwicklungsumgebungen. Die Projekte unterschied nur, dass EGCS bewusst versuchte, die Basar-Taktik anzuwenden, die ich beschrieben hatte, während GCC seine mehr Kathedralen-artige Organisation beibehielt, die aus einer geschlossenen Entwicklergruppe bestand und nur selten Freigaben durchführte.

Das kam einem kontrollierten Experiment so nahe, wie man es sich nur wünschen konnte, und die Resultate waren dramatisch. Innerhalb von Monaten lagen die EGCS-Versionen bei den Features bedeutend in Führung; bessere Optimierung, bessere Unterstützung für FORTRAN und C++. Viele Leute befanden die EGCS-Entwicklungs-Snapshots für zuverlässiger als die neuesten stabilen Versionen von GCC und bedeutende Linux-Distributionen begannen, auf EGCS zu wechseln.

Im April 1999 löste die Free Software Foundation (die offiziellen Sponsoren von GCC) die ursprüngliche GCC Development Group auf und händigte die Kontrolle über das Projekt dem EGCS Steering Team aus.

[SP] Natürlich werfen *Kropotkins* Kritik und *Linus*' Gesetz allgemeinere Fragen über die Kybernetik von sozialen Organisationen auf. Eine davon wird durch ein anderes Volkstheorem der Software-Entwicklung nahegelegt: *Conways* Gesetz. Üblicherweise wird es so formuliert: **Wenn man vier Gruppen hat, die an einem Compiler arbeiten, bekommt man einen 4-Pass-Compiler.** Ursprünglich hatte diese Aussage die Form **Organisationen, die Systeme entwerfen, sind auf das Hervorbringen von Entwürfen beschränkt, die Kopien der Kommunikationsstrukturen dieser Organisationen sind.** Prägnanter könnten wir sagen: **Die**

Mittel bestimmen das Ergebnis, oder noch zackiger: **Prozess wird Produkt**.

Dementsprechend ist es wertvoll festzuhalten, dass in der Open Source-Gemeinde die organisatorische Form die Funktion auf vielen Ebenen widerspiegelt. Das Netzwerk ist alles und allgegenwärtig -- nicht nur das Internet, sondern auch die Seilschaften der Teilnehmer formen eine verteilte, locker gekoppelte Arbeitsgruppe (" peer-to-peer network"), die viel an Redundanz bietet und in der alle graziöse Zurückhaltung zeigen. In beiden Netzwerken ist jeder Knoten nur so weit wichtig, als dass andere Knoten mit ihm kooperieren wollen.

Die **peer-to-peer**-Charakteristik ist hier sehr bedeutend für die erstaunliche Produktivität der Gemeinde. Was *Kropotkin* versuchte im Zusammenhang mit Machtverhältnissen deutlich zu machen, wird durch das snafu-Prinzip (Situation normal -- all fucked up) weiter entwickelt: **Wirkliche Kommunikation kann es nur zwischen Gleichgestellten geben, weil Untergebene in aller Regel eher für das Erzählen von gefälligen Lügen belohnt werden als für das Sagen der Wahrheit.** Kreative Teamarbeit hängt sehr von echter Kommunikation ab und wird daher durch die Präsenz von Machtverhältnissen ernsthaft behindert. Die Open Source-Gemeinde ist wirklich frei von solchen Machtverhältnissen und lehrt uns durch ein dazu in krassem Gegensatz stehendes Beispiel, wie teuer Macht ist: hohe Kosten durch Bugs, geringere Produktivität und verpennte Gelegenheiten.

Darüber hinaus sagt das snafu-Prinzip voraus, dass es in autoritär orientierten Organisationen einen Realitätsverlust unter Entscheidungsträgern gibt. Er kommt durch die Erscheinung zustande, dass nach und nach mehr und mehr der Information aus gefälligen Lügen besteht -- die Folgen sind in konventionellen Softwareprojekten klar zu erkennen. Es gibt für Untergebene sehr starke Anreize für das Herunterspielen, Verstecken und Ignorieren von Problemen. Wenn dieser Prozess Produkt wird, ist die Software ein Desaster.

16 Version and Change History

Ich zirkulierte 1.16 auf dem Linux Kongress am 21. Mai 1997.

Die Bibliographie fügte ich am 7. Juli 1997 der Version 1.20 hinzu.

Die Anekdote von der Perl-Konferenz fügte ich am 18. November 1997 der Version 1.27 hinzu.

Ich änderte am 9. Februar 1998 in Version 1.29 "Freie Software" in "Open Source" ab.

Ich fügte den Epilog "Netscape geht auf den Basar" am 10. Februar 1998 in 1.31 hinzu.

Ich entfernte Paul Eggerts Graphen über GPL vs. Basar als Reaktion auf Einwände von RMS am 28. Juli 1998.

Ich fügte am 20. November 1998 in 1.40 eine Korrektur von Brooks auf Grund der Halloween Dokumente hinzu.

Ich fügte am 8. August 1999 in 1.45 die Fußnoten über das Snafu-Prinzip (Situation normal -- all fucked up), (prä)historische Beispiele für Basar-orientierte Entwicklung und Originalität am Basar hinzu.

Andere Änderungen beinhalten kleinere redaktionelle und Markup-Fixes.