Produktion von Open-Source-Software

Wie man ein erfolgreiches freies Software-Projekt führt

Karl Fogel
Manuel Barkhau
Sebastian Menge
Reiner Pittinger
Wolf Peuker

Produktion von Open-Source-Software: Wie man ein erfolgreiches freies Software-Projekt führt

von Karl Fogel, Manuel Barkhau, Sebastian Menge, Reiner Pittinger und Wolf Peuker

Widmung

Dieses Buch ist zwei lieben Freunden gewidmet, ohne die es nicht möglich gewesen wäre: Karen Underhill und Jim Blandy.

Inhaltsverzeichnis

Vorwort

Warum ich dieses Buch schreibe?

Wenn ich auf einer Party davon erzähle, dass ich freie Software schreibe, wird mir heutzutage nicht mehr mit leerem Blick begegnet; vielmehr ist die Reaktion: "Ach ja, Open Source – wie Linux?" Und ich nicke eifrig zustimmend: "Ja, genau! Das mache ich." Es ist angenehm, nicht mehr so ganz am Rand zu stehen. In der Vergangenheit war die nächste Frage leicht vorhersehbar: "Wie verdient man denn dabei Geld?" Als Antwort fasste ich die wirtschaftlichen Hintergründe von Open Source kurz zusammen: Es gibt Organisationen, die daran interessiert sind, dass eine bestimmte Software existiert, ohne sie verkaufen zu müssen, sie wollen schlicht sicher gehen, dass sie verfügbar ist und gepflegt wird: als Werkzeug, und nicht als Ware.

In letzer Zeit jedoch dreht sich die nächste Frage nicht immer um Geld. Die Wirtschaftlichkeit von Open-Source-Software[1] ist nicht mehr so rätselhaft, und viele Nicht-Programmierer verstehen mittlerweile – oder sind zumindest nicht überrascht darüber – , dass manche damit ihr Geld verdienen. Stattdessen ist die Frage, die ich immer häufiger höre, "*Ach ja, wie funktioniert das eigentlich?*"

Dafür hatte ich keine gute Antwort parat, und je mehr ich versuchte eine zu finden, desto bewusster wurde mir, was für ein komplexes Thema es wirklich ist. Ein Open-Source-Projekt zu leiten, ist nicht unbedingt wie die Leitung einer Firma (stellen Sie sich vor, ständig mit einer Gruppe von Freiwilligen über die Natur Ihres Produkts diskutieren zu müssen, von denen Sie den meisten noch nicht einmal begegnet sind!). Es ist auch nicht wie die Leitung einer gewöhnlichen, gemeinnützigen Organisation oder eines Staates. Es gibt Ähnlichkeiten zu all dem, aber ich bin langsam zu dem Schluss gekommen, dass freie Software in dieser Hinsicht einzigartig ist. Es gibt vieles womit man sie sinnvoll vergleichen kann, aber nichts womit man sie gleichsetzen kann. Tatsächlich geht die Annahme etwas weit, dass man ein Open-Source-Projekt überhaupt "führen" könne. Ein Open-Source-Projekt kann *begonnen* und durch interessierte Beteiligte beeinflusst werden, oft sogar relativ stark. Aber niemand kann es sich aneignen, und solange es irgendwo Beteiligte gibt, die es fortführen wollen, kann es niemals von einer Partei stillgelegt werden. Jeder hat unendliche Macht; jeder hat keine Macht. Daraus ergibt sich eine interessante Dynamik.

Das ist der Grund für dieses Buch. Freie Software-Projekte haben eine ausgeprägte Kultur entwickelt, mit einer Gesinnung, in der die Freiheit, Software zu entwickeln, die alles tut was man will, der zentrale Grundsatz ist. Das Ergebnis dieser Freiheit ist jedoch nicht eine Zerstreuung der einzelnen Beteiligten, so dass jeder seinen eigenen Weg mit seinem Code geht, sondern eine enthusiastische Zusammenarbeit. Tatsächlich ist die Kompetenz zur Zusammenarbeit eine der am meisten geschätzten Fähigkeiten in Open-Source-Software. Diese Projekte zu verwalten, ist wie das Angehen einer Art hypertrophisierten Kooperation, in der sowohl die eigene Fähigkeit mit anderen zusammenzuarbeiten, als auch neue Wege für diese Zusammenarbeit zu finden, konkrete Vorteile für die Software bringen kann. Dieses Buch versucht Techniken hierzu zu beschreiben. Es ist keineswegs vollständig, aber immerhin ein Anfang.

Gute freie Software ist an und für sich schon ein würdiges Ziel, und ich hoffe, dass Leser die hierin nach neuen Wegen suchen, mit dem zufrieden sein werden, was sie hier finden. Darüber hinaus hoffe ich etwas von der Freude an der Arbeit mit einem motivierten Team von Open-Source-Entwicklern und der wundervoll direkten Art mit Benutzern zu interagieren, zu der Open Source ermutigt, vermitteln zu können. An einem erfolgreichen Open-Source-Projekt teilzunehmen, macht *Spaß*, und letztlich ist es das, was das ganze System am Laufen hält.

[1] Im Prinzip sind die Begriffe "Open Source" und "Frei" in diesem Kontext Synonyme. Sie werden in „"Frei" kontra "Open Source"“ im Kapitel Kapitel 1, *Einleitung* ausführlicher behandelt.

Wer sollte dieses Buch lesen?

Dieses Buch richtet sich an Softwareentwickler und Führungskräfte, die mit dem Gedanken spielen, ein Open-Source-Projekt zu starten, oder bereits eines begonnen haben, und sich nun fragen wie es weitergeht. Es sollte auch für alle hilfreich sein, die sich zum ersten Mal an einem Open-Source-Projekt beteiligen wollen.

Der Leser braucht kein Programmierer zu sein, sollte jedoch grundsätzliche Konzepte der Softwareentwicklung wie Quellcode, Compiler und Patches kennen.

Erfahrung mit Open-Source-Software, ob nun als Anwender oder als Entwickler, ist nicht nötig. Wer sich bereits an Open-Source-Projekten beteiligt hat, wird vielleicht manchen Abschnitt als überflüssig empfinden, und ggf. überspringen wollen. Aufgrund der potenziell breit gefächerten Erfahrung des Publikums habe ich versucht, Abschnitte klar zu kennzeichnen und darauf hinzuweisen, wann bereits mit der Materie vertraute Leser beruhigt weiterblättern können.

Quellen

Ein Großteil des Materials für dieses Buch stammt aus fünf Jahren Arbeit am Subversion-Projekt (http://subversion.tigris.org/). Subversion ist ein Open-Source-System zur Versionsverwaltung[2], das von Grund auf neu geschrieben wurde, und als Ablösung für CVS konzipiert ist, dem Quasi-Standard für Versionsverwaltung in der Open-Source-Gemeinschaft. Das Projekt wurde von meinem Arbeitgeber, CollabNet (http://www.collab.net/), im Frühjahr 2000 begonnen, und zum Glück verstand CollabNet, es gleich von Beginn an als echtes gemeinschaftliches, verteiltes Unterfangen zu betreiben. Von Anfang an bekamen wir großen Zulauf an Freiwilligen. Heute sind um die 50 Entwickler am Projekt beteiligt, von denen nur wenige bei CollabNet angestellt sind.

Subversion ist in vielerlei Hinsicht ein klassisches Beispiel für ein Open-Source-Projekt, und ich griff darauf mehr zurück, als ich ursprünglich erwartet hatte. Zum Teil einfach aus Bequemlichkeit: Wenn ich nach einem Beispiel für ein bestimmtes Phänomen suchte, konnte ich mich meist spontan an einem Fall aus Subversion erinnern. Obwohl ich an anderen Open-Source-Projekten in unterschiedlichem Maß beteiligt bin, und mit Freunden und Bekannten rede, die ebenfalls an vielen weiteren beteiligt sind, wurde mir beim Schreiben schnell klar, dass ich meine Behauptungen durch Fakten stützen sollte. Allein auf der Grundlage dessen, was ich aus den öffentlichen Archiven ihrer Mailinglisten entnehmen konnte, mochte ich keine Äußerungen über Ereignisse in anderen Projekten machen. Würde jemand dasselbe mit Subversion versuchen, da bin ich sicher, läge er nur in der Hälfte der Fälle richtig. Wenn ich also Inspiration oder Beispiele aus Projekten nahm, mit denen ich keine direkte Erfahrung hatte, versuchte ich zuerst, mit einem Beteiligten aus dem Projekt zu reden, mit jemandem der wusste, was sich tatsächlich abgespielt hatte.

Ich arbeite seit 5 Jahren an Subversion, habe aber schon seit 12 Jahren mit freier Software zu tun. Unter anderem haben folgende Projekte dieses Buch beeinflusst:

- Das GNU Emacs Text-Editor-Projekt der GNU Software Foundation, bei der ich ein paar kleine Pakete pflege.
- Concurrent Versions System (CVS), an dem ich intensiv mit Jim Blandy von 1994 bis 1995 arbeitete, seitdem aber nur sporadisch beteiligt bin.
- Die Open-Source-Projekte unter dem gemeinschaftlichen Namen Apache Software Foundation, insbesondere die Apache Portable Runtime (APR) und der Apache HTTP Server.

[2] engl. Version Control

- OpenOffice.org, die Berkeley-Datenbank von Sleepycat, und die MySQL-Datenbank, an denen ich zwar nicht persönlich beteiligt war, die ich aber beobachtet und zu denen ich Kontakt gepflegt habe.
- GNU Debugger (GDB) (ebenso).
- Das Debian-Projekt (ebenso).

Die Liste ist natürlich unvollständig. Wie die meisten Open-Source-Programmierer halte ich beiläufig Kontakt zu vielen verschiedenen Projekten, nur um ein allgemeines Gefühl für ihren Zustand zu behalten. Ich werde sie hier nicht alle beim Namen nennen, aber sie werden im Text an entsprechender Stelle erwähnt.

Danksagung

Es hat vier mal länger gedauert dieses Buch zu schreiben, als ich ursprünglich geplant hatte, und die meiste Zeit davon fühlte ich mich, als würde beständig ein Konzertflügel über meinem Kopf schweben. Erst die Hilfe vieler Menschen hat es mir ermöglicht, dieses Buch fertigzustellen, ohne dabei den Verstand zu verlieren.

Mein Redakteur, Andy Oram bei O'Reilly, war der Traum eines jeden Schriftstellers. Er kennt nicht nur das Thema (viele Vorschläge kamen von ihm), sondern hat auch die seltene Gabe zu verstehen, was man meint und kann einem helfen, es in die richtigen Worte zu fassen. Es war eine Ehre, mit ihm zu arbeiten. Danke auch an Chuck Toprek, der den Vorschlag zu diesem Buch direkt an Andy geleitet hatte.

Brian Fitzpatrik hat das ganze Material durchgesehen, während ich es schrieb, was nicht nur dazu führte, dass das Buch besser wurde, sondern mich auch dann am Schreiben hielt, wenn ich lieber an jedem anderen Ort auf der Welt gewesen wäre, als vor dem Bildschirm. Auch Ben Collins-Sussman und Mike Pilato prüften meinen Fortschritt und waren stets erfreut, mit mir – manchmal sehr ausführlich – zu diskutieren, egal welches Thema ich in der Woche zu behandeln versuchte. Sie bemerkten auch, wenn ich erlahmte und munterten mich durch kleine Nörgeleien auf. Danke Jungs!

Biella Coleman schrieb zur selben Zeit an ihrer Doktorarbeit, wie ich an diesem Buch. Sie weiß was es heißt, sich jeden Tag hinzusetzen und zu schreiben. Sie war ein inspirierendes Beispiel und bot mir ein mitfühlendes Ohr. Sie hat auch einen faszinierenden Anthropologen-Blickwinkel auf die Bewegung der freien Software und lieferte Ideen und Referenzen, die ich im Buch verwenden konnte. Alex Golub – auch ein Anthropologe mit einem Fuß in der Welt der freien Software, der ebenfalls zu der Zeit an seiner Doktorarbeit schrieb – bot am Anfang eine außerordentliche Unterstützung, was eine große Hilfe war.

Micah Anderson schien irgendwie nie besonders bedrückt durch seine eigene Schreibarbeit, was auf eine krankhafte, Neid erzeugende Art inspirierend war. Er war aber immer mit Freundschaft, Gesprächsbereitschaft und (mindestens einmal) mit technischer Unterstützung zur Stelle. Danke Micah!

Jon Trowbridge und Sander Striker gaben sowohl Aufmunterung als auch konkrete Hilfe – ihre breite Erfahrung hinsichtlich freier Software lieferte Material, an das ich sonst nie herangekommen wäre.

Danke an Greg Stein, nicht nur für seine Freundschaft und zeitige Ermunterung, sondern dafür, dass er dem Subversion-Projekt gezeigt hat, wie wichtig der regelmäßige Code-Review für den Aufbau einer Entwickler-Gemeinschaft ist. Danke auch an Brian Behlendorf, der taktvoll in unsere Köpfe hämmerte, wie wichtig es ist, Diskussionen öffentlich zu führen; Ich hoffe, dass dieses Prinzip sich durchweg im Buch wiederspiegelt.

Danke an Benjamin "Mako" Hill und Seth Schoen für verschiedene Gespräche über freie Software und ihre Politik; an Zack Urlocker und Louis Suarez-Potts dafür, dass sie sich Zeit für Gespräche aus ihren

Terminkalendern nahmen; an Shane auf der Slashcode-Liste für die Erlaubnis, seinen Beitrag zitieren zu dürfen; und an Hagen So für seine enorm hilfreichen Vergleiche verschiedener Anbieter von Hosting-Paketen.

Danke an Alla Dekhtyar, Polina und Sonya für ihre unermüdliche und geduldige Ermutigung. Ich bin sehr froh, unsere Abende nicht mehr vorzeitig beenden zu müssen (bzw. dies ohne Erfolg zu versuchen), um "das Buch" weiter zu schreiben.

Danke an Jack Tepenning für seine Freundschaft, Unterhaltung, und seine sture Weigerung, eine einfache und falsche Analyse jemals einer schwierigeren aber richtigen vorzuziehen. Ich hoffe, dass ein Teil seiner langjährigen Erfahrung sowohl in der Softwareentwicklung als auch in der Softwareindustrie auf dieses Buch abgefärbt hat.

CollabNet war besonders großzügig darin, mir eine flexible Zeitplanung zu erlauben, und beschwerte sich nicht, als ich viel länger brauchte als ursprünglich geplant. Ich kenne die komplexen Wege nicht, auf denen eine Betriebsleitung zu solchen Entscheidungen gelangt, aber ich vermute es hatte etwas mit Sandyha Klute, und später mit Mahesh Murthy, zu tun – meinen Dank ihnen beiden.

Das gesamte Subversion-Team ist in den letzten fünf Jahren eine Inspiration gewesen, und vieles in diesem Buch habe ich durch die Zusammenarbeit in ihm gelernt. Ich werde hier nicht jeden beim Namen nennen, es sind einfach zu viele, aber ich beschwöre jeden Leser, dem ein Mitwirkender von Subversion über den Weg läuft, diesem ein Getränk seiner Wahl zu spendieren – ich jedenfalls habe das vor.

Oftmals wetterte ich gegenüber Rachel Scollen über den Zustand des Buches; stets war sie bereit mir zuzuhören und irgendwie schaffte sie es immer, die Probleme kleiner zu machen als vor dem Gespräch. Das war eine ungeheure Hilfe – danke.

Danke (nochmals) an Noel Taylor, der sich sicherlich fragte, warum ich noch ein Buch schreiben wollte, wenn man bedenkt wie oft ich mich beim letzten Mal beschwert hatte, dessen Freundschaft und Führung bei Golosá mir half, Musik und Kameradschaft in meinem Leben zu halten, selbst zu den betriebsamsten Zeiten. Danke auch an Matthew Dean und Dorothea Samtleben, Freunde und lange musikalische Leidensgenossen, die sehr verständnisvoll waren, als meine Ausreden für Proben sich stapelten. Megan Jennings blieb durchweg unterstützend und ehrlich interessiert an dem Thema, obwohl es ihr fremd war – eine großartige Stärkung für einen unsicheren Schriftsteller. Danke!

Ich hatte vier sachkundige und gewissenhafte Kritiker für dieses Buch: Yoav Shapira, Andrew Stellman, Davanum Srinivas und Ben Hyde. Das Buch wäre sicherlich besser, wenn ich all ihre ausgezeichneten Vorschläge hätte aufnehmen können. Zeitmangel zwang mich leider, mich aufs Herauspicken zu beschränken, aber die Verbesserungen waren dennoch erheblich. Alle übrig gebliebenen Fehler sind voll und ganz mir zuzuschreiben.

Meine Eltern, Frances und Henry waren wunderbare Unterstützer wie immer, und da dieses Buch weniger technisch ist als das letzte, hoffe ich, sie werden es etwas leserlicher finden.

Schließlich möchte ich Karen Underhill und Jim Blandy danken, denen dieses Buch gewidmet ist. Die Freundschaft und das Verständnis von Karen haben mir alles bedeutet, nicht nur während ich an diesem Buch schrieb, sondern auch die letzten sieben Jahre hindurch. Ohne ihre Hilfe hätte ich es sicher nicht geschafft. Dasselbe gilt für Jim, einen echten Freund und Hacker von einem Hacker, der mir beibrachte was freie Software ist, ähnlich wie ein Vogel, der ein Flugzeug fliegen lehrt.

Haftungsausschluss

Die Gedanken und Meinungen in diesem Buch sind meine eigenen. Sie stellen nicht unbedingt die Sicht von CollabNet oder die des Subversion-Projekts dar.

Anmerkungen der Übersetzer

Manuel Barkhau, März 2008

Diese Buch wurde mit großzügiger Unterstützung der mg.softech GmbH http://www.mgsoftech.com ins Deutsche übersetzt, der ich an dieser Stelle danken möchte. Es ist ein gutes Beispiel für die gemeinsamen Interessen von Open-Source-Gemeinschaft und Wirtschaft, die sogar über die tatsächliche Produktion von offenem Quellcode hinausgeht. Als abgeschlossen möchte ich diese Übersetzung jedoch nicht betrachten, und ich lade Sie als Leser herzlich ein, sich an der Fertigstellung zu beteiligen. Ich bin mir sicher, Sie werden über manches in dem Buch stolpern und wir freuen uns natürlich über jeden Beitrag.

Wolf Peuker, September 2012

Als ich mich vor einigen Jahren mit einer Projektmanagement-Frage an Google wandte, ahnte ich nichts davon, in welcher Ausführlichkeit ich sie beantwortet bekommen würde: ich fand mich mitten in einem Kapitel dieses großartigen Buches und im festen Griff einer spannenden Lektüre.

Producing Open Source Software ist nicht nur selbst ein vorbildliches Beispiel eines Open-Source-Projekts, es stellt auch den Autor und seine vielen Übersetzer vor die besondere Herausforderung, mit einem atemberaubenden Fortschritt der Open-Source-Bewegung mitzuhalten.

Die Alltagssprache der (deutschen) Softwareentwickler ist voll von englischem Fachjargon. Ich versuche von diesem so viel wie möglich zu übersetzen, um das Thema einem möglichst großen Kreis von Lesern (oftmals Open-Source-Anwender) zugänglich zu machen.

Kapitel 1. Einleitung

Die meisten freien Software-Projekte scheitern.

Wir hören nicht viel über Fehlschäge. Nur erfolgreiche Projekte ziehen die Aufmerksamkeit auf sich, und es gibt insgesamt so viele Open-Source-Projekte [1], dass obwohl nur ein Bruchteil Erfolg hat, immer noch eine Menge Projekte sichtbar sind. Von den Fehlschlägen hören wir auch deshalb nicht, weil Versagen kein Ereignis ist. Es gibt keinen bestimmten Zeitpunkt, an dem ein Projekt aufhört realisierbar zu sein; Teilnehmer gleiten irgendwie langsam davon und hören auf daran zu arbeiten. Es gibt vielleicht einen Moment, an dem die letzte Änderung gemacht wird, aber derjenige, der sie durchführt, weiß für gewöhnlich nicht, dass es die letzte sein wird. Es gibt nicht einmal eine klare Definition, wann ein Projekt abgelaufen ist. Ist es, wenn an einem Projekt seit sechs Monaten nicht aktiv gearbeitet wurde? Sobald seine Nutzergemeinde aufhört zu wachsen, ohne die Entwicklergemeinde übertroffen zu haben? Was ist wenn die Entwickler ihr Projekt verlassen, sobald sie merken, dass ein anderes Projekt die gleichen Ziele verfolgt, und sie nur unnötig einen doppelten Aufwand betreiben, und sie sich diesem anderen Projekt anschließen und ihre früheren Anstrengungen mit einzubeziehen? Wurde das Projekt beendet, oder ist es nur umgezogen?

Durch diese Komplexität ist es unmöglich die Ausfallquote genau zu beziffern. Einzelberichte aus über einem Jahrzehnt freier Software, kurzes Stöbern auf SourceForge.net und ein wenig Googeln, deuten jedoch alle auf den gleichen Schluss: die Rate ist extrem hoch, wahrscheinlich in der Größenordnung von 90%–95%. Die Zahl wird größer, wenn man überlebende, aber nicht vernünftig laufende Projekte mit einbezieht: solche die zwar laufenden Code produzieren, aber weder ein angenehmer Aufenthaltsort sind, noch so schnell oder zuverlässig Fortschritte machen, wie sie könnten.

In diesem Buch geht es darum, Versagen zu vermeiden. Es untersucht nicht nur wie man Sachen richtig macht, sondern wie man sie falsch macht, um frühzeitig Probleme erkennen und korrigieren zu können. Ich hoffe Ihnen zeigen zu können, wie man häufige Stolpersteine vermeidet, und erfolgreich Wachstum und der Pflege eines Projekts meistert. Erfolg ist kein Nullsummenspiel, und in diesem Buch geht es nicht darum zu Gewinnen oder der Konkurrenz voraus zu sein. Tatsächlich ist ein wichtiger Teil beim Betrieb eines Open-Source-Projekts die reibungslose Zusammenarbeit mit anderen verwandten Projekten. Auf lange Sicht trägt jedes erfolgreiche Open-Source-Projekt zum Wohl der gesamten Welt der freien Software bei.

Man ist versucht zu sagen, freie Software-Projekte würden aus den selben Gründen fehlschlagen, wie proprietäre Software-Projekte. Freie Software hat sicherlich kein Monopol auf unrealistische Anforderungen, vage Spezifikationen, schlechte Verwaltung von Ressourcen, zu kurze Entwurfsphasen, oder irgend einen der anderen Kobolde die der Software-Industrie bereits wohl bekannt sind. Es gibt genügend Material zu diesem Thema und ich werde in diesem Buch nicht versuchen es zu duplizieren. Stattdessen werde ich versuchen die Probleme zu beschreiben, die der freien Software eigen sind. Wenn ein freies Software-Projekt an die Wand gefahren wird, liegt es oft daran dass die Entwickler (oder die Projektleiter) nicht um die spezifischen Probleme von Open-Source-Software wussten, auch wenn sie durchaus auf die bekannteren Probleme der Closed-Source-Entwicklung vorbereitet waren.

Einer der häufigsten Fehler sind unrealistische Erwartungen über die Vorteile von offenem Quellcode. Eine offene Lizenz garantiert weder, dass eine Horde aktiver Entwickler urplötzlich anfangen, von sich aus, Zeit in Ihr Projekt zu investieren, noch wird die Offenlegung die Krankheiten des Projekts automatisch heilen. Tatsächlich kann es sogar genau das Gegenteil bewirken: Es kann eine ganze Reihe neuer Komplexitäten hinzufügen und kurzfristig *mehr* kosten, als die Software einfach in Betrieb zu halten. Das Projekt zu öffnen bedeutet, den Quellcode so zu gestalten, dass er für völlig Fremde verständlich ist, eine Site für Entwickler aufzubauen, Mailinglisten einzurichten und oft auch zum erstem Mal

[1] Die beliebte Hosting-Seite SourceForge.net, verzeichnete Mitte April 2004 79,225 Projekte. Natürlich ist das nicht einmal annähernd die Gesamtzahl der freien Software-Projekte im Internet, sondern lediglich die Teilmenge die Sourceforge als Plattform gewählt haben.

eine Dokumentation zu schreiben. All das ist eine Menge Arbeit. Und *falls* interessierte Entwickler auftauchen, gibt es die zusätzliche Bürde, eine Zeit lang ihre Fragen beantworten zu müssen, bevor man aus ihrer Anwesenheit einen Nutzen zieht. Der Entwickler Jamie Zawinski hatte folgendes über die Anfangstage des Mozilla Projekts zu sagen:

> *Open Source funktioniert schon, aber es ist sicherlich kein Allheilmittel. Falls es hier eine warnende Lehre gibt, dann die, dass man ein sterbendes Projekt nicht einfach mit dem magischen Elfenstaub des "Open Source" bestreuen kann, und danach alles wie von selbst läuft. Die Probleme sind nicht so einfach.*
>
> (aus **http://www.jwz.org/gruntle/nomo.html**)

Ein verwandter Fehler ist an Aufmachung und Präsentation zu sparen, in der Annahme, dass diese Sachen auch später erledigt werden können, sobald das Projekt erst einmal läuft. Aufmachung und Präsentation umfasst eine weite Reihe von Aufgaben, die sich alle um das Thema drehen, die Einstiegshürden niedrig zu legen. Das Projekt für Außenstehende einladend zu machen bedeutet, Anleitungen für Nutzer und Entwickler zu schreiben, eine Webseite aufzubauen, die für Neuankömmlinge informativ ist, möglichst viele der Kompilierungs- und Installationsvorgänge der Software zu automatisieren, usw. Viele Entwickler behandeln diese Aufgaben leider so als wären sie nur von zweitrangiger Bedeutung im Verhältnis zum eigentlichen Quellcode. Es gibt hierfür ein paar Gründe. Erstens, kann es einem wie Fleißarbeit vorkommen, denn die Vorteile sind zumeist nur für diejenigen sichtbar, die noch am wenigsten mit dem Projekt zu tun haben und umgekehrt. Schließlich kann jemand, der bereits mit dem Projekt vertraut ist, auf diese ganze Aufmachung verzichten. Er weiß schon, wie man die Software installiert, administriert und benutzt, schließlich hat er sie ja entwickelt. Zweitens sind die Fähigkeiten, die Aufmachung und Präsentation vernünftig zu gestalten, andere als solche die man benötigt um Quellcode zu schreiben. Menschen neigen dazu, sich auf das zu konzentrieren, was sie gut können, selbst wenn es für das Projekt besser wäre, ein wenig Zeit in etwas zu investieren, was ihnen weniger liegt. Kapitel 2, *Der Einstieg* behandelt Aufmachung und Präsentation im Detail und erklärt, warum es entscheidend ist, dass sie gleich von Anfang an Priorität haben sollten.

Der nächste Trugschluss ist, dass bei freier Software wenig bis gar kein Projektmanagement erforderlich ist bzw. umgekehrt, dass dieselben Management-Verfahren, die für die Entwicklung im Betrieb benutzt werden, sich genauso gut auf ein Open-Source-Projekt anwenden lassen. Die Verwaltung ist bei einem Open-Source-Projekt nicht immer besonders offensichtlich, aber bei erfolgreichen Projekten gibt es sie in irgendeiner Form im Hintergrund. Ein kleines Gedankenexperiment reicht um die Gründe dafür zu zeigen. Ein Open-Source-Projekt besteht aus einem zufällig zusammengewürfeltem Haufen von Programmierern – bereits an und für sich notorisch eigensinnige Leute –, von denen untereinander wahrscheinlich keiner einem anderen je begegnet ist, und von denen jeder u.U. unterschiedliche eigene persönliche Ziele verfolgt. Das Gedankenexperiment ist einfach: Stellen Sie sich vor, was mit einer solchen Gruppe passieren würde, *ohne* eine Verwaltung. Wenn kein Wunder geschieht, würden sie sehr schnell auseinander gehen. Auch wenn wir es uns anders wünschen, läuft das Ganze nicht einfach von alleine. Die Verwaltung ist aber, wenn auch ziemlich aktiv, meistens informell, subtil und unauffällig. Das Einzige, was die Entwickler zusammenhält, ist, dass sie zusammen mehr erreichen können als jeder für sich. Deshalb muss die Aufgabe einer Verwaltung hauptsächlich darin bestehen, sie weiterhin daran glauben zu lassen, indem sie Richtlinien für die Kommunikation festlegt, dafür sorgt, dass brauchbare Entwickler nicht aufgrund persönlicher Eigenheiten an den Rand gedrängt werden, und allgemein das Projekt als einen Ort zu gestalten, an den Entwickler gern zurückkehren. Im Verlauf des Buchs soll gezeigt werden, wie man diese Ziele erreicht.

Schließlich gibt es eine generelle Problematik, die man als "Versagen kultureller Navigation" bezeichnen könnte. Vor zehn, vielleicht auch nur fünf Jahren hätte es als voreilig gegolten, von einer globalen Kultur der freien Software zu reden, heute jedoch nicht. Eine erkennbare Kultur ist langsam gewachsen, und obwohl sie sicherlich nicht monolitisch ist – sie ist mindestens so anfällig für interne Meinungsverschiedenheiten und Parteigeist wie irgend eine geographisch gebundene Kultur – hat sie doch einen im Grunde genommen beständigen Kern. Die meisten erfolgreichen Open-Source-Projekte, zeigen alle,

oder zumindest einen großen Teil der Merkmale von diesem Kern. Sie belohnen bestimmte Verhaltensmuster und bestrafen andere; sie schaffen eine Atmosphäre, die spontane Beteiligung fördert, manchmal auf Kosten zentraler Koordination; sie haben Konzepte von Unhöflichkeit und gutem Benehmen, die von den anderswo vorherrschenden erheblich abweichen können. Vielleicht am wichtigsten sind die erfahrenen Teilnehmer, die diese Konzepte meistens verinnerlicht haben, sodass sie einen groben Konsens über das zu erwartenden Benehmen teilen. Nicht erfolgreiche Projekte weichen für gewöhnlich wesentlich von diesem Kern ab, wenn auch unabsichtlich, und haben oft keinen Konsens darüber, welches Benehmen als angemessen einzustufen ist. Das hat zur Folge, dass sobald Probleme auftreten, die Situation schnell eskalieren kann, da den Teilnehmern ein etablierter Grundbestand kultureller Reflexe fehlt, um Meinungsverschiedenheiten zu klären.

Dieses Buch ist ein praktischer Führer, keine anthropologische Studie oder Historie. Grundkenntnisse über die Herkunft der Kultur der freien Software sind dennoch, eine erforderliche Grundlage bei jedem praktischen Ratschlag. Jemand der die Kultur versteht, kann weit und breit in der Open-Source-Welt reisen, viele lokale Unterschiede in den Gebräuchen und Dialekten begegnen, und trotzdem in der Lage sein, sich überall bequem und effektiv zu beteiligen. Im Gegensatz dazu wird eine Person ohne Verständnis für die Kultur, die Organisation und die Art sich an einem Projekt zu beteiligen, die Bräuche als schwierig und voller Überraschungen empfinden. Da die Anzahl der Menschen, die freie Software entwickeln, immer noch stark ansteigt, gibt es viele in der letzten Kategorie – diese sind zum größten Teil vor kurzem eingewandert und das wird auch eine Weile lang so bleiben. Wenn Sie meinen, vielleicht eine von ihnen zu sein, gibt der nächste Abschnitt einen Hintergrund für spätere Diskussionen, sowohl im Buch als auch im Internet. (Wenn Sie andererseits bereits eine Weile lang mit Open Source arbeiten, und u.U. bereits eine Menge seiner Geschichte kennen, können Sie den nächsten Abschnitt getrost überspringen.)

Die Geschichte

Software wurde schon immer an Andere weitergegeben und miteinander geteilt. In den frühen Tagen der Computerindustrie waren Hersteller der Meinung, vor allem durch Innovationen in der Hardware Wettbewerbsvorteile zu erreichen, und schenkten der Software als Vorteil gegenüber der Konkurrenz wenig Aufmerksamkeit. Viele Kunden dieser frühen Maschinen waren Wissenschaftler oder Techniker, die in der Lage waren, die Software die mit der Maschine selbst ausgeliefert wurde, selbst zu verändern und zu erweitern. Kunden gaben ihre Verbesserungen nicht nur an den Hersteller zurück, sondern teilten es manchmal auch mit den Besitzern ähnlicher Maschinen. Den Herstellern war das nur recht, sie ermutigten sogar dazu: Aus ihrer Sicht machte jede Verbesserung an der Software, egal aus welcher Quelle, ihre Maschine attraktiver für andere potenzielle Kunden.

Obwohl diese frühe Zeit in vielerlei Hinsicht der heutigen freien Software-Kultur ähnelte, gab es zwei wesentliche Unterschiede. Erstens gab es noch kaum standardisierte Hardware – es war eine Zeit florierender Innovationen bei den Architekturen in Computern, aber diese Vielfalt bedeutete, dass alles inkompatibel zueinander war. Deshalb lief Software, die für eine Maschine geschrieben wurde, im Allgemeinen auf keiner anderen. Programmierer eigneten sich im Allgemeinen Fachkenntnisse zu einer bestimmten Architektur oder Familie von Architekturen an (im Gegensatz dazu würden sie heute eher Fachkenntnisse in einer Programmiersprache oder Familie von Programmiersprachen sammeln, mit der Zuversicht, dass ihre Kenntnisse, auf welcher Hardware sie auch arbeiten mögen, übertragbar wären). Weil die Erfahrungen einer Person dazu neigten sich auf einen Computer zu beschränken, hatte die Aneignung von Erfahrung die Folge, diesen Computer für sie und ihre Kollegen attraktiver zu machen. Deshalb war es im Interesse des Herstellers, dass für seine Maschinen spezifisches Wissen und Code sich so weit wie möglich verbreitete.

Zweitens gab es kein Internet. Obwohl es weniger rechtliche Einschränkungen gab als heute, die den Austausch von Software verhinderten, gab es mehr technische: Es gab ein ein paar wenige, lokale Netzwerke, die gut waren um Informationen unter Mitarbeitern der gleichen Forschungseinrichtung auszutauschen. Aber es blieben Barrieren die es zu überwinden galt, wenn man mit jedem teilen wollte, unab-

hängig von seinem Standort. Diese Barrieren *wurden* in vielen Fällen überwunden. Manchmal stellten verschiedene Gruppen eigenständig Kontakt zueinander her. Sie sandten einander Disketten oder Magnetbänder mittels Post zu und manchmal fungierten die Hersteller selbst als zentrale Anlaufstelle für Patches. Es half auch, dass viele frühe Computer-Entwickler an Universitäten arbeiteten, wo die Veröffentlichung des eigenen Wissens erwartet wurde. Die physische Realität der Datenübertragung bedeutete jedoch, dass der Austausch immer einen Widerstand mit sich brachte, der proportional zur Entfernung (echte oder organisatorische) anwuchs, die von der Software überwunden werden musste. Weitverbreitetes reibungsloses Tauschen, wie wir es heute erleben, war nicht möglich.

Der Aufstieg proprietärer und freier Software

Mit der Reifung der Industrie ergaben sich mehrere zusammenhängende Veränderungen. Der Wildwuchs bei den Hardware-Architekturen konsolidierte sich auf einige wenige Gewinner – Gewinner durch überlegene Technologie, überlegenes Marketing, oder eine Kombination beider. Gleichzeitig, und nicht ganz zufällig, hatte die Entwicklung sogenannter "höherer" Programmiersprachen zur Folge, dass man Anwendungen in einer Sprache schreiben konnte, und es automatisch übersetzen ("kompilieren") lassen konnte, sodass es auf viele verschiedene Computer laufen konnte. Die Folgen hiervon blieben den Hardware-Herstellern nicht verborgen: Ein Kunde konnte jetzt ein großes Software-Projekt in Angriff nehmen, ohne sich auf eine bestimmte Computer-Architektur festzulegen. Diese Tatsache, zusammen mit den allmählich kleiner werdenden Unterschieden zwischen den verschiedenen Marken (da weniger effiziente Architekturen ausgesiebt wurden), zwang Hersteller, die ihre Hardware als ihr einziges Gut behandelten, sich auf sinkende Gewinnmargen einzustellen. Rohe Rechenleistung wurde ein ersetzbares Gut, während Software zum Unterscheidungsmerkmal wurde. Software zu verkaufen, oder zumindest es als integrierten Bestandteil der Hardware zu verkaufen, wurde zu einer verlockenden Strategie.

Hersteller fingen also an, die Urheberrechte auf ihren Quellcode strenger durchzusetzen. Wenn Nutzer einfach weiterhin frei miteinander Code tauschen und modifizieren konnten, würden sie vielleicht manche der Verbesserungen eigenständig neu implementieren, die der Hersteller als "Mehrwert" verkaufen wollte. Schlimmer noch, getauschter Quellcode könnte an die Konkurrenz gelangen. Ironisch dabei ist, dass all das zur selben Zeit geschah, als das Internet endlich anfing abzuheben. Gerade als der echte reibungslose Austausch von Software endlich technisch machbar wurde, machten Veränderungen in der Computerindustrie es wirtschaftlich nicht wünschenswert, zumindest aus Sicht der einzelnen Unternehmen. Die Hersteller wurden restriktiver, entweder verwehrten sie den Nutzern den Zugriff auf den Quellcode der auf ihren Maschinen lief, oder sie bestanden auf Vertraulichkeitsvereinbarungen, die das Tauschen praktisch unmöglich machten.

Bewusster Widerstand

Langsam ließ der unbeschränkte Austausch von Quellcode überall in der Softwareindustrie nach. Überall? Zumindest im Kopf eines Programmierers kristallisierte sich eine Gegenreaktion. Richard Stallman arbeitete im Labor für künstliche Intelligenz am Massachusetts Institute of Technology in den 1970ern und frühen '80ern. Es sollte sich herausstellen, dass dies die goldene Zeit und der goldene Ort für den freien Austausch von Quellcode war. Das KI-Labor hatte eine starke "Hacker-Ethik", [2] und Leute wurden nicht nur dazu ermutigt sondern es wurde von ihnen erwartet, alle Verbesserungen am System mit Anderen zu teilen. Stallman schrieb später:

> *Wir nannten unsere Software nicht "freie Software" weil es diesen Begriff noch nicht gab; aber genau das war es. Immer als Leute von anderen Universitäten oder Firmen eine Anwendung benutzen und portieren wollte, konnten sie das gerne machen. Wenn du jemand bei der Nutzung einer unbekannten und interessanten Anwendung gesehen hast, konntest du immer darum bitten dir den Quellcode anzuschauen, um es zu lesen, zu verändern oder Teile davon für eine neue Anwendung auszuschlachten.*

[2] Stallman benutzt das Wort "Hacker" im ursprünglichen Sinne von "Jemand der es liebt zu Programmieren und Spaß daran hat sich dabei geschickt anzustellen", nicht im Sinne der relativ neuen Bedeutung von "Jemand der in Computer einbricht".

(von **http://www.gnu.org/gnu/thegnuproject.html**)

Die Edenische Gemeinschaft um Stallman herum, brach kurz nach 1980 zusammen, als die Veränderungen der Industrie letztendlich das KI-Labor einholten. Eine Startup-Firma stellte viele Programmierer des Labors ein, um an einem Betriebssystem zu arbeiten, ähnlich zu dem welches sie im Labor programmiert hatten, diesmal aber unter einer exklusiven Lizenz. Gleichzeitig, schaffte sich das KI-Labor neue Ausrüstung an, welches mit einem proprietären Betriebssystem ausgeliefert wurde.

Stallman sah in den Geschehnissen das größere Muster:

> *Die modernen Computer dieser Ära, wie z.B. der VAX oder die 68020, hatten ihre eigenen Betriebssysteme, aber keines davon war freie Software: Man musste eine Vertraulichkeitsvereinbarung unterschreiben, nur um eine ausführbare Kopie zu bekommen.*
>
> *Um einen neuen Computer zu nutzen, musste man also als allererstes versprechen, seinen eigenen Nachbarn nicht zu helfen. Eine gemeinschaftliche Zusammenarbeit war verboten. Die Hersteller proprietärer Software stellten die Regel auf: "Wenn du mit deinem Nachbarn teilst, bist du ein Pirat. Wenn du irgendwelche Änderungen haben möchtest, dann musst du bei uns darum betteln."*

Stallman entschied sich aus irgend eine persönliche Eigenart heraus Widerstand gegen diese Entwicklung zu leisten. Anstatt für das nunmehr dezimierte KI-Labor weiterzuarbeiten, oder eine Arbeit bei einer der neuen Firmen anzunehmen, wo die Ergebnisse seiner Arbeit verschlossen in einer Kiste wären, kündigte er dem Labor und gründete das GNU Projekt und die Free Software Foundation (FSF). Das Ziel von GNU [3] war es ein komplett freies und offenes Betriebssystem und eine Reihe von Anwendungen zu entwickeln, die jedem Benutzer ermöglichen sollte an der Software zu hacken, sowie ihre Änderungen untereinander zu teilen. Im wesentlichen machte er sich auf den Weg wiederherzustellen, was im KI-Labor zerstört wurde, aber in einer weltumspannenden Größenordnung und ohne die Schwachstellen welche die Kultur des KI-Labors verwundbar gemacht hatten.

Zusätzlich zur Arbeit am neuen Betriebssystem, entwarf er eine urheberrechtliche Lizenz, deren Bedingungen garantierten, dass sein Code für immer frei bleiben würde. Die GNU General Public License (GPL) ist ein cleveres Stück juristisches Judo: Es besagt, dass Code ohne Einschränkungen kopiert und verändert werden darf, und dass sowohl Kopien als auch abgeleitete Werke (das heißt veränderte Versionen) unter derselben Lizenz wie das Original, ohne weitere Einschränkungen, freigegeben werden müssen. Tatsächlich benutzt es das Urheberrecht um genau das Gegenteil zu erreichen wozu es üblicherweise benutzt wird: Anstatt die Verbreitung der Software einzuschränken, hindert es *jeden*, sogar den Autor, an seiner Einschränkung. Für Stallman war das besser als seinen Code einfach als öffentliches Gut freizugeben. Ohne eine Lizenz könnte jede beliebige Kopie in eine proprietäre Anwendung aufgenommen werden (ein bekanntes Phänomen bei Code welcher unter toleranten Lizenzen veröffentlicht wird). Obwohl solch eine Einbindung in keiner Weise die weitere Verfügbarkeit des Codes einschränken würde, könnten die Anstrengungen von Stallman dadurch dem Feind – proprietäre Software – zum Vorteil gereichen. Die GPL kann man als einen Schutz für freie Software betrachten, da es nicht-freie Software daran hindert, seinen GPL Code komplett auszunutzen. Die GPL und ihre Beziehung zu anderen freien Software-Lizenzen werden im Kapitel ??? ausführlich behandelt.

Mit der Unterstützung vieler anderer Programmierer, teils Gleichgesinnte von Stallman, die seine Ideologie teilten, teils andere, die einfach nur möglichst viel freien Quellcode wollten, fing das GNU Projekt an, freien Ersatz für viele der wichtigsten Komponenten eines Betriebssystems zu veröffentlichen. Die nunmehr weit verbreitete Standardisierung von Hardware und Software erlaubte den Einsatz von GNU Software in ansonsten proprietären Systemen, was von vielen gemacht wurde. Der GNU Text-Editor (Emacs) und C Compiler (GCC) fanden großen Anklang, nicht nur wegen der Ideologie, sondern ein-

[3] Abkürzung für "GNU's Not Unix", und das "GNU" in dieser Erweiterung steht für... dasselbe.

fach aufgrund ihrer technischen Vorzüge. Bis ca. 1990, hatte GNU den Großteil eines freien Betriebssystems fertiggestellt. Es fehlte noch ein Kernel – also die Software die beim Hochfahren geladen wird und für die Verwaltung von Arbeitsspeicher, Festplatten und anderer Ressourcen des Systems zuständig ist.

Leider hatte das GNU Projekt eine Kernelstruktur gewählt, die schwieriger zu implementieren war, als sie es erwartet hatten. Die dadurch entstandene Verzögerung hinderte die Free Software Foundation daran, die erste Veröffentlichung eines komplett freien Betriebssystems zu machen. Das letzte Stück wurde statt dessen von Linus Torvalds, einem finnischen Informatik-Studenten hervorgebracht, der mit der Hilfe von vielen Freiwilligen, verteilt auf der ganzen Welt, einen freien Kernel fertig gestellt hatte, der auf einem viel konservativeren Aufbau basierte. Er nannte es Linux, und als es mit den bereits existierenden GNU Anwendungen kombiniert wurde, war das Ergebnis ein komplett freies Betriebssystem. Zum ersten Mal konnte man seinen Computer ohne proprietäre Software hochfahren und damit arbeiten. [4]

Viel Software dieses neuen Betriebssystems wurde nicht vom GNU Projekt produziert. In der Tat war GNU nicht einmal die einzige Gruppe die daran arbeitete ein freies Betriebssystem herzustellen (ein Beispiel ist der Code der letztendlich in NetBSD und FreeBSD aufging, an dem zu dieser Zeit bereits entwickelt wurde). Die Free Software Foundation war nicht nur aufgrund des Codes den sie produzierten von Bedeutung, sondern auch wegen ihrer Rhetorik. Indem sie von freier Software eher als Glaubenssache sprachen wofür es sich zu kämpfen lohnt, und nicht als eine Sache der Bequemlichkeit, machten sie es für Programmierer schwierig *nicht* ein politisches Bewusstsein darüber zu haben. Selbst wenn man der FSF nicht zustimmte musste man sich mit dem Thema befassen, wenn auch nur um eine andere Position einzunehmen. Die Wirksamkeit der FSF als Propagandisten bestand darin, ihren Code mittels der GPL und anderer Texte an eine Botschaft zu binden. Zusammen mit ihrem Code verbreitete sich auch ihre Botschaft.

Zufälliger Widerstand

Es gab viele andere Vorgänge in der aufkeimenden Szene der freien Software und wenige waren derart offen ideologisch wie das GNU Projekt von Stallman. Einer der wichtigsten war die *Berkeley Software Distribution* (*BSD*), eine schrittweise Neuimplementierung des Unix-Betriebssystems – bis in die späten 1970ern ein loses proprietäres Forschungsprojekt von Programmierern bei AT&T – an der Universität von Kalifornien in Berkeley. Die BSD Gruppe machte keine offenkundigen politischen Äußerungen darüber, dass Programmierer sich verbünden und miteinander teilen mussten, aber sie *praktizierten* die Idee mit Spürsinn und Enthusiasmus, indem sie eine massive verteilte Anstrengung koordinierten, bei dem Unix-Konsolen-Anwendungen, Code-Bibliotheken und schließlich das Betriebssystem selbst von Grund auf, größtenteils von Freiwilligen, neu geschrieben wurde. Das BSD Projekt wurde zum Aushängeschild für Entwicklung freier Software ohne ideologischen Hintergrund und diente als Ausbildungsstätte für viele Entwickler die später in der Open-Source-Welt weiterhin aktiv bleiben sollten.

Eine weitere Feuerprobe der kooperativen Entwicklung, war das *X Window System*, eine freie netzwerktransparente grafische Benutzerumgebung, welches Mitte der 1980er am MIT in Zusammenarbeit mit Hardware-Anbietern entwickelt wurde, die ein gemeinsames Interesse daran hatten ihren Kunden ein Fenstersystem anbieten zu können. Weit davon entfernt sich proprietärer Software entgegenzustellen, erlaubte die X Lizenz ganz bewusst proprietäre Erweiterungen auf seinem freien Kern – alle Beteiligten des Konsortiums wollten die Möglichkeit die gängige X Version zu verbessern und sich dadurch von den anderen Mitgliedern abzuheben. X Windows [5] selbst war freie Software, aber hauptsächlich als Mittel um das Spielfeld zwischen konkurrierenden wirtschaftlichen Interessen zu ebnen, nicht als Wunsch die Vormacht proprietärer Software zu brechen. Ein weiteres Beispiel, das dem GNU Projekt ein paar Jahre vorausging war TeX von Donald Knuth, ein freies Textsatzsystem für druckfertige Dokumente. Er

[4] Rein technisch gesehen war Linux nicht das erste freie Betriebssystem, sondern das kurz zuvor erschienene 386BSD für IBM-kompatible Rechner. 386BSD war jedoch viel schwieriger zum Laufen zu bringen. Linux sorgte für Aufregung, nicht nur weil es frei war, sondern weil man es auf seinem Computer tatsächlich zum Laufen bringen konnte.

[5] Sie bevorzugen den Namen "X Window System", aber für gewöhnlich nennt man es "X Windows", da drei Wörter einfach zu schwerfällig sind.

veröffentlichte es unter einer Lizenz, welche jedem erlaubte es zu modifizieren und zu veröffentlichen, solange man das Ergebnis nicht "TeX" nannte, wenn es nicht einen strikten Satz an Prüfungen bestand, die Kompatibilität gewährleisten sollten (TeX ist ein Beispiel für eine Klasse von Lizenzen für freie Software die ein Markenzeichen schützen sollen, welches im Kapitel Kapitel 9, *Lizenzen, Urheberrecht und Patente* ausführlicher behandelt wird). Knuth bezog nicht Stellung für die eine oder andere Partei in Bezug auf die Frage freier gegen proprietäre Software, er brauchte nur ein besseres Textsatzsystem um sein *echtes* Ziel zu erreichen – ein Buch über Softwareentwicklung zu schreiben – und sah, als er fertig war, keinen Grund sein System nicht der Welt zur Verfügung zu stellen.

Ohne hier jedes Projekt und jede Lizenz aufzulisten, kann man doch mit Sicherheit sagen, dass Ende der 1980er eine Menge freier Software unter einer breiten Auswahl an Lizenzen zu Verfügung stand. Die Vielfalt an Lizenzen spiegelte eine entsprechende Vielfalt an Motivationen wider. Selbst einige der Programmierer welche die GNU GPL wählten waren viel weniger ideologisch getrieben als das GNU Projekt selbst. Obwohl sie es genossen an freier Software zu arbeiten, betrachteten viele Entwickler proprietäre Software nicht als soziales Übel. Es gab Menschen die einen moralischen Drang spürten die Welt von "Software-Hortung" (der Begriff den Stallman für nicht freie Software benutzt) zu befreien, andere waren jedoch eher durch technische Begeisterung motiviert, durch die Freude an der Zusammenarbeit mit Gleichgesinnten, sogar durch das einfache menschliche Bedürfnis nach Ruhm. Dennoch beeinflussten sich diese ungleichen Motivationen größtenteils nicht negativ. Teilweise liegt das daran, dass Software im Gegensatz zu anderen kreativen Aktivitäten wie Prosa oder die bildende Kunst einige mehr oder weniger objektive Prüfungen bestehen muss um als erfolgreich erachtet zu werden: Es muss Laufen und zu einem gewissen Maß frei von Fehlern sein. Dadurch haben alle Teilnehmer eines Projekts automatisch grundlegende gemeinsame Interessen, eine Basis und ein Rahmenwerk um miteinander zu arbeiten, ohne sich all zu viele Sorgen um eine Qualifizierung außerhalb des technischen machen zu müssen.

Entwickler hatten noch einen weiteren Grund zusammenzuhalten: es stellte sich heraus, dass die Welt freier Software qualitativ sehr hochwertigen Code produzierte. Manchmal war es aus technischer Sicht der nächstbesten nicht-freien Alternative nachweislich überlegen; manchmal war es zumindest vergleichbar und natürlich kostete es weniger. Auch wenn nur wenige motiviert gewesen wären freie Software aus rein philosophischen Gründen zu nutzen, waren viele mehr als glücklich sie wegen ihrer technischen Überlegenheit zu nutzen. Und von denen die es benutzten war immer irgend ein Bruchteil bereit ihre Zeit und Fähigkeiten zu spenden, um bei der Pflege und Verbesserung der Software mitzuhelfen.

Diese Tendenz guten Code zu produzieren war sicherlich nicht überall gegeben, aber bei freien Software-Projekten auf der ganzen Welt zunehmend oft der Fall. Geschäftszweige die zu einem gewichtigen Teil von Software abhingen fingen langsam an davon Wind zu bekommen. Viele bemerkten, dass in ihren Betrieben bereits jeden Tag freie Software eingesetzt wurde, ohne es gewusst zu haben (die Geschäftsleitung wird nicht immer über alles informiert, was sich in der IT-Abteilung abspielt). Firmen fingen an eine aktivere Rollen bei freien Software-Projekte einzunehmen, manchmal trugen sie mit Zeit und Ausrüstung bei, manchmal auch direkt durch finanzielle Unterstützung der Entwicklung freier Software-Anwendungen. Solche Investitionen konnten sich im Idealfall um ein Vielfaches auszahlen. Der Geldgeber stellt lediglich eine kleine Gruppe erfahrener Entwickler ein, die sich ganztags einem Projekt widmen, profitiert aber von den Beiträgen *aller* Beteiligten, also auch von unbezahlten Freiwilligen und Programmierern anderer Unternehmen.

"Frei" kontra "Open Source"

Mit zunehmender Aufmerksamkeit aus der Unternehmenswelt wurden Programmierer freier Software mit Fragen der Präsentation konfrontiert. Eines war das Wort "frei"[6] selbst. Wenn man den Begriff

[6]Anm. d. Übersetzer Das Wort "frei" wird hier für das englische Wort "free" benutzt. Es hat eine ähnliche Doppeldeutigkeit wie im Englischen, auch wenn man im deutschen zur besseren Unterscheidung den Begriff "kostenlos" verwenden kann was im Englischen nicht so leicht möglich ist.

"freie Software" zum ersten Mal hört, denken viele es bedeute lediglich "gratis Software". Es stimmt zwar, dass freie Software auch kostenlos ist[7], aber nicht jede Software die nichts kostet ist auch frei. Ein Beispiel sind die Browser-Kriege der 1990er, indem sowohl Netscape als auch Microsoft ihre konkurrierenden Browser kostenlos anboten, um eilig einen möglichst großen Marktanteil zu erlangen. Keiner von ihnen war jedoch frei im Sinne "freier Software". Man hatte keinen Zugriff auf den Quellcode und selbst wenn, hätte man nicht die Rechte ihn zu modifizieren und weiterzugeben[8]. Man konnte lediglich eine ausführbare Datei herunterladen und laufen lassen. Die Browser waren nicht freier als eingeschweißte Software aus dem Geschäft; sie waren lediglich etwas günstiger.

Die Verwirrung um das Wort "frei" ist einer ganz und gar unglücklichen Doppeldeutigkeit der Englischen Sprache zuzuschreiben. Die meisten anderen Sprachen unterscheiden zwischen einem niedrigen Preis und Freiheit (die Unterscheidung zwischen *gratis* und *libre* leuchtet den meisten Sprechern romanischer Sprachen sofort ein). Die Stellung der englischen Sprache als de-facto Brückensprache des Internets hat zur Folge, dass dieses Problem zu einem gewissen Grad ein Problem aller ist. Die Missverständnisse um das Wort "free" waren so verbreitet, dass die Programmierer freier Software schließlich eine Standard-Formulierung als Reaktion parat hielten: "Es ist *frei* im Sinne von *Freiheit* – denke an die *Redefreiheit*, nicht an *Freibier*."[9]. Diese Erklärung dauernd wiederholen zu müssen ist aber auf Dauer ermüdend. Viele Programmierer fühlten, teils zu Recht, dass die Zweideutigkeit des Worts "frei" das Verständnis in der Öffentlichkeit behinderte.

Das Problem war aber viel schwerwiegender. Das Wort "frei" trug eine unausweichliche moralische Konnotation: Wenn die Freiheit ein Ziel für sich war, dann machte es keinen Unterschied ob die Software zufällig auch besser oder unter bestimmten Bedingungen auch für bestimmte Geschäfte profitabler war. Das waren lediglich angenehme Nebeneffekte einer Motivation, die im Grunde genommen weder technischer noch geschäftlicher, sondern moralischer Natur war. Zusätzlich wurde Firmen durch den Standpunkt "frei im Sinne von Freiheit" eine grelle Inkonsistenz aufgezwungen, die für einen Geschäftszweig freie Anwendungen unterstützen wollten, aber für andere weiterhin proprietäre Software vertrieben.

Diese Dilemma trafen auf eine Gemeinschaft die sich bereits mit einer Identitätskrise konfrontiert sah. Programmierer die wirklich freie Software *schreiben*, waren sich nie sonderlich einig über das Endziel der freien Software-Bewegung, wenn es überhaupt ein solches gibt. Zu sagen, dass die Meinungen von einem Extrem zum anderen laufen wäre aber irreführend, da es fälschlicherweise eine lineare Reihe implizieren würde, wo es in Wirklichkeit eine mehrdimensionale Verteilung gibt. Wir können jedoch grob zwischen zwei Glaubensrichtungen unterschieden, wenn wir uns darauf einigen können Feinheiten außen vor zu lassen. Eine Gruppe vertritt die Ansicht von Stallman, dass die Freiheit zu Teilen und zu Modifizieren das Wichtigste ist. Sollte man also aufhören über Freiheit zu reden, dann hat man die Kernfrage weggelassen. Andere sind der Meinung, die Software selbst sei das wichtigste Argument für freie Software und fühlen sich unwohl bei der Behauptung, proprietäre Software sei an und für sich schlecht. Manche, aber nicht alle Programmierer freier Software, glauben der Autor (bzw. bei bezahlter Arbeit der Arbeitgeber) *sollte* das Recht haben die Bedingungen zu bestimmen, mit denen die Software verteilt werden darf und dass keine moralische Beurteilung an eine bestimmte Wahl von Bedingungen geknüpft sein muss.

Lange Zeit mussten diese Meinungsverschiedenheiten nicht klar untersucht oder formuliert werden. Durch den anbrechenden Erfolg in der Geschäftswelt wurde die Angelegenheit aber unausweichlich. 1998 wurde der Begriff *Open Source* als Alternative zu "frei" erschaffen, durch eine Vereinigung von Programmierern die letztendlich zur Open-Source-Initiative (OSI) wurde.[10] Die OSI war der Meinung, dass der Begriff "freie Software" nicht nur potentiell verwirrend war, sondern dass es ein Symptom

[7] Man kann eine Gebühr für die Aushändigung von Kopien freier Software verlangen, da man aber den Empfänger nicht daran hindern kann es danach kostenlos weiterzugeben, läuft der Preis effektiv sofort gegen null

[8] Der Quellcode vom Netscape Navigator *wurde* letztendlich 1998 unter eine Open-Source-Lizenz gestellt, und wurde zur Grundlage für den Mozilla-Browser. Siehe http://www.mozilla.org/.

[9] engl.: "It's *free* as in *freedom*—think *free speech*, not *free beer*."

[10] Die Webseite von OSI ist http://www.opensource.org/.

eines allgemeinen Problems war: Die Bewegung brauchte eine Marketing-Kampagne um das Konzept der Geschäftswelt schmackhaft zu machen und um zu verhindern, dass das Gerede über Moral, soziale Vorteile und zügellosen Austausch niemals in den Vorstandsetagen ankommen würde. Mit anderen Worten:

> *Die Open-Source-Initiative ist eine Marketingkampagne für freie Software. Es ist die Anpreisung freier Software auf einer pragmatischen Basis anstatt auf geschwollenem ideologischem Gerede. An der erfolgreichen Substanz hat sich nichts geändert, an der schlechten Einstellung schon. ...*
>
> *Was den meisten Technikfreaks klargemacht werden muss ist nicht das Konzept von Open Source, sondern der Name. Warum sollen wir es nicht wie bisher "freie Software" nennen?*
>
> *Ein Grund ist, dass der Begriff "freie Software" leicht missverstanden wird und dadurch zu Konflikten führen kann. ...*
>
> *Der echte Grund für die Umbenennung ist aber eines der Vermarktung. Wir versuchen jetzt unser Konzept der Geschäftswelt zu verkaufen. Wir haben ein gutes Produkt, waren aber bisher furchtbar aufgestellt. Der Begriff "freie Software" wurde von Geschäftsleuten falsch verstanden, die den Wunsch zu teilen als Anti-Kommerz verstanden; gar schlimmer noch, als Diebstahl.*
>
> *Die durchschnittliche Firmenleitung wird niemals "freie Software" kaufen. Wenn wir aber genau die gleiche Tradition, die selben Menschen und die selben freien Software-Lizenzen nehmen und den Namen in "Open Source" ändern? Dann kaufen Sie es gerne.*
>
> *Manche Hacker finden das schwer zu glauben, was aber nur daran liegt, dass sie logisch denken und nur konkrete Tatsachen betrachten. Sie verstehen nicht wie wichtig das Erscheinungsbild ist, wenn man etwas verkaufen will.*
>
> *Bei der Vermarktung entspricht das Erscheinungsbild auch der Wirklichkeit. Der Anschein, dass wir bereit sind von unseren Barrikaden herunterzukommen um mit Geschäftsleuten zu arbeiten zählt genauso viel wie unser tatsächliches Verhalten, unsere Überzeugungen und unsere Software.*
>
> (von **http://opensource.feratech.com/advocacy/faq.php** und **http://opensource.feratech.com/advocacy/case_for_hackers.php#marketing**)

Die Spitzen vieler Eisberge sind in diesem Text sichtbar. Es spricht von "unseren Überzeugungen", vermeidet aber schlauerweise klar zu formulieren was diese sind. Für manche mag es die Überzeugung sein, dass Code der in einem offenen Prozess entwickelt wurde besser sei; für andere mag es wiederum die Überzeugung sein, dass alle Informationen geteilt werden sollten. Das Wort "Diebstahl" wird benutzt, um (vermutlich) auf illegales Kopieren hinzuweisen – wozu viele Vorbehalte auf der Grundlage hegen, dass es kein Diebstahl ist, wenn der ursprüngliche Besitzer nachher noch den Gegenstand besitzt. Der Text gibt einen verlockenden Hinweis, dass die Bewegung der freien Software versehentlich als anti-kommerziell beschuldigt werden könnte, untersucht aber vorsichtigerweise nicht ob solch eine Beschuldigung vielleicht einen Grundlage hat.

Das soll nicht heißen die Webseite der OSI wäre inkonsistent oder irreführend. Das ist sie nicht. Es ist vielmehr ein Beispiel für genau das was die OSI behauptet der freien Software-Bewegung fehlen würde: eine gute Vermarktung, wobei "gut" hier "brauchbar für die Geschäftswelt" bedeutet. Die Open-Source-Initiative gab vielen genau das wonach sie gesucht hatten – ein Wortschatz um über freie Software als Entwicklungsprinzip und Geschäftsstrategie zu sprechen, anstatt als moralischen Kreuzzug.

Die Entstehung der Open-Source-Initiative veränderte die Landschaft der freien Software. Es formalisierte einen Zwiespalt der lange ohne Namen geblieben war, und zwang die Bewegung so, die Tatsache anzuerkennen, dass es sowohl eine interne als auch eine externe Politik hatte. Beide Seiten waren dadurch gezwungen eine gemeinsame Basis zu finden, denn die meisten Projekte haben Programmierer aus beiden Lagern, sowie solche die sich nicht klar einer Kategorie zuordnen lassen. Was nicht bedeutet, dass die Leute nie über moralische Motivationen reden – Manchmal wird zum Beispiel auf Fehler in der traditionellen "Hacker-Ethik" hingewiesen. Aber es ist selten, dass ein Entwickler freier / Open-Source-Software offen die Motive anderer im Projekt in Frage stellt. Die Beteiligung ist wichtiger als der Beteiligte. Wenn jemand guten Code schreibt, sollte man sie nicht fragen ob sie es aus moralischen Gründen machen, weil sie dafür bezahlt werden, weil sie ihren Lebenslauf erweitern wollen, oder aus sonstwelchen Gründen. Man beurteilt den Beitrag den sie leisten auf technischer Ebene und antwortet auf technischer Ebene. Selbstorganisationen wie Debian, die explizit politischer Natur sind, dessen Ziel es ist eine komplett freie Rechenumgebung bereitzustellen, sind relativ locker wenn es darum geht mit nicht-freiem Code zu funktionieren und mit Programmierern zusammenzuarbeiten, die nicht genau die selben Ziele teilen.

Die Situation heute

Beim Betrieb eines freien Software-Projekts werden Sie nicht täglich über derart schwerwiegende philosophische Themen reden müssen. Kein Programmierer wird darauf bestehen, dass jeder andere im Projekt die gleichen Ansichten vertritt wie er selbst (und wer es tut wird ganz schnell feststellen, dass er sich an keinem Projekt beteiligen kann). Sie sollten sich aber darüber im Klaren sein, dass die Kontroverse "frei" kontra "Open Source" existiert, teils um Aussagen zu vermeiden die von manchen Teilnehmern als feindlich aufgefasst werden könnten, teils da ein Verständnis über die Motivationen der Entwickler der beste – in gewissem Sinne der *einzige* – Weg ist ein Projekt zu leiten.

Freie Software ist eine Kultur die man sich aussucht. Um darin erfolgreich wirken zu können, muss man verstehen warum Leute überhaupt die Wahl treffen sich daran zu beteiligen. Sie zu zwingen funktioniert nicht. Wenn Leute in einem Projekt unglücklich sind, gehen sie einfach zu einem anderen. Freie Software ist selbst unter Gemeinschaften von Freiwilligen bemerkenswert darin nur eine geringe Investition zu fordern. Die meisten Beteiligten sind einander noch nie wirklich von Angesicht zu Angesicht begegnet und spenden kleine Häppchen ihrer Zeit, wann immer ihnen danach ist. Die Fäden, die Menschen für gewöhnlich verbindet, um eine beständige Gruppe zu formen, sind auf einen winzigen Kanal reduziert: das geschriebene Wort, übertragen durch elektrische Leitungen. Aus diesem Grund kann es lange dauern, bis sich eine geschlossene, engagierte Gruppe bildet. Umgekehrt kann eine Gruppe schon während der ersten fünf Minuten ihrer Begegnung mit einem potentiellen Freiwilligen sein Interesse sehr leicht verspielen. Wenn ein Projekt keinen guten ersten Eindruck macht werden Neulinge ihm selten eine zweite Chance geben.

Die Vergänglichkeit der Beziehungen, bzw. ihre *potenzielle* Vergänglichkeit, ist das vielleicht größte Problem dem ein neues Projekt sich stellen muss. Was soll all diese Menschen dazu bewegen lange genug zusammen zu bleiben um etwas Nützliches zu produzieren? Die Antwort auf diese Frage ist komplex genug, um sie zum Thema dieses Buchs zu machen. Müsste man sie jedoch in einem Satz zusammenfassen, wäre es folgender:

> *Menschen sollten spüren, dass ihre Verbindung zu einem Projekt und ihr Einfluss darauf proportional zu ihrer Beteiligung ist.*

Kein Entwickler oder potentieller Entwickler sollte sich jemals aus nicht-technischen Gründen herabgesetzt oder diskriminiert fühlen. Projekte mit Unterstützung durch Firmen müssen in dieser Hinsicht ganz besonders vorsichtig sein. Der Abschnitt Kapitel 5, *Geld* behandelt dieses Thema im Detail. Keine Unterstützung durch eine Firma zu haben, bedeutet natürlich nicht, dass man sich um nichts Sorgen machen muss. Geld ist nur einer von vielen Faktoren, die den Erfolg eines Projekts beeinflussen können. Es sind ebenso die Fragen zu klären, welche Sprache, welche Lizenz und welche Art der Entwicklung

man wählen sollte, welche Infrastrukturen man aufbauen sollte, wie man die Gründung des Projekts am besten bekanntgeben sollte und vieles mehr. Wie man ein Projekt auf dem richtigen Fuß startet ist Thema des nächsten Kapitels.

Kapitel 2. Der Einstieg

Den klassischen Verlauf eines freien Software-Projektes beschreibt Eric Raymond in seinem nunmehr berühmten Text über Open Source, *The Cathedral and the Bazaar*:

> *Jede gute Software entsteht aus einem persönlichen Bedürfnis eines Programmierers.*
>
> (von **http://www.catb.org/~esr/writings/cathedral-bazaar/**)

Raymond sagte wohlgemerkt nicht, dass Open-Source-Projekte nur aus dem persönlichen Bedürfnis eines Programmierers entstehen, sondern dass *gute* Software dann entsteht, wenn der Programmierer ein persönliches Interesse daran hat ein Problem zu lösen; was insofern für freie Software relevant ist, da sich herausstellte, dass die meisten Open-Source-Projekte von Programmierern begonnen wurden, die eines ihrer eigenen Probleme lösen wollten.

Die Motivation für die meisten freien Software-Projekte ist auch heute noch dieselbe, wenn auch in geringerem Maße als um 1997, zu der Zeit als Raymond diese Worte schrieb. Heute beobachten wir das Phänomen, dass Organisationen – auch gewinnorientierte Unternehmen – große, zentral organisierte Open-Source-Projekte anfangen. Der einsame Programmierer, der ein bisschen Code produziert um ein lokales Problem zu lösen und dann feststellt, dass das Ergebnis eine breitere Anwendbarkeit besitzt, ist immer noch die Quelle für vieles an freier Software, jedoch nicht mehr die einzige.

Raymonds Sicht ist aber immer noch aufschlussreich. Diejenigen, die Software produzieren, müssen ein direktes Interesse an ihrem Erfolg haben, weil sie auch Benutzer sind. Wenn die Software nicht macht was es soll, wird die Person oder Organisation, die sie produziert, es bei der täglichen Arbeit merken. Ein gutes Beispiel ist das OpenAdapter Projekt (http://www.openadapter.org/) der Investmentbank Dresdner Kleinwort Wasserstein, das als Open-Source-Framework zur Integration unterschiedlicher finanzieller Informationssysteme begonnen wurde. Es kann wohl kaum als Bedürfnis eines eines einzelnen Programmierers bezeichnet werden, sondern als institutionelles Bedürfnis. Dieses Bedürfnis entsteht aber direkt aus den Erfahrungen der Institution und ihrer Partner. Wenn die Software ihre Aufgabe nicht erfüllt, wird es bemerkt. Aus diesem Arrangement entsteht gute Software, da Rückmeldungen an der richtigen Stelle ankommen. Die Software muss nicht verkauft werden, also können sie sich auf *ihre* Probleme konzentrieren. Sie wird geschrieben um die *eigenen* Probleme zu lösen, um diese Lösung dann mit allen zu teilen. Es ist so, als wäre das Problem eine Krankheit und die Software die entsprechende Medizin um eine Epidemie in den Griff zu bekommen.

In diesem Kapitel geht es um die Frage, wie man der Welt ein neues freies Software-Projekt vorstellt. Viele seiner Empfehlungen klingen sehr nach einer Gesundheitsorganisation, die Medizin verteilen will. Die Ziele sind sich sehr ähnlich: Man will klarstellen, was die Medizin macht, sie in die Hände der richtigen Leute bringen und sicherstellen, dass diejenigen, die es erhalten, damit umzugehen wissen. Bei freier Software will man aber auch ein paar der Empfänger zu einer Beteiligung an der fortwährenden Forschungsarbeit zur Verbesserung der Medizin bewegen.

Beim Vertrieb freier Software gibt es zwei Ziele. Die Software muss sowohl Nutzer als auch Entwickler anziehen. Diese beiden Anforderungen stehen zueinander nicht zwangsläufig im Widerspruch, aber sie machen die anfängliche Präsentation des Projekts etwas komplexer. Manche Informationen sind für beide Gruppen nützlich, manche nur für die eine oder die andere. Beide Arten der Information sollten das Prinzip der skalierenden Präsentation verfolgen; d.h. dass die Menge an Informationen sich jederzeit mit der Zeit und Anstrengung decken sollte, die vom Leser aufgebracht wird. Eine größere Anstrengung sollte auch immer eine größere Belohnung mit sich bringen. Wenn beides nicht eng miteinander korreliert, werden die Leser schnell die Hoffnung aufgeben und aufhören Zeit zu investieren.

Daraus folgt: *das Erscheinungsbild ist wichtig*. Programmierern fällt es besonders schwer, das zu glauben. Ihre Liebe zum Inhalt gegenüber dem Äußeren gehört fast schon zum Berufsethos. Es ist kein

Zufall, dass so viele Programmierer gegen Marketing und Public Relations eine Antipathie hegen, oder dass professionelle Grafiker über so manches erschrocken sind, worauf Programmierer von sich aus kommen.

Das ist schade, denn es gibt Situationen, in denen das Aussehen auch *wirklich* dem Inhalt entspricht. Die Präsentation eines Projekts ist genau solch ein Fall. Die erste Information, die ein Besucher über ein Projekt erhält, ist die Gestaltung seiner Webseite. Diese Information wird erfasst, bevor irgendein tatsächlicher Inhalt der Seite verstanden wird – bevor ein Text gelesen wurde oder auf einen Link geklickt wurde. Egal wie ungerecht es sein mag, Menschen können sich nicht anders helfen, als sich sofort einen ersten Eindruck zu verschaffen. Im Erscheinungsbild der Seite wird deutlich, ob man sich beim Aufbau der Präsentation des Projekts Mühe gemacht hat. Menschen haben eine extrem sensible Antenne dafür, wieviel Mühe in etwas investiert wurde. Die meisten können mit einem Blick erkennen, ob eine Seite eilig zusammengebastelt wurde oder ob man sich ernsthafte Gedanken gemacht hat. Das ist die erste Information, die Ihr Projekt nach außen gibt, und der hierdurch vermittelte Eindruck überträgt sich auf das übrige Projekt.

Auch wenn sich dieses Kapitel thematisch um inhaltliche Fragen dreht, sollten sie daran denken, dass das Erscheinungsbild eine Rolle spielt. Da die Webseite für zwei Arten von Besuchern – Benutzer und Entwickler – geeignet sein muss, muss besonders auf Klarheit und Führung geachtet werden. Auch wenn hier nicht die richtige Stelle für eine allgemeine Abhandlung über Web-Design ist, gibt es ein erwähnenswertes Prinzip, insbesondere wenn die Seite mehrere (überlappende) Zielgruppen ansprechen soll: Besucher sollten eine grobe Vorstellung haben, wo ein Link hinführt, bevor sie darauf klicken. Das Ziel eines Links zur Benutzer-Dokumentation sollte *allein vom Anblick her* deutlich sein, und keine Missverständnisse aufkommen lassen, ob es sich nicht etwa um die Dokumentation für Entwickler handelt. Beim Betrieb eines Projekts geht es zu einem Teil darum, Informationen bereitzustellen, aber auch darum, ein Gefühl der Bequemlichkeit zu vermitteln. Allein schon die Verfügbarkeit bestimmter grundsätzlicher Angebote an der richtigen Stelle gibt Benutzern und Entwicklern eine Sicherheit bei ihrer Entscheidung, ob sie sich beteiligen wollen oder nicht. Es sagt ihnen, dass dieses Projekt seine Siebensachen beisammen hat, Fragen erahnt, die gestellt werden, und sich die Mühe gemacht hat diese so zu beantworten, dass Fragesteller möglichst wenig Einsatz aufbringen müssen. Indem das Projekt eine Aura des Vorbereitetseins ausstrahlt, sendet es die folgende Botschaft aus: "Du verschwendest deine Zeit nicht, wenn du dich beteiligst". Das ist genau die Botschaft, die Menschen hören wollen.

Schauen Sie sich zuerst um

Bevor Sie ein Open-Source-Projekt anfangen, gibt es noch eine wichtige Warnung:

Schauen Sie sich vorher um, ob nicht schon ein Projekt existiert das Ihre Anforderungen erfüllt. Die Wahrscheinlichkeit ist hoch, dass unabhängig von dem Problem, das Sie lösen wollen, Ihnen jemand zuvorgekommen ist. Wenn das der Fall ist und der entsprechende Code unter eine freie Lizenz gestellt wurde, gibt es keinen Grund das Rad neu zu erfinden. Es gibt natürlich Ausnahmen: Falls Sie ein Projekt um der Lernerfahrung willen beginnen wollen, wird Ihnen bereits existierender Code nicht weiterhelfen. Vielleicht wissen Sie bereits von vornherein, dass Ihr Problem so spezifisch ist, dass es mit Sicherheit noch von keinem gelöst wurde. Im Allgemeinen gibt es aber keinen Grund sich nicht umzuschauen, und der Lohn kann beträchtlich sein. Sollten die üblichen Suchmaschinen keine brauchbaren Ergebnisse liefern, sollte Sie es bei http://freshmeat.net/ (eine Nachrichtenseite über Open-Source-Projekte (zu dieser Seite, später mehr), bei http://www.sourceforge.net/ oder beim Verzeichnis freier Software der Free Software Foundation http://directory.fsf.org/ versuchen.

Selbst wenn Sie nicht genau das finden, wonach Sie suchen, könnten Sie etwas derart ähnliches finden, dass es sinnvoller ist, sich an diesem Projekt zu beteiligen und es um die fehlenden Funktionen zu erweitern, als komplett von vorne anzufangen.

Mit dem Vorhandenen beginnen

Sie haben sich umgeschaut, herausgefunden dass nichts Ihre Anforderungen erfüllt und sich entschlossen, ein neues Projekt zu starten.

Was nun?

Das Schwierigste beim Start eines neuen freien Software-Projekts ist die Verwandlung einer persönlichen Vision in eine öffentliche. Sie oder Ihre Organisation mögen sehr wohl wissen was Sie wollen. Dieses Ziel so auszudrücken, dass die Welt es versteht, erfordert ein beträchtliches Maß an Arbeit. Allerdings ist es von so grundsätzlicher Bedeutung, dass Sie sich die Zeit dazu nehmen sollten. Sie und die anderen Gründer müssen entscheiden, worum es in dem Projekt wirklich geht – so müssen Sie über seine Grenzen entscheiden, was es abdecken wird und was *nicht* – und ein Missionsziel (engl. mission statement) verfassen. Dieser Teil ist für gewöhnlich nicht allzu schwer, auch wenn es manchmal unerwähnt gebliebene Annahmen, sogar Meinungsverschiedenheiten über die Natur des Projekts aufdecken kann, was sogar gut ist: es ist besser diese jetzt auszuräumen als später. Der nächste Schritt ist das Projekt für die öffentliche Wahrnehmung aufzubereiten, was im Grunde genommen eine Schinderei ist.

Diese Arbeit ist deshalb so mühselig, weil man hauptsächlich Dinge organisiert und dokumentiert, die jeder bereits kennt – d.h "jeder" der bislang am Projekt beteiligt ist. Für diejenigen, die die Arbeit machen, gibt es deshalb keinen direkten Nutzen. Sie brauchen weder die `README`-Datei für einen Überblick über das Projekt, noch Entwurfsdokumente oder Benutzerhandbücher. Sie brauchen keine sorgsam aufbereitete Code-Struktur die mit den zwar informellen aber verbreiteten Normen zur Veröffentlichung von Quellen konform ist. Wie auch immer der Quellcode strukturiert ist, es ist ihnen recht; sie sind ja bereits daran gewöhnt, und solange der Code läuft, wissen sie wie man ihn benutzt. Es macht ihnen nicht einmal etwas aus, wenn grundlegende Annahmen über die Architektur des Projekts undokumentiert bleiben; auch mit ihnen sind sie ja bereits vertraut.

Neulinge andererseits brauchen all diese Dinge. Glücklicherweise jedoch nicht alle auf einmal. Sie müssen nicht jede erdenklich Ressource gleich parat haben, bevor Sie mit einem Projekt an die Öffentlichkeit gehen. In einer perfekten Welt wäre ein neues Open-Source-Projekt von Beginn an ausgestattet mit einem ausführlichen Entwurfsdokument, einem vollständigen Benutzerhandbuch (inklusive der Hinweise auf Funktionen, die zwar geplant, aber noch nicht implementiert sind), wunderschönem und portabel gegliedertem Code, der auf jeder Plattform läuft, und so weiter. In Wirklichkeit wäre es unvertretbar zeitaufwendig, auf all diese Dinge zu achten; und überhaupt sind dies Arbeiten, bei denen man davon ausgehen kann, dass sich Freiwillige beteiligen sobald das Projekt läuft.

Wirklich notwendig ist jedoch, so viel in die Präsentation zu investieren, dass Neulinge die ersten Hürden des Unbekannten überwinden können. Stellen Sie sich das wie den ersten Schritt beim Hochfahren vor, um dem Projekt zu einer Art minimaler Aktivierungsenergie zu verhelfen. Dieser Grenzbereich wird mitunter *Hacktivierungs-Energie* genannt: die Energie, die Neulinge investieren müssen, bevor sie etwas zurückbekommen. Je geringer die Hacktivierungs-Energie ist, desto besser. Ihre erste Aufgabe ist es, die Hacktivierungs-Energie auf ein Niveau zu senken, das Leute dazu ermutigt sich zu beteiligen.

Jeder der folgenden Unterabschnitte, beschreibt einen wichtigen Aspekt beim Start eines neuen Projekts. Sie werden ungefähr in der Reihenfolge präsentiert, in der neue Besucher sie wahrnehmen werden, natürlich kann die tatsächliche Reihenfolge auch abweichen. Betrachten Sie sie als Checkliste. Wenn Sie ein Projekt starten, gehen Sie die Liste durch und stellen Sie sicher, dass alle Punkte erledigt sind oder zumindest dass Sie mit den möglichen Folgen zurechtkommen, wenn Sie einen Punkt auslassen.

Wählen Sie einen guten Namen

Versetzen Sie sich in die Lage von jemandem, der gerade von Ihrem Projekt erfahren hat, vielleicht ganz zufällig, bei der Suche nach einer Software, um ein Problem zu lösen. Das erste, womit er in Berührung kommen wird, ist der Name Ihres Projekts.

Ein guter Name wird Ihr Projekt nicht automatisch erfolgreich machen, und ein schlechter Name wird nicht seinen Untergang besiegeln – nun ja, ein *wirklich* schlechter Name könnte das vielleicht tatsächlich, aber wir gehen davon aus, dass niemand versuchen wird, sein Projekt aktiv zu sabotieren. Allerdings kann ein schlechter Name die Aufmerksamkeit für das Projekts schmälern, entweder weil die Leute ihn nicht ernst nehmen oder schlicht deshalb, weil Sie ihn sich nicht merken können.

Ein guter Name:

- Gibt eine ungefähre Vorstellung davon, was das Projekt tut oder steht zumindest in einer so offensichtlichen Beziehung dazu, dass man weiß was das Projekt tut, wenn man den Namen kennt, und es deshalb später leicht hat, sich an den Namen zu erinnern.
- Ist einfach zu behalten. Man kommt hier nicht um die Tatsache herum, dass Englisch zur Standardsprache im Internet geworden ist: "einfach zu behalten" bedeutet in diesem Fall "Einfach zu behalten für jemanden, der Englisch lesen kann". Wortspiele, die auf der einheimischen Aussprache beruhen, werden vielen Menschen, deren Muttersprache nicht Englisch ist, unverständlich bleiben. Ist das Wortspiel besonders verlockend und einprägsam, kann es das wert sein; denken Sie aber daran, dass viele, die den Namen sehen, nicht dasselbe heraushören werden wie ein englischer Muttersprachler.
- Gleicht nicht dem eines anderen Projekts und verletzt auch kein Markenrecht. Das ist einerseits höflich und andererseits auch rechtlich sinnvoll, denn Sie wollen keine Verwirrung über Identitäten anstiften. Es ist schwierig genug, im Blick zu behalten, was das Netz zu bieten hat, auch ohne unterschiedliche Dinge mit demselben Namen.

  Die zuvor in „Schauen Sie sich zuerst um“ erwähnten Quellen können Ihnen dabei helfen herauszufinden, ob ein anderes Projekt bereits den Namen trägt, den Sie im Sinn haben. Die kostenlose Suche nach Markenzeichen ist über http://www.nameprotect.org/ und http://www.uspto.gov/ verfügbar.
- Ist idealerweise als Domain-Name verfügbar in den Top-Level-Domeins `.com`, `.net`, und `.org`. Sie sollten eine von ihnen auswählen, vielleicht `.org`, um sie als offizielle Seite des Projekts zu bewerben; die anderen beiden sollten darauf verweisen und einfach nur dazu dienen, andere zu hindern, Verwirrung bezüglich Ihres Projektnamens zu stiften. Selbst wenn Sie vorhaben, das Projekt auf einer anderen Seite zu betreiben (siehe „Hosting-Pakete“), können Sie immer noch die Projekt-spezifische URL registrieren und diese auf die Seiten des Betreibers weiterleiten. Es hilft dem Nutzer ungemein, nur eine einfache URL im Kopf behalten zu müssen.

Formulieren Sie ein klares Missionsziel

Sobald Besucher ihre Projektseite gefunden haben, werden sie nach einer kurzen Beschreibung, dem Ziel des Projekts suchen, um (innerhalb von 30 Sekunden) entscheiden zu können, ob sie interessiert daran sind, mehr zu erfahren. Das Ziel sollte auf der Frontseite einen auffälligen Platz einnehmen, vorzugsweise gleich unter dem Projektnamen.

Die Formulierung des Missionsziels sollte anschaulich, klar umrissen und vor allem kurz sein. Hier ist ein gutes Beispiel von http://www.openoffice.org/:

> *Gemeinschaftlich die führende internationale Office-Lösung erschaffen, die auf allen wichtigen Plattformen läuft und den Zugriff auf alle Funktionen und Daten durch offene Schnittstellen und ein XML-basiertes Dateiformat erlaubt.*

In nur wenigen Worten haben sie alle wichtigen Punkte erfasst, hauptsächlich indem sie sich auf das Vorwissen der Leser stützen. Mit "*gemeinschaftlich*", signalisieren sie, dass keine einzelne Firma die Entwicklung dominieren wird; "*international*" bedeutet, dass die Software es Menschen erlauben wird, in mehreren Sprachen zu arbeiten; "*alle wichtigen Plattformen*" heißt für Unix, Macintosh und Windows. Das Übrige signalisiert, dass offene Schnittstellen und leicht verständliche Dateiformate ein wichtiger Teil ihres Ziels sind. Sie sagen nicht offen, dass sie eine freie Alternative zu Microsoft Office sein

wollen, aber die meisten Menschen können zwischen den Zeilen lesen. Auch wenn dieser Satz auf den ersten Blick weitgreifend erscheint, ist es tatsächlich recht begrenzt: Der Begriff "*Office-Lösung*" bedeutet etwas ganz bestimmtes für diejenigen, die mit solcher Software vertraut sind. Die mutmaßlichen Vorkenntnisse der Leser (in diesem Fall wahrscheinlich mit MS Office) werden ausgenutzt, um das Missionsziel kompakt zu halten.

Die Gestaltung des Missionsziels hängt teilweise davon ab, wer es schreibt und nicht von der Software die es beschreibt. So ist es für Open Office beispielsweise sinnstiftend, das Wort "*gemeinschaftlich*" zu verwenden, denn das Projekt wurde gestartet und noch immer großteils gesponsort von Sun Microsystems. Mit dieser Wortwahl zeigt man sich sensibel gegenüber Befürchtungen, die Entwicklung könne einmal seitens Sun dominiert werden. In einer solchen Angelegenheit kann schon allein der Hinweis auf die *Möglichkeit* eines Problems einen wesentlichen Schritt für seine Ausräumung darstellen. Andererseits können Projekte, die nicht durch eine einzige Firma unterstützt werden, auf solche Formulierungen verzichten; schließlich ist die Entwicklung durch eine Gemeinschaft das Übliche, es gibt also normalerweise keinen Grund, dies im Missionsziel aufzuführen.

Sagen Sie, dass das Projekt frei ist

Wer die Missionziele gelesen hat und noch interessiert ist, wird nun weitere Einzelheiten erfahren wollen, vielleicht durch die Benutzer- oder Entwicklerdokumentation, schließlich wird er etwas herunterladen wollen. Doch vor alledem will er sicher sein, dass es sich um Open Source handelt.

Die Hauptseite muss unmissverständlich klar machen, dass das Projekt Open Source ist. Das mag offensichtlich klingen, Sie wären aber überrascht darüber, wie viele Projekte es vergessen. Ich habe schon Projekte gesehen, deren Hauptseite nicht nur versäumte zu sagen, unter welcher Lizenz ihre Software veröffentlicht wurde, sondern nicht einmal erwähnte, dass es sich um freie Software handelt. Manchmal erschien die entscheidende Information erst auf der Download-Seite, der Entwickler-Seite oder irgend einer andere Stelle die einen Klick mehr erforderte. Im Extremfall wurde die Lizenz überhaupt nicht auf der Site angegeben – die einzige Möglichkeit sie herauszufinden war, die Software herunterzuladen und hineinzuschauen.

Vermeiden Sie diesen Fehler. Durch solch ein Versäumnis können Ihnen viele potentieller Entwickler und Nutzer verloren gehen. Sagen Sie gleich vorweg, direkt unterhalb des Missionsziels, dass das Projekt freie Software oder Open-Source-Software ist, und geben Sie die genaue Lizenz an. Eine kurze Anleitung zur Wahl einer Lizenz bietet der Abschnitt „Die Wahl einer Lizenz“später in diesem Kapitel; Lizenzfragen werden ausführlich im Kapitel Kapitel 9, *Lizenzen, Urheberrecht und Patente* behandelt.

Bis hierhin hat sich unser hypothetischer Besucher entschieden – wahrscheinlich innerhalb der ersten Minute oder schon vorher – ob er interessiert ist, sagen wir, zumindest weitere fünf Minuten in das Projekt zu investieren. Der nächste Abschnitt beschreibt, was er innerhalb dieser fünf Minuten vorfinden sollte.

Funktionen und Anforderungen

Es sollte eine kurze Liste der Funktionen geben, die von der Software unterstützt werden (Wenn etwas noch nicht fertig ist, können Sie es trotzdem auflisten, schreiben Sie aber "*geplant*" oder "*in Arbeit*" daneben), und der Anforderungen, die die Software an die Hardware stellt. Stellen Sie sich die Liste der Funktionen/Anforderungen wie etwas vor, das Sie jemandem geben würden, der eine Kurzzusammenfassung zu dieser Software wünscht. Oft ist sie einfach nur eine logische Erweiterung der Missionsziele. Das Missionsziel könnte zum Beispiel folgendes beinhalten:

> *Erstellung einer Volltext-Indexierungs- und -Suchmaschine mit einer umfangreichen Schnittstelle für Programmierer, die Suchdienste über große Mengen von Text anbieten wollen.*

Die Liste der Funktionen und Anforderungen würde Details bieten, um das Missionsziel zu verdeutlichen:

Funktionen:

- *Durchsucht Klartext, HTML, und XML*
- *Suche nach Wörtern oder Ausdrücken*
- *(geplant) Unscharfe Suche*
- *(geplant) Inkrementelle Aktualisierung der Indexe*
- *(geplant) Indexierung von Ressourcen im Netzwerk*

Anforderungen:

- *Python 2.2 oder höher*
- *Genug Festplattenspeicher für die Indexe (ungefähr die doppelte Menge der Originaldaten)*

Durch diese Informationen gewinnen Leser schnell ein Gefühl dafür, ob ihnen diese Software etwas nützen könnte, und sie können gleichzeitig überlegen, ob sie sich als Entwickler beteiligen wollen.

Stand der Entwicklung

Leute wollen immer wissen wie es einem Projekt geht. Bei neuen Projekten wollen sie wissen wie weit seine Versprechen und sein derzeitiger Stand auseinanderliegen. Bei einem ausgereiften Projekt wollen sie wissen, wie aktiv es gepflegt wird, wie oft neue Versionen veröffentlicht werden, wie schnell es wahrscheinlich auf Bug-Meldungen reagieren wird, usw.

Um diese Fragen zu beantworten, sollten Sie eine Seite zum Fortschritt der Entwicklung einrichten, auf der die kurzfristigen Ziele und Anfragen des Projekts aufgelistet werden (es könnte z.B. auf der Suche nach Entwicklern mit bestimmten Fachkenntnissen sein). Die Seite kann auch eine Übersicht vergangener Versionen haben, mit einer Auflistung der jeweiligen Funktionen, damit Besucher sich ein Bild machen können, was in diesem Projekt unter "Fortschritt" verstanden wird und wie schnell es nach diesem Verständnis vorankommt.

Fürchten Sie sich nicht davor, einen unfertigen Eindruck zu vermitteln und widerstehen Sie der Versuchung, den Entwicklungsstand besser darzustellen als er wirklich ist. Jeder weiß, dass sich Software in Schritten entwickelt; es ist keine Schande zu sagen "Es ist Alpha-Software und sie hat bekannte Fehler. Sie läuft zwar und funktioniert zumindest teilweise, trotzdem gilt: Nutzung auf eigene Gefahr". Diese Ausdrucksweise wird nicht die Art von Entwicklern verschrecken, die Sie zu dieser Zeit brauchen. Was die Nutzer angeht: Einer der schlimmsten Fehler, die ein Projekt machen kann, ist Nutzer anzulocken, für die die Software noch nicht bereit ist. Der Ruf, instabil oder fehlerträchtig zu sein, ist schwer wieder loszuwerden, wenn er dem Projekt einmal anhaftet. Auf lange Sicht zahlt es sich aus, konservativ zu sein; es ist besser wenn die Software *stabiler* läuft als erwartet, und angenehme Überraschungen sorgen für die beste Mundpropaganda.

Alpha und Beta

Der Begriff *alpha* bedeutet für gewöhnlich eine erste Version, mit der Benutzer echte Arbeit erledigen können, die alle geplanten Funktionen hat, aber auch bekannte Fehler. Der vorrangige Sinn von Alpha-Software ist, Rückmeldungen zu erhalten, damit die Entwickler wissen, woran sie arbeiten sollen. Die Stufe *beta* bedeutet, dass in der Software alle groben Fehler behoben wurden, aber noch nicht genug getestet wurde, um als zur Herausgabe geeignet zu gelten. Der Sinn von Beta-Software ist entweder zur offiziellen Version zu werden, wenn keine Fehler gefunden wurden, oder detaillierte Rückmeldungen an die Entwickler zu geben, um die zügige Fertigstellung der Version zu unterstützen. Der Unterschied zwischen alpha und beta ist zum größten Teil eine Sache der Einschätzung.

Downloads

Man sollte die Software als Quelltext in den üblichen Formaten herunterladen können. Wenn ein Projekt noch am Anfang ist, sind binäre (ausführbare) Dateien nicht nötig, es sei denn der Build-Vorgang ist derart kompliziert und voller Abhängigkeiten, dass es für die meisten Leute einen Menge Arbeit wäre, die Software überhaupt zum Laufen zu bringen. (Wenn das aber der Fall ist, wird das Projekt sowieso Schwierigkeiten haben, Entwickler anzuziehen.

Die veröffentlichten Dateien herunterzuladen, sollte so bequem, standardkonform und mühelos sein wie möglich. Um eine Krankheit auszurotten, würden Sie die Medizin nicht so verteilen, dass man zur Anwendung eine unübliche Spritzengröße bräuchte. Ebenso sollte Software die üblichen Build- und Installationsmethoden beachten, denn je mehr sie von diesen Standards abweicht, desto mehr potenzielle Benutzer und Entwickler werden aufgeben und sich verwirrt abwenden.

Das hört sich offensichtlich an, aber viele Projekte machen sich diese Mühe sehr lange Zeit nicht, in der Annahme, dass sie es jederzeit tun könnten: *"Wir erledigen den Kram sobald der Code näher an der Fertigstellung ist."* Hierbei übersehen sie jedoch, dass das Hinausschieben der langweiligen Arbeiten an Build- und Installations-Vorgängen auch die Fertigstellung vom Code hinausschiebt – denn sie entmutigen Entwickler, die ansonsten etwas zum Code beigetragen hätten. Das Heimtückische daran ist, dass nicht einmal jemand davon *erfährt*, dass Entwickler verlorengegangenen sind, denn der Vorgang ist eine Ansammlung von Nicht-Ereignissen: Jemand geht auf die Webseite, lädt die Software herunter, versucht einen Build zu machen, scheitert, gibt auf und geht seiner Wege. Wer außer der Person selbst wird jemals davon erfahren? Keiner im Projekt wird je bemerken, wie Interesse und Wohlwollen von jemanden lautlos verschwendet wurde.

Langweilige Arbeit mit einem hohen Nutzen sollte immer frühzeitig erledigt werden, und das Herabsetzen der Einstiegshürden für ein Projekt durch sauberes Packen zahlt sich mit vielfachem Gewinn aus.

Wenn Sie ein Paket zum Herunterladen freigeben ist es wichtig, eine eindeutige Versionsnummer zu vergeben, damit die Ausgaben unterschieden werden können und zu sehen welche die aktuellere ist. Eine ausführliche Diskussion über Versionsnummern finden Sie in „Versionszählung“, und Details zur Standardisierung von Build- und Installations-Vorgängen werden im Abschnitt „Erstellung der Pakete“, sowie in Kapitel 7, *Paket-Erstellung, Veröffentlichung, und tägliche Entwicklung* behandelt.

Zugriff auf Versionsverwaltung und Bugtracker

Den Quellcode herunterzuladen mag für diejenigen ausreichend sein, die lediglich die Software installieren und benutzen wollen. Das genügt jedoch nicht für diejenigen, die sie weiterentwickeln wollen. Nächtliche Quelltext-Schnappschüsse können helfen, sind aber nicht ausreichend fein für eine lebendige Entwicklergemeinschaft. Diese Leute brauchen Echtzeit-Zugriff auf den neuesten Quellcode; ihn bereitzustellen wird erst durch die Benutzung einer Versionsverwaltung möglich. Anonymer Zugriff auf den

Quellcode, der unter Versionsverwaltung steht, ist ein Zeichen – für Entwickler wie auch für Nutzer – dass das Projekt sich Mühe gibt, den Leuten das für eine Beteiligung Nötige zu geben. Wenn Sie nicht sofort eine Versionsverwaltung bereitstellen können, sollten Sie zumindest darauf hinweisen, dass Sie dies demnächst vorhaben. Das Thema Infrastruktur der Versionsverwaltung wird ausführlich in „Versionsverwaltung“ im Kapitel Kapitel 3, *Technische Infrastruktur* behandelt.

Gleiches gilt für den Bugtracker. Die Bedeutung des Bugtracker liegt nicht allein in seinem Nutzen für die Entwickler, sondern er ist auch ein Signal an Außenstehende. Für viele ist eine öffentliche Bug-Datenbank eines der stärksten Anzeichen dafür, dass ein Projekt ernstgenommen werden sollte. Desweiteren ist ein Projekt um so besser, je mehr Fehler darin protokolliert sind. Auch wenn es sich widersprüchlich anhört, sollte man bedenken, dass die Anzahl der erfassten Fehler von drei Dingen abhängt: Die absolute Anzahl in der Software enthaltene Fehler, die Anzahl seiner Benutzer und wie bequem es für diese Benutzer ist, neue Fehler einzutragen. Von diesen dreien sind die letzten beiden wesentlich. Jede ausreichend große und komplexe Software enthält eine im Grunde genommen beliebige Menge an Fehlern, die nur darauf warten, entdeckt zu werden. Die eigentliche Frage ist, wie gut das Projekt diese Fehler erfassen und priorisieren kann. Ein Projekt mit einer großen und gut gepflegten Fehler-Datenbank (ein Zeichen dafür, dass schnell auf Bugs reagiert wird, Duplikate markiert werden, usw.) macht deshalb einen besseren Eindruck als ein Projekt ohne oder mit einer fast leeren Fehler-Datenbank.

Am Anfang des Projekts wird die Fehler-Datenbank natürlich nur sehr wenige Meldungen enthalten und es gibt nicht viel, das Sie dagegen tun könnten. Wenn die Statusseite aber das junge Alter hervorhebt und wenn Leute, die die Bug-Datenbank betrachten, sehen können, dass die meisten Einträge vor kurzem gemacht wurden, können sie leicht schlussfolgern, dass das Projekt immer noch eine gesunde *Rate* an Einträgen hat und werden dementsprechend über die niedrige absolute Anzahl an Bug-Meldungen nicht wirklich beunruhigt sein.

Man sollte auch beachten, dass Bugtracker oft nicht nur zur Verfolgung von Bugs, sondern auch für Verbesserungen an der Software, Änderungen an der Dokumentation, ausstehende Aufgaben und mehr benutzt werden. Weiteres zum Betrieb eines Bugtrackers, wird in „Bugtracker“ im Kapitel Kapitel 3, *Technische Infrastruktur* behandelt, also werde ich hier nicht näher darauf eingehen. Das Wichtige aus Sicht der Präsentation ist, überhaupt einen Bugtracker zu *haben* und sicherzustellen, dass dieser Umstand bereits auf der Hauptseite deutlich wird.

Kommunikationskanäle

Besucher wollen oft wissen, wie sie am Projekt beteiligte Menschen erreichen können. Veröffentlichen Sie deshalb Adressen von Mailinglisten, Chat-Räumen, IRC-Kanälen und anderen Foren, auf denen Beteiligte erreicht werden können. Stellen Sie klar, dass Sie und die anderen Autoren des Projekts auf diesen Listen eingetragen sind, um den Besuchern zu zeigen , dass sie Rückmeldungen an die Entwickler richten können. Eine Anmeldung auf den Mailinglisten beinhaltet für Sie nicht die Verpflichtung, alle Fragen zu beantworten oder alle Wünsche nach neuen Funktionen zu verwirklichen. Auf lange Sicht betrachtet, nutzt die Mehrheit diese Foren sowieso nie, aber es wird sie ermutigen zu wissen, dass sie es *könnten*, sollte es einmal nötig sein.

Am Anfang eines Projekts hat es keinen Sinn, die Foren für Benutzer und Entwickler getrennt zu halten. Es ist viel besser, wenn alle Projektbeteiligten miteinander reden: in einem "Raum". Unter den ersten Interessenten eines Projekts ist die Unterscheidung zwischen Entwickler und Nutzer oft verwaschen. Sofern sie überhaupt gemacht werden kann, gibt es in den frühen Tagen wesentlich mehr Entwickler im Verhältnis zu Nutzern als es später der Fall ist. Obwohl Sie nicht annehmen können, dass jeder, der sich früh für das Projekt interessiert, ein Programmierer ist, der am Quelltext der Software arbeiten will, können Sie annehmen, dass sie zumindest daran interessiert sind, die Diskussionen um die Entwicklung mitzuverfolgen und ein Gefühl für die Richtung des Projekts zu entwickeln.

Da es in diesem Kapitel nur darum geht wie man ein Projekt startet, belassen wir es dabei zu sagen, dass diese Foren existieren sollten. Später, in „Handhabung von Wachstum“ im Kapitel Kapitel 6, *Kommu-*

nikation, werden wir untersuchen, wo und wie man diese Foren aufbaut, inwiefern Sie möglicherweise Moderation erfordern und wie man Foren für Nutzer und Foren für Entwickler voneinander löst wenn es nötig wird, ohne eine unüberwindliche Kluft aufzureißen.

Richtlinien für Entwickler

Wenn jemand sich überlegt, etwas zu einem Projekt beizutragen, wird er sich nach Richtlinien für Entwickler umschauen. Diese sind weniger technischer, als viel mehr sozialer Natur: Sie erklären, wie Entwickler miteinander und mit Benutzern umgehen, also letzlich wie die Dinge laufen sollten.

Dieses Thema wird ausführlich in „Schriftliche Regeln“ im Kapitel Kapitel 4, *Soziale und politische Infrastruktur* behandelt, aber die wesentlichen Elemente der Richtlinien sind:

- Hinweise auf Foren für die Zusammenarbeit mit anderen Entwicklern
- Anleitungen wie man Fehler meldet und Patches einreicht
- Einige Hinweise darauf *wie* die Entwicklung für gewöhnlich abläuft – ob das Projekt eine gütige Diktatur, eine Demokratie oder etwas anderes ist

Übrigens soll "Diktatur" in keiner Weise herabsetzend wirken. Es ist völlig in Ordnung eine Tyrannei zu betreiben, bei dem ein bestimmter Entwickler das letzte Wort über alle Änderungen haben kann. Viele erfolgreiche Projekte arbeiten in dieser Weise. Das Wichtige dabei ist, dass das Projekt dies von vornherein klarstellt. Eine Tyrannei, die vorgibt eine Demokratie zu sein, wird sich Menschen abspenstig machen; eine Tyrannei die klar sagt was sie ist, wird gut zurecht kommen, sofern der Tyrann kompetent und vertrauenswürdig ist.

Siehe http://subversion.apache.org/docs/community-guide/ für ein Beispiel besonders gründlicher Richtlinien für Entwickler oder http://www.openoffice.org/dev_docs/guidelines.html für allgemeinere Richtlinien, die sich mehr auf die Steuerung und Teilnahme am Projekt als auf technische Angelegenheiten konzentrieren.

Das etwas andere Thema, die Bereitstellung einer Projekt-Einführung für Programmierer, wird im Abschnitt „Entwickler-Dokumentation“ später in diesem Kapitel behandelt.

Dokumentation

Dokumentation ist unerlässlich. Es muss *irgendetwas* zum Lesen geben, selbst wenn es nur rudimentär und unvollständig ist. Die Dokumentation ist ganz klar ein Teil der vorhin erwähnten "Plackerei" und sie ist oft das erste, das in einem neuen Open-Source-Projekt zu kurz kommt. Die Missionsziele und eine Liste von Funktionen zu schreiben, die Wahl einer Lizenz, den Stand der Entwicklung zusammenzufassen – das sind alles relativ kleine Aufgaben, die mit einem Schlag erledigt werden können; und wenn sie erledigt sind, muss man sich normalerweise nicht weiter damit beschäftigen. Die Dokumentation hingegen ist nie wirklich fertig, was vielleicht ein Grund dafür ist, dass man es manchmal hinauszögert, sie überhaupt in Angriff zu nehmen.

Das heimtückischste daran ist, dass die Autoren der Dokumentation keinen direkten Nutzen aus ihr ziehen, während sie für neue Nutzer unerlässlich ist. Die wichtigste Dokumentation für neue Benutzer sind die Grundlagen: Wie richte ich die Software zügig ein, eine Übersicht über ihre Funktionsweise, vielleicht auch Anleitungen für häufige Aufgaben. All dies ist den *Autoren* nur allzu gut bekannt – so bekannt, dass es für sie schwierig sein kann, sich in die Lage der Leser zu versetzen, und mühsam die offensichtlichen Einzelschritte zu buchstabieren, die aus ihrem Blickwinkel kaum der Erwähnung wert scheinen.

Es gibt keine magische Lösung für dieses Problem. Es muss sich nur jemand die Zeit nehmen, alles aufzuschreiben und die Brauchbarkeit der Dokumentation dann an neuen Nutzern zu testen. Benutzen Sie ein einfaches, leicht zu bearbeitendes Format wie HTML, Klartext oder eine XML-Variante – etwas geeignetes für kleine und spontane Verbesserungen. Das reduziert nicht nur den Aufwand für die ersten Autoren, die Dokumentation schrittweise zu verbessern, sondern auch allen, die später zum Projekt hinzukommen.

Eine Möglichkeit, eine erste grundlegende Dokumentation abzusichern ist es, ihren Umfang von vornherein einzuschränken. So erscheint die Aufgabe zumindest nicht bodenlos. Als Richtlinie könnte gelten, dass folgende minimale Bedingungen erfüllt werden:

- Sagen Sie dem Leser klar, welche technischen Kenntnisse erwartet werden.

- Beschreiben sie klar und deutlich, wie man die Software einrichtet, und nennen Sie dem Benutzer irgendwo am Anfang der Dokumentation ein Merkmal oder einen Befehl, mit dem man prüfen kann, ob sie richtig eingerichtet wurde. Die erste Dokumentation ist in mancherlei Hinsicht wichtiger als eine echte Bedienungsanleitung. Je mehr Mühe jemand in die Installation und Einrichtung der Software investiert hat, desto beharrlicher wird er darin sein, fortgeschrittene, unzureichend dokumentierte Funktionen zu erfassen. Wenn Leute aufgeben, passiert es meistens gleich am Anfang; deshalb sind es die frühesten Phasen wie die Installation, bei der man die meiste Unterstützung braucht.

- Geben Sie Tutorial-artige Beispiele für typische Aufgaben. Natürlich sind viele Beispiele für viele Aufgaben noch besser, aber wenn die Zeit knapp ist, wählen Sie einen Punkt aus und schreiben Sie dazu eine ausführliche Anleitung. Sobald jemand sieht, dass die Software für eine Sache benutzt werden *kann*, wird er beginnen alleine herauszufinden, wofür sie noch zu gebrauchen ist – und wenn Sie Glück haben, dazu übergehen, die Dokumentation selbst zu erweitern. Was uns zum nächsten Punkt bringt...

- Kennzeichnen Sie unvollständige Bereiche der Dokumentation als solche. Indem Sie dem Leser zeigen, dass Sie sich über die Defizite im Klaren sind, stellen Sie sich auf seine Sicht ein. Durch Einfühlungsvermögen geben Sie ihm zu verstehen, dass das Projekt nicht erst überzeugt werden muss, was wichtig ist. Solche Kennzeichen entsprechen keiner Verpflichtung, die Lücken bis zu einem bestimmten Datum auszufüllen – sie können auch als offenen Anfragen um Hilfe von Freiwilligen betrachtet werden.

Der letzte Punkt hat tatsächlich umfassendere Bedeutung und kann auf das ganze Projekt angewendet werden, nicht nur auf die Dokumentation. Eine genaue Buchführung über bekannte Defizite ist in der Open-Source-Welt die Norm. Sie müssen die Mängel des Projekts nicht hochspielen, sondern einfach gewissenhaft und leidenschaftslos aufzählen, wo die Veranlassung gegeben ist (das kann in der Dokumentation, im Bugtracker oder in einer Diskussion auf einer Mailingliste geschehen). Keiner wird das als vom Projekt ausgehende Miesmacherei ansehen, oder als Verpflichtung die Probleme bis zu einem bestimmten Datum zu lösen, es sei denn, das Projekt geht ausdrücklich eine solche Verpflichtung ein. Da jeder Nutzer diese Mängel selbst finden wird, ist es besser, sie psychologisch darauf vorzubereiten – das gibt den Eindruck, dass im Projekt ein Bewusstsein über seinen Zustand besteht.

Die Pflege einer FAQ-Liste

Eine *FAQ* ("Frequently Asked Questions"[1]) kann eine der besten Investitionen des Projekts hinsichtlich ihres Informationsgehalts sein. Eine FAQ ist auf Fragen abgestimmt, die von Benutzern und Entwicklern wirklich gestellt werden – nicht auf Fragen die Sie vielleicht *erwarten* würden – und deshalb neigt eine gut gepflegte FAQ dazu, denjenigen, die sie zu Rate ziehen, genau das zu geben wonach sie suchen. Die FAQ ist oft die erste Stelle, die Benutzer durchsuchen, wenn sie auf ein Problem stoßen, sie ziehen es oft sogar dem offiziellen Handbuch vor und es ist wahrscheinlich das Dokument in ihrem Projekt, worauf andere Seiten am ehesten verweisen.

Leider können Sie die FAQ nicht am Anfang eines Projekts schreiben. Eine gute FAQ schreibt man nicht, man lässt sie wachsen. Sie sind schon per Definition auf Rückmeldungen angewiesen. FAQs entwickeln sich erst mit der Zeit durch die tägliche Nutzung der Software. Da es unmöglich ist, die Fragen der Benutzer zu erahnen, ist es unmöglich sich hinzusetzen und von Grund auf eine nützliche FAQ zu schreiben.

Verschwenden Sie also nicht Ihre Zeit damit, es zu versuchen. Sie können jedoch eine größtenteils leere FAQ einrichten, als Vorlage und offensichtlichen Ort, an dem Leute Fragen und Antworten eintragen können, wenn das Projekt erst einmal läuft. Zunächst ist wichtigste Eigenschaft einer FAQ aber nicht ihre Vollständigkeit, sondern ihre Einfachheit: Wenn es einfach ist, neue Einträge zu anzufügen, wird dies auch getan werden. (Die vernünftige Pflege einer FAQ ist eine nicht ganz triviale, aber faszinierende Angelegenheit, die in „FAQ-Verwalter“ im Kapitel Kapitel 8, *Leitung von Freiwilligen* weiter behandelt wird.)

Erreichbarkeit der Dokumentation

Die Dokumentation sollte an zwei Stellen erreichbar sein: Online (direkt auf der Website) *und* in der zum Download verfügbaren Ausgabe der Software (siehe „Erstellung der Pakete“ im Kapitel Kapitel 7, *Paket-Erstellung, Veröffentlichung, und tägliche Entwicklung*). Sie sollte online in durchsuchbarer Form vorliegen, weil Interessierte die Dokumentation oft lesen, *bevor* sie die Software zum ersten Mal herunterladen, um besser entscheiden zu können, ob sie dies überhaupt tun sollen. Prinzipiell sollte die Dokumentation aber auch der Software beiliegen, denn ein veröffentlichtes Paket sollte alles enthalten (d.h. lokal verfügbar machen), was man braucht um die Software zu benutzen.

Im Falle der Online-Dokumentation sollten Sie für einen Link sorgen, der auf eine *vollständige* Dokumentation auf einer einzigen HTML-Seite verweist (schreiben Sie einen Hinweis wie "monolitisch" oder "umfangreiche Einzelseite" daneben, damit die Leser nicht überrascht sind wenn es beim Laden etwas Zeit braucht). Das ist nützlich, da Leute oft die ganze Dokumentation nach einem bestimmten Wort oder einer Wendung absuchen wollen. Im Allgemeinen wissen sie schon, wonach sie suchen, können sich nur nicht erinnern an welcher Stelle es stand. In dieser Situation gibt es nichts frustrierenderes, als je eine HTML-Seite für Inhaltsangabe, eine für die Einleitung, eine weitere für die Installationsanleitung, usw zu haben. Wenn die Seiten so aufgeteilt sind, machen sie die Suchfunktion des Browsers wertlos. Eine in mehrere Seiten aufgebrochene Dokumentation ist gut, wenn man weiß, welchen Abschnitt man braucht oder die ganze Dokumentation von vorne nach hinten durchlesen möchte. Das ist aber *nicht* die häufigste Art auf die Dokumentation zuzugreifen. Viel häufiger kennt sich jemand im Grunde genommen mit der Software aus und kehrt zurück, um nach einem bestimmten Wort oder Ausdruck zu suchen. Keine monolitische Datei bereitzustellen, würde in solchen Fällen das Leben unnötig erschweren.

Entwickler-Dokumentation

Die Entwickler-Dokumentation wird geschrieben, damit Programmierer den Code verstehen, um ihn reparieren und erweitern zu können. Sie unterscheidet sich ein wenig zu den vorhin erwähnten *Richt-*

[1] Häufig gestellte Fragen

linien für Entwickler, die eher sozialer als technischer Natur sind. Entwickler-Richtlinien sagen den Programmierern, wie sie miteinander zurecht kommen; die Entwickler-Dokumentation hingegen sagt ihnen, wie sie mit dem Code zurechtkommen. Beide werden oft der Bequemlichkeit halber gemeinsam in einem Dokument angeboten (wie im oben erwähnten Beispiel http://subversion.apache.org/docs/community-guide/), müssen dies jedoch nicht.

Obwohl die Entwickler-Dokumentation sehr hilfreich sein kann, gibt es keinen Grund, um ihretwillen eine Freigabe zu verzögern. So lange die ursprünglichen Autoren verfügbar (und bereit) sind, Fragen zum Code zu beantworten, genügt das für den Anfang. Tatsächlich ist eine häufige Motivation zum Schreiben der Dokumentation, dass man wieder und wieder die immer gleichen Fragen beantworten muss. Aber selbst bevor sie geschrieben ist, wird es entschlossenen Freiwilligen gelingen, sich einen Weg durch den Code zu bahnen. Was Menschen dazu treibt, ihre Zeit mit dem Erarbeiten einer Codebasis zu verbringen, ist, dass der Code aus ihrer Sicht etwas Nützliches tut. Solange sie sich dessen gewiss sind, nehmen sich die Zeit, Probleme zu lösen; ohne diese Zuversicht wird sie keine auch noch so gute Dokumentation anlocken oder halten können.

Wenn Sie also nur die Zeit zum Schreiben einer Dokumentation haben, so schreiben Sie eine für Benutzer. Jede Dokumentation für Benutzer ist auch für die Entwickler effektiv; jeder Programmierer, der an einer Software arbeitet, muss auch damit vertraut sein, wie man sie benutzt. Wenn Sie später sehen, wie Programmierer andauernd die gleichen Fragen stellen, nehmen Sie sich die Zeit, eine paar separate Dokumente eigens für sie zu schreiben.

Manche Projekte nutzen Wikis für die allererste Dokumentation, oder sogar für die Hauptdokumentation. Nach meiner Erfahrung funktioniert das nur dann, wenn das Wiki aktiv von einer Handvoll Leuten bearbeitet wird, die hinsichtlich der Organisation und des Tonfalls der Dokumentation einig sind. Mehr dazu steht in „Wikis“ im Kapitel Kapitel 3, *Technische Infrastruktur*.

Beispiel-Ausgaben und Screenshots

Ein Projekt mit graphischer Benutzeroberfläche oder graphischen oder anderen markanten Ausgaben sollte Beispiele auf der Website des Projekts anbieten. Im Fall einer Benutzeroberfläche wären es Screenshots; für Ausgaben können es Screenshots oder vielleicht nur Dateien sein. Beide befriedigen das Bedürfnis des Menschen nach direkter Belohnung: Ein einziges Bild kann überzeugender sein, als ganze Abstätze von Beschreibungen oder Geplapper auf Mailinglisten, denn ein Bild ist ein unverkennbarer Beweis dafür, dass die Software *funktioniert*. Sie mag ihre Fehler haben, schwer zu installieren und unvollständig dokumentiert sein, aber ein Bild ist immerhin ein Beweis, dass man sie zum Laufen bringen kann, wenn man sich nur genug Mühe gibt.

Screenshots

Screenshots können einschüchtern, wenn man noch nie welche gemacht hat, deshalb hier ein kleine Anleitung. Mit "Gimp" (http://www.gimp.org/), öffnen Sie Datei->Erstellen-> Bildschirmfoto..., und wählen Sie Ein einzelnes Fenster aufnehmen oder Den ganzen Bildschirm aufnehmen, klicken Sie dann auf Aufnehmen. Ziehen Sie nun den Kreuz-Cursor auf das gewünschte Fenster (dieser Schritt entfällt bei ganzem Bildschirm) um das Bild in Gimp aufzunehmen. Ändern und schneiden Sie anschließend das Bild bei Bedarf nach den Anweisungen auf http://www.gimp.org/tutorials/Lite_Quickies/#crop.

Sie können der Website des Projekts noch vieles mehr hinzufügen, wenn die Zeit dazu reicht, oder wenn es aus irgendeinem Grund besonders passend erscheint: Eine Seite mit Neuigkeiten, eine Seite mit der Historie des Projekts, eine Seite mit verwandten Links, eine Suchfunktion, ein Link für Spenden, usw. Nichts davon ist am Anfang notwendig, aber man sollte es für später im Hinterkopf behalten.

Hosting-Pakete

Es gibt ein paar Sites, die kostenlos die Infrastruktur für Open-Source-Projekte bereitstellen: einen Web-Bereich, Versionsverwaltung, einen Bugtracker, einen Download-Bereich, Foren, regelmäßiges Backup, usw. Die Details unterscheiden sich zwar von Site zu Site, aber das Wesentliche wird bei allen angeboten. Durch die Nutzung dieser Sites erhalten Sie vieles umsonst, müssen dafür aber die Kontrolle über die Benutzerführung teilweise aufgeben. Der Hosting-Dienst entscheidet darüber, welche Software die Site benutzt, und kann so Look-and-Feel der Projektseiten bestimmen oder zumindest beeinflussen.

Siehe „Hosting-Pakete“ im Kapitel Kapitel 3, *Technische Infrastruktur* für eine detailliertere Diskussion über Vor- und Nachteile von Hosting-Paketen und eine Liste von Sites die sie anbieten.

Die Wahl einer Lizenz

Dieser Abschnitt soll eine schnelle und sehr grobe Anleitung zur Wahl einer Lizenz bieten. Lesen Sie Kapitel 9, *Lizenzen, Urheberrecht und Patente* um die rechtlichen Folgen der verschieden Lizenzen im Detail besser zu verstehen und zu erkennen, inwiefern sich Ihre Lizenz-Entscheidung auf die Möglichkeit auswirkt, Ihre Software mit anderer freier Software zu kombinieren.

Sie können aus einer Menge von Lizenzen für freie Software wählen. Die meisten müssen wir hier nicht beachten, da sie für die speziellen rechtlichen Bedürfnisse einer bestimmten Firma oder Person geschrieben wurden und für Ihr Projekt nicht angemessen wären. Wir beschränken uns auf die am häufigsten verwendeten Lizenzen; in den meisten Fällen werden Sie sich für eine von ihnen entscheiden wollen.

"Alles ist erlaubt"-Lizenzen

Wenn Sie kein Problem damit haben, dass der Code Ihres Projekts in proprietären Anwendungen benutzt wird, können Sie eine *MIT/X*-artige Lizenz nutzen. Es ist die einfachste der verschiedenen Minimal-Lizenzen, die wenig mehr leisten als die Vervielfältigungsrechte abzusichern (ohne dabei wirklich die Vervielfältigung einzuschränken) und zu erklären, dass keiner für den Code haftet. Weiteres darüber finden Sie in „Die MIT- / X-Window-System-Lizenz“.

Die GPL

Wenn Sie nicht wollen, dass Ihr Code in proprietären Anwendungen verwendet wird, benutzen Sie die GNU General Public License[2] (http://www.gnu.org/licenses/gpl.html). Die GPL ist heute die wahrscheinlich bekannteste Lizenz für freie Software der Welt. Das ist an und für sich schon ein großer Vorteil, da viele potentielle Benutzer und Teilnehmer bereits mit ihr vertraut sein werden und deshalb keine zusätzliche Zeit aufbringen müssen, um Ihre Lizenz zu lesen und zu verstehen. Für weitere Details siehe „Die GNU GPL“ in Kapitel 9, *Lizenzen, Urheberrecht und Patente*.

> Wenn die Benutzer auf Ihren Code vorwiegend über ein Netzwerk zugreifen – d.h. wenn die Software typischerweise Teil eines Hostingdienstes ist – dann sollten Sie alternativ die Wahl der *GNU Affero GPL* in Betracht ziehen. Siehe The GNU Affero GPL: A Version of the GNU GPL for Server-Side Code in Kapitel 9, *Lizenzen, Urheberrecht und Patente* für weitere Informationen.

Eine Lizenz für Ihre Software

Wenn Sie sich einmal für eine Lizenz entschieden haben, müssen Sie diese auf Ihre Software anwenden.

[2] Allgemeine Öffentliche Lizenz

Dazu sollten Sie diese zunächst auf Ihrer Website klar benennen. Sie müssen dort nicht den vollständigen Text dieser Lizenz angeben; der Name und ein Verweis auf den vollständigen Text genügen für's erste. So erklären Sie der Öffentlichkeit, unter welcher Lizenz Sie *beabsichtigen*, die Software zu veröffentlichen, rechtlich gesehen reicht das natürlich nicht. Dafür muss die Software selbst die Lizenz enthalten.

Der übliche Weg dies zu tun ist, den vollständigen Text der Lizenz in einer Datei namens `COPYING` (oder `LICENSE`) abzulegen; dann fügen sie jeder Quellcode-Datei am Beginn eine knappe Notiz hinzu, in der Sie Copyright-Datum, -Inhaber und Lizenz benennen und angeben, wo der vollständige Text der Lizenz zu finden ist.

Davon gibt es viele Variationen, also werden wir uns hier nur ein Beispiel anschauen. Die GNU GPL sagt, dass man einen Hinweis wie diesen an den Anfang jeder Datei des Quellcodes setzen sollte.

```
Copyright (C) <Jahr>  <Name des Autors>

Dieses Programm ist Freie Software: Sie können es unter den Bedingungen
der GNU General Public License, wie von der Free Software Foundation,
Version 3 der Lizenz oder (nach Ihrer Wahl) jeder neueren
veröffentlichten Version, weiterverbreiten und/oder modifizieren.

Dieses Programm wird in der Hoffnung, dass es nützlich sein wird, aber
OHNE JEDE GEWÄHRLEISTUNG, bereitgestellt; sogar ohne die implizite
Gewährleistung der MARKTFÄHIGKEIT oder EIGNUNG FÜR EINEN BESTIMMTEN ZWECK.
Siehe die GNU General Public License für weitere Details.

Sie sollten eine Kopie der GNU General Public License zusammen mit diesem
Programm erhalten haben. Wenn nicht, siehe <http://www.gnu.org/licenses/>.
```

Damit drücken Sie nicht explizit aus, dass der vollständige Text der Lizenz in der Datei `COPYING` steht, es ist aber meistens der Fall. (Sie können den obigen Hinweis abändern um dies direkt auszudrücken.) Diese Vorlage enthält auch die Post-Adresse, über die eine Kopie der Lizenz bezogen werden kann. Noch häufiger wird auf die eine Webseite verwiesen, die den kompletten Text der Lizenz enthält.

Konsultieren Sie einen Juristen (oder vertrauen Sie Ihrem eigenen Rechtsverständnis, falls Sie keinen Juristen zur Hand haben) hinsichtlich der Entscheidung, wo die beständigste Kopie dieser Lizenz gepflegt wird, vielleicht ist es auch einfach die Webseite Ihres Projekts. Im Allgemeinen muss der Hinweis am Anfang jeder Datei nicht genau so aussehen wie der obige, sofern er mit dem Hinweis auf Inhaber des Copyrights, Datum und Lizenz ausgestattet ist und dem Hinweis, wo der komplette Lizenztext zu finden ist.

Den Ton angeben

Bis jetzt haben wir Aufgaben behandelt die einmal beim Aufbau des Projekts zu erledigen sind: Eine Lizenz wählen, die Website einrichten, usw. Die wichtigsten Aspekte beim Start eines Projekts sind aber dynamisch. Es ist einfach, eine Adresse für eine Mailingliste zu wählen; aber dafür zu sorgen, dass dort die Kommunikation beim Thema und produktiv bleibt, ist eine ganz andere Sache. Wenn das Projekt nach Jahren der geschlossenen Entwicklung geöffnet wird, verändert sich der Entwicklungsprozess, und Sie werden die bestehenden Entwickler auf diesen Wandel vorbereiten müssen.

Die ersten Schritte sind die schwersten, da es für zukünftiges Verhalten noch keine Beispiele oder Erwartungen gibt nach denen man sich richten könnte. Beständigkeit in einem Projekt entsteht nicht durch formale Richtlinien, sondern durch eine von allen geteilte, schwer greifbare kollektive Weisheit, die sich mit der Zeit entwickelt. Oft gibt es auch geschriebene Regeln, die aber im wesentlichen eine

Zusammenfassung der sich fortwährend weiterentwickelnden Vereinbarungen sind, an denen sich das Projekt wirklich orientiert. Die schriftlichen Richtlinien legen die Kultur des Projekts nicht fest, sondern beschreiben sie eher, und selbst das nur näherungsweise.

Dafür gibt es ein paar Gründe. Wachstum und hohe Fluktuation sind keineswegs so schädlich für die Herausbildung sozialer Normen, wie man vielleicht denken würde. So lange Veränderungen nicht *zu* schnell ablaufen, haben auch Neulinge Zeit, Abläufe zu lernen. Später werden sie diese Regeln selbst anzuwenden und durchsetzen. Bedenken Sie, wie Kinderlieder Jahrhunderte überdauern. Es gibt heute Kinder, die ungefähr die gleichen Lieder singen wie Kinder vor Hunderten von Jahren, auch wenn keines von ihnen heute noch am Leben ist. Jüngere Kinder hören die Lieder, wie sie von den älteren gesungen werden und wenn sie wiederum älter sind, singen sie vor den anderen jüngeren Kindern. Dabei geben sie die Lieder natürlich nicht bewusst weiter, aber die Lieder überleben trotzdem, weil sie regelmäßig und wiederholt weitergegeben werden. Die Lebenszeit freier Software-Projekte wird vielleicht nicht in Jahrhunderten gemessen (zumindest bis jetzt noch nicht), aber die Dynamik der Übertragung ist ziemlich dieselbe. Die Fluktuation ist allerdings viel höher und muss bei der Weitergabe durch eine aktivere und bewusstere Anstrengung ausgeglichen werden.

Diese Anstrengung wird dadurch unterstützt, dass Neuankömmlige für gewöhnlich soziale Normen erwarten und auch suchen. Das liegt einfach in der Natur des Menschen. In einer Gruppe, die durch ein gemeinsames Bestreben vereint ist, sucht man instinktiv nach Verhaltensmustern, um sich als Mitglied dieser Gruppe darzustellen. Sie sollten früh Beispiele setzen, um das Verhalten der Mitglieder so zu beeinflussen, dass es für das Projekt nützlich ist; einmal etabliert, werden sie überwiegend von selbst weiterbestehen.

Es folgen ein paar Beispiele, um gute Präzedenzfälle zu setzen. Es ist keine ausführliche Liste, sondern lediglich eine Veranschaulichung der Idee, dass es hilfreich ist, die Stimmung für die Zusammenarbeit im Projekt bereits frühzeitig zu prägen. Rein physikalisch mögen die einzelnen Entwickler jeder für sich in einem Raum arbeiten, Sie können jedoch eine Menge dafür tun, ihnen das *Gefühl* zu geben, sie würden alle in einem gemeinsamen Raum arbeiten. Je mehr sie sich so fühlen, desto mehr Zeit werden sie für das Projekt aufwenden wollen. Ich habe diese Beispiele gewählt, da sie in dem Subversion-Projekt aufkamen, (http://subversion.tigris.org/), das ich seit seiner Gründung tätig und beobachtend begleite. Sie gelten aber nicht allein für Subversion; diese Situationen werden in den meisten Open-Source-Projekten aufkommen, und sie sollten als Gelegenheit betrachtet werden, die Dinge auf dem richtigen Fuß zu erwischen.

Private Diskussionen vermeiden

Selbst nachdem Sie ein Projekt an die Öffentlichkeit gebracht haben, werden Sie und die anderen Gründungsmitglieder manchmal damit konfrontiert werden, schwierige Fragen innerhalb eines kleineren Kreises durch private Kommunikation lösen zu müssen. Das gilt besonders am Anfang des Projekts, wo viele wichtige Entscheidungen zu treffen sind und es meist nur wenige Freiwillige gibt, die qualifiziert wären, sie zu treffen. All die offensichtlichen Nachteile öffentlicher Diskussionen auf Mailinglisten werden greifbar vor Ihnen liegen: Die bei E-Mail-Diskussionen unvermeidbare Verzögerung, die für einen Konsens erforderliche Zeit, die Mühe sich mit naiven Freiwilligen auseinandersetzen zu müssen, die meinen, alles zu verstehen (solche gibt es in jedem Projekt; manchmal bringen sie im nächsten Jahr die besten Beiträge, manchmal bleiben sie ewig naiv), die Person die nicht versteht, warum Sie nur das Problem X lösen wollen, wenn es offensichtlich eine Untermenge des größeren Problems Y ist, usw. Die Verlockung, diese Unterhaltungen hinter verschlossenen Türen zu führen und sie als vollendete Tatsache zu präsentieren oder zumindest als nachdrückliche Empfehlung einer vereinigten und einflussreichen Wählergruppe, ist tatsächlich groß.

Tun Sie's nicht.

So langsam und mühselig öffentliche Diskussionen auch sein mögen, sie sind auf lange Sicht trotzdem vorzuziehen. Wichtige Entscheidungen privat zu treffen ist Gift für Freiwillige. Kein Freiwilliger, der

seine Sache ernst meint, würde es lange in einer Umgebung aushalten, in dem alle wichtigen Entscheidungen von einer geheimen Versammlung getroffen werden. Desweiteren hat die öffentliche Diskussion den Vorteil, dass ihre positiven Nebenwirkungen viel länger fortbestehen als die kurzlebige technische Frage um die es geht:

- Die Diskussion wird dabei helfen, neue Entwickler auszubilden und zu unterrichten. Sie können nie wissen, wie viele Augen die Diskussion beobachten; selbst wenn die Meisten sich nicht beteiligen, kann es sein, dass viele sie im Stillen mitverfolgen, um Informationen über das Projekt zu sammeln.

- Bei der Diskussion werden *Sie* die Kunst erlernen, technische Angelegenheiten für Leute zu erklären die mit der Software nicht so vertraut sind wie Sie. Das ist eine Fähigkeit, die Übung erfordert und nicht durch die Unterhaltung mit Ebenbürtigen erlangt werden kann.

- Die Diskussion und ihre Ergebnisse werden auf ewig in den öffentlichen Archiven verfügbar sein und es zukünftigen Diskussionen ermöglichen, Wiederholungen zu vermeiden. Siehe „Auffällige Nutzung der Archive“ im KapitelKapitel 6, *Kommunikation*.

Schließlich besteht die Möglichkeit, dass jemand auf der Mailingliste einen echten Beitrag zu der Diskussion leisten könnte, indem er eine Idee aufbringt, an die Sie nie gedacht hätten. Es ist schwer zu sagen, wie wahrscheinlich das ist; es hängt schlicht von der Komplexität des Problems und dem erforderlichen Grad der Spezialisierung ab. Wenn ich aber ein Beispiel anführen darf, wage ich zu behaupten, dass es viel wahrscheinlicher ist, als man es intuitiv erwarten würde. Im Subversion-Projekt glaubten wir (die Gründer), mit einer tiefen und komplexen Problematik konfrontiert zu sein, über die wir uns seit ein paar Monaten viele Gedanken gemacht hatten; und offen gesagt zweifelten wir daran, dass irgendjemand auf der kürzlich eingerichteten Mailingliste etwas Wertvolles zu der Diskussion beizutragen hätte. Wir nahmen also den einfachen Weg und begannen, unsere technischen Ideen in privaten E-Mails untereinander auszutauschen, bis ein Beobachter des Projekts[3] davon Wind bekam und uns bat, die Diskussion auf die öffentliche Mailingliste zu verlegen. Wir verdrehten zwar ganz schön die Augen, taten es aber – und waren ganz erstaunt über die Anzahl aufschlussreicher Kommentare und Vorschläge, die sich recht schnell ergaben. In vielen Fällen boten Leute Ideen an, die uns nie in den Sinn gekommen wären. Es stellte sich heraus, dass ein paar *sehr* kluge Köpfe auf dieser Liste waren; sie hatten nur auf den richtigen Köder gewartet. Es ist wahr, dass die resultierenden Diskussionen länger dauerten, als wenn wir sie privat gehalten hätten, allerdings waren sie um vieles produktiver, was die zusätzliche Zeit in jedem Fall wert war.

Ohne in allzu verallgemeinernde Aussagen wie "Die Gruppe ist immer schlauer als der Einzelne" abzugleiten (wir kennen alle genügend Gruppen, die daran zweifeln lassen), muss man doch anerkennen, dass es bestimmte Aktivitäten gibt, für die Gruppen besonders geeignet sind. Ausführliche Gutachten sind eine; schnell auf viele Ideen zu kommen eine weitere. Die Qualität dieser Ideen hängt natürlich davon ab, wie hochwertig die Gedanken waren, die man hineingesteckt hat. Sie werden aber nie erfahren, was für geistreiche Denker da draußen sind, solange Sie ihnen keine echten Herausforderungen bieten.

Es gibt natürlich auch Diskussionen, die man im Privaten führen muss; im Verlauf des Buches werden wir Beispiele dafür sehen. Das leitende Prinzip sollte aber sein: *Solange es keinen Grund gibt, etwas privat zu regeln, sollte es öffentlich geschehen.*

Dies in Gang zu setzen erfordert Ihre Einflussnahme. Es reicht nicht lediglich sicherzustellen, dass Ihre eigenen Nachrichten an die öffentliche Mailingliste gehen; Sie müssen auch andere dazu bewegen, ihre unnötig privaten Unterhaltungen öffentlich zu halten. Wenn jemand versucht, eine private Diskussion anzufangen, und es keinen Grund gibt, sie privat zu halten, sollten Sie sich verpflichtet fühlen, sofort eine angemessene übergeordnete Diskussion zu eröffnen. Sie sollten nicht einmal direkt auf das Thema

[3] Wir haben zwar noch nicht das Thema der Namensnennung und Anerkennung angesprochen, aber um auch zu praktizieren, was ich später predigen werde: Der Name des Beobachters war Brian Behlendorf; und er war es, der darauf hingedeutet hat, wie wichtig es ist, alle Diskussionen öffentlich zu halten, es sei denn es gibt einen bestimmten Grund für Geheimhaltung.

eingehen, bevor Sie nicht entweder die Diskussion erfolgreich an einem öffentlichen Ort gelenkt haben oder deutlich erkannt haben, dass sie tatsächlich besser privat gehalten werden sollte. Wenn Sie sich konsequent so verhalten, werden die Beteiligten es ziemlich schnell mitbekommen und gleich die öffentlichen Foren benutzen.

Unhöflichkeit im Keim ersticken

Sie sollten von Anfang an, gleich nachdem Ihr Projekt an die Öffentlichkeit geht, null Toleranz gegenüber unhöflichem oder beleidigendem Verhalten in seinem Foren zeigen. Null Toleranz heißt nicht unbedingt, diese technisch durchzusetzen. Sie sollten diese Leute nicht aus der Liste entfernen, wenn sie einen anderen Teilnehmer "flamen", oder ihnen wegen abfälligen Bemerkungen den Commit-Zugriff entziehen. (Theoretisch müssten Sie eventuell auf solche Mittel zurückgreifen, aber erst nachdem alle anderen erschöpft sind – was am Anfang eines Projekts per Definition noch nicht der Fall ist.) Null Toleranz bedeutet, schlechtes Benehmen niemals einfach unbemerkt geschehen zu lassen. Wenn jemand zum Beispiel eine technische Bemerkung zusammen mit einem *argumentum ad hominem* gegen einen anderen Entwickler koppelt, ist es zwingend notwendig, dass Ihre Reaktion als *erstes* den persönlichen Angriff anspricht, als separate Angelegenheit, und erst dann auf den technischen Inhalt eingeht.

Leider ist es sehr leicht und allzu üblich, dass konstruktive Diskussionen in destruktive "flame wars" ausarten. Menschen werden Dinge sagen, die sie nie von Angesicht zu Angesicht sagen würden. Die Themen dieser Diskussionen verstärken nur diesen Effekt: Bei technischen Angelegenheiten glauben Menschen oftmals, dass es nur eine richtige Antwort zu den meisten Fragen gibt und dass eine Abweichung von ihrer Antwort nur durch Ignoranz oder Dummheit des Anderen erklärt werden kann. Den Vorschlag einer Person als dämlich zu bezeichnen, ist oft nur einen winzigen Schritt davon entfernt, die Person selbst als dämlich zu bezeichnen. Tatsächlich ist es oft schwer zu unterscheiden, wo die technische Diskussion aufhört und die persönliche Beleidigung anfängt, was auch ein Grund ist warum drastische Maßnahmen oder Bestrafungen nicht angebracht sind. Sie sollten stattdessen, sobald Ihnen derartiges auffällt, eine Nachricht schreiben, die nachdrücklich darauf hinweist wie wichtig es ist, die Unterhaltung in einem freundlichen Ton zu führen, ohne dabei jemanden zu beschuldigen, absichtlich giftig gewesen zu sein. Leider hören sich diese "Netter-Polizist"-Nachrichten meist an wie die Predigt einer Vorschullehrerin, die ihre Klasse über gutes Benehmen belehrt:

> *Lasst uns bitte zuerst mit den u.U. ad hominem Bemerkungen aufhören; zum Beispiel, J's Entwurf der Sicherheitsschicht als "naiv und dumm gegenüber allen Grundprinzipien der Informationssicherheit" zu bezeichnen. Ob das so stimmt oder nicht, es ist in jedem Fall nicht die Art, eine Diskussion zu führen. J hat seinen Entwurf in guter Absicht vorgeschlagen. Wenn er Fehler aufweist, weist darauf hin und wir werden sie beheben oder einen neuen Entwurf suchen. Ich bin mir sicher, dass M niemanden persönlich beleidigen wollte, aber er hat sich unglücklich ausgedrückt, und wir wollen versuchen, hier konstruktiv zu bleiben.*
>
> *Und jetzt zu dem Entwurf. Ich denke M hatte recht als er sagte, ...*

So gestelzt sich solche Antworten auch anhören, sie haben doch eine messbare Wirkung. Wenn Sie immer wieder auf solches Verhalten hindeuten, aber keine Entschuldigung von der angreifenden Partei fordern, lassen Sie ihr die Möglichkeit, sich abzuregen und ihre bessere Seite zu zeigen, indem sie sich beim nächsten mal anständiger benimmt – und das wird sie. Eines der Geheimnisse, erfolgreich gegen solches Verhalten vorzugehen, ist niemals die untergeordnete Diskussion zum Hauptthema werden zu lassen. Es sollte immer nur nebenbei erwähnt werden, ein kurzes Vorwort zu der eigentlichen Antwort. Weisen Sie im Vorbeigehen darauf hin, dass "wir hier so nicht arbeiten", aber gehen Sie dann weiter zum echten Inhalt, damit die Beteiligten immer etwas zum Thema haben, worauf sie antworten können. Wenn jemand protestiert, dass er unrechtmäßig zurechtgewiesen wurde, sollten Sie sich nicht in einen Streit verzetteln. Antworten Sie entweder gar nicht (wenn Sie denken, dass die Person nur Dampf ablassen will und nicht wirklich eine Antwort erwartet), oder entschuldigen Sie sich für die übertriebe-

ne Reaktion und schreiben Sie, dass es schwer ist, Nuancen aus einer E-Mail herauszulesen. Gehen Sie danach aber wieder zum eigentlichen Thema über. Bestehen Sie niemals auf ein Eingeständnis, privat oder öffentlich, von jemandem, der sich unangemessen verhalten hat. Wenn er sich von sich aus entschuldigt, ist das großartig, aber es von ihm zu verlangen, würde nur Verbitterung heraufbeschwören.

Das übergeordnete Ziel ist, dass gute Umgangsformen zu einem wesentlichen Merkmal in der Kerngruppe werden. Das hilft dem Projekt, da Entwickler durch "flame wars" vertrieben werden können (sogar von Projekten die sie mögen und unterstützen). Es kann passieren, dass Sie ihre Vertreibung nicht einmal mitbekommen; Mancher könnte sich die Liste anschauen, erkennen dass er ein dickeres Fell bräuchte, um an diesem Projekt teilzunehmen, und verzichtet daraufhin besser gleich ganz auf die Teilnahme. Foren freundlich zu halten ist auf lange Sicht eine Überlebenstaktik und das ist einfacher, wenn das Projekt noch klein ist. Ist es erst zu einem Teil der Kultur geworden, werden Sie nicht mehr der einzige sein, der sich darum bemüht. Jeder wird daran mitwirken.

Code Review

Eine der besten Möglichkeiten, eine produktive Entwicklergemeinschaft zu fördern ist es, Leute zu überreden, sich gegenseitig Ihren Code anzuschauen. Das effektiv zu gewährleisten, erfordert ein wenig technische Infrastruktur – insbesondere sollten Commit-E-Mails angeschaltet werden; siehe „Commit-E-Mails“ für Details hierzu. Commit-E-Mails sorgen dafür, dass jede Änderung am Quellcode eine E-Mail zur Folge hat mit dem zugehörigen Kommentar des Autors und den Diffs (siehe *Diff*[7] im Kapitel „Vokabular der Versionsverwaltung“). *Code Review* heißt, diese Commit-E-Mails beim Eintreffen auch durchzulesen und nach Fehlern und möglichen Verbesserungen zu suchen. [4]

Code Review dient mehreren Zwecken gleichermaßen. Er ist das offensichtlichste Beispiel für "Peer Review" in der Open-Source-Welt und hilft unmittelbar, die Qualität der Software zu erhalten. Jeder Fehler, der in einer Software ausgeliefert wird, kam durch einen Commit zustande der übersehen wurde; es werden umso weniger Fehler in einer veröffentlichten Version sein, je mehr Augen auf jeden Commit gerichtet sind. Code Review dient aber auch einem indirekten Zweck: Es bestätigt Menschen, dass ihre Arbeit Bedeutung hat, man würde sich schließlich nicht die Zeit nehmen über einen Commit zu schauen, wenn es einen nicht interessieren würde, welche Auswirkungen er hat. Menschen leisten dann die beste Arbeit wenn sie wissen, dass andere sich die Zeit nehmen diese zu prüfen.

Jeder Review sollte öffentlich durchgeführt werden. Selbst wenn ich im selben Raum mit anderen Entwicklern bin und einer von uns einen Commit macht, achten wir darauf, den Review nicht verbal im Raum zu führen, sondern ihn über die Entwickler-Liste zu schicken. Jeder profitiert davon, wenn der Review sichtbar ist. Leute folgen den Erläuterungen und finden darin manchmal Mängel und selbst wenn nicht, erinnert es sie zumindest daran, dass Code Review eine zu erwartende regelmäßige Aktivität ist, wie Geschirrspülen oder Rasenmähen.

Im Subversion-Projekt hatten wir am Anfang nicht diese Gewohnheit. Es gab keine Garantie, dass jeder Commit überprüft wurde, wir schauten zwar manchmal über bestimmte Änderungen, wenn ein bestimmter Bereich im Quellcode besonders interessant schien. Es schlichen sich Fehler ein, die man eigentlich hätte sehen sollen und müssen. Ein Entwickler namens Greg Stein, der aus seiner vorherigen Arbeit den Wert von Code Review kannte, entschied sich, ein Beispiel zu setzen, indem er sich jede Zeile von *jedem einzelnen Commit* anschaute. Auf jeden Commit, den irgendjemand machte, folgte bald eine E-Mail von Greg an die Entwickler-Liste, indem er den Commit sezierte, mögliche Probleme analysierte und ab und zu, für besonders cleveren Code, ein Lob aussprach. Sofort begann er, Fehler und problematische Programmierpraktiken zu entdecken, die ansonsten durchgerutscht wären, ohne je bemerkt zu werden. Er beschwerte sich wohlgemerkt nie, dass er der einzige sei, der Code Review betrieb, auch wenn es keinen unwesentlichen Teil seiner Zeit in Anspruch nahm, er lobte bei jeder Gelegenheit die

[7] kurz für *difference*(de. Unterschied)

[4] So wird Review zumindest in Open-Source-Projekten praktiziert. Für zentralisierte Projekte kann "Code Review" auch bedeuten, dass mehrere Personen gemeinsam den ausgedruckten Code durchgehen und nach bestimmten Problemen und Mustern suchen.

Vorteile von Code Review. Ziemlich bald fingen andere an, mich eingeschlossen, regelmäßig Commits zu überprüfen. Was war unsere Motivation? Greg hatte uns nicht bewusst durch Beschämung dazu gebracht. Er hatte bewiesen, dass Code Review eine wertvolle Investition von Zeit ist und dass man durchaus auch zum Projekt beitragen kann, wenn man die Änderungen anderer durchsieht, anstatt selbst neuen Code zu schreiben. Nachdem er das demonstriert hatte, wurde es zum erwarteten Verhalten. Das ging sogar soweit, dass man sich, wenn auf einen Commit keine Reaktion folgte, als Committer Sorgen machte und sogar auf der Liste nachfragte, ob denn niemand die Zeit gefunden hätte, es sich anzuschauen. Später bekam Greg eine Arbeit, die ihm nicht so viel Zeit für Subversion ließ, und er musste mit dem regelmäßigen Code Review aufhören. Aber inzwischen war die Angewohnheit unter uns anderen bereits so weit verbreitet, als sei es nie anders gewesen.

Beginnen Sie mit Code Reviews vom allerersten Commit an. Probleme, die man am einfachsten bei der Durchsicht von Diffs erkennt, sind Sicherheitslücken, Speicherlecks, ungenügende Kommentare oder Dokumentation von Schnittstellen, Eins-daneben-Fehler, Aufrufer-Aufgerufener-Divergenzen und andere Probleme, deren Entdeckung keinen großen Kontext erfordern. Selbst Angelegenheiten größeren Umfangs, wie z.B. häufig auftauchende Muster, nicht an einer gemeinsamen Stelle zu abstrahieren, werden leichter erkennbar, wenn man Code Review regelmäßig betreibt, weil man sich an vergangene Diffs erinnert.

Machen Sie sich keine Sorgen, dass Sie nichts finden, worüber es etwas zu sagen gäbe, oder dass Sie nicht genug über alle Bereiche im Code wissen. Es gibt meistens irgendetwas über so ziemlich jeden Commit zu sagen; selbst wenn Sie nichts Bedenkliches finden, kann es sein, dass Sie etwas Lobenswertes finden. Wichtig ist, jedem der Committer klar zu machen, dass ihre Arbeit gesehen und verstanden wird. Natürlich befreit Code Review die Programmierer nicht von der Verantwortung, ihre Änderungen vor dem Commit zu überprüfen und zu testen; niemand sollte sich auf Code Review verlassen, um Fehler zu finden, die er eigentlich selbst hätte finden sollen.

Der Übergang ehemals geschlossener Projekte zu Open Source

Wenn Sie ein bestehendes Projekt mit aktiven Entwicklern, die eine Umgebung mit geschlossenem Code gewohnt sind, öffnen, sorgen Sie dafür, dass Klarheit über den Umfang der Änderungen besteht, die auf die Entwickler zukommen – und versuchen Sie sich so gut wie möglich in ihre Lage zu versetzen.

Versuchen Sie sich die Situation aus ihrer Sicht vorzustellen: vorher wurden Entscheidungen über Code und Architektur in einer Gruppe von Programmierern getroffen, die sich alle mehr oder weniger gleich gut mit der Software auskannten, die alle den gleichen Druck von oben zu spüren bekamen und die sich alle gegenseitig in ihren Stärken und Schwächen kannten. Jetzt verlangen Sie von ihnen, ihren Code freizugeben, nur damit irgendwelche Fremden ihn auseinandernehmen, untersuchen und sich Meinungen über ihn allein anhand des Quellcodes bilden, ohne zu wissen, welcher geschäftliche Druck Sie zu bestimmten Entscheidungen gezwungen hat. Diese Fremden werden viele Fragen stellen, Fragen die vorhandene Entwickler aufrütteln werden, wenn Sie feststellen müssen, dass die Dokumentation, an der sie so hart gearbeitet haben, *immer noch* lückenhaft ist (das ist unvermeidbar). Um dem ganzen noch ein Sahnehäubchen aufzusetzen: die Neulinge sind unbekannte, gesichtslose Wesen. Wenn einer Ihrer Entwickler sich seiner Fähigkeiten als Programmierer unsicher ist, stellen Sie sich vor, wie es ihn erst verbittern wird, wenn die Neulinge auf Mängel in seinem Code hinweisen, noch dazu vor seinen Kollegen. Wenn Sie kein Team von perfekten Programmierern haben, ist sowas unvermeidlich – tatsächlich wird es am Anfang wahrscheinlich allen passieren. Nicht weil sie schlechte Programmierer sind; sondern weil jedes Projekt oberhalb einer bestimmten Größe zwangsläufig Fehler beinhaltet, und die Überprüfung durch eine Gemeinschaft manche dieser Fehler aufdecken wird (siehe „Code Review“ früher in diesem Kapitel). Gleichzeitig werden die neuen Freiwilligen selber nicht so sehr dieser Prüfung unterliegen, da sie selber noch keinen Code beisteuern können, bis sie mehr mit dem Projekt vertraut sind. Für Ihre Ent-

wickler kann das erscheinen, als ob die ganze Kritik immer nur auf sie gerichtet ist und nie nach außen geht. Es besteht deshalb die Gefahr, dass sich unter den alten Hasen eine Belagerungsmentalität einstellt.

Am besten kann man das verhindern, indem man alle vorwarnt, was auf Sie zukommt; es ihnen erklärt, ihnen sagt, dass ein anfängliches Unbehagen völlig normal ist, und ihnen versichert, dass es mit der Zeit besser wird. Manche dieser Warnungen sollten privat geschehen, bevor das Projekt geöffnet wird. Es könnte aber auch hilfreich sein, die Leute auf der Mailingliste zu erinnern, dass es für das Projekt eine neue Art der Entwicklung ist und dass die Anpassung eine gewisse Zeit brauchen wird. Das beste was Sie machen können, ist als gutes Beispiel voranzugehen. Wenn Sie sehen, dass Ihre Entwickler nicht genügend Fragen der neuen Freiwilligen beantworten, nützt es nichts, ihnen zu sagen, dass sie mehr antworten sollten. Es mag sein, dass sie kein gutes Gefühl dafür haben, wann eine Reaktion gerechtfertigt ist, oder es kann sein, dass sie nicht wissen, welche Priorität die eigentliche Arbeit am Code gegenüber der neuen Bürde einnimmt, mit Außenstehenden zu kommunizieren. Man kann sie am ehesten dazu überreden, sich zu beteiligen, indem man sich selbst beteiligt. Beobachten Sie die öffentliche Mailingliste und beantworten Sie ein paar Fragen. Wenn Sie nicht genügend Erfahrung haben, um die Fragen zu beantworten, sollten Sie es für alle sichtbar an einem anderen Entwickler weitergeben, der die nötige Erfahrung hat – und achten Sie darauf, dass er eine Antwort oder zumindest eine Reaktion gibt. Natürlich wird es für die älteren Entwickler verlockend sein, in private Diskussionen zu verfallen, schließlich sind sie daran gewöhnt. Beobachten Sie deshalb auch die interne Mailingliste und bitten Sie ggf. darum, bestimmte Diskussionen besser gleich auf die öffentliche Liste zu verlegen.

Es gibt andere, langfristigere Bedenken beim Öffnen vormals geschlossener Projekte. Kapitel 4, *Soziale und politische Infrastruktur* untersucht Techniken um bezahlte und unbezahlte Entwickler erfolgreich zu mischen und Kapitel 9, *Lizenzen, Urheberrecht und Patente* behandelt die nötige rechtliche Sorgfalt beim Öffnen von privatem Code, mit bestimmten Komponenten, die einer anderen Partei "gehört", bzw. von ihnen geschrieben wurde.

Bekanntgabe

Sobald das Projekt in einem vorzeigbarem Zustand ist – nicht perfekt, lediglich vorzeigbar – ist es bereit, der Welt bekannt gemacht zu werden. Tatsächlich geht das relativ einfach: Gehen Sie auf http://freshmeat.net/, klicken Sie in der oberen Navigationsleiste auf Submit und füllen Sie das Formular aus, um Ihr Projekt bekanntzumachen. Freshmeat ist der Ort, auf dem alle nach Ankündigungen neuer Projekte Ausschau halten. Sie müssen dort nur ein paar Augen erwischen und ihre Nachricht wird sich von da an über Mundpropaganda weiterverbreiten.

Wenn Sie Mailinglisten oder Newsgroups kennen, auf denen eine Ankündigung Ihres Projekts thematisch passen würde und von Interesse wäre. Sollten Sie dort einen Eintrag machen, achten Sie aber darauf, genau *einen* Eintrag pro Forum zu machen, und verweisen Sie dabei auf Ihre eigenen Foren für weitere anschließende Diskussionen (indem Sie den `Reply-to` Header setzen). Die Einträge sollten kurz und prägnant sein:

```
An: discuss@lists.example.org
Betreff: [ANN] Das Scanley-Projekt für Volltext-Indizierung
Antwort-An: dev@scanley.org

Diese Nachricht ist die einmalige Bekanntgabe über die Gründung des
Scanley-Projekts, eine Open-Source-Volltextindizierungs- und Suchmaschine,
mit einer reichen API für Programmierer, die Suchfunktionen für große
Mengen an Text implementieren wollen. Der Code von Scanley läuft, wird
aktiv entwickelt, und wir suchen sowohl nach Entwickler als auch Nutzern
die testen wollen.
```

```
Webseite: http://www.scanley.org/

Funktionen:
   - Durchsucht Klartext, HTML, und XML
   - Suche nach Wörtern oder Ausdrücken
   - (geplant) Unscharfe Suche
   - (geplant) Inkrementelle Aktualisierung der Indizes
   - (geplant) Indizierung von Webseiten

Voraussetzungen:
   - Python 2.2 oder neuer
   - Genügend Festplattenplatz für die Indizes (ca. 2x die Größe der
     ursprünglichen Daten)

Weiteres finden Sie auf scanley.org.

Vielen Dank,
-H. Mustermann
```

(Siehe „Öffentlichkeit“ im Kapitel Kapitel 6, *Kommunikation* für Ratschläge über die Bekanntmachung neuer Versionen und anderer Ereignisse im Projekt.)

Es gibt eine anhaltende Diskussion in der Open-Source-Gemeinschaft darüber, ob ein Projekt schon am Anfang laufenden Code haben muss, oder ob es einem Projekt hilft, selbst in den frühen Phasen des Entwurfs offen zu sein. Ich dachte früher, dass es am allerwichtigsten sei, mit laufendem Code anzufangen, dass man so erfolgreiche Projekte vom Spielzeug unterscheiden könnte, und ernstzunehmende Entwickler nur etwas anfassen würden, was auch schon etwas Handfestes machte.

Wie es sich herausstellte, war das nicht der Fall. Beim Subversion-Projekt, fingen wir mit einem Entwurf an, ein Kern interessierter und miteinander vertrauter Entwickler, viel Fanfare und *keine einzige Zeile* lauffähigen Codes. Zu meiner völligen Überraschung, schaffte es das Projekt von Anfang an, aktive Freiwillige anzulocken und bis wir tatsächlich etwas Laufendes hatten, waren bereits eine ziemliche Menge freiwilliger Entwickler beteiligt. Subversion ist nicht das einzige Beispiel; das Mozilla-Projekt fing auch ohne laufenden Code an und bietet heute einen erfolgreichen und beliebten Web-Browser.

Angesichts solcher Beweise muss ich von meiner ursprünglichen Behauptung zurücktreten, dass laufender Code absolut notwendig ist, um ein Projekt anzufangen. Trotzdem ist laufender Code immer noch eines der besten Grundlagen für Erfolg, und eine gute Grundregel wäre es, mit der Bekanntgabe zu warten, bis Sie solchen haben. Es mag allerdings Umstände geben, unter denen eine frühere Bekanntmachung Sinn macht. Ich denke, dass zumindest ein gut ausgearbeiteter Entwurf oder irgend ein Grundgerüst für den Code notwendig ist – dieser kann natürlich wegen öffentlicher Rückmeldungen überarbeitet werden müssen, aber es muss etwas Handfestes und Greifbares geben, etwas mehr als gute Absichten, von dem aus Leute ausgehen und weiterarbeiten können.

Wenn Sie die Ankündigung machen, sollten Sie jedoch nicht gleich darauf ein Schar Freiwilliger erwarten. Für gewöhnlich ist das Resultat einer Bekanntmachung, dass Sie nebenbei ein paar Anfragen bekommen, es melden sich ein paar Leute auf der Mailingliste an, abgesehen davon geht so ziemlich alles weiter wie bisher. Mit der Zeit werden Sie aber eine stete Zunahme von Beiträgen neuen Mitarbeiter und Benutzer bemerken. Die Ankündigung ist lediglich das Pflanzen eines Samenkorns. Es braucht Zeit, bis die Nachricht sich fortpflanzt. Wenn das Projekt beständig diejenigen honoriert, die sich beteiligen, wird sich die Nachricht verbreiten, denn Menschen teilen einander mit, wenn sie etwas Gutes entdecken. Wenn alles gut läuft, wird die Dynamik der exponenziellen Kommunikationsnetze das Projekt langsam in eine komplexe Gemeinschaft verwandeln, in der Sie nicht unbedingt jeden Namen kennen und nicht länger jede Unterhaltung mitverfolgen können. Die nächsten Kapitel handeln vom Arbeiten in einer solchen Umgebung.

Kapitel 3. Technische Infrastruktur

Freie Software-Projekte beruhen auf Techniken, die das selektive Aufgreifen und Integrieren von Information ermöglichen. Je gewandter Sie mit diesen Techniken umgehen und andere dazu bewegen können, sie zu benutzen, desto erfolgreicher wird Ihr Projekt sein. Mit zunehmender Größe Ihres Projekts, gewinnt diese Regel umso mehr an Bedeutung. Eine gute Informationsverwaltung verhindert, dass ein Open-Source-Projekt unter der Last von Brooks Gesetz ,[1] zusammenbricht. Sie besagt: Der Einsatz zusätzlicher Arbeitskräfte bei bereits verspäteten Softwareprojekten vergrößert nur die Verspätung. Er beobachtete, dass die Komplexität eines Projekts mit der Anzahl der Teilnehmer *Quadratisch* zunimmt. Wenn nur wenige beteiligt sind, kann jeder leicht mit jedem reden, wenn aber hunderte beteiligt sind, ist es nicht weiter möglich, dass jeder über die Arbeit aller anderen Bescheid weiß. Wenn es bei der Verwaltung eines freien Software-Projekts darum geht, jedem das Gefühl zu geben in einem Raum mit allen Anderen zu sitzen, stellt sich die offensichtliche Frage: Was passiert wenn alle in einem gedrängten Raum versuchen auf einmal zu reden?

Dieses Problem ist nicht neu. In der Praxis wird es gelöst wie in einem Parlament: Es gibt formelle Richtlinien für Diskussionen innerhalb von großen Gruppen, um sicherzustellen, dass wichtige Meinungsverschiedenheiten nicht durch Zwischenrufe verloren gehen. Weitere Richtlinien gibt es für die Bildung von Untergremien, sowie um erkennen zu können, wann Entscheidungen getroffen werden, usw. Ein wichtiger Teil der parlamentarischen Diskussion ist zu spezifizieren, wie die Gruppe ihre Informationen verwaltet. Manche Anmerkungen werden "fürs Protokoll" gemacht, andere nicht. Das Protokoll selbst ist Gegenstand direkter Änderungen, und wird nicht als wörtliche Niederschrift der tatsächlichen Ereignisse angesehen, sondern als Auflistungen der gemeinsam *anerkannten* Ereignisse. Das Protokoll ist nicht monolitisch, sondern nimmt verschiedene Formen an, für verschiedene Zwecke. Es umfasst die einzelnen Sitzungen, die komplette Sammlung aller Sitzungen, Zusammenfassungen, Tagesordnungen mit Anmerkungen, Berichte von Gremien und nicht anwesenden Korrespondenten, Abläufe, usw.

Da das Internet nicht wirklich ein Raum ist, müssen wir nicht die Teile der parlamentarischen Diskussion replizieren, die manche Leute ruhig hält, während andere Reden. Wenn es aber um Techniken zur Verwaltung von Informationen geht, sind gut betriebene Open-Source-Projekte wie eine parlamentarische Diskussionen auf Steroiden. Weil fast die gesamte Kommunikation in Open-Source-Projekten schriftlich abläuft, haben sich wohl durchdachte Systeme entwickelt um Daten angemessen zu markieren und an die richtige Stellen zu leiten; um Wiederholungen zu minimieren, damit Diskussionen nicht auseinander laufen; um Daten zu speichern und abzurufen; um schlechte oder veraltete Informationen zu korrigieren; und um getrennte Informationen miteinander zu verbinden, wenn Zusammenhänge gefunden werden. Aktive Teilnehmer verinnerlichen viele dieser Techniken und werden oft komplexe manuelle Aufgaben durchführen, um sicherzustellen, dass Information am richtigen Ziel ankommen. Diese Bestrebungen hängen aber letztendlich von der Unterstützung ausgeklügelter Software ab. Die Kommunikationsmedien sollten so weit möglich, diese Aufgaben selbstständig übernehmen und Information für Menschen so bequem wie möglich aufbereiten. In der Praxis werden Menschen natürlich an vielen Stellen eingreifen müssen und es ist wichtig, dass diese Eingriffe möglichst einfach sind. Im allgemeinen sollte aber, solange die Menschen sich bemühen Informationen sorgfältig zu Kennzeichnen und am richtigen Ziel zu leiten, die Software so konfiguriert sein, dass sie den größtmöglichen Nutzen aus diesen Metadaten zieht.

Die Ratschläge in diesem Kapitel sind sehr praxisnah und basieren auf ganz bestimmter Software und Nutzungsverhalten. Es geht hierbei aber nicht nur darum bestimmte Techniken zu zeigen. Es geht auch darum, durch viele kleine Beispiele, die beste Haltung zu demonstrieren, um in Ihrem Projekt eine möglichst gute Informationsverwaltung zu fördern. Diese Haltung wird eine Kombination aus technischen Fähigkeiten und sozialer Kompetenz sein. Technische Fähigkeiten sind unentbehrlich, da die Software

[1] Aus seinem Buch *The Mythical Man Month*, 1975. Siehe http://en.wikipedia.org/wiki/The_Mythical_Man-Month und http://en.wikipedia.org/wiki/Brooks_Law.

konfiguriert, sowie gelegentlich gepflegt und Angepasst werden muss, sobald neue Anforderungen auftauchen (siehe z.B. den Abschnitt über die Handhabung von Wachstum, im Abschnitt „Vor-Filterung des Bugtrackers“ später in diesem Kapitel). Soziale Kompetenz ist nötig, da eine menschliche Gemeinschaft auch gepflegt werden muss: Es ist nicht immer sofort ersichtlich wie diese Hilfsmittel am besten genutzt werden, und manchmal haben Projekte widersprüchliche Konventionen (siehe z.B. Beispiel die Diskussion um die Einstellung des `Reply-to` Headers bei ausgehende Nachrichten der Mailingliste, in „Mailinglisten“). Jeder der mit dem Projekt etwas zu tun hat, wird zur richtigen Zeit und auf die richtigen Art, dazu ermutigt werden müssen, seinen Teil beizutragen, die Information des Projekts geordnet zu halten. Je größer die Beteiligung des Freiwilligen, desto komplexer und spezieller werden die Techniken die man von ihr erwarten kann, sich anzueignen.

Es gibt keine schablonenhafte Lösung für Informationsverwaltung. Es gibt einfach zu viele Variablen. Es kann sein das Sie irgendwann alles richtig eingerichtet haben, genau wie Sie es haben möchten, und die meisten Freiwilligen überredet haben mitzumachen. Sobald das Projekt jedoch wächst, können sich manche dieser Vorgänge als nicht skalierbar erweisen. Wenn der Wachstum sich stabilisiert und die Entwickler und Nutzer sich an die technische Infrastruktur gewöhnen, kann eine völlig neues Verwaltungssystem für Informationen aufkommen und neue Freiwillige werden ziemlich bald fragen, warum Ihr Projekt diese nicht einsetzt – viele freie Software-Projekte die vor der Erfindung von Wikis gegründet wurden, erleben das derzeit (siehe http://en.wikipedia.org/wiki/Wiki). Viele Fragen sind Ansichtssache und sind Kompromisse zwischen dem Komfort derjenigen die Informationen produzieren und dem Komfort derjenigen die diese konsumieren, oder der Zeit die erforderlich ist, die Software einzurichten und ihrem Nutzen für das Projekt.

Hüten Sie sich vor der Verlockung allzuviel zu automatisieren, dass eigentlich die Aufmerksamkeit eines Menschen erfordert. Technische Infrastruktur ist wichtig, aber ein freies Software-Projekt funktioniert in Wirklichkeit durch die Fürsorge – und die kluge Formulierung dieser Fürsorge – der beteiligten Menschen. Die technische Infrastruktur ist hauptsächlich diesen Menschen das möglichst zu vereinfachen.

Das nötige Werkzeug

Die meiste Open-Source-Projekte haben ein Mindestmaß an üblichen Hilfsmitteln um Informationen zu Verwalten:

Webseite
: Hauptsächlich ein zentralisierter Kanal für Informationen vom Projekt in die Öffentlichkeit. Die Webseite kann auch eine administrative Schnittstelle für andere Hilfsmittel des Projekts sein.

Mailinglisten
: Meistens das aktivste Forum in einem Projekt und das Medium "fürs Protokoll".

Versionsverwaltung
: Ermöglicht es den Entwicklern Änderungen am Code bequem zu verwalten, u.a. auch rückgängig zu machen. Erlaubt es allen zu sehen was mit dem Code passiert.

Bugtracker
: Ermöglicht Entwicklern ihre Arbeit im Blick zu behalten, miteinander zu koordinieren und neue Versionen zu planen. Erlaubt es jedem den Status von einem Bug zu überprüfen und Informationen über den Bug festzuhalten (z.B. wie man ihn reproduziert). Kann außer zur Beobachtung von Fehlern auch für die Planung von Aufgaben, neuen Versionen, Funktionen usw. benutzt werden.

Chat
: Ein Ort für kurze, oberflächliche Diskussionen und um Fragen und Antworten auszutauschen. Wird nicht immer vollständig archiviert.

Jedes dieser Hilfsmittel ist dazu gedacht, einen bestimmten Bedarf zu decken, ihre Funktionen sind aber auch mit einander verwandt und müssen so eingerichtet werden, dass sie zusammen funktionieren. Weiter unten werden wir untersuchen wie man sie so einrichten kann und viel wichtiger noch, wie man Leute dazu bewegt sie zu benutzen. Die Webseite wird erst zum Schluss behandelt, da es eher als Klebstoff dient denn als eigenständig zu sehendes Hilfsmittel.

Sie können sich eine Menge Kopfschmerzen bei der Einrichtung dieser Hilfsmittel ersparen, mit der Hilfe von Hosting-Bündel: Ein Server der vorkonfigurierte Seiten und Vorlagen bereitstellt, sowie alle zugehörigen Hilfsmittel die man braucht um ein freies Software-Projekt zu betreiben. Siehe „Hosting-Pakete“ später in diesem Kapitel indem die Vor- und Nachteile von Hosting-Lösungen behandelt werden.

Mailinglisten

Mailinglisten sind im Projekt das tägliche Brot der Kommunikation. Sieht der Nutzer irgend ein anderes Forum außer der Webseite, wird es wahrscheinlich eine der Mailinglisten sein. Vorher werden sie sich aber mit der Anmeldung beschäftigen müssen. Das bringt uns zur ersten Regel von Mailinglisten:

> *Versuchen Sie nicht, die Mailingliste händisch zu verwalten – Besorgen Sie sich dazu die nötige Software.*

Es wird verlockend sein, diese Aufgabe hinauszuschieben. Am Anfang scheint es überflüssig diese Software einzurichten. Kleine Verteiler mit geringem Nachrichtenverkehr, scheinen verlockend einfach zu verwalten: Man richtet einfach eine Adresse für die Anmeldung ein, die Anmeldungen werden an Sie weiterleitet, worauf Sie seine Adresse in einer Text-Datei eintragen, indem alle Adressen enthalten sind. Was könnte einfacher sein?

Es gibt allerdings einen Haken. Ein gut verwalteter Verteiler – was viele mittlerweile erwarten – ist alles andere als einfach. Es geht nicht nur darum Leute an- und abzumelden wenn sie eine entsprechende Anfrage stellen. Es geht u.a. darum Spam zu verhindern, nicht einzelne Nachrichten, sondern Zusammenfassungen zu verteilen und Informationen über den Verteiler und das Projekt mittels automatisierter Antworten zu verschicken. Ein Mensch der die Adresse eines Verteilers beobachtet, bietet nur ein Mindestmaß an Funktionen, und selbst dann nicht so zuverlässig und schnell wie es eine Software könnte.

Moderne Mailinglisten-Software bietet für gewöhnlich mindestens folgende Funktionen:

Anmeldung sowohl per E-Mail als auch über die Webseite.
: Wenn ein Nutzer sich bei der Mailingliste anmeldet, sollte er *umgehend* eine automatisierte Willkommensnachricht bekommen, in der beschrieben steht, wofür er sich angemeldet hat, wie man weiter mit der Mailingliste umgeht und (am wichtigsten) wie man sich abmeldet. Diese automatische E-Mail kann natürlich bearbeitet werden, um projektspezifische Informationen einzuschließen, wie z.B. die Webseite des Projekts, wo man die FAQ findet, usw.

Nachrichten in Kurzform oder als Einzelnachrichten
: Im Digest-Modus wird einmal am Tag eine E-Mail verschickt. Sie ist eine Zusammenfassung der gesamten Aktivität auf der Mailingliste, die im Verlauf des Tages aufgekommen ist. Wer die Liste nur nebenher verfolgt, ohne selbst beteiligt zu sein, bevorzugt oft diese Form, da dies ihnen erlaubt, alle Themen auf einmal durchzusehen und hat den Vorteil, die Ablenkung durch ständig eintreffende E-Mails zu vermeiden.

Moderation
: "Moderieren" bedeutet sicherzustellen, dass Nachrichten a) kein Spam sind und b) zum Thema gehören, bevor sie verteilt werden. Moderation muss zwangsläufig von Menschen erledigt werden,

aber Software kann eine Menge dazu beitragen, die Aufgabe einfacher zu gestalten. Zu diesem Thema später mehr.

Administrative Schnittstellen
: Diese ermöglichen es einem Administrator u.a. ohne Umstände veraltete Adressen zu löschen. Was dringend werden kann, wenn die Adresse eines Empfängers anfängt, Antworten wie "Ich bin nicht mehr bei dieser Adresse" auf jede Nachricht von der Liste zu schicken. (Manche Listensoftware kann so etwas automatisch erkennen und die Person abmelden.)

Manipulieren der Header
: Viele Leute haben ausgeklügelte Filter- und Antwort-Regeln für ihre E-Mail-Programm eingerichtet. Mailinglisten-Software kann bestimmte Header hinzufügen und manipulieren, die die Empfänger ausnutzen können (mehr dazu später).

Archivierung
: Alle Nachrichten an die Liste werden gespeichert und im Netz bereitgestellt; alternativ bietet manche Software spezielle Schnittstellen an um externe Archivierungssoftware einzubinden wie z.B. (http://www.mhonarc.org/). Archivierung ist unerlässlich, mehr dazu in „Auffällige Nutzung der Archive“ im Kapitel Kapitel 6, *Kommunikation*.

Der Sinn dieser Punkte ist lediglich zu betonen, dass die Verwaltung einer Mailingliste ein komplexes Problem ist, mit dem sich schon viele beschäftigt haben und weitestgehend gelöst haben. Sie müssen sicherlich kein Experte auf diesem Gebiet werden. Aber Sie sollten sich bewusst machen, dass es immer mehr zu lernen gibt, und dass die Verwaltung der Liste im Verlauf eines freien Software Projekts ab und zu etwas Zeit in Anspruch nehmen wird. Im weiteren werden wir ein paar der häufigsten Punkte beim Konfigurieren einer Antworten angehen.

Schutz vor Spam

Seit ich diesen Satz schrieb, bis zu seiner Veröffentlichung, ist das Internet-weite Spam-Problem wahrscheinlich doppelt so schlimm geworden – oder zumindest wird es einem so vorkommen. Es gab eine Zeit, nicht all zu lange her, in der man eine Liste betreiben konnte ohne überhaupt irgend welche Maßnahmen gegen Spam vornehmen zu müssen. Gelegentlich verirrte sich eine Nachricht, aber selten genug um nur eine geringes Ärgernis zu sein. Diese Ära ist für immer vorbei. Heutzutage wird eine Mailingliste, die keine Maßnahmen gegen Spam unternimmt, schnell von Werbemüll überflutet und dadurch unbenutzbar. Schutz vor Spam ist unerlässlich.

Wir unterteilen Schutz vor Spam in zwei Kategorien: Spam daran zu hindern auf Ihrer Liste aufzutauchen, und zu verhindern, dass Ihre Liste für Spam harvester zu einer Quelle von Adressen wird. Ersteres ist wichtiger also untersuchen wir es zuerst.

Filterung von Nachrichten

Es gibt drei grundsätzliche Arten, Spam zu vermeiden und die meisten Mailinglisten bieten alle drei an. Gemeinsam arbeiten sie am effizientesten:

1. **Es sollten nur Nachrichten von angemeldeten Nutzern automatisch angenommen werden.**

   Diese Einstellung ist weitestgehend effektiv und ist nicht besonders schwer einzurichten, da es für gewöhnlich nur eine kleine Änderung an der Konfiguration der Software des Verteilers bedeutet. Beachten Sie aber, dass Nachrichten, die nicht automatisch akzeptiert werden, nicht einfach verworfen werden sollten. Es gibt zwei Gründe, warum sie stattdessen für weitere Moderation aufbewahrt werden sollten. Erstens wollen Sie auch unangemeldeten Benutzer erlauben, Nachrichten an die Lis-

te zu senden, schließlich sollte eine Person mit einer Frage oder einem Vorschlag sich nicht gleich anmelden müssen, um eine einzige E-Mail zu senden. Zweitens kann es vorkommen, dass auch angemeldete Benutzer eine anderen E-Mail-Adresse benutzen als die, mit der sie angemeldet sind. E-Mail-Adressen sind keine zuverlässige Möglichkeit, Menschen zu identifizieren und sollten nicht als solche behandelt werden.

2. **Benutzen Sie einen Spam-Filter**

   Wenn die Software des Verteilers es ermöglicht (was die meisten tun), können Sie Nachrichten durch einen Spam-Filter laufen lassen. Automatisierte Spam-Filterung ist nicht perfekt und wird es aufgrund des andauernden Wettrüstes zwischen Spammern und den Autoren der Filtersoftware auch nie sein. Es kann aber einen großen Teil des Spams reduzieren, der bis zu den Moderatoren durchkommt. Jede solche Filterung ist vorteilhaft, da die Moderatoren weniger Zeit mit dem Untersuchen von Nachrichten verbringen müssen, und deshalb sehr wünschenswert.

   Hier ist nicht der Raum für eine detailierte Anleitung, wie man einen Spam-Filter einrichtet. Sie werden hierzu die Dokumentation Ihrer Mailinglisten-Software lesen müssen (siehe „Mailinglisten-Software“ später in diesem Kapitel). E-Mail-Verteiler haben oft eingebaute Möglichkeiten um Spam zu verhindern, es kann aber sein, dass Sie weitere Filter von einem dritten Anbieter hinzufügen wollen. Mit diesen Beiden habe ich gute Erfahrungen gemacht: SpamAssassin (http://spamassassin.apache.org/) und SpamProbe (http://spamprobe.sourceforge.net/). Das soll allerdings kein Urteil über andere Open-Source-Filter sein, von denen einige scheinbar auch ziemlich gut sind. Ich habe zufällig diese Beiden benutzt und war mit ihnen zufrieden.

3. **Moderation.**

   Die Letzte Stufe für Nachrichten die nicht von einem angemeldeten Benutzer stammen und es durch den Spamfilter schaffen, ist die *Moderation*: Die E-Mail wird an eine bestimmte Adresse geleitet, wo es von einem Menschen untersucht wird, der sie entweder annimmt oder ablehnt.

   Eine E-Mail zu akzeptieren, kann eine von zwei Formen annehmen: Sie können die E-Mail nur dieses eine Mal annehmen oder Sie können die Software anweisen, diese eine und alle weiteren Nachrichten von dieser Adresse anzunehmen. Letzteres ist fast immer vorzuziehen, um die zukünftige Bürde der Moderation zu verringern. Wie man Nachrichten annimmt, kann sich von System zu System unterscheiden, für gewöhnlich reicht es aber eine E-Mail, an eine bestimmte Adresse zu senden, mit einem Befehl wie "accept" (um nur diese eine E-Mail zu erlauben) oder "allow" (um diese und alle weiteren Nachrichten zu erlauben).

   Um eine Nachricht abzulehnen reicht es meistens sie einfach zu ignorieren. Wenn der Verteiler niemals eine Bestätigung erhält, wird er die E-Mail auch nicht verteilen. Das erwünschte Ergebnis erreicht man also, indem man die E-Mail einfach ignoriert. Manchmal haben Sie auch die Möglichkeit mit "reject" oder "deny" an den Verteiler zu antworten, was dazu führt, dass alle weiteren Nachrichten von dieser Adresse automatisch abgelehnt werden, ohne moderiert zu werden. Es gibt selten einen Grund dafür, da es bei der Moderation meistens darum geht, Spam zu vermeiden und Spammer selten zwei mal die gleiche Adresse benutzen.

Die Moderierung sollte allerdings *ausschließlich* für Spam und Nachrichten benutzt werden, die eindeutig nicht auf die Liste gehören, wie z.B. wenn jemand aus versehen eine Nachricht an die falsche Adresse schickt. Das System wird Ihnen meistens eine Möglichkeit geben eine E-Mail direkt an den Absender zu schicken, nutzen Sie diese aber nicht, um Fragen zu beantworten die eigentlich auf die Mailingliste gehören, selbst wenn Ihnen sofort die Antwort einfällt. Das würde der Gemeinschaft des Projekts nur die Möglichkeit entziehen sich ein klares Bild zu machen, welche Fragen von der Öffentlichkeit gestellt werden und Sie der Gelegenheit berauben, diese Fragen selber zu beantworten und/oder die Antworten anderer zu sehen. Die Moderation einer Mailingliste sollte sich ausschließlich auf Werbemüll und irrelevante Nachrichten beschränken und sonst nichts.

Verschleierung von Adressen im Archiv

Um zu verhindern, dass Ihr Verteiler zu einer Quelle von E-Mail-Adressen für Spammer wird gibt es die gebräuchliche Methode, Adressen in den Archiven zu verschleiern indem man z.B.

```
hmustermann@einedomain.de
```

mit

```
hmustermann_AT_einedomain.de
```

oder

```
hmustermannKEINSPAM@einedomain.de
```

oder irgendetwas (für Menschen) ähnlich offensichtliches ersetzt. Da Spam-Harvester oft nach dem Prinzip funktionieren, Webseiten abzugrasen – auch die Archive Ihres Verteilers – und nach Zeilen mit einem "@" suchen, ist diese Art der Verschleierung eine effektive Methode E-Mail-Adressen vor Spammern zu verstecken oder unbrauchbar zu machen. Was natürlich nichts am Spam, der an die Liste selbst geschickt wird ändert, aber es kann die Menge an Spam reduzieren, die direkt an die persönlichen Adressen der Nutzer des Verteilers gesandt wird.

Die Verschleierung von Adressen kann kontrovers sein. Manche finden, dass es eine gute Idee ist und werden sich wundern wenn Ihre Archive es nicht automatisch machen. Andere denken es ist eine zu große Unbequemlichkeit (da Menschen auch die Adresse wieder korrigieren müssen, vor sie benutzt werden können). Manche behaupten, dass es nicht effektiv sei, da Harvester theoretisch jede konsistente Verschleierung, ausgleichen können. Es gibt jedoch empirische Beweise, dass die Verschleierung von Adressen *tatsächlich* funktioniert, siehe http://www.cdt.org/speech/spam/030319spamreport.shtml.

Im Idealfall würde die Listensoftware diese Entscheidung jedem Benutzer überlassen, entweder durch einen zusätzlichen Header oder in den Einstellungen seines Listenkontos. Ich kenne allerdings keine Software, die diese Entscheidung für jede Nachricht oder jeden Nutzer ermöglicht, also wird vorerst der Administrator der Liste, diese Entscheidung für alle übernehmen müssen (angenommen die Archivierungssoftware bietet diese Einstellung überhaupt an, was nicht immer der Fall ist). Ich ziehe es vor die Verschleierung anzuschalten. Manche sind sehr vorsichtig Ihre Adresse nicht auf Webseiten oder irgendwo anders zu platzieren, an dem ein Spam-Harvester sie finden könnte und sie wären enttäuscht, wenn all ihre Mühe vom Archiv einer Mailingliste zunichte gemacht würden; zudem ist der Aufwand der durch die Verschleierung den Nutzern auferlegt wird nur sehr gering, da es trivial ist eine verschleierte Adresse zu korrigieren, sollte man die Person erreichen wollen. Behalten Sie aber das Wettrüsten im Hinterkopf: Bis Sie diese Zeilen lesen kann es durchaus sein, dass die Harvester sich so weit entwickelt haben, dass sie die häufigsten Verschleierungen erkennen können und wir müssen uns wieder etwas neues einfallen lassen.

Umgang mit E-Mail-Headern

Die Nutzer einer Mailingliste wollen oft die davon stammenden E-Mails in einem bestimmten Ordner sortieren, getrennt von ihren anderen E-Mails. E-Mail-Software kann das automatisch übernehmen, indem sie die *Header* der E-Mails untersucht. Header sind die Felder im Kopf einer E-Mail, wie Absender, Empfänger, Betreff, Datum und verschiedenes andere das Informationen über die Nachricht enthält. Bestimmte Header sind weit verbreitet und im wesentlichen Pflichtangaben:

```
Von: ...
```

```
An: ...
Betreff: ...
Datum: ...
```

Andere sind optional, wenn auch ziemlich üblich. E-Mails müssen z.B. genau genommen keinen

```
Antwort An: absender@email.adresse.hier
```

Header angeben, tun es aber trotzdem, da es den Empfängern eine narrensichere Möglichkeit gibt den Autoren zu erreichen (was besonders nützlich ist wenn der Autor die E-Mail von einer anderen Adresse senden musste, als diejenige an die die Antworten gerichtet sein sollten).

In manchen E-Mail-Programmen ist es einfach, E-Mails anhand von verschiedenen Mustern im Betreff unterschiedlich abzulegen. Das führt zum Bedarf, dass der Verteiler automatisch einen Präfix vor jeden Betreff setzen soll, damit E-Mail-Programme danach suchen und automatisch Nachrichten in den richtigen Ordnern ablegen können. Die Idee ist, dass der Autor folgendes schreiben würde:

```
Betreff: Erstelle die Version 2.5.
```

die Nachricht aber wie folgt ankommen würde:

```
Betreff: [diskussion@verteiler.beispiel.org] Erstelle die Version 2.5.
```

Obwohl die meisten Listenlösungen diese Möglichkeit bieten, empfehle ich diese Option nicht anzuschalten. Das Problem, kann viel einfacher auf eine sehr viel weniger aufdringliche Art gelöst werden, und der Preis den man durch verlorenen Platz im Betreff verliert ist viel zu hoch. Erfahrene Nutzer von Mailinglisten, suchen die Betreffzeilen ihrer im Laufe des Tages eingetroffenen E-Mails ab und entscheiden daran, ob sie eine Nachricht lesen oder auf sie antworten. Den Namen der Liste vor dem eigentlichen Betreff zu setzen, kann die rechte Seite des Betreffs über den Bildschirmrand hinausschieben, wodurch er verschwindet. Die Informationen auf denen sich Leser verlassen um zu entscheiden, welche Nachrichten Sie öffnen sollen gehen damit z.T. verloren und der Nutzen der Liste wird dadurch insgesamt für alle verringert.

Anstatt die Betreffzeile zu verunstalten, sollten Sie Ihren Nutzern beibringen, andere übliche Header zu verwenden, angefangen mit dem To("An:")-Header indem die Adresse der Liste angeben werden sollte:

```
An: <diskussion@verteiler.beispiel.org>
```

Jede E-Mail-Software die nach Betreff filtern kann, sollte ebenso einfach auch in der Lage sein, nach dem To-Header zu filtern.

Es gibt ein paar weitere optionale, aber üblicherweise angegebene Header, die bei Mailinglisten erwartet werden. Nach ihnen zu filtern ist noch zuverlässiger als die "To" oder "Cc" Header; da diese Header von dem Verteiler an jede Nachrichten angehängt werden, verlassen sich manche Nutzer möglicherweise auf sie:

```
list-help: <mailto:discuss-help@lists.example.org>
```

```
list-unsubscribe: <mailto:discuss-unsubscribe@lists.example.org>
list-post: <mailto:discuss@lists.example.org>
Delivered-To: mailing list discuss@lists.example.org
Mailing-List: contact discuss-help@lists.example.org; run by ezmlm
```

Zum größten Teil sind diese selbsterklärend. Siehe http://www.nisto.com/listspec/list-manager-intro.html für eine weitergehende Erklärung, oder http://www.faqs.org/rfcs/rfc2369.html für die ausführliche formale Spezifikation.

Beachten Sie auch, dass diese Header durch den Prefix "list" implizieren, dass Sie auch administrative Adressen mit den Namen "list-help" und "list-unsubscribe" eingerichtet haben. Weitere üblich Adressen sind "list-subscribe", um die Liste zu abonnieren und "list-owner", um seine Administratoren zu erreichen. Je nachdem welche Software Sie für Ihre Liste verwenden, werden diese und verschiedene andere eingerichtet sein; die Dokumentation wird dazu weitere Angaben bietem. Eine vollständige Liste aller Adressen wird meistens jedem Nutzer als Teil der "Willkommensnachricht", bei der Anmeldung zugeschickt. Sie werden selber wahrscheinlich eine Kopie dieser E-Mail bekommen. Wenn nicht sollten Sie jemand um eine Kopie bitten, damit Sie wissen was Ihre Nutzer sehen, sobald sie die Liste abonnieren. Behalten Sie sie in Griffweite um Fragen über die Funktionen der Liste beantworten zu können, bzw. noch besser wäre, es irgendwo auf Ihre Webseite zu stellen, um bei Fragen der Form "Wie kann ich mich abmelden?", Sie einfach auf eine URL weisen können.

Manche Verteiler bieten die Option, an jede Nachricht Anweisungen anzuhängen, um sich abmelden. Wenn es sie gibt, sollten sie angeschaltet werden. Es verbraucht für jede Nachricht nur ein paar zusätzliche Zeilen an einer harmlosen Stelle und kann viel Zeit ersparen, indem es die Anzahl der Nutzer reduziert die an Sie – oder schlimmer noch an die ganze Liste – Anfragen schicken, wie man sich abmelden kann.

Die große "reply-to"-Debatte

Ich habe vorhin in „Private Diskussionen vermeiden“ betont, wie wichtig es ist, Diskussionen in den öffentlichen Foren zu halten und erwähnte, dass aktive Maßnahmen manchmal nötig sind, um zu verhindern, dass Unterhaltungen ins private abgleiten; weiterhin geht es in diesem Kapitel darum, die Kommunikationssoftware soweit zu konfigurieren, dass sie möglichst viel Arbeit übernimmt. Es wäre deshalb anzunehmen, dass Verteiler mit einer Möglichkeit, alle Nachrichten öffentlich zu halten, diese Option offensichtlich anschalten sollte.

Nun ja, nicht ganz. Es gibt solch eine Funktion, allerdings hat sie ein paar schwerwiegende Nachteile. Die Frage ob man sie benutzen soll oder nicht, ist eine der am heißesten debattierten bei der Verwaltung von Mailinglisten – zugegeben, es ist nicht unbedingt eine Kontroverse die es auf die Titelseite Ihres Tageblatts schaffen würde, aber Sie kann in einem freien Software-Projekt ab und zu aufflammen. Unten werde ich die Funktion beschreiben, die Hauptargumente beider Seiten erläutern und die mir bestmögliche Empfehlung geben.

Die Funktion selbst ist relativ einfach: Wenn Sie wollen, kann die Listen-Software automatisch den Reply-to (de. Antwort an) Header auf die Adresse der Liste setzen. Was so viel heißt, dass egal was der Autor einer E-Mail in den Reply-to Header schreibt (bzw. selbst wenn er nicht einmal angegeben wird), bei den Empfängern der Header mit der Adresse der Liste erscheint:

```
Reply-to: discuss@lists.example.org
```

Oberflächlich scheint das eine gute Sache zu sein. Praktisch jede E-Mail-Software berücksichtigt den Reply-to Header. Wenn nun jemand auf eine Nachricht antwortet, wird seine Nachricht an die gesamte Liste gerichtet sein und nicht nur an den Autor der ursprünglichen Nachricht. Natürlich kann der Antwortende immer noch händisch den Empfänger ändern, wichtig ist aber, dass Antworten *standardmäßig*

an die Liste gerichtet sind. Es ist ein perfektes Beispiel für eine Technik, um gemeinschaftliche Arbeit zu unterstützen.

Leider gibt es einige Nachteile. Der erste ist bekannt als das *Ich-finde-nicht-den-Weg-nach-Hause*-Problem: Manchmal setzt der ursprüngliche Absender seine "echte" E-Mail-Adresse in den Reply-to Header, da er aus irgendeinem Grund die Nachricht von einer anderen Adresse absendet, als er Antworten empfangen möchte. Personen, die immer von der gleichen Adresse absenden und empfangen, kennen dieses Problem nicht und viele sind überrascht, dass es das Problem überhaupt gibt. Für Leute mit einer ungewöhnlichen E-Mail-Konfiguration oder ohne Einfluss auf die Gestalt ihrer E-Mails (vielleicht weil sie von der Arbeit schreiben und keinen Einfluss auf ihre IT-Abteilung haben), kann der Reply-to Header die einzig sichere Möglichkeit sein, Antworten an die richtige Adresse zu leiten. Wenn Sie nun an eine Liste schreiben, ohne in dieser angemeldet zu sein, wird ihre Reply-to Einstellung zu einer unabdingbaren Information. Wenn der Verteiler diese nun überschreibt, wird Sie die Rückmeldung auf Ihre Nachricht vielleicht niemals erreichen.

Der zweite Nachteil betrifft eine Erwartungshaltung, und das ist meiner Meinung nach das stärkste Argument gegen die Verunstaltung von Reply-to. Die meisten erfahrenen E-Mail-Nutzer, sind an zwei grundsätzliche Arten zu antworten gewohnt: *reply-to-all (Antwort an alle)* und *reply-to-author (Antwort an Autor)*. Alle modernen E-Mail-Programme bieten beide Optionen an. Nutzer wissen, dass sie reply-to-all wählen sollten um an alle zu antworten (d.h. inklusive der Personen auf der Liste), und dass sie reply-to-author wählen sollten, um eine private Nachricht an den Autor zu schicken. Auch wenn Sie an jeder möglichen Stelle zu offenen Diskussionen ermutigen sollten, gibt es doch Situationen, bei denen der Antwortende eine private Nachricht bevorzugen sollte – zum Beispiel wenn er etwas im Vertrauen an den Autor schreiben möchte, was unangemessen für die öffentliche Liste wäre.

Schauen Sie, was nun passiert, wenn der Verteiler den ursprünglichen Reply-to Eintrag überschreibt und der Antwortende auf den reply-to-author Knopf drückt, in der Erwartung eine private E-Mail an den Absender zu schicken. Aufgrund seiner Erwartungshaltung, wird er die Adresse des Empfängers vielleicht nicht nochmals überprüfen. Er hat dabei vielleicht eine geheime, vertrauliche Nachricht verfasst, mit peinlichen Details über ein Mitglied des Verteilers und drückt nun auf absenden. Seine Nachricht erscheint nun unerwartet, etwas später *auf dem Verteiler!* Zugegeben, theoretisch hätte er sorgfältig auf das Empfänger-Feld achten sollen, und keine Annahmen über den Reply-to Header machen sollen. Autoren setzen aber fast immer Reply-to auf ihre eigene persönliche Adresse (bzw. ihre E-Mail-Software macht es für sie), und viele langjährige Nutzer, erwarten es mittlerweile. Das geht sogar soweit, dass wenn jemand absichtlich reply-to auf irgendeine andere Adresse setzt als den Verteiler, er es explizit in dem Text der E-Mail erwähnt, damit Leute sich bei ihrer Antwort nicht wundern.

Aufgrund der potentiell schwerwiegenden Folgen dieses unerwarteten Verhaltens, bevorzuge ich es, Verteiler so zu konfigurieren, dass sie den reply-to Header niemals anfassen. Es ist ein Beispiel für eine Technik um Zusammenarbeit zu unterstützen, mit wie es mir scheint, potentiell gefährlichen Nebenwirkungen. Auf der anderen Seite dieser Debatte gibt es jedoch auch starke Argumente. Es werden Leute, egal wie Sie sich entscheiden, ab und zu fragen, warum Sie sich nicht anders entschieden haben. Da sowas niemals zum Hauptthema einer Diskussion werden sollte, kann es angebracht sein, hierfür eine vorformulierte Antwort parat zu haben, die so gestaltet ist, dass sie die Diskussion eher beendet als anfeuert. Stellen Sie klar, dass Sie *nicht* darauf bestehen, die einzig richtige und sinnvolle Entscheidung getroffen zu haben, (selbst wenn Sie das denken). Deuten statt dessen darauf wie alt diese Debatte ist, dass es gute Argumente auf beiden Seiten gibt, keine Entscheidung alle zufriedenstellen wird und dass Sie einfach die Ihnen bestmögliche Entscheidung getroffen haben. Bitten Sie höfflich darum diese Diskussion nicht weiterzuführen, es sei denn jemand hat etwas wirklich neues zu sagen. Halten Sie sich danach aus dem Thread heraus und hoffen Sie, dass er eines natürlichen Todes stirbt.

Jemand wird vielleicht vorschlagen eine Wahl darüber zu halten. Wenn Sie möchten können Sie das tun, ich persönlich finde es in diesem Fall jedoch unzureichend einfach Köpfe zu zählen. Für jemanden, der ein gewisses Verhalten erwartet, wäre die Strafe unangemessen hoch (versehentliche Sendung einer privaten E-Mail an die Liste) und die Unbequemlichkeit für die übrigen Teilnehmer ist relativ gering (ab

und zu jemand daran erinnern an den ganzen Verteiler zu antworten, statt nur an den Autor), weshalb man nicht eindeutig sagen kann dass die Mehrheit, auch wenn sie das ist, eine Minderheit in solch eine Gefahr bringen darf.

Ich habe nur die wichtigsten Aspekte dieses Themas hier angesprochen. Für eine vollständige Behandlung des Themas verweise ich auf folgende zwei anerkannten Dokumente, die bei dieser Debatte immer wieder zitiert werden:

- **Lass Reply-to in Ruhe**, *von Chip Rosenthal*

  http://www.unicom.com/pw/reply-to-harmful.html

- **Setze Reply-to auf die Liste**, *von Simon Hill*

  http://www.metasystema.net/essays/reply-to.mhtml

Trotz meiner angedeuteten leichten Präferenz, denke ich nicht, dass es auf diese Frage eine "richtige" Antwort gibt und beteilige mich auch gerne auf vielen Listen die Reply-to setzen. Das wichtigste zu diesem Thema ist, sich frühzeitig für das eine oder andere zu entscheiden und sich danach nicht zu Debatten über das Thema verleiten zu lassen.

Zwei Fantasien

Eines Tages wird jemand auf die geniale Idee kommen, einen *reply-to-list* Schlüssel in einer E-Mail Software zu implementieren. Es würde irgendwelche der vorher erwähnten spezifischen Header benutzen, um die Adresse der Mailingliste herauszufinden und direkt bzw. nur an die Liste zu antworten, ohne gesonderte Adresse für den Empfänger, da diese höchst wahrscheinlich sowieso die Liste abonniert haben. Mit der Zeit werden andere E-Mail-Programme diese Funktion übernehmen und diese ganze Debatte wird sich auflösen. (Tatsächlich gibt es sogar eine E-Mail-Software namens Mutt [http://www.mutt.org/] mit einer solchen Funktion.[2])

Eine noch bessere Lösung wäre es, die Verunstaltung von Reply-to jedem selbst zu überlassen. Wer haben möchte, dass der Verteiler ihre Reply-to Header ändert (entweder nur für ihren eigenen oder für alle Nachrichten) könnten darum bitten und solche die es nicht wollen, könnten bitten es in Ruhe zu lassen. Ich kenne jedoch keine Software die so etwas für jeden Benutzer einzeln einstellen lässt. Es scheint, dass wir derzeit mit einer globalen Einstellung für alle leben müssen.[3]

Archivierung

Wie man genau ein Archiv für Mailinglisten einrichtet, ist bei jeder Listen-Software unterschiedlich, und würde den Rahmen dieses Buchs sprengen. Wenn Sie ein Programm zur Archivierung wählen oder konfigurieren, sollten Sie auf folgendes achten:

Zeitnahe Aktualisierung
: Teilnehmer werden oft auf eine archivierte Nachricht verweisen wollen, die erst vor ein oder zwei Stunden gepostet wurde. Wenn möglich sollte die Software jede Nachricht sofort archivieren, sodass im gleichen Moment, indem es vom Verteiler versandt wird, es auch im Archiv ist. Wenn diese Option nicht verfügbar ist, versuchen Sie die Software zumindest so einzustellen, dass es sich ca. jede Stunde aktualisiert. (Standardmäßig lässt mache Software die Aktualisierung ein Mal jede Nacht laufen, was aber bei einer aktiven Mailingliste in der Praxis eine viel zu große Verzögerung bedeutet.)

[2] Kurz nachdem dieses Buch erschien, schrieb mir Michael Bernstein [http://www.michaelbernstein.com/] folgende Nachricht: "Es gibt weitere E-Mail-Programme außer Mutt, die eine reply-to-list Funktion implementiert haben. Evolution bietet diese Funktion z.B. wenn auch ohne eigene Schaltfläche, über die Tastenkombination (Strg+L)."

[3] Seitdem ich das schrieb, habe ich zumindest von einer Mailinglisten-Software erfahren, die diese Funktion anbietet: Siesta [http://siesta.unixbeard.net/]. Siehe dazu auch diesen Artikel: http://www.perl.com/pub/a/2004/02/05/siesta.html

Link Stabilität
: Sobald eine E-Mail unter einer bestimmten URL archiviert wurde, sollte sie ewig, bzw. so lang wie möglich unter genau der gleichen URL erreichbar sein. Selbst wenn die Archive neu aufgebaut werden, aus einem Backup wiederhergestellt werden oder sonstwie repariert werden, sollte jede öffentlich bekannte URL weiterhin gültig sein. Stabile Verweise ermöglichen es Suchmaschinen die Archive zu indexieren und das ist für Nutzer die auf der Suche nach einer bestimmten Nachricht sind, ein großer Segen. Stabile Verweise sind auch wichtig, da der Bugtracker (siehe „Bugtracker“) später in diesem Kapitel oder Dokumente des Projekts oftmals auf darin enthaltene Nachrichten verweisen.

  Idealerweise würde die Mailinglisten-Software eine URL der jeweiligen Nachricht im Archiv, in einem Header mitschicken oder zumindest den URL-Teil, spezifisch für die E-Mail. So weiß jeder mit einer Kopie der E-Mail, wo es im Archiv zu finden ist, ohne die Archive tatsächlich aufsuchen zu müssen, was hilfreich wäre da jeder Vorgang mit dem Browser, automatisch einen größeren Zeitaufwand bedeutet. Ich weiß nicht ob irgendeine Listen-Software diese Funktion anbietet; diejenigen die ich benutzt habe, können es leider nicht. Es ist allerdings etwas nachdem man suchen sollte (oder falls Sie Mailinglisten-Software schreiben, wäre es eine Funktion, die Sie bitte in Erwägung ziehen sollten).

Backup
: Es sollte ziemlich offensichtlich sein wie man Sicherungen des Archivs macht und der Vorgang um sie wiederherzustellen sollte nicht zu schwierig sein. Mit anderen Worten, behandeln Sie Ihr Archiv nicht wie eine Blackbox. Sie (oder jemand aus Ihrem Projekt) sollte wissen wo die Nachrichten gespeichert werden und wie die Seiten des Archivs sich wiederherzustellen lassen, sollte es jemals nötig werden. Diese Archive sind wertvolle Daten – wenn ein Projekt sie verliert, geht auch ein großer Teil seiner kollektiven Erinnerung verloren.

Unterstützung für Threads
: Es sollte möglich sein, von jeder Nachricht aus, zu seinem zugehörigen *thread* (eine Gruppe verwandter Nachrichten) zu gehen. Jeder Thread sollte auch seine eigene URL haben, getrennt von denen seiner einzelnen Nachrichten.

Durchsuchbarkeit
: Ein Archiv das man nicht durchsuchen kann – sowohl Volltext, als auch Betreffzeilen – ist nahezu wertlos. Bedenke, dass manche Archive ihre Suchfunktion über eine externe Suchmaschinen wie Google [http://www.google.com/] anbieten, indem sie die Arbeit einfach auslagern. Das ist zwar akzeptable, aber eine direkte Suchfunktion ist meistens besser abgestimmt, da es dem Suchenden z.B. erlaubt, nur die Betreffzeile zu durchsuchen oder den gesamten Text.

Obiges ist lediglich eine Checkliste, um Ihnen die Evaluierung und Einrichtung der Software für ihre Archive zu erleichtern. Leute zu überreden es wirklich zum Vorteil des Projekts zu *benutzen* wird in späteren Kapiteln behandelt, insbesondere im Abschnitt „Auffällige Nutzung der Archive“.

Mailinglisten-Software

Hier sind einige Open-Source-Programme für die Verwaltung von Listen und Archives. Wenn Ihre Projektseite bereits vorkonfiguriert wurde, werden Sie sich u.U. niemals für eines entscheiden müssen. Wenn Sie es jedoch selber einrichten müssen, sind hier einige Optionen. Zu der Software, die ich tatsächlich benutzt habe gehört Mailman, Etmlm, MHonArc, und Hypermail, was aber keine Aussage über andere sein soll (und natürlich gibt es bestimmt auch andere Programme die ich einfach nicht gefunden habe, betrachten Sie diese Liste also nicht als vollständig).

Software für Mailinglisten:

- **Mailman** – http://www.list.org/

(Mit einem eingebauten Archiv und die Möglichkeit externe einzubinden. Mailman ist seit langer Zeit der Standard; seine administrative Schnittstellen, insbesondere für die Moderation von Spam und frisch angemeldeten Mitgliedern, wirken recht angestaubt und können besonders auf diejenigen frustrierend wirken, die modernere Schnittstellen gewohnt sind.)

- **GroupServer** – http://www.groupserver.org/

  (Hat eingebaute Archivierung und integriertes Web-Interface. GroupServer ist etwas schwerer einzurichten als Mailman, und erfordert (Stand: Anfang 2012) eine sehr spezielle Menge an Voraussetzungen, aber wenn Sie ihn erst einmal zum Laufen gebracht haben, bietet es den Benutzern ein besseres Arbeiten.)

- **Sympa** — http://www.sympa.org/

  (Entwickelt und gepflegt durch einen Verband französischer Universitäten, sowohl für sehr große Listen ausgelegt (> 700000 Mitglieder) als auch eine große Anzahl von Listen. Es kann mit einer Vielzahl von Abhängigkeiten umgehen; z.B. können Sie es mit sendmail, postfix, qmail oder exim als unterliegenden Message Transfer Agent nutzen. Eine Web-basierte Archivierung ist eingebaut.)

- **SmartList** – http://www.procmail.org/

  (Für die Nutzung mit Procmail gedacht.)

- **Ecartis** – http://www.ecartis.org/

- **ListProc** – http://listproc.sourceforge.net/

- **Ezmlm** – http://cr.yp.to/ezmlm.html

  (Entworfen für Qmail [http://cr.yp.to/qmail.html].)

- **Dada** – http://mojo.skazat.com/

  (Trotz des bizarren Versuchs der Webseite die Tatsache zu verstecken, dass sie freie Software ist, wird sie unter der GNU GPL veröffentlicht. Es hat auch einen eingebauten Archiv.)

Software zur Archivierung:

- **MHonArc** – http://www.mhonarc.org/

- **Hypermail** – http://www.hypermail.org/

- **Lurker** – http://sourceforge.net/projects/lurker/

- **Procmail** – http://www.procmail.org/

  (Begleitend zu SmartList, eine allgemeine Software zur Verarbeitung von E-Mails und scheinbar auch als Archiv konfigurierbar.)

Versionsverwaltung

Eine *Versionsverwaltung*[4] (en. Version Control) ist eine Kombination verschiedener Techniken und Verfahren um Änderung an den Dateien eines Projekts, insbesondere Quellcode, Dokumentation und

[4] im Deutschen auch manchmal "Versionskontrolle" genannt.

Webseiten, zu verfolgen und verwalten. Wenn Sie Versionsverwaltung noch nie benutzt haben, sollten Sie sich als Erstes jemand suchen die es kennt und sie überreden dem Projekt beizutreten. Heutzutage wird jeder erwarten, dass zumindest Ihr Quellcode unter Versionsverwaltung steht und keiner wird Ihr Projekt ernst nehmen, wenn es nicht zumindest halbwegs kompetent mit seiner Versionsverwaltung umgeht.

Versionsverwaltung ist beim Betrieb eines Projekts allgegenwärtig, weile es in nahezu jedem Bereich hilft: Kommunikation unter den Entwicklern, Veröffentlichung neuer Versionen, Bug-Verwaltung, Code-Stabilität und experimentelle Entwicklungen sowie die Annahme von und Anerkennung für Änderungen durch bestimmte Entwickler. Die Versionsverwaltung kann all diese Bereiche zentral koordinieren. Der Kern der Versionsverwaltung ist die Verwaltung von Änderungen (en. *change management*: Sie identifiziert jede einzelne Änderung an den Dateien eines Projekts, fügt ihnen Metadaten bei, wie das Datum der Änderung, den Namen des Autors und kann jedem der danach sucht, diese auf die gewünschte Art aufbereiten. Sie ist eine Methode zur Kommunikation, bei dem eine Änderung die grundlegende Einheit der Information ist.

Dieser Abschnitt behandelt nicht alle Aspekte der Bedienung einer Versionsverwaltung. Dieses Thema ist derart umfassend, dass es im Verlaufe des Buchs immer wieder angesprochen werden muss. Hier werden wir uns darauf konzentrieren ein Versionsverwaltung auszuwählen und einzurichten, die später die gemeinschaftliche Entwicklung unterstützt.

Vokabular der Versionsverwaltung

Dieses Buch kann Ihnen die Bedienung einer Versionsverwaltung nicht beibringen ohne vorherige Erfahrung, es wäre aber unmöglich das Thema zu behandeln, ohne ein paar Begriffe zu klären. Diese sind unabhängig von der eingesetzten Versionsverwaltung: Sie sind die grundsätzlichen Nomen und Verben der gemeinsamen Arbeit im Netzwerk und sie werden immer wieder im Verlaufe des Buches aufkommen. Selbst wenn es keine Versionsverwaltung gäbe, bestünde das Problem der Verwaltung von Änderungen und diese Wörter geben uns eine Sprache um präzise und prägnant über das Problem zu reden.

"Version" kontra "Revision"

Das Wort *Version* wird manchmal als Synonym für "Revision" benutzt, ich werde es jedoch in diesem Buch nicht auf diese Art verwenden da es zu leicht mit "Version" im Sinne einer bestimmten Version einer Software – also eine Veröffentlichung, mit einer Versionsnummer wie "Version 1.0", zu verwechseln ist . Da der Begriff Versionsverwaltung bereits geläufig ist, werden ich diesen trotzdem weiterhin verwenden.

Commit
: Eine Änderung an dem Projekt vornehmen; formeller gesagt, eine Änderung in die Versionsverwaltung zu speichern, sodass es in zukünftige Versionen des Projekts eingebunden werden kann. "Commit" (de. festlegen) kann als Nomen oder als Verb benutzt werden. Als Nomen ist es im wesentlichen ein Synonym für Änderung. Beispiel: "Ich habe eben einen Bugfix der bei Nutzern von Max OS X, Abstürze ihrer Server verursacht hat committed, Jay könntest du dir bitte den Commit anschauen und überprüfen, dass ich dort mit der Speicher Zuweisung nicht falsch umgehe?"

Commit-Log
: Ein Kommentar der an jedem Commit angehängt wird, mit einer Beschreibung über die Änderung und sein Nutzen. Commit-Kommentare sind mitunter die wichtigsten Dokumente in einem Projekt: Sie sind die Brücke zwischen der äußerst technischen Sprache der einzelnen Änderungen am Code und der eher Nutzer orientierten Sprache der Funktionen, Bugfixes und dem Projektfortschritt. Später in diesem Abschnitt werden wir uns Möglichkeiten anschauen, Commit-Logs für das entspre-

chend angemessene Publikum zu veröffentlichen; ebenso sind in „Festschreiben von Traditionen“ im Kapitel Kapitel 6, *Kommunikation* Methoden beschrieben, Beteiligte dazu anzuregen, kurze prägnante und nützliche Commit-Kommentare zu schreiben.

Update
: Eine Anfrage die Änderungen (Commits) anderer Teilnehmer in die eigenen lokalen Kopie des Projekts einzubinden; bzw. Ihre Kopie zu aktualisieren. Dies ist ein sehr häufiger Vorgang; die meisten Entwickler aktualisieren ihren Code mehrmals am Tag um sicherzustellen, dass sie ungefähr das Gleiche benutzen, wie die anderen Entwickler und beim Auffinden eines Fehlers sicher sein zu können, dass er noch nicht behoben wurde. z.B.: "Hallo, ich habe bemerkt, dass der Code für die Indexierung immer das letzte Byte fallen lässt. Ist das ein neuer Bug?". "Ja, aber er wurde letzte Woche behoben – versuch mal ein Update zu machen, dann sollte er verschwinden."

Projektarchiv
: Die Datenbank der Versionsverwaltung in der Änderungen gespeichert werden. Manche Systeme sind zentralisiert: Es gibt ein Projektarchiv, in dem alle Änderungen am Projekt gespeichert werden. Andere sind dezentralisiert: dort hat jeder Entwickler sein eigenes Projektarchiv, und Änderungen können beliebig hin und her getauscht werden. Die Versionsverwaltung verfolgt die Abhängigkeiten zwischen den Änderungen und wenn es Zeit wird, eine neue Version zu herauszugeben, bekommt ein bestimmter Satz von Änderungen den Zuspruch als neue Version. Die Frage welche der beiden besser ist, ist ein weiterer der andauernden heiligen Kriege der Softwareentwicklung; versuchen Sie nicht in die Falle zu tappen, auf Ihrer Mailingliste darüber zu streiten.

Checkout
: Sich eine Kopie des Projekts aus dem Projektarchiv zu beschaffen. Ein Checkout produziert meistens eine Verzeichnisstruktur, auch als Arbeitsverzeichnis (siehe unten) bekannt, von dem aus Änderungen wieder zurück ins Projektarchiv übertragen werden können. Bei manchen dezentralisierten Versionsverwaltungen ist jedes Arbeitsverzeichnis selbst ein eigenes Projektarchiv, von dem aus Änderungen an jedes Projektarchiv hoch- oder heruntergeladen werden können, das sie annehmen möchte.

Arbeitkopie[5]
: Der private Verzeichnisbaum eines Entwicklers, mit dem Quellcode des Projekts und möglicherweise seine Webseite oder andere Dokumente. Eine Arbeitskopie enthält auch ein paar Metadaten, die von der Versionsverwaltung benutzt werden um zu Kennzeichen, von welchem Projektarchiv sie kommt, welche "Revision" (siehe unten) der Dateien vorliegen, usw. Im allgemeinen hat jeder Entwickler seine eigenes Arbeitskopie, indem er seine Änderungen macht, prüft und anschließend als Commit an das Projektarchiv schickt.

Revision[6]
: Eine "Revision" ist für gewöhnlich eine bestimmte Version einer Datei oder einem Verzeichnis. Wenn das Projekt z.B. mit der Revision 6 der Datei D anfängt und dann jemand eine Änderung an D committed, entsteht die Revision 7 von D. Manche Systeme benutzen "Revision" auch als Bezeichnung für einen ganzen Satz an Änderungen die zusammen als Einheit committed wurden.

  Diese Begriffe haben ab und zu eine Bestimmte technische Bedeutung abhängig von der Versionsverwaltung, im Allgemeinen ist die Idee jedoch immer die gleiche: Sie ermöglichen es genau über bestimmte Zeitpunkte in der Geschichte einer Datei zu reden (wie, direkt vor und nachdem ein Fehler behoben wurde). Beispielsweise: "Ja, sie hat das in Revision 10 behoben" oder "Sie hat das in Revision 10 von foo.c behoben."

  Wenn man von einer Datei oder einer Sammlung von Dateien spricht ohne eine bestimmte Revision anzugeben, geht man im Allgemeinen von der aktuellsten Revision aus.

[5] engl. "working copy"
[6] im engl. auch *change* oder *changeset* (de. Satz von Änderungen)

Diff[7]
: Eine textuelle Representation einer Änderung. Ein Diff zeigt wie und welche Zeilen geändert wurden, sowie ein paar zusätzliche Zeilen um einen Kontext zu geben. Ein Entwickler der bereits ein wenig mit dem Code vertraut ist, kann für gewöhnlich ein Diff lesen und verstehen was die Änderung gemacht hat und sogar Fehler bemerken.

Tag[8]
: Eine Beschriftung einer bestimmen Menge an Dateien ganz bestimmter Revisionen. Tags werden üblicherweise benutzt um interessante Revisionen des Projekts zu bewahren. Für jede neue veröffentlichte Version wird z.B. ein neuer "Tag" erstellt, um später genau dieselben Dateien/Revisionen aus der Versionsverwaltung herunterladen zu können. Häufige "Tag" Bezeichnungen sind `Version_X_Y`, `Auslieferung_00456`, usw.

Branch[9]
: Eine Kopie des Projekts in der Versionsverwaltung, die aber vom Hauptzweig isoliert ist, damit Änderungen nicht das Übrige Projekt beeinflussen und umgekehrt, außer wenn Änderungen absichtlich von einer Seite zur Anderen portiert werden (siehe unten). Ein Branch kann man auch als Entwicklungszweig bezeichnen. Selbst wenn ein Projekt nicht explizit irgendwelche Zweige hat, gibt es dennoch einen sogenannten "Hauptzweig"[10]als den Zweig auf dem die Entwicklung stattfindet.

  Zweige bieten bei der Entwicklung, die Möglichkeit verschiedene Richtungen getrennt zu verfolgen. Ein Zweig kann z.B. für experimentelle Entwicklung benutzt werden, die für den Hauptzweig nicht stabil genug wären. Umgekehrt kann ein Zweig auch als Ort benutzt werden um eine neue Version zu stabil zu bekommen. Während der Entwicklung kann die reguläre Entwicklung im Hauptzweig ohne Unterbrechung weiterlaufen; währenddessen werden auf dem Zweig der neuen Version keine Änderungen mehr zugelassen, außer sie werden von einem Versionsverwalter genehmigt. Auf diese Art, muss eine neue Version die laufende Entwicklung nicht stören. Siehe „Benutze Zweige, um Engpässe zu vermeiden“später in diesem Kapitel für eine detailliertere Erörterung über Zweige.

Merge[11]
: Eine Änderung von einem Zweig in ein Anders übernehmen. Was auch portieren von Änderungen aus dem Hauptzweig in einem anderen Zweig oder umgekehrt bedeuten kann. Tatsächlich ist das sogar die häufigste Art zu mergen; man portiert selten eine Änderung zwischen zwei Zweige, die nicht beide Hauptzweige sind. Siehe „Eindeutigkeit von Informationen“ für mehr zu dieser Art zu portieren.

  "Merge" hat eine zweite, verwandte Bedeutung: Die Versionsverwaltung macht einen Merge, wenn zwei Leute die gleiche Datei bearbeitet haben, sodass die Änderungen sich nicht überlappen. Da die Änderungen nicht miteinander kollidieren, werden die Änderungen in die eigenen Kopie (mit eigenen Änderungen) übertragen, bzw. die Kopie wird aktualisiert. Das kommt sehr häufig vor, besonders in Projekten bei dem mehrere Entwickler am gleichen Code arbeiten. Wenn zwei verschiedene Änderungen *doch* überlappen, gibt es einen "Konflikt"; siehe unten.

Konflikt
: Was geschieht wenn zwei Personen gleichzeitig unterschiedliche Änderungen vornehmen, an der gleichen Stelle im Code. Jede Versionsverwaltung erkennt Konflikte automatisch, und benachrichtigt mindestens einen der Beteiligten, dass ihre Änderungen mit denen von anderen kollidieren. Der Konflikt muss dann von einem Menschen bereinigt (engl. *resolve*) und an die Versionsverwaltung übermittelt werden.

[7] kurz für *difference*(de. Unterschied)
[8] de. Etikett
[9] de. Zweig/Ast
[10] engl. "main branch" auch "main line" oder "*trunk*" de. "Stamm"
[11] im engl. auch "port" de. Zusammenführung/Portierung

Lock

(de. Schloss/Sperre) Eine Möglichkeit eine exklusive Absicht auf eine Datei oder ein Verzeichnis zu erklären. z.B.: "Ich kann gerade keine Änderungen an der Webseite machen. Es scheint das Alfred alles gesperrt hat während er die Hintergrundbilder korrigiert". Nicht jede Versionsverwaltung bieten überhaupt die Möglichkeit Dateien zu sperren, und solche die es tun erfordern nicht alle, dass sie auch benutzt wird. Das liegt daran, dass parallele, gleichzeitige Entwicklung der Normalfall ist und Sperren auf Dateien, diesem Ideal (üblicherweise) widersprechen.

Eine Versionsverwaltung die einen Lock erfordert um einen Commit zu machen, benutzt das sogenannte *lock-modify-unlock* Verfahren. Solche die es nicht erfordern, nutzen das *copy-modify-merge* Verfahren. Eine ausgezeichnete tiefgehende Erklärung und Vergleich der beiden Methoden ist auf http://svnbook.red-bean.com/svnbook-1.0/ch02s02.html zu finden. Im allgemeinen ist die copy-modify-merge Methode besser für die Open-Source-Entwicklung und jede Versionsverwaltung in diesem Buch unterstützen sie.

Wahl einer Versionsverwaltung

Zum Zeitpunkt dieses Schreibens sind die beiden verbreitetsten Systeme für Versionsverwaltung in der Welt der freien Software das *Concurrent Versions System* oder auch *CVS* (http://www.cvshome.org/) und *Subversion* (*SVN*, http://subversion.tigris.org/).

CVS gibt es schon lange. Die meisten erfahrenen Entwickler sind bereits damit vertraut, es erledigt die Aufgabe mehr oder weniger gut und da es die Norm ist, werden Sie keine lange Debatten darüber führen müssen, ob es die richtige Wahl war. CVS hat jedoch einige Nachteile. Es bietet keine einfache Möglichkeit an, Änderungen über mehrere Dateien abzufragen; es erlaubt nicht, Dateien im Projektarchiv umzubenennen oder zu kopieren (was besonders nervt, wenn Sie Ihren Code neu organisieren wollen, nachdem Sie das Projekt gestartet haben); es bietet nur dürftige Merge-Unterstützung; es kann nicht sonderlich gut mit großen oder binären Dateien umgehen; und manche Vorgänge sind bei vielen Dateien sehr langsam.

Kein Fehler von CVS ist fatal und es ist immer noch ziemlich beliebt. In den vergangenen Jahren hat das neuere Subversion an Boden gewonnen, inbesondere bei neuen Projekten.[12]. Wenn Sie ein neues Projekt anfangen, empfehle ich Ihnen Subversion.

Da ich andererseits selbst an dem Subversion-Projekt arbeite, könnte man meine Objektivität berechtigt in Frage stellen. In den letzden Jahren sind ein paar neue Versionsverwaltungssysteme erschienen. Anhang A, *Systeme zur Versionsverwaltung* listet alle mir bekannten auf. Wie diese Liste klar macht kann die Entscheidung für eine Versionsverwaltung zu einem lebenslangen Forschungsprojekt werden. Möglicherweise wird Ihnen die Entscheidung erspart bleiben weil sie von Ihrer Hosting-Seite bereits getroffen wurde. Wenn Sie sich aber für eines entscheiden müssen, fragen Sie andere Entwickler, finden Sie heraus womit Andere bereits Erfahrung haben, suchen Sie sich eines aus und bleiben Sie dabei. Jede stabile, ausgereifte Versionsverwaltung reicht aus; Sie müssen sich keine Sorgen darüber machen, dass Sie eine furchtbar schlechte Entscheidung treffen werden. Wenn Sie sich einfach nicht entscheiden können, dann nehmen Sie CVS. Es ist immer noch die Norm und wird es auch wahrscheinlich ein paar Jahre lang bleiben. Viele andere Systeme unterstützen auch die Konvertierung in eine Richtung von einem CVS Archiv, Sie können sich also später auch umentscheiden.

Nutzung einer Versionsverwaltung

Die Empfehlungen in diesem Abschnitt sind nicht auf eine bestimmte Versionsverwaltung abgestimmt und sollten in allen Systemen einfach zu implementieren sein. Für weitere Details, schlagen Sie in der Dokumentation Ihrer Versionsverwaltung nach.

[12]Siehe http://cia.vc/stats/vcs und http://subversion.tigris.org/svn-dav-securityspace-survey.html für beweise für dieses Wachstum.

Versioniere alles

Benutzen Sie die Versionsverwaltung nicht nur für den Quellcode Ihres Projekts, sondern auch die Webseite, Dokumentation, FAQ, Entwurfsskizen und alles andere, was jemand vielleicht bearbeiten möchte. Behalten Sie alles direkt neben dem Quellcode, im selben Projektarchiv. Jede Information, die sich lohnt niederzuschreiben, ist es auch Wert im Projektarchiv zu sein – also jede Information die sich ändern könnte. Sachen die sich nicht ändern, sollten archiviert und nicht versioniert werden. Eine E-Mail ändert sich beispielsweise nicht, wenn sie einmal abgeschickt wurde; deshalb würde es keinen Sinn machen sie zu versionieren (es sei denn sie wird zu einem Teil eines größeren, sich entwickelnden Dokuments).

Der Grund warum es wichtig ist alles an einem Ort zu versionieren ist, dass Personen nur eine Methode lernen müssen um Änderungen einzureichen. Oftmals, wird ein Beteiligter damit anfangen Änderungen an der Webseite oder der Dokumentation zu machen, und gehen später dazu über kleine Beiträge am Quellcode zu machen. Wenn das Projekt dasselbe System für alle Beiträge verwendet, müssen Beteiligte nur eine Methode lernen. Alles zusammen zu versionieren, bedeutet auch, dass neue Funktionen gleich zusammen mit ihrer zugehörigen Aktualisierungen an der Doku eingereicht werden können, sodass ein Zweig für den Code auch ein Zweig für die Doku mit sich bringt, usw.

Behalten Sie keine *generierte Dateien* im Projektarchiv. Sie sind nicht wirklich bearbeitbare Daten, da sie aus anderen Daten erzeugt werden. Manche Build-Systeme erzeugen beispielsweise `configure` aus der Vorlage `configure.in`. Um eine Änderung an `configure` vorzunehmen, würde man `configure.in` bearbeiten und es daraus neu erzeugen lassen; weßhalb lediglich die Vorlage `configure.in` "bearbeitbar" ist. Versionieren Sie lediglich die Vorlage – wenn Sie die erzeugten Dateien auch versionieren, wird man zwangsläufig vergessen sie neu zu erzeugen nachdem man eine Änderung an einer Vorlagen eingespielt hat. Die daraus resultierende Inkonsistenz wird endlose Verwirrung stiften.[13]

Zu der Regel, dass alle bearbeitbare Dateien im Projektarchiv sein sollten, gibt es eine Ausnahme: den Bugtracker. Eine Bug-Datenbank enthält eine Menge bearbeitbarer Daten, kann aber aus technischen Gründen diese Daten im allgemeinen nicht im Projektarchiv speichern. (Manche Tracker haben eigene primitive Funktionen zur Versionierung, die allerdings von der Versionsverwaltung des Projekts getrennt ist.)

Zugang per Browser

Das Projektarchiv eines Projekts sollte per Browser erreichbar sein. Das bedeutet nicht nur die neuste Version der einzelnen Dateien einsehen zu können, sondern auch in der Zeit zurück zu gehen und frühere Revisionen der Dateien zu sehen, die Unterschiede zwischen den verschiedenen Versionen der Dateien sehen zu können, die commit-logs bestimmter Änderungen lesen zu können, usw.

Der Browser-Zugang ist wichtig, denn er bietet ein leichtgewichtiges Portal zu den Projekt-Daten. Ist der Zugriff per Browser nicht möglich, dann würde jemand, nur um eine bestimmte Datei zu untersuchen (sagen wir, um nachzuschauen ob ein bestimmter Bugfix es in den Code geschafft hat), zuerst lokal eine Versionsverwaltungs-Software installieren müssen, was eine einfache zweiminutige Anfrage zu einer halbstündigen Aufgabe macht.

Der Browser-Zugang erlaubt auch normale URLs auf bestimmte Revisionen von Dateien oder die jeweils aktuellste Revisionen. Das kann bei technischen Diskussionen sehr nützlich sein, oder falls man jemanden auf die Dokumentation verweisen will. Man könnte zum Beispiel anstatt "Eine ausführliche Anleitungen zum Fehlermanagement enthält die Datei community-guide/index.html deiner lokalen Arbeitskopie." folgendes schreiben: "Eine ausführliche Anleitungen zum Fehlermanagement kannst du

[13] Eine andere Meinung im Bezug auf die Versionierung von `configure` Dateien, kann man in "*configure.in and version control*" bei http://versioncontrolblog.com/2007/01/08/configurein-and-version-control/ von Alexey Makhotkin nachlesen.

unter *http://subversion.apache.org/docs/community-guide/* nachlesen.", wobei die URL auf die aktuelle Version der Datei zeigt. Die URL ist überlegen: sie ist eindeutig und erübrigt die Frage, ob der Angesprochene wirklich eine aktuelle Arbeitskopie verwendet.

Manche Versionsverwaltungssysteme haben eine eingebaute Funktion um das Projektarchiv online zu durchsuchen, andere verlassen sich hierfür auf zusätzliche Software. Drei Beispiele hierfür sind *ViewCVS* (http://viewcvs.sourceforge.net/), *CVSWeb* (http://www.freebsd.org/projects/cvsweb.html), und *WebSVN* (http://websvn.tigris.org/). Ersteres funktioniert sowohl mit CVS als auch Subversion, das Zweite nur mit CVS und Letzteres nur mit Subversion.

Commit-E-Mails

Jeder Commit an das Projektarchiv sollte eine E-Mail erzeugen, die zeigt, wer den Commit gemacht hat, wann er gemacht wurde, welche Dateien und Verzeichnisse sich geändert haben und was sich an ihnen geändert hat. Die E-Mail sollte auf eine bestimmte Mailingliste gehen, eine andere als die Liste der Entwickler. Diese und andere interessierte Beteiligte sollten ermutigt werden, sich auf der Commit-Liste anzumelden, da es die effektivste Art ist sich über die Ereignisse auf Code-Ebene des Projekts auf dem Laufenden zu halten. Abgesehen von den offensichtlichen technischen Vorteilen der Überprüfung durch andere Entwickler (siehe „Code Review“), können die Commit E-Mails dazu beitragen eine Gemeinschaft aufzubauen, da sie eine gemeinsame Umgebung schaffen indem Personen auf Ereignisse (Commits) reagieren können von denen sie wissen das Andere sie auch wahrnehmen.

Wie man Commit E-Mails einrichtet, hängt von Ihrer Versionsverwaltung ab, meistens gibt es aber hierfür einen Script oder eine andere Einrichtung bei ihrer Software. Wenn Sie Schwierigkeiten bekommen es zu finden, schlagen Sie in Ihrer Dokumentation den Begriff *hooks*, insbesondere auch *post-commit hook* nach, bei CVS auch *loginfo hook* genannt. Diese Commit-Hooks sind eine allgemeine Möglichkeit, Befehle nach jedem Commit zu schalten. Der Hook wird von einem Commit ausgelöst, ihm werden alle Informationen über den Commit übergeben, mit denen er dann vordefinierte Aufgaben ausführen kann, wie z.B. eine E-Mail abschicken.

Bei fertig eingerichteten Systemen für Commit E-Mails werden Sie möglicherweise das übliche Verhalten ändern wollen:

1. Manchmal beinhalten die Commit E-Mails nicht die tatsächlichen Diffs und geben stattdessen eine URL an, bei dem man die Änderungen über das Web-Portal des Projektarchivs einsehen kann. Obwohl es gut ist eine URL anzugeben, auf der man später verweisen kann, ist es auch *sehr* wichtig, dass die Diffs in den Nachrichten enthalten sind. E-Mails zu lesen gehört schon zum Alltag der Leute, wenn also die Änderungen gleich in der E-Mail zu lesen sind, werden Entwickler sie auf der Stelle untersuchen, ohne ihre E-Mail-Anwendung verlassen zu müssen. Wenn sie erst auf eine URL klicken müssen werden es die Meisten bleiben lassen, da es eine weitere Aktion erfordert anstatt fortzusetzen was sie bereits angefangen hatten. Desweiteren geht es bei Fragen über die Änderung viel schneller einfach auf die E-Mail zu antworten und eine Bemerkung an entsprechender Stelle zu schreiben als eine Webseite zu besuchen und mühselig den Diff aus dem Webbrowser heraus in die E-Mail zu kopieren.

   (Wenn der Diff natürlich riesig ist, wie z.B. wenn eine große Menge neuer Code im Projektarchiv eingefügt wurde, macht es natürlich Sinn den Diff wegzulassen und nur die URL anzubieten. Die meisten Systeme für Commit E-Mails können diese Art der Verkürzung automatisch. Wenn Ihres es nicht kann, ist es immer noch besser, die Diffs mitzuschicken und gelegentlich mit riesigen E-Mails zu leben, als die Diffs komplett auszuschalten. Bequeme Möglichkeiten zur Überprüfung und Bewertung sind ein Eckstein der gemeinschaftlichen Entwicklung und deshalb unerlässlich.)

2. Der Reply-to-Header der Commit-E-Mails sollte auf die Mailingliste der Entwickler eingestellt sein, nicht auf die Commit-Liste. Wenn also jemand eine Commit-E-Mail durchgelesen und bewertet hat und daraufhin eine Antwort schreibt, sollte die Antwort an die Liste der Entwickler gehen auf dem

technische Angelegenheiten üblicherweise stattfinden. Es gibt hierfür ein paar Gründe. Erstens wollen Sie alle technischen Diskussionen auf einer Liste behalten, Leute erwarten nämlich, dass sie dort gehalten werden und so auch nur ein Archiv durchsucht werden muss. Zweitens könnte es interessierte Parteien geben, die nicht bei der Commit-Liste angemeldet sind. Drittens wird die Commit-Liste als Dienst verstanden um Commits zu verfolgen und nicht um Commits zu verfolgen *und* gelegentlich auch technische Diskussionen zu führen. Wer sich auf den Commit-Liste angemeldet hat, will nichts anderes als Commit-E-Mails; wenn Ihnen also anderes Material über diese Liste zugesandt wird, bricht das ein ungesprochenes Übereinkommen. Viertens schreiben Beteiligte oft Programme um die Commit-E-Mails zu lesen und zu verarbeiten (z.B. um sie auf einer Webseite anzuzeigen). Diese Programme sind auf eine konsistente Formatierung ausgelegt, nicht jedoch auf inkonsistente von Menschen geschriebene E-Mails.

Beachten Sie dass dieser Ratschlag, Reply-to umzuschreiben nicht den Empfehlungen aus „Die große "reply-to"-Debatte“ in einem früheren Abschnitt dieses Kapitels widerspricht. Es ist immer in Ordnung, wenn der *Absender* einer Nachricht Reply-to setzt. In diesem Fall ist der Absender die Versionsverwaltungs selber und es setzt Reply-to um anzudeuten, dass der angemessene Ort für Antworten die Entwickler-Liste ist und nicht die Commit-Liste.

CIA: Eine weitere Möglichkeit für Commit-Benachrichtigungen

Commit E-Mails sind nicht die einzige Möglichkeit um Nachrichten über Änderungen zu verbreiten. Vor kurzem wurde eine weitere Möglichkeit namens CIA (http://cia.navi.cx/) entwickelt. CIA fasst Commit-Statistiken zusammen und verbreitet sie in Echtzeit. Die verbreitetste Verwendung von CIA ist Commit Benachrichtigungen an IRC-Kanäle zu senden, damit die Leute dort in Echtzeit mitbekommen, wann Änderungen gemacht werden. Obwohl diese Methode nicht ganz so nützlich ist wie Commit E-Mails, da Beteiligte unter Umständen nicht während einem Commit im IRC anwesend sind, hat diese Methode trotzdem einen immensen *sozialen* Wert. Leute bekommen das Gefühl ein Teil von etwas lebendigem und aktivem zu sein und der Fortschritt vor ihren Augen geschieht.

Es funktioniert indem Sie die CIA Anwendung automatisch nach jedem Commit aufrufen. Es formatiert die Information über den Commit in eine XML Nachricht, und sendet es an einen zentralen Server (typischerweise `cia.navi.cx`). Der Commit-Server verteilt diese Information dann an andere Foren.

CIA kann auch konfiguriert werden um einen RSS [http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html] feed zu erzeugen. Weiteres dazu können Sie in der Dokumentation bei http://cia.navi.cx/ nachlesen.

Sie können sich von CIA einen Eindruck verschaffen, indem Sie sich mit Ihrem IRC-Client bei `#commits` auf dem Server `irc.freenode.net` einloggen.

Benutze Zweige, um Engpässe zu vermeiden

Laien im Umgang mit Versionsverwaltung scheuen sich manchmal vorm Verzweigen und Zusammenführen. Das ist wahrscheinlich ein Nebeneffekt des Erfolgs von CVS: Die Schnittstellen von CVS um Zweige zu machen und wieder zusammenzuführen sind nicht ganz eingängig, weshalb viele sich angeeignet haben diese Operationen komplett zu vermeiden.

Wenn Sie zu diesen Leuten gehören, nehmen Sie sich vor alle Ängste die Sie haben mögen zu besiegen und nehmen Sie sich die Zeit um zu lernen, wie man Zweige macht und wieder zusammenführt. Es sind keine schwierigen Vorgänge, wenn man sich erst einmal daran gewöhnt hat und sie werden mit der Zeit immer wichtiger, sowie ein Projekt immer mehr Entwickler aufnimmt.

Zweige sind wichtig da sie eine knappe Ressource – den Platz im Code des Projekts – im Überfluss bereitstellt. Normalerweise arbeiten alle Entwickler im gleichen Sandkasten und bauen an der gleichen Burg. Wenn jemand eine neue Zugbrücke anbauen will, jedoch nicht alle überzeugen kann, dass es eine Verbesserung wäre, ermöglicht ein Zweig mit ihr in einer eigenen isolierte Ecke zu experimentieren. Wenn es funktioniert kann sie die anderen Entwickler einladen, sich das Ergebnis anzuschauen. Wenn alle zustimmen, dass das Ergebnis gut ist können sie mittels der Versionsverwaltung die Zugbrücke aus dem Burgflügel in die Haupt-Burg übernehmen (einen "Merge" machen).

Es ist einfach zu erkennen, wie diese Fähigkeit die gemeinschaftliche Entwicklung fördert. Menschen brauchen die Freiheit Neues auszuprobieren ohne das Gefühl zu bekommen andere bei der Arbeit zu stören. Gleichermaßen gibt es Zeiten an denen es wichtig ist, bestimmten Code von der alltäglichen Entwicklung zu isolieren, um einen Fehler zu beheben oder eine neue Version stabil zu machen (siehe „Stabilisierung einer neuen Version“ und „Wartung mehrerer Versionszweige“ im Kapitel Kapitel 7, *Paket-Erstellung, Veröffentlichung, und tägliche Entwicklung*) ohne sich über ein bewegliches Ziel Gedanken zu machen.

Seien Sie großzügig bei der Nutzung von Zweigen und ermutigen Sie andere, ebenso zu verfahren. Achten Sie aber auch darauf, dass jeder Zweig nur so lange aktiv bleibt wie nötig. Jeder aktive Zweig beansprucht die Aufmerksamkeit der Gemeinschaft ein klein wenig. Selbst diejenigen die nicht an einem Zweig arbeiten, behalten noch einen groben Überblick über die Ereignisse darin. Diese Aufmerksamkeit ist natürlich wünschenswert und Commit-Nachrichten sollten auch für Zweige eingeschaltet sein, genau wie für jeden anderen Commit. Zweige sollten jedoch nicht zu einer Methode werden, die Gemeinschaft zu spalten. Mit seltenen Ausnahmen sollte jeder Zweig das Ziel haben, irgendwann wieder zurück in den Hauptzweig überzugehen und zu verschwinden.

Eindeutigkeit von Informationen

Merges haben eine wichtige Konsequenz: Dieselbe Änderung sollte niemals doppelt committet werden. D.h. eine bestimmte Änderung sollte durch das Versionsverwaltungssystem nur genau einmal übernommen werden. Die Revision (oder die Gruppe von Revisionen) in dem diese Änderung eingebracht wurde, ist von da an seine eindeutige Kennung. Wenn sie auf weitere Zweige angewendet werden soll, sollte sie von ihrem ursprünglichen Eintrittspunkt aus in diese anderen Ziele portiert werden – man sollte also nicht mehrere textgleiche Änderungen committen, was zwar die gleiche Wirkung auf den Code haben würde, aber eine genaue Buchführung unmöglich machen würden.

Die praktische Auswirkung dieser Empfehlung sind unterschiedlich, je nach Versionsverwaltungssystem. Manche Systeme erfassen einen Merge als besonderes Ereignis, grundsätzlich unterschiedlich zu einem normalen Commit, mit eigenen Metadaten. Bei anderen wird das Ergebnis eines Merges genau wie jeder andere Commit übernommen, in solchen Fällen sollte man dafür sorgen, dass sich ein "Merge-Commit" im Commit-Log klar von einem "Änderungs-Commit" unterscheidet. In dem Commit-Log von einem Merge sollte nicht die Nachricht der ursprünglichen Änderung wiederholt werden. Stattdessen sollten Sie lediglich angeben, dass es sich um einen Merge handelt, und die Revisionsnummer der Ursprünglichen Änderung angeben und höchstens einen Satz, um die Auswirkungen der Änderung zusammenzufassen. Wenn jemand den kompletten Commit-Log sehen will, kann er die ursprünglichen Revisionen aufsuchen.

Es ist wichtig zu vermeiden, die Commit-Logs zu wiederholen, da Log-Nachrichten manchmal nach dem Commit geändert werden. Wenn die Log- Nachricht einer Änderung bei jedem Merge wiederholt würde, blieben selbst bei einer Korrektur der ursprünglichen Nachricht alle Kopien unverändert – was später unweigerlich zu Verwirrung führen kann.

Dasselbe Prinzip gilt beim Rückgängigmachen einer Änderung. Wenn eine Änderung wieder vom Code entfernt wird, sollte sein Commit-Log lediglich festhalten, dass die Änderung einer bestimmten Revision rückgängig gemacht wird und *nicht* eine Beschreibung der tatsächlichen Änderungen am Code, da die-

se Information aus der ursprünglichen Änderungen und sein Log ersichtlich ist. Selbstverständlich sollte der Log auch den Grund für die Entfernung nennen, jedoch nichts aus dem ursprünglichen Log wiederholen. Wenn möglich gehen Sie zurück und Ändern Sie den Log der ursprünglichen Änderung und weisen Sie darauf hin, dass sie zurückgedreht wurde.

Die vorhergehenden Passagen implizieren die Verwendenung einer konsistenten und gleichbleibende Syntax um auf Revisionen Bezug zu nehmen. Das ist nicht nur in den Logs hilfreich, sondern auch in E-Mails, dem Bugtracker und anderswo. Wenn Sie CVS verwenden, schlage ich "`path/to/file/in/project/tree:REV`" als Format vor, wobei REV eine CVS Revisionsnummer wie "1.76" darstellt. Wenn Sie Subversion verwenden, ist die übliche Syntax für die Revision 1729 "r1729" (Datei-Pfade sind bei Subversion nicht nötig, da es globale Revisionsnummern verwendet). Andere Systemen haben meistens ihre eigene übliche Syntax um Änderungen zu kenzeichnen. Konsistente Bezeichnungen erleichtern die Buchhaltung eines Projekts ungemein (was wir in Kapitel 6, *Kommunikation* und Kapitel 7, *Paket-Erstellung, Veröffentlichung, und tägliche Entwicklung*) sehen werden und da ein großer Teil der Buchhaltung von Freiwilligen erledigt wird, muss es so einfach wie Möglich sein.

Siehe auch „Neue Versionen und tägliche Entwicklung“ im Kapitel Kapitel 7, *Paket-Erstellung, Veröffentlichung, und tägliche Entwicklung* .

Autorisierung

Die meistene Versionsverwaltungen bieten Funktionen, um bestimmten Personen Schreibzugriff auf einzelne Bereiche des Projektarchivs zu erlauben oder zu verwehren. Nach dem Grundsatz, dass jemand, sobald man ihm einen Hammer in die Hand gibt, anfangen wird, überall Nägel zu sehen, wird diese Funktion von vielen Projekten benutzt, um den Beteiligten lediglich Schreibzugriff auf Bereiche zu geben mit denen sie sich offensichtlich auskennen und so sichergestellt ist, dass sie nirgendwo sonst Schreibzugriff haben. (Siehe „Committer“ im Kapitel Kapitel 8, *Leitung von Freiwilligen* indem beschrieben ist, wie Projekte entscheiden wer wo Änderungen machen kann.)

Mit einer solch scharfen Kontrolle können Sie wahrscheinlich kaum Schaden anrichten, eine lockerere Haltung ist aber auch in Ordnung. Manche Projekte benutzen einfach ein Vertrauenssystem: Wenn einer Person Commit-Zugriff gewährt wird, wenn auch nur für einen Teilbereich des Projektarchivs, erhält diese in Wirklichkeit den Schlüssel, um überall im Projekt zu Änderungen vorzunehmen. Sie wird einfach darum gebeten, sich auf ihren Bereich zu beschränken. Denken Sie daran, dass hierin keine echte Gefahr besteht: In einem aktiven Projekt wird sowieso jeder Commit überprüft. Wenn jemand in einem Bereich etwas ändert, in dem er nichts zu suchen hat, werden es andere bemerken und etwas sagen. Wenn eine Änderung rückgängig gemacht werden muss, ist das auch kein Problem – es ist sowieso alles in der Versionsverwaltung, also kann man es einfach rückgängig machen.

Die Sache locker anzugehen hat mehrere Vorteile. Erstens gibt es keinen weiteren Aufwand um Entwickler zusätzliche Rechte einzuräumen, sobald sie sich auf andere Bereiche ausweiten (was meistens irgendwann passiert, wenn sie länger beim Projekt bleiben). Sobald die Entscheidung getroffen wurde, kann die Person gleich anfangen Änderungen im neuen Bereich zu machen.

Zweitens kann die Erweiterung viel feiner granuliert werden. Allgemein wird ein Commit-Berechtigter im Bereich X der auch im Bereich Y arbeiten will, anfangen Patches für Y einzureichen und darum bitten, dass sie überprüft werden. Wenn jemand der bereits Zugriff auf dem Bereich Y hat solch einen Patch sieht und ihm zustimmt, können sie dem Autor einfach sagen, dass sie die Änderung gleich selber einspielen können(natürlich mit Namensangabe vom Überprüfenden bzw, Zustimmenden im Commit-Log). Auf diese Art stammt der Commit von dem, der ihn auch geschrieben hat, was sowohl aus Sicht der Informationsverwaltung als auch der Anerkennung vorzuziehen ist.

Schließlich, und das ist vielleicht das Wichtigste, regt ein System, das auf Ehre basiert, eine Atmosphäre des Vertrauens und des gegenseitigen Respekts an. Jemandem Commit-Zugriff auf einem Teilbereich zu geben, ist eine Aussage über ihre fachliche Vorbereitung – es sagt: "Wir sehen, dass du die Kennt-

nisse hast, um auf einem Gebiet Änderungen zu machen, also leg los". Strikte Autorisierung aufzuerlegen, sagt aber: "Wir behaupten nicht nur, dass deine Kenntnisse begrenzt sind, wir sind auch ein wenig skeptisch im Bezug auf deine *Absichten*". Das ist keine Behauptung, die Sie in den Raum stellen wollen, wenn es sich vermeiden lässt. Jemanden an dem Projekt als commit-berechtigt zu beteiligen, ist eine Gelegenheit, ihn in einen Kreis vertrauter Personen aufzunehmen. Das erreicht man am besten, indem man ihm mehr Macht gibt als er letztlich braucht, und ihn darüber informiert, dass es an ihm ist, sich innerhalb der genannten Grenzen zu bewegen.

Das Subversion-Projekt arbeitet schon seit vier Jahren nach dem Ehren-Prinzip, mit 33 voll -und 43 teil-berechtigten Entwicklern zum Zeitpunkt dieses Schreibens. Der einzige Unterschied, den das System macht, ist zwischen Commit-Berechtigten und nicht Commit-Berechtigten; weitere Unterteilungen bestehen allein auf zwischenmenschlicher Ebene. Dennoch hatte man dort nie Probleme mit den Grenzen der Berechtigungen. Es gab ein oder zwei Missverständnisse über das Ausmaß der Commit-Berechtigungen, die jedoch immer schnell und freundlich gelöst wurden.

Offensichtlich muss man sich auf strikte Autorisierung verlassen können, wenn Selbstkontrolle nicht sinnvoll ist. Solche Situationen sind jedoch selten. Selbst bei Unmengen Code und hunderten oder tausenden Entwicklern, sollte ein Commit zu jedem beliebigen Modul von denen zuständigen Personen überprüft werden, die auch erkennen können ob jemand an eine Stelle etwas geändert hat die er nicht sollte. Wenn Änderungen *nicht* regelmäßig überprüft werden, dann hat das Projekt ohnehin größere Probleme als die Commit-Berechtigung.

Insgesamt sollte man nicht allzuviel Zeit damit verbringen, die Berechtigungen der Versionsverwaltung auszutüfteln, es sei denn Sie haben einen ganz bestimmten Grund dazu. Es wird für meistens wenig handfesten Nutzen bringen und es hat seine Vorteile sich stattdessen auf menschliche Kontrolle zu verlassen.

Natürlich sollte man nichts hiervon so auffassen, dass die Beschränkungen an und für sich unwichtig sind. Es wäre schlecht für das Projekt, Teilnehmer anzuregen Bereiche zu ändern, für die sie nicht qualifiziert sind. Desweiteren geben viele Projekte dem vollen (uneingeschränkten) Zugriff auf das Projektarchiv einen besonderen Status: Es impliziert, dass der Teilnehmer das Recht hat, über Fragen die das ganze Projekt betreffen abzustimmen. Dieser politische Aspekt der Commit-Berechtigung wird weiter in „Wahlberechtigung“ im Kapitel Kapitel 4, *Soziale und politische Infrastruktur* behandelt.

Bugtracker

Bugtracking ist ein weites Feld, von dem viele Aspekte im Verlauf dieses Buchs besprochen werden. Ich werde versuchen, mich hier auf Einrichtung und technische Erwägungen zu konzentrieren. Zuvor müssen wir jedoch eine politische Frage stellen: Welche Informationen sollen im Bugtracker gehalten werden?

Der Begriff *Bugtracker* ist irreführend. Diese Systeme werden häufig auch verwendet, um Anfragen für neue Funktionen, einmalige Aufgaben, unaufgeforderte Patches – im Prinzip alles zu verfolgen (engl. to track), was einen eindeutigen Anfangs- und Endzustand sowie optionale Übergangszuständen hat und über dessen Lebenszyklus Informationen anfallen. Aus diesem Grund haben Bugtracker auch häufig Namen[14] wie *Issue Tracker*, *Defect Tracker*, *Artifact Tracker*, *Request Tracker*, *Trouble Ticket System*, usw. Eine Liste verfügbarer Software finden Sie im Anhang Anhang B, *Freie Bugtracker*.

In diesem Buch werde ich weiterhin "Bugtracker" für Software benutzen, die jegliche der vorher erwähnten Angelegenheiten verfolgt, da es weithin so bezeichnet wird, ein einzelnes Element in der Datenbank des Bugtracker werde ich jedoch als *Ticket* (im engl. auch "issue") bezeichnen. So können wir zwischen Verhalten oder Fehlverhalten unterscheiden, die ein Nutzer beobachtet hat, (also einen

[14]Im Deutschen Raum sind die Begriffe *Bugtracker* und *Ticket-System* am weitesten verbreitet.

Fehler) und die *Erfassung* seiner Entdeckung, Diagnose und Lösung im Tracker. Behalten Sie im Hinterkopf, dass auch wenn es bei den meisten Tickets um Fehler handelt, sie auch benutzt werden können um andere Aufgaben zu verfolgen.

Der klassische Verlauf eines Tickets sieht wie folgt aus:

1. Jemand reicht das Ticket ein. Sie machen eine Zusammenfassung, eine anfängliche Beschreibung (einschließlich dessen wie man den Fehler reproduziert, falls anwendbar, siehe „Behandeln Sie jeden Nutzer wie einen möglichen Freiwilligen“ im Kapitel Kapitel 8, *Leitung von Freiwilligen* in der beschrieben wird, wie man gute Bug-Meldungen fördert ermutigen kann), und sonstige Informationen. die der Tracker verlangt. Die Person die den Fehler meldet, kann dem Projekt völlig unbekannt sein – Bug-Meldungen und Anfragen für Funktion können genau so aus der Nutzer-Gemeinschaft kommen, wie von den Entwicklern.

   Sobald das Ticket gemeldet wurde sagt man, dass es *offen* ist. Da bisher nichts unternommen wurde, kennzeichnen manche Tracker diese auch als *unbestätigt* (engl. "unverified") oder *nicht angefangen* (engl. "unstarted"). Er wurde noch niemandem zugewiesen; oder, in manchen Systemen, wird es einem fiktiven Benutzer zugewiesen um anzudeuten, dass es nicht wirklich jemandem zugewiesen wurde. Zu diesem Zeitpunkt steht es in einer Warteschlange: Das Ticket wurde erfasst, ist jedoch nicht im Bewusstsein des Projekts aufgenommen.

2. Andere lesen den Ticket, sie fügen Kommentare hinzu, und bitten vielleicht dem Meldenden einige Punkte zu klären.

3. Der Fehler wird *reproduziert*. Dieser Augenblick mag der Wichtigste in seinem Lebenszyklus sein. Auch wenn der Bug noch nicht behoben wurde ist die Tatsache, dass jemand außer dem der ihn gemeldet hat, es geschafft hat selbst den Fehler zu finden. Dies beweist, dass der Bug valide ist und bestätigt nicht zuletzt den Berichtenden, dass sie durch die Meldung eines echten Fehlers etwas zum Projekt beigetragen haben.

4. Der Fehler wird *untersucht*: seine Ursache wird identifiziert und wenn möglich, wird der nötige Aufwand geschätzt um ihn zu beheben. Stellen Sie sicher, dass diese Sachen in dem Ticket erfasst werden; wenn die Person die den Bug untersucht hat, plötzlich eine längere Zeit vom Projekt wegtreten muss (was bei freiwilligen Entwicklern häufig passieren kann), sollte jemand anders in der Lage sein, seine Arbeit wieder aufzunehmen.

   In dieser Phase oder manchmal schon vorher, kann ein Entwickler das Ticket in Besitz nehmen, und es sich selber *zuweisen* (In „Unterscheiden Sie eindeutig zwischen Anfrage und Anweisung“ im Kapitel Kapitel 8, *Leitung von Freiwilligen* wird der diese Zuweisung genauer untersucht. Die *Priorität* kann in dieser Phase auch bestimmt werden. Wenn er z.B. derart schwerwiegend ist, dass er die nächste Version verzögern würde, muss diese Tatsache frühzeitig erkannt werden und der Tracker sollte eine Möglichkeit bieten das zu erfassen.

5. Es wird geplant wann das Ticket aufgelöst werden soll, wobei dabei nicht unbedingt ein bestimmtes Datum gemeint ist an dem es behoben sein soll. Manchmal bedeutet es einfach ein Entscheiden bis zu welcher Version (nicht unbedingt die Nächste) der Bug behoben sein soll, oder dass er keine bestimmte Version blockieren muss. Wenn der Bug schnell behoben werden kann, ist die Planung wahrscheinlich überflüssig.

6. Der Bug wird behoben (oder die Aufgabe wird erledigt, oder der Patch angewandt, oder was auch immer). Die Änderung die ihn beheben sollten in einem Kommentar des Tickets protokolliert werden, worauf er *geschlossen* wird und/oder als *gelöst* markiert wird.

Dieser Lebenszyklus variiert häufig. Manchmal wird ein Ticket frühzeitig nach seiner Meldung geschlossen, da sich herausstellt, dass es sich nicht um einen Fehler handelt, sondern um ein Missver-

ständnis seitens des Anwenders. Sowie ein Projekt immer mehr Benutzer aufnimmt, werden immer mehr dieser ungültigen Tickets aufkommen und Entwickler werden sie zunehmend gereizt schließen. Versuchen Sie der letzteren Neigung entgegenzuwirken. Sie hilft keinem, da der einzelne Nutzer in jedem Einzelfall nicht für alle vorhergehenden ungültigen Tickets verantwortlich ist; die statistische Zunahmen ist lediglich für die Entwickler sichtbar, nicht für die Nutzer. (In „Vor-Filterung des Bugtrackers“ später in diesem Kapitel, werden wir die Methoden untersuchen, um die Anzahl der ungültigen Tickets zu verringern). Wenn verschiedene Nutzer immer wieder dasselbe Missverständnis haben, kann das ein Hinweis sein, dass ein bestimmter Bereich der Software überdacht werden muss. Diese Muster können am einfachsten von einem Ticket-Verwalter bemerkt werden, der die Bug-Datenbank überwacht; siehe „Ticketverwalter“ im Kapitel Kapitel 8, *Leitung von Freiwilligen*.

Eine weitere häufige Abweichung zu diesem Lebenszyklus ist, dass das Ticket als *Duplikat* gleich nach dem ersten Schritt geschlossen wird. Ein Duplikat erscheint wenn jemand ein Ticket meldet der dem Projekt bereits bekannt ist. Duplikate beschränken sich nicht auf offene Tickets: Es ist auch möglich, dass ein Bug wiederkehrt, nachdem er behoben wurde (auch als *Regression* bekannt). In solchen Fällen öffnet man Vorzugsweise wieder das ursprüngliche Ticket und alle neuen Meldungen werden als Duplikate des Originals geschlossen. Der Bugtracker sollte diese Beziehung in beiden Richtungen verfolgen können, um Informationen wie man den Bug reproduziert dem ursprünglichen Ticket verfügbar zu machen und umgekehrt.

Eine dritte Variante ist, dass Entwickler ein Ticket schließen, in der Annahme, der Fehler wurde bereits behoben, was der Meldende allerdings abweist und es erneut öffnet. Meistens geschieht das, wenn die Entwickler nicht die nötige Umgebung haben, um den Fehler zu reproduzieren oder nicht mit genau derselben Anleitung zur Reproduktion beim Testen genutzt haben wie der Meldende.

Abgesehen von diesen Abweichungen, kann es andere kleine Details im Lebenszyklus geben, die sich abhängig von der Software des Bugtrackers unterscheiden. Die grundsätzliche Form ist jedoch bei allen die gleiche, und obwohl der Lebenszyklus nicht eigen zu Open-Source-Software ist, hat es Auswirkungen darauf wie Open-Source-Projekte ihre Bugtracker benutzen.

Wie der erste Schritt andeutet, bietet ein Bugtracker der Öffentlichkeit genau so sehr ein Bild des Projekts wie seine Mailinglisten oder seine Webseite. Jeder kann ein Ticket aufmachen, jeder kann sich ein Ticket anschauen und jeder kann die Liste der offenen Tickets anschauen. Daraus folgt, dass Sie niemals wissen können, wieviele Leute darauf warten, Fortschritte für ein bestimmtes Ticket zu sehen. Auch wenn die Größe und Erfahrung der Entwicklergemeinschaft die Geschwindigkeit einschränken kann, mit der Tickets abgearbeitet werden, sollte das Projekt zumindest versuchen, jedes Ticket gleich nach seiner Meldung zu bestätigen. Selbst wenn es eine Weile lang liegen bleibt, ermutigt eine Reaktion dem Melder gegenüber, sich weiterhin an seiner Lösung zu beteiligen, da er das Gefühl bekommt, dass ein Mensch sich seiner Mühe bewusst ist (bedenken Sie, dass ein Ticket aufzumachen für gewöhnlich einen größeren Aufwand bedeutet, als sagen wir eine E-Mail zu schreiben). Desweiteren tritt das Ticket, sobald es von einem Entwickler bemerkt wird, in das Bewusstsein des Projekts, womit gemeint ist, dass dieser Entwickler darauf achten kann, ob das Ticket an anderer Stelle irgendwo auftaucht, mit anderen Entwicklern darüber reden kann, usw.

Die Notwendigkeit zeitnaher Reaktionen, impliziert zweierlei:

- Der Tracker muss mit einer Mailingliste verbunden sein, damit jede Änderung an einem Ticket, inklusive seine erste Meldung, eine E-Mail erzeugt die beschreibt was passiert ist. Diese Liste ist normalerweise eine andere als die gewöhnliche Entwickler-Liste, da nicht alle Entwickler automatisierte Bug-Mails empfangen wollen, der Reply-to-Header sollte aber (genau wie bei Commit-Mails) auf die Entwickler-Liste weisen.

- Das Formular um Tickets zu melden sollte die E-Mail-Adresse der Berichterstatter erfassen, damit sie für weitere Informationen erreicht werden können. (Die E-Mail-Adresse sollte jedoch nicht *erzwun-*

gen werden, da manche Leute es vorziehen, Fehler anonym zu melden. Siehe „Anonymität und Beteiligung“ später in diesem Kapitel in dem die Bedeutung von Anonymität behandelt wird.)

Interaktion mit Mailinglisten

Sorgen Sie dafür, dass der Bugtracker nicht zu einem Diskussionsforum wird. Obwohl die menschliche Mitwirkung im Bugtracker wichtig ist, ist er nicht wirklich für Diskussionen geeignet. Betrachten Sie ihn eher als Archiv, eine Möglichkeit Tatsachen und Verweise auf andere Diskussionen zu organisieren, die hauptsächlich auf Mailinglisten stattfinden.

Es gibt Zwei Gründe diese Unterscheidung zu machen. Erstens ist die Bedienung des Bugtrackers umständlicher als die einer Mailingliste (oder, wo wir gerade dabei sind, IRC oder andere Echtzeit-Foren). Das liegt nicht an der schlechten Benutzeroberfläche im Bugtracker, sondern an seiner Ausrichtung auf die Erfassung und Darstellung getrennter Zustände und nicht auf offene Diskussionen. Zweitens beobachtet nicht jeder den Bugtracker, der auch an der Diskussion eines Tickets beteiligt sein sollte. Ein Teil guter Ticker-Verwaltung (siehe „Teilen sie sowohl Verwaltungsaufgaben als auch technische Aufgaben“ im Kapitel Kapitel 8, *Leitung von Freiwilligen*) besteht darin sicherzustellen, dass jedes Ticket eher die Aufmerksamkeit der richtigen Leute erhält, als dass jeder Entwickler über jedes Ticket Beischeid wissen muss. In „Keine Unterhaltungen im Bugtracker“ im Kapitel Kapitel 6, *Kommunikation*, werden wir Wege untersuchen, Benutzer davon abzuhalten, versehentlich Diskussionen von den ihnen angemessenen Foren auf den Bugtracker auszulagern.

Manche Bugtracker können Mailinglisten überwachen und automatisch alle E-Mails protokollieren, die sich um ein bekanntes Ticket drehen. Typischerweise erkennen sie das anhand der Identifikationsnummer des Tickets in der Betreffzeile der E-Mail, als Teil einer bestimmten Zeichenfolge; Entwickler lernen diese Zeichenfolgen in ihren E-Mails zu benutzen, um die Aufmerksamkeit des Bugtrackers anzulocken. Der Tracker kann entweder die E-Mail als ganzes speichern oder (besser noch) einen Link zu dem Archiv der Liste. So oder so ist diese Funktion sehr nützlich; wenn Ihr Tracker sie hat, sollten Sie sie unbedingt einschalten und Teilnehmer erinnern sie zu nutzen.

Vor-Filterung des Bugtrackers

Die meisten Bugtracker leiden irgendwann an demselben Problem: Eine erstickende Anzahl doppelter oder ungültiger Einträge die mit guter Absicht, aber von unerfahrenen oder schlecht informierten Nutzern gemeldet werden. Der erste Schritt dieser Entwicklung entgegenzuwirken ist üblicherweise, einen deutlichen Hinweis auf der Hauptseite des Bugtrackers anzubringen, der erklärt woran man erkennen kann, ob ein Bug wirklich ein Bug ist, wie man nach bereits gemeldete Fehler suchen kann und letztendlich, wie man seine Meldung effektiv gestaltet, wenn man immer noch der Meinung ist, dass sie einen neuen Bug beinhaltet.

Der Geräuschpegel sollte danach eine Weile reduziert sein, sowie die Anzahl der Nutzer zunimmt, wird das Problem jedoch wiederkehren. Kein einzelner Nutzer ist daran Schuld. Jeder versucht nur zum Wohl des Projekts beizutragen und auch wenn ihre erste Meldung nicht hilfreich ist, sollten Sie dennoch dazu ermutigen sich weiterhin zu beteiligen und zukünftig bessere Tickets zu schreiben. In der Zwischenzeit muss das Projekt die Ticket-Datenbank so frei von Müll halten wie möglich.

Die zwei größten Abhilfen sind: Sicherzustellen, dass Leute den Bugtracker beobachten, die genügend wissen um ungültige oder doppelte Tickets gleich nach ihrer Meldung zu schließen und von Nutzern erfordern (oder nachdrücklich dazu anregen) ihre Bugs von anderen bestätigen zu lassen bevor sie eine Meldung im Tracker eintragen.

Die erste Methode scheint universelle Anwendung zu finden. Selbst Projekte mit riesigen Ticket-Datenbanken (wie der Debian-Bugtracker bei http://bugs.debian.org/, mit 315,929 Tickets zum Zeitpunkt

dieses Schreibens) organisieren sich so, dass *irgendjemand* jedes eintreffende Ticket sieht. Es können verschiedene Personen sein, abhängig von der Kategorie des Tickets. Das Debian-Projekt ist z.B. eine Sammlung verschiedener Software-Pakete, also leitet Debian automatisch jedes Ticket an die entsprechend Zuständigen für das Paket. Natürlich kann es manchmal vorkommen, dass Nutzer ein Ticket falsch einordnen mit dem Ergebnis, dass das Ticket zunächst an die falsche Person geleitet wird die es dann möglicherweise wieder umleiten muss. Wichtig dabei ist, dass diese Last trotzdem verteilt wird – ob der Nutzer beim Ausfüllen des Formulars richtig oder falsch rät, die Beobachtung der Tickets sollte dennoch mehr oder weniger gleichmäßig auf die Entwickler aufgeteilt sein, damit jedes Ticket eine zeitige Antwort erhalten kann.

Die zweite Technik ist weniger verbreitet, wahrscheinlich da sie sich schwerer automatisieren lässt. Der Grundgedanke ist jedem Ticket einen "Buddy" zuzuordnen. Wenn ein Nutzer denkt er hat ein Problem gefunden, wird er darum gebeten, es auf einer der Mailinglisten oder im IRC zu beschreiben und sich von jemandem bestätigen zu lassen, dass es sich auch wirklich um einen Bug handelt. Ein zweites Augenpaar frühzeitig einzubeziehen kann viele störende Meldungen verhindern. Manchmal kann die zweite Partei erkennen, dass das Verhalten kein Fehler ist, oder dass er in einer neuen Version behoben wurde. Sie kann auch mit den Symptomen aus einem früheren Ticket vertraut sein und einen doppelten Eintrag verhindern, indem sie den Nutzer auf das ältere Ticket hinweist. Oftmals reicht es auch den Nutzer zu fragen, "Haben Sie im Bugtracker geschaut ob er bereits gemeldet wurde?" Viele denken einfach nicht daran, haben jedoch kein Problem es zu tun wenn sie wissen, dass jemand es von ihnen *erwartet*.

Das Buddy-System kann die Ticket-Datenbank wirklich sauber halten, hat aber auch einige Nachteile. Viele machen trotzdem alleine Meldungen, entweder weil sie die Anweisungen, sich für neue Tickets einen Buddy zu suchen, nicht sehen oder ignorieren. Von daher ist es immer noch nötig, die Ticket-Datenbank von Freiwilligen überwachen zu lassen. Desweiteren ist es nicht gerechtfertigt, sie für ihre Ignoranz gegenüber den Richtlinien allzusehr zurechtzuweisen, da die meisten die ihre erste Meldung machen, nicht verstehen wie schwer es ist, die Ticket-Datenbank in Stand zu halten. Die Freiwilligen müssen deshalb wachsam sein und dennoch Zurückhaltung üben wenn sie Tickets ohne einen Buddy wieder an seinen Autor zurückweisen. Das Ziel ist jedem beizubringen, dass er zukünftig das Buddy-System verwenden soll, um eine wachsende Gemeinschaft entstehen zu lassen, die das Filter-System für die Tickets verstehen. Bei der Sichtung eines Tickets ohne einen Buddy ist dies das idealisierte Vorgehen:

1. Antworten Sie sofort auf das Ticket, bedanken Sie sich bei dem Nutzer für die Meldung, weisen Sie dabei aber auf die Buddy-Richtlinien (die natürlich auf der Webseite deutlich dargestellt sein sollten).

2. Falls das Ticket eindeutig gültig und kein Duplikat ist, bestätigen Sie es und und starten den normalen Lebenszyklus. So ist der Berichterstatter über die Zuordnung informiert und es gibt keinen Grund die investierte Arbeit zu verschwenden, indem man einen gültigen Bug wegen eines Formfehlers schließt.

3. Wenn andererseits das Ticket nicht klar berechtigt ist, schließen Sie es und bitten Sie den Autor darum, ihn wiederzueröffnen, sobald er eine Buddy-Bestätigung bekommt, dann jedoch zusammen mit einem Verweis auf den Thread der Mailingliste, der die Bestätigung enthält (z.B. per URL in das Archiv der Mailingliste).

Denken Sie daran, obwohl dieses System mit der Zeit das Signal-/Rauschverhältnis in der Ticket-Datenbank verbessern wird, es niemals alle Falschmeldungen unterbinden kann. Der einzige Weg Falschmeldungen komplett zu verhindern, ist den Bugtracker komplett für alle außer die Entwickler abzuschalten – eine Medizin die meistens schlimmer ist als die Krankheit. Es ist besser sich damit abzufinden, dass die Entfernung ungültiger Tickets immer ein Teil der üblichen Wartungsarbeiten am Projekt bleiben wird und so viele Leute wie möglich dazu zu überreden, dabei zu helfen.

Siehe auch „Ticketverwalter“ im Kapitel Kapitel 8, *Leitung von Freiwilligen*.

IRC / Echtzeit-Nachrichtendienste

Viele Projekte bieten Chat-Foren über *Internet Relay Chat* (*IRC*) an, in denen Nutzer und Entwickler einander Fragen stellen können und sofort Antworten erhalten können. Auch wenn Sie einen IRC-Server über Ihre eigene Webseite betreiben *können*, ist es im allgemeinen den Aufwand nicht wert. Machen Sie statt dessen was alle anderen auch machen: Betreiben Sie Ihre *IRC-Kanäle* auf Freenode (http://freenode.net/). Freenode bietet Ihnen die nötige Kontrolle um die IRC-Kanäle für Ihr Projekt zu verwalten[15], da Sie Ihnen die nicht unwesentliche Mühe ersparen, einen eigenen IRC-Server zu betreiben.

Das erste was Sie tun müssen, ist einen Namen für den Kanal auszusuchen. Die offensichtlichste Wahl ist der Namen Ihres Projekts – wenn dieser bei Freenode verfügbar ist, nehmen Sie ihn. Wenn nicht, wählen Sie etwas möglichst ähnliches und leicht zu merkendes. Bewerben Sie den Kanal auf Ihrer Webseite, damit Besucher mit kurzen Fragen hat ihn gleich finden. Folgender Hinweis erscheint z.B. markant in einem Kasten oben auf der Hauptseite von Subversion:

> *Wenn Sie Subversion nutzen, empfehlen wir Ihnen die* `users@subversion.tigris.org` *Mailingliste zu abonnieren und das Subversion-Buch [http://svnbook.red-bean.com/] sowie die FAQ [http://subversion.tigris.org/faq.html] zu lesen. Sie können auch Fragen im IRC stellen auf* `irc.freenode.net` *im Kanal* `#svn`.

Manche Projekte haben mehrere Kanäle, einen für jedes Unterthema. Einen z.B. für Probleme bei der Installation, einen weiteren für Fragen der Nutzer, noch einen für Diskussionen der Entwickler, usw. („Handhabung von Wachstum“ im Kapitel Kapitel 6, *Kommunikation* behandelt wie man die verschieden Kanäle aufteilt). Wenn Ihr Projekt noch jung ist, sollte es nur einen Kanal geben, indem alle zusammen miteinander reden. Später, sobald das Verhältnis von Nutzer zu Entwickler zunimmt, kann es nötig werden separate Kanäle zu haben.

Wie sollen Leute von allen Kanälen wissen und woher sollen sie gar wissen in welchen sie reden sollen? Und wenn sie reden, woher sollen sie die hiesigen Konventionen kennen?

Die Antwort ist, es ihnen zu sagen, indem man den *channel topic* entsprechend setzt.[16] Der Topic des Kanals ist eine kurze Nachricht, die jeder Nutzer beim betreten des Kanals sieht. Es gibt Neulinge eine kurze Anleitung und Hinweise wie sie an weitere Informationen kommen wie z.B.:

```
Sie sind jetzt in #svn

Topic: #svn ist ein Forum für Fragen von Subversion-Benutzern, siehe
auch http://subversion.tigris.org/. || Entwickler-Diskussionen finden
in #svn-dev statt. || Fügen Sie hier bitte keine langen Protokolle ein,
nutzen Sie hierzu eine Pastebin-Seite wie http://pastebin.ca/. ||
Neuigkeiten: Freigabe von Subversion 1.1.0, weiteres dazu auf
http://svn110.notlong.com/.
```

Das ist zwar knapp, aber es sagt Neueinsteigern, was sie wissen müssen. Es sagt genau wofür der Channel ist, enthält den Link zur Webseite des Projekts (falls jemand an dem Channel vorbei kommt, ohne vorher darauf gewesen zu sein), erwähnt einen verwandten Channel, und gibt ein wenig Anleitung dazu, wie umfangreiche Einfügungen zu handhaben sind.

[15] Es gibt keine Voraussetzung oder Erwartung, dass Sie an Freenode spenden, wenn Ihr Projekt es sich aber leisten kann, sollten Sie es in Erwägung ziehen. Sie sind eine von Steuern befreite gemeinnützige Einrichtung in den USA und sie betreiben einen wertvollen Dienst.

[16] Um den Topic in einem Kanal zu setzen, benutzen Sie den `/topic` Befehl. Alle Befehle im IRC fangen mit einem "/" an. Siehe http://www.irchelp.org/ wenn Sie nicht mit der Nutzung und Administration von IRC vertraut sind; insbesondere ist http://www.irchelp.org/irchelp/irctutorial.html eine hervorragende Anleitung.

Pastebin-Sites

Der Platz in einem IRC-Kanal wird von allen geteilt: jeder sieht was der andere sagt. Normalerweise ist das etwas gutes, da es Leute erlaubt sich an eine Unterhaltung zu beteiligen, wenn sie meinen etwas beitragen zu können und Beobachtern erlaubt beim zusehen etwas zu lernen. Wenn jemand aber einen langen Text zitieren muss, wie ein Protokoll aus einem Debugger wird das problematisch, da eine derart viele Zeilen andere Unterhaltungen stören würde.

Die Lösung ist eine der *pastebin* oder auch *pastebot* -Sites zu benutzen. Wenn Sie bei jemandem nach umfangreichen Daten anfragen, bitten Sie ihn darum, diese nicht direkt in den Kanal einzufügen, sondern ihre Daten (z.B.) auf http://pastebin.ca/ abzulegen und die sich ergebende URL in den Kanal zu kopieren. Diese kann dann jeder besuchen, um sich die Daten anzuschauen.

Es gibt eine Vielzahl freier Sites, zu viele um alle zu nennen, hier nur diejenigen, von denen ich weiß, dass sie benutzt werden: http://www.nomorepasting.com/, http://pastebin.ca/, http://nopaste.php.cd/ http://rafb.net/paste/ http://sourcepost.sytes.net/, http://extraball.sunsite.dk/notepad.php, und http://www.pastebin.com/.

Bots

Viele IRC-Kanäle mit einem technischen Thema haben einen nicht-menschlichen Teilnehmer, einen sogenannten *Bot*, der in der Lage ist, auf bestimmte Befehle zu reagieren, indem er Informationen speichert und wiedergibt. Typischerweise wird der Bot genau wie jeder andere im Kanal angesprochen, d.h. die Befehle werden übermittelt, indem man den Bot "anspricht". Z.B.:

```
<kfogel> ayita: learn diff-cmd = http://subversion.tigris.org/faq.html#diff-cmd
<ayita>  Thanks!
```

Dem Bot (mit dem Namen ayita im Kanal angemeldet) wurde dadurch gesagt, dass er sich eine bestimmte URL merken soll und bei einer Anfrage nach "diff-cmd" wiedergeben soll. Jetzt können wir ayita ansprechen und bitten einem anderen Nutzer etwas über das diff-cmd zu erzählen:

```
<kfogel> ayita: tell hmustermann about diff-cmd
<ayita>  hmustermann: http://subversion.tigris.org/faq.html#diff-cmd
```

Dafür gibt es auch ein Kürzel:

```
<kfogel> !a hmustermann diff-cmd
<ayita>  hmustermann: http://subversion.tigris.org/faq.html#diff-cmd
```

Der genaue Befehl und das Verhalten unterscheidet sich bei jedem Bot. Das obige Beispiel benutzt `ayita` (http://hix.nu/svn-public/alexis/trunk/), wovon üblicherweise in `#svn` eine Instanz läuft. Andere Bots sind unter anderem (http://dancer.sourceforge.net/) und Supybot (http://supybot.com/). Beachten Sie, dass man keine besonderen Rechte auf den Server haben muss um einen Bot zu betreiben. Ein Bot kann auf einem beliebigen Rechner betrieben werden; jeder kann einen Einrichten und anweisen einen bestimmten Server/Kanal zu betreten.

Wenn Sie in Ihrem Kanal bemerken, dass immer wieder die gleichen Fragen gestellt werden, empfehlen ich Ihnen dringend, einen Bot einzurichten. Nur wenige Nutzer im Kanal werden herausfinden wie der

Bot zu bedienen ist, diese Personen werden aber umso mehr Fragen beantworten können, wenn der Bot es Ihnen ermöglicht, um so vieles effektiver zu antworten.

IRC-Archivierung

Auch wenn es möglich ist, alles in einem Kanal zu archivieren, wird das nicht unbedingt erwartet. Unterhaltungen im IRC sind im Grunde genommen öffentlich, aber viele betrachten es sie als informelle, halbwegs private Unterhaltungen. Nutzer neigen dazu nicht so sehr auf ihre Grammatik zu achten und äußern Meinungen (zum Beispiel über andere Entwickler im Projekt) die sie nicht unbedingt archiviert haben möchten.

Es wird natürlich manchmal *Auszüge* geben, die erhalten werden sollte und das ist auch in Ordnung. Die meisten IRC-Anwendungen können, wenn der Nutzer das möchte, Unterhaltungen mitschneiden oder wenn das nicht geht, kann die Unterhaltung immer noch aus dem Fenster der Anwendung heraus kopiert und in ein beständigeres Forum (meistens der Bugtracker) einfügt werden. Alles mitzuschneiden kann aber manche Nutzer unruhig machen. Wenn sie doch alles archivieren, stellen Sie sicher, dass es klar im Topic erklärt wird und geben Sie die URL des Archivs an.

RSS-Feeds

RSS (Really Simple Syndication) ist ein Mechanismus, um mit Metadaten versehene Kurznachrichten an "Abonnenten" zu verteilen, sprich Menschen, die ein Interesse bekundet haben, diese Neuigkeiten zu erfahren. Eine gegebene RSS-Quelle wird üblicherweise als *Feed* bezeichnet, und die Schnittstelle des Benutzers wird *Feed-Reader* oder *Feed-Aggregator* genannt. RSS Bandit [http://www.rssbandit.org/] und der namengebende Feedreader [http://www.feedreader.com/] z.B. sind zwei quelloffene RSS-Reader.

Hier ist nicht der Raum für eine detaillierte technische Erklärung von RSS[17], aber Sie sollten sich zumindest zweier Dinge bewusst sein. Erstens: die Feed-Reader-Software ist durch den Benutzer bestimmt und ist *dieselbe* für alle Feeds, die dieser abonniert hat – tatsächlich ist das der springende Punkt bei RSS: dass der Abonnent selbst die Schnittstelle für alle Feeds bestimmt, und so sollte sich jeder Feed ausschließlich auf das Verteilen der Inhalte konzentrieren. Zweitens: RSS ist inzwischen so allgegenwärtig, dass die meisten Menschen gar nicht bemerken. dass sie es damit zu tun haben. Im großen Ganzen scheint es sich bei RSS einfach um einen kleinen Knopf auf einer Webseite zu handeln mit einer Beschriftung etwa "diese Seite abonnieren" oder "News feed". Sie klicken auf diesen Knopf und von da an aktualisiert sich Ihr Feed-Reader (der z.B. auch ein in Ihre Startseite eingebettetes Applet sein könnte) automatisch wann immer es Neuigkeiten von dieser Site gibt.

Das heißt dass Ihr Open-Source-Project möglicherweise einen RSS-Feed anbieten sollte (beachten Sie, dass viele Hosting-Sites – siehe „Hosting-Pakete“ – es fix und fertig anbieten). Widerstehen Sie der Versuchung, täglich so viele Neuigkeiten zu verbreiten, dass Ihre Abonnenten die Spreu nicht mehr vom Weizen trennen können. Wenn es zu viele Neuigkeiten gibt, werden die Benutzer den Feed einfach ignorieren, wenn nicht sogar gleich ganz abbestellen aus Verärgerung. Idealerweise würde ein Projekt unterschiedliche Feeds anbieten: einen für die großen Neuigkeiten, einen z.B. für Bugtracker-Aktivitäten, einen weiteren je Mailingliste usw. In der Praxis ist es schwierig, hier alles richtig zu machen; so kann es leicht zur Verwirrung bei den Schnittstellen für Besucher und Administratoren der Website führen. Zumindest aber sollte das Projekt einen RSS-Feed auf der Hauptseite anbieten, der die herausragenden Ankündigungen anbieter wie z.B. Freigaben und Sicherheitswarnungen. [18]

[17] Siehe dafür http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html

[18] Ehre, wem Ehre gebührt: dieser Abschnitt war nicht Teil der ersten Ausgabe dieses Buches, aber Brian Aker's Blog-Eintrag "Release Criteria, Open Source, Thoughts On..." [http://krow.livejournal.com/564980.html] erinnerte mich an die Nützlichkeit von RSS-Feeds für Open-Source-Projekte.

Wikis

Ein *Wiki* ist eine Website die es jedem Besucher erlaubt, ihren Inhalt zu bearbeiten; der Begriff "wiki" (von dem hawaiischen Wort für "schnell" oder "super-schnell") ist auch der Name für Software, die solche Bearbeitungen ermöglicht. Wikis wurden 1995 erfunden, richtig populär wurden sie aber erst 2000 bzw. 2001, teilweise befeuert durch den Erfolg von Wikipedia (http://www.wikipedia.org/), ein wikibasiertes freies Lexikon. Sie können Wikis zwischen IRC und Webseiten einordnen: Wikis sind kein Echtzeit-Medium, also haben Benutzer die Möglichkeit, ihre Beiträge zu überdenken und auszufeilen, es ist aber auch sehr leicht beizutragen, da der Bearbeitungsaufwand geringer ist als bei einer normalen Webseite.

Wikis gehören noch nicht zur Standardausstattung für Open-Source-Projekte, werden es aber wahrscheinlich bald sein. Sie sind eine relativ neue Technik, und es wird noch mit ihrer Nutzung experimentiert. Hier ein paar Worte der Vorsicht – derzeit ist es einfacher, den Missbrauch von Wikis zu untersuchen als ihre Erfolge.

Wenn Sie sich entschließen, ein Wiki zu betreiben, verwenden Sie große Aufmerksamkeit auf klare Organisation der Seiten und einen angenehmes Seitenlayout, damit Besucher (bzw. potentielle Bearbeiter) instinktiv wissen, wo sie ihre Beiträge einordnen sollen. Genauso wichtig ist es, diese Richtlinien in das Wiki hinein zu schreiben, damit die Benutzer etwas zur Orientierung haben. Viel zu oft verrennen sich Administratoren in dem Glauben, dass wenn eine Horde Besucher für sich genommen qualitativ hochwertige Beiträge leistet, das Gesamtresultat automatisch auch insgesamt qualitativ hochwertig sein müsste. So jedoch funktionieret ein Wiki nicht. Jede individuelle Seite, jeder Abschnitt, mag für sich genommen gut sein, nicht aber eingebettet in ein unorganisiertes oder verwirrendes Ganzes. Wikis leiden viel zu häufig an:

- **Mangel an Orientierungsrichtlinien.** Eine gut organisierte Webseite gibt dem Besucher das Gefühl, jederzeit Bescheid zu wissen, wo er ist. Bei Seiten mit einer guten Aufmachung kann man beispielsweise intuitiv den Unterschied zwischen dem Bereich für das "Inhaltsverzeichnis" und dem für den "Inhalt" erkennen. Autoren des Wikis werden solche Unterschiede auch respektieren, aber nur wenn die Unterschiede schon am Anfang vorhanden sind.

- **Doppelte Informationen.** Häufig passiert es in Wikis, dass irgendwann verschiedene Seiten ähnliches sagen, da die einzelnen Teilnehmer nicht unbedingt mitbekommen, was andere schreiben. Teilweise kann das am Fehlen der oben erwähnten Orientierungsprinzipien liegen, insofern als Benutzer die bereits vorhandenen Inhalte nicht finden, wenn sie nicht liegen, wo man sie erwarten würde.

- **Unklarheit der Zielgruppe.** Zu einem gewissen Grad ist dieses Problem bei einer so großen Anzahl von Autoren unvermeidbar, kann aber verringert werden, wenn es schriftliche Richtlinien gibt, wie man die Inhalte erstellt. Es hilft auch, neue Beiträge am Anfang aggressiv zu bearbeiten, damit sich ein Standard bildest.

Die übliche Lösung für all diese Probleme ist die gleiche: Redaktionelle Normen anzusetzen und sich nicht allein auf ihre Bekanntgabe zu verlassen, sondern sie durch die Bearbeitung von Seiten durchzusetzen und sich ihnen auch selbst zu unterwerfen. Im Allgemeinen werden Wikis alle Schwächen im Ausgangsmaterial verstärken, da die Teilnehmer alle Muster nachahmen werden, die sie vorfinden. Richten Sie das Wiki nicht nur ein und hoffen Sie, dass sich alles fügt. Es muss auch mit gut geschriebenen Inhalten angefangen werden, damit die Besucher Beispiele haben, denen sie folgen können.

Das leuchtende Beispiel eines gut betriebenen Wikis ist Wikipedia, obwohl das sicherlich teilweise daran liegt, dass der Inhalt (Lexikon-Einträge) von Natur aus für das Wiki-Format geeignet ist. Wenn Sie die Wikipedia aber genauer ansehen, werden Sie feststellen, dass die Administratoren eine *äußerst* gründliche Basis für die Zusammenarbeit ausgelegt haben. Es gibt eine ausführliche Dokumentation dazu, wie man neue Einträge schreiben soll, wie man einen angemessenen, neutralen Standpunkt

bewahrt, welche Art von Änderungen man vornehmen oder vermeiden sollte, welche Verfahren dazu taugen, Konflikte zwischen sich widersprechende Bearbeitungen aufzulösen (mit mehreren Schritten, mitunter auch durch das letzte Mittel eines Schiedsgerichts) usw. Sie haben auch eine Verwaltung für die Autorisierung, damit sie eine Seite für die Bearbeitung sperren können, sollt sie Ziel mehrfacher unangemessener Bearbeitungen werden, bis das Problem gelöst wurde. Mit anderen Worten: werfen sie nicht einfach ein paar Vorlagen auf eine Webseite und drücken die Daumen! Wikipedia funktioniert deshalb, weil ihre Gründer sorgfältig darüber nachgedacht haben, wie man tausende Fremde dazu bringt, ihren Schreibstil einer gemeinsamen Vision anzupassen. Auch wenn Sie nicht gleichermaßen vorbereitet sein müssen, um in einem freien Software-Projekt ein Wiki zu betreiben, lohnt es diesem Geist nachzueifern.

Weitere Informationen über Wikis finden Sie unter http://en.wikipedia.org/wiki/Wiki. Das erste Wiki ist weiterhin wohlauf und beinhaltet viel Material darüber wie man Wikis betreibt: Siehe http://www.c2.com/cgi/wiki?WelcomeVisitors, http://www.c2.com/cgi/wiki?WhyWikiWorks, und http://www.c2.com/cgi/wiki?WhyWikiWorksNot für verschiedene Ansichten.

Website

Es gibt aus technischer Sicht nicht viel darüber zu sagen, wie man die Website für das Projekt einrichtet: Einen Webserver aufzubauen und Webseiten zu schreiben, sind relativ einfache Aufgaben, und das wichtigste im Bezug auf das Layout und die Anordnung wurde bereits im vorherigen Kapitel abgedeckt. Die Hauptfunktion der Website ist, das Projekt klar und einladend zu präsentieren und die anderen Werkzeuge einzubinden (Versionsverwaltung, Bugtracker, usw.). Wenn Ihnen die Kenntnisse dazu fehlen, selbst einen Webserver aufzusetzen, ist es normalerweise nicht schwer, jemanden zu finden, der es kann und bereit ist zu helfen. Trotzdem ziehen es viele vor, bestehende Hosting-Pakete zu nutzen, um sich Zeit und Mühe zu sparen.

Hosting-Pakete

Es gibt zwei Hauptvorteile bei der Nutzung von Hosting-Paketen. Der erste ist die Kapazität ihrer Server und ihre Bandbreite: Ihre Server sind schnelle Kisten mit dicken Leitungen. Egal wie erfolgreich Ihr Projekt wird, Ihnen wird niemals der Plattenplatz oder die Bandbreite ausgehen. Der zweite Vorteil ist seine Einfachheit. Sie haben bereits einen Bugtracker, eine Versionsverwaltung, ein Malinglisten-System, ein Archiv-System und alles was Sie sonst benötigen, um eine Site zu betreiben. Sie haben die Programme konfiguriert, und kümmern sich um die Sicherung aller Daten dieser Programme. Sie müssen nicht viele Entscheidungen treffen. Sie müssen lediglich ein Formular ausfüllen, einen Knopf drücken und plötzlich haben Sie eine Website für Ihr Projekt .

Das sind ziemlich schwerwiegende Vorteile. Der Nachteil ist natürlich, dass Sie sich mit der *angebotenen* Auswahl und Konfiguration abfinden müssen, selbst wenn etwas anderes für Ihr Projekt besser wäre. Meistens lassen sich diese Sites innerhalb enger Grenzen konfigurieren, aber Sie werden niemals die feingranulare Kontrolle haben wie bei einer selbstgebauten Site mit vollem administrativen Zugriff auf den Server.

Ein perfektes Beispiel hierfür ist der Umgang mit generierten Dateien. Bestimmte Webseiten des Projekts können generierte Dateien sein – es gibt z.B. Systeme, um die Daten für eine FAQ in ein einfaches Quellformat zu schreiben, von dem aus HTML, PDF und andere Darstellungsformate generiert werden können. Wie in „Versioniere alles“ früher in diesem Kapitel beschrieben, wollen Sie nicht die generierten Formate in der Versionsverwaltung haben, sondern nur die Quellen. Wenn Ihre Website aber auf den Server von jemand anderem betrieben wird, kann es unmöglich sein, ein eigenes Script einzubinden, um die HTML-Version automatisch zu erzeugen, gleich nachdem etwas an der Quelldatei geändert wurde. Die einzige Abhilfe ist hier, die generierten Formate zusätzlich in der Versionsverwaltung abzulegen, damit sie auch in der Website auftauchen.

Es kann auch schwerwiegendere Folgen geben. Sie werden vielleicht nicht so viel Kontrolle über die Präsentation haben, wie Sie es sich wünschen würden. Manche Hosting-Sites erlauben es Ihnen, Ihre Webseiten anzupassen, der vorgegebene Aufbau hinterlässt aber aber meistens an verschiedenen Stellen erkennbare Spuren. Manche auf SourceForge gehosteten Projekte haben ausgefeilte eigene Webseiten, verweisen Entwickler aber immer noch auf ihre "SourceForge Seite" für weitere Informationen. Die SourceForge-Seite ist, was die Webseite des Projekts gewesen wäre, wenn das Projekt keine angepasste Seite benutzt hätte. Auf der SourceForge-Seite sind Verweise auf den Bugtracker, zum CVS-Repository, Downloads, usw. Eine SourceForge-Seite beinhaltet unglücklicherweise auch eine ganze Menge belanglosen Rauschens. Oben ist ein Werbebanner, oftmals eine Animation. Links ist eine vertikale Anordnung verschiedener Verweise die für jemandem, der am Projekt interessiert ist, kaum von Bedeutung sein dürften. Rechts ist meistens noch mehr Werbung. Lediglich der mittlere Bereich der Seiten ist dem Material des Projekts gewidmet und selbst dieser ist verwirrend angeordnet, so dass Benutzer unsicher sind, auf was sie als nächstes klicken sollen.

Hinter jedem einzelnen Aspekt von SourceForge's Design steht zweifellos ein guter Grund – gut für SourceForge, wie z.B. die Werbung. Für das einzelne Projekt kann das Ergebnis aber eine nicht ganz optimale Website sein. Es ist nicht meine Absicht, auf SourceForge herumzuhacken; ähnliche Bedenken lassen sich auf viele andere Sites übertragen, die Hosting-Pakete anbieten. Das Wesentliche ist hierbei, den Kompromiss zu erkennen den man eingeht. Es wird Ihnen die technische Bürde abgenommen, eine eigene Website zu betreiben, dafür müssen Sie akzeptieren, dass jemand anderes bestimmt, wie sie betrieben wird.

Sie allein können entscheiden, ob ein Hosting-Paket für Ihr Projekt das Beste ist. Wenn Sie eine solche Seite wählen, halten Sie sich die Möglichkeit offen, im nachhinein auf Ihre eigenen Server zu wechseln, indem Sie einen gesonderten Domain-Namen als Adresse verwenden. Sie können diese URL für komplizierte Funktionen auf die Hosting-Seite leiten. Ordnen Sie aber unbedingt alles so, dass Sie die die Adresse des Projekts nicht ändern müssen, sollten Sie sich später umentscheiden.

Die Wahl des Hosting-Anbieters

Derzeit (Anfang 2011) dominieren Drei Große den Markt freier Hosting-Angebote. Alle drei hosten Open-Source-lizenzierte Projekte kostenlos. Sie bieten Versionskontrolle, Bugtracking und Wikis an (manche sogar mehr, wie z.B. Downloadmöglichkeiten für Binärpakete):

- GitHub [http://github.com/] Die Versionskontrolle ist auf Git beschränkt – aber falls Sie ohnehin Git benutzen, ist das vermutlich genau der richtige Ort für Ihr Projekt. GitHub hat sich für Git-Projekte zum Zentrum des Universums entwickelt und integriert all seine Dienste mit Git. GitHub bietet auch eine vollständige API [http://develop.github.com/], um seine Dienste aus Programmen heraus zu nutzen. Es bietet keine Mailinglisten; allerdings werden diese von so vielen anderen geboten, dass dies kein allzu großes Problem sein dürfte.

- Google Code Hosting [http://code.google.com/hosting/] Bietet Subversion und Mercurial zur Versionkontrolle (kein Git, zumindest noch nicht), Wikis, einen Downloadbereich, und einen ziemlich schicken Bugtracker. Es hat auch APIs zu bieten: die Versionskontollsysteme sind naturgemäß ihre eigene API; der Bugtracker hat eine eigene API [http://code.google.com/p/support/wiki/IssueTrackerAPI]; der Wiki-Inhalte ist direkt durch die Versionsverwaltun [http://code.google.com/p/support/wiki/WikiFAQ#How_do_I_edit_a_wiki_page_directly_through_subversion?] zugänglich und ist so auch durch Scripts bearbeitbar; der Downloadbereich bietet gescriptete Uploads [http://code.google.com/p/support/wiki/ScriptedUploads], sprich: eine API. Mailinglisten werden über Google Groups [http://groups.google.com/] geboten (diese Aussage freilich lässt sich auch auf alle anderen Hostinganbieter übertragen).

- SourceForge [http://sourceforge.net/] Dies ist der älteste und in vielerlei Hinsicht auch der größte Anbieter für freies Hosting von Projekten. Es bietet alle Features, die auch die anderen bieten, und seine Benutzerschnittstelle ist gut benutzbar; Allerdings empfinden viele Menschen die Werbeblöcke

auf den Projektseiten als abstoßend. Es scheint der einzige der drei Großen, der im Moment alle wichtigen Versionskontrollsysteme bietet (Git, Subversion, Mercurial, und CVS). SourceForge bietet eigene Mailinglisten, aber natürlich können Sie dafür auch einen anderen Anbieter nutzen.

Einige Organisationen, wie z.B. die Apache Software Foundation [http://www.apache.org/], bieten auch freies Hosting für solche Open-Source-Projekte, die ihrer Mission entsprechen und sich gut in den Bestand der dort bereits existierenden Projekte eingliedern.

Falls Sie im Zweifel sind, wo Sie Ihr Projekt unterbringen sollen, empfehle ich Ihnen dringend, sich an einen der großen drei Anbieter zu halten. Wenn Sie noch mehr Anleitung wünschen: nutzen Sie GitHub, falls Ihr Projekt Git zur Verionierung benutzt, ansonsten nutzen Sie Google Code. Natürlich ist auch SourceForge akzeptabel, aber es bietet keinen Vorteil gegenüber den anderen, und die Werbung kann schon nerven.

Viele Menschen haben beobachtet, dass freie Projekt-Hosting-Sites oftmals die Software, die sie benutzen, um diesen Service zu bieten, selbst nicht unter eine freie Software-Lizenz stellen. (einige, die es tun sind Launchpad [http://launchpad.net], Gitorious [http://gitorious.org/] and GNU Savannah [http://savannah.gnu.org/]). Aus meiner Sicht wäre es zwar ideal, Zugriff auf all den Code zu haben, der die Site ausmacht, aber der entscheidende Punkt ist einen Weg zu haben, die eigenen Daten zu exportieren und die Möglichkeit, mit den Daten automatisiert zu interagieren. Eine solche Site würde Sie nie wirklich einsperren können und würde durch die programmgesteuerte Anbindung sogar zu einem gewissen Grade erweiterbar sein. Während es einen gewissen Wert darstellen würde, den Code, auf dem eine Hosting-Site beruht, unter Open-Source-Bedingungen zur Verfügung zu haben, würden die praktischen Anforderung, diesen Code in eine produktive Umgebung zu verwandeln, die meisten Benutzer völlig überfordern. Diese Sites benötigen etliche Server, angepasste Netzwerke, in Vollzeit beschäftigte Angestellte, die diese warten; allein den Code zu haben, würde nicht ausreichen, den Service zu kopieren oder zu "forken". Die Hauptsache ist sicherzustellen, dass Ihre Daten nicht in die Gefangenschaft geraten.

Wikipedia enthält einen ausführlichen Vergleich von Open-Source-Hosting-Angeboten [http://en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities]; es ist der erste Ort, den Sie bezüglich aktueller umfangreicher Informationen über Hosting-Optionen für Open-Source-Projekte konsultieren sollten. Auch Haggen So unternahm eine Evaluierung der vielfältigen Hosting-Angebote im Rahmen seiner Dr. Phil. Dissertation *Construction of an Evaluation Model for Free/Open Source Project Hosting (FOSPHost) sites*. Auch wenn diese aus dem Jahre 2005 stammt, scheint er sie zuletzt 2007 aktualisiert zu haben und seine Kriterien scheinen langfristig gültig. Seine Ergebnisse finden Sie unter http://www.ibiblio.org/fosphost/, siehe insb. auch die sehr lesbare Vergleichstabelle unter http://www.ibiblio.org/fosphost/exhost.htm.

Anonymität und Beteiligung

Ein Problem, das nicht allein auf Seiten beschränkt ist, die Hosting-Pakete benutzen, dort aber am häufigsten auftritt, ist der Missbrauch der Login-Funktion. Die Funktion selbst ist relativ einfach: Die Seite erlaubt jedem Besucher, sich mit Benutzernamen und Passwort zu registrieren. Von da an speichert sie ein Profil für diesen Nutzer und die Administratoren der Projekte können dem Nutzer bestimmte Rechte einräumen, z.B. Schreibzugriff auf das Projektarchiv.

Das kann äußerst nützlich sein und es ist tatsächlich eines der wesentlichen Argumente für fertige Hosting-Pakete. Das Problem ist, dass Nutzer sich manchmal für Aufgaben anmelden müssen, die eigentlich auch ohne Anmeldung möglich sein sollten, z.B. ein Ticket im Bugtracker zu erstellen. Indem man eine Anmeldung für solche Aufgaben voraussetzt, hebt das Projekt die Einstiegshürden für Angelegenheiten, die möglichst schnell und einfach sein sollten. Natürlich möchte man jemanden erreichen können, der Daten in den Bugtracker eingetragen hat, dazu reicht aber ein Feld, in dem er seine E-Mail-Adresse eingeben kann (falls er möchte). Wenn ein neuer Benutzer einen Bug findet und ihn melden möchte, wird er durch die Anforderung verärgert sein, ein Konto erstellen zu müssen, nur um dem Bugtracker einen Eintrag hinzuzufügen: leicht könnte er sich entscheiden, es gleich ganz zu lassen.

Die Vorteile der Nutzerverwaltung wiegen im Allgemeinen ihre Nachteile auf. Wenn Sie es sich aussuchen können, welche Aufgaben Sie anonym erlauben wollen, stellen Sie sicher nicht nur *alle* Abläufe einzubeziehen, die nur Lesezugriff erfordern, sondern nehmen Sie auch bestimmte Eingabe-Aktionen hinzu, insbesondere im Bugtracker sowie, falls vorhanden, auf den Seiten des Wikis.

Kapitel 4. Soziale und politische Infrastruktur

Das erste, was viele über freie Software wissen wollen, ist meist: "Wie funktioniert das? Was hält ein Projekt am Laufen? Wer trifft die Entscheidungen?". Ich bin immer unzufrieden mit dem faden Geschmack der Antworten über Meritokratie, den Geist der Zusammenarbeit, Code der für sich selbst spricht, usw. Tatsache ist, die Frage ist nicht so leicht zu beantworten. Leistung, Zusammenarbeit und laufender Code gehören zwar alle dazu, sagen aber nichts über den täglichen Ablauf in einem Projekt, oder wie Konflikte gelöst werden.

Dieses Kapitel soll zeigen, welche Grundbausteine zu einem erfolgreichen Projekt gehören. Mit "erfolgreich" meine ich nicht nur die technische Qualität, sondern auch die Gesundheit der Gemeinschaft und seine Überlebensfähigkeit. Die Gesundheit bezieht sich auf die Fähigkeit eines Projeks, neue Codebeiträge und Entwickler aufzunehmen, sowie auf eingehende Bug-Meldungen zu reagieren. Überlebensfähigkeit bedeutet, unabhängig von irgendeinem Beteiligten oder Geldgeber fortbestehen zu können – betrachten Sie es als die Wahrscheinlichkeit für das Weiterleben eines Projekts, selbst wenn alle Gründungsmitglieder ihre Sachen packen und sich etwas anderem widmen. Technischer Erfolg ist leicht zu erreichen, aber ohne eine solide Entwicklergemeinschaft wird ein Projekt scheitern, sobald es wächst, Erfolg hat oder ein charismatisches Mitglied verliert.

Es gibt verschiedene Wege zu diesem Erfolg. Manche benutzen formelle Leitungsstrukturen um ihre Debatten zu lösen, neue Entwickler einzuladen (manchmal auch auszuladen), neue Funktionen zu planen usw. Andere haben weniger formelle Strukturen, dafür halten sich die Teilnehmer bewusst zurück, um eine gerechte Atmosphäre herzustellen, in der sich Leute auf diese Quasi-Regierungsform verlassen können. Beide Wege führen zum gleichen Ziel: das Gefühl einer beständigen Institution, die durch Gewohnheiten und Abläufe unterstützt wird, die alle Beteiligten gut verstehen. Diese Eigenschaften sind bei selbstorganisierenden Systemen noch wichtiger, als bei solchen mit einer zentralen Verwaltung, da jeder sich darüber im Klaren ist, dass ein paar faule Äpfel die ganze Kiste verderben können, zumindest eine Zeit lang.

Aufspaltbarkeit

Entwickler in einem freien Softwareprojekt sind durch eine unverzichtbare Zutat verbunden: die Möglichkeit, es zu *forken*[1]: jeder kann eine Kopie des Quellcodes nehmen und daraus ein neues Projekt starten. Dieses wird als Fork bezeichnet und steht meistens in Konkurrenz zu dem ursprünglichen Projekt. Das paradoxe daran ist, dass die *Möglichkeit* von Forks zumeist ein viel größerer Antrieb ist, als ein tatsächlich bestehender Fork, der übrigens sehr selten eintritt. Ein Fork ist schlecht für alle (die Gründe dafür werden in „Abspaltungen“ im Kapitel Kapitel 8, *Leitung von Freiwilligen* genauer untersucht), je ernster die Gefahr eines Forks wird, desto größer ist die Bereitschaft der Projektbeteiligten, Kompromisse einzugehen, um diese Gefahr abzuwenden.

Ein Fork, oder vielmehr das Potential dafür, ist der Grund, dass freie Software-Projekte keine wirklichen Diktatoren haben. Das klingt überraschend, wenn man bedenkt, wie viele "Diktatoren" oder "Tyrannen" es bei verschiedenen Open-Source-Projekten gibt. Diese Art der Tyrannei ist aber etwas Besonderes, völlig verschieden von dem gewöhnlichen Verständnis des Wortes. Stellen Sie sich einen König vor, dessen Untertanen jederzeit eine Kopie seines Königreichs machen könnten, in dem sie selbst nach eigenem Ermessen regieren könnten. Würde dieser König sich nicht völlig anders benehmen, als einer dessen Untertanen auf Gedeih und Verderb an ihn gebunden sind?

[1] de. Gabel im Sinne einer Gabelung

Projekte sind deshalb, auch ohne formal demokratische Struktur, faktisch Demokratien, zumindest bei wichtigen Entscheidungen. Die Kopierbarkeit von Projekten impliziert die Aufspaltbarkeit; diese wiederum impliziert Konsens. Möglich, dass eine große Bereitschaft besteht, einem Anführer zu vertrauen (das bekannteste Beispiel ist wohl Linus Torvalds bei der Entwicklung des Linux-Kernels), aber nur deshalb, weil sie es sich so *ausgesucht* haben, und das ist ganz und gar nicht ironisch oder zynisch gemeint. Der Diktator hat keine magische Macht über das Projekt. Eine wesentliche Eigenschaft aller Open-Source-Lizenzen ist, dass sie keiner einzelnen Partei die Entscheidungsgewalt über Änderungen am Code oder seine Nutzung einräumen. Wenn der Diktator plötzlich anfangen würde, schlechte Entscheidungen zu treffen, würde das Unruhe hervorrufen, gefolgt von einer Revolte und einem Fork. Aber so weit kommt es meist nicht, da der Diktator vorher Kompromisse eingeht.

Aber auch wenn die Aufspaltbarkeit die Obergrenze für die Macht Einzelner im Projekt bestimmt, heißt das nicht, dass es nicht große Unterschiede in der Leitung von Projekten geben würde. Sie werden nicht jede einzelne Entscheidung auf die Frage hinauslaufen lassen wollen, ob das jemanden zu einem Fork veranlassen könnte. Das würde ermüdend wirken und eine Menge Energie von der eigentlichen Arbeit abziehen. Die nächsten beiden Abschnitte untersuchen die verschiedenen Wege, Projekte so zu organisieren, dass die meisten Entscheidungen reibungslos verlaufen. Diese beiden Beispiele sind zugegeben idealisierte Grenzfälle; viele Projekte bewegen sich irgendwo dazwischen.

Gütige Diktatoren

Das Modell des *gütigen Diktators* entspricht genau dem, wonach es sich anhört: Die letzte Entscheidungsgewalt liegt bei einer Person, von der man aufgrund ihrer Persönlichkeit und Erfahrung erwartet, dass sie mit dieser weise umgeht.

Auch wenn der Begriff "gütiger Diktator", im Englischen bekannt als "benevolent dictator" (oder *BD*), der übliche Begriff für diese Rolle ist, wäre es besser, ihn als einen von der Gemeinschaft anerkannten Vermittler oder Richter zu betrachten. Im allgemeinen treffen gütige Diktatoren nicht alle oder auch nur die meisten Entscheidungen. Es ist ohnehin unwahrscheinlich, dass eine einzelne Person genügend Kenntnisse hat, um in einem größeren Projekt durchweg gute Entscheidungen treffen zu können. Hochrangige Entwickler werden sich nicht lange am Projekt beteiligen, wenn sie nicht zumindest ein Stückweit Einfluss auf seine Richtung haben. Deshalb diktieren gütige Diktatoren nicht besonders viel. Stattdessen lassen sie möglichst viel ohne ihr Zutun, durch Diskussionen oder Experimente entscheiden. Sie nehmen zwar auch selber an diesen Diskussionen teil, jedoch nur als gewöhnliche Entwickler, oftmals verweisen sie auf einen Zuständigen, mit mehr Kenntnissen über einen bestimmten Bereich. Nur wenn offensichtlich wird, dass kein Konsens erreicht werden kann und dass der größte Teil der Gruppe eine Entscheidung haben *will*, damit die Entwicklung fortgesetzt werden kann, sprechen sie ein Machtwort. Der Widerwille gegen Entscheidungen von oben ist ein Wesenszug, den praktisch alle erfolgreichen gütigen Diktatoren teilen; er ist einer der Gründe, dass sie es schaffen, diese Rolle zu behalten.

Wer kann ein gütiger Diktator sein?

Ein gütiger Diktator braucht eine Kombination verschiedener Wesenszüge. Erstens bedarf es eines feinsinniges Gespürs für den eigenen Einfluss im Projekt, das wiederum Zurückhaltung mit sich bringt. In der frühen Phase einer Diskussion sollte man nicht seine Meinungen und Folgerungen mit einer solchen Sicherheit ausdrücken, dass andere das Gefühl bekommen, es wäre sinnlos zu widersprechen. Menschen sollten die Freiheit haben, ihre Ideen auszudrücken, selbst blöde Ideen. Es lässt sich natürlich nicht vermeiden, dass der gütige Diktator ab und zu selber eine blöde Idee vorschlägt, weshalb die Rolle auch die Fähigkeit erfordert, die eigenen schlechten Entscheidungen zu erkennen und sie als solche zu akzeptieren – wobei das ein Wesenszug ist, den *jeder* gute Entwickler aufweisen sollte, insbesondere wenn er lange beim Projekt bleibt. Der Unterschied ist aber, dass der gütige Diktator sich ab und zu solche Patzer leisten kann, ohne sich über sein Ruf all zu viele Sorgen machen zu müssen. Neue Entwickler werden sich weniger selbstsicher fühlen, also sollte der gütige Diktator seine Kritik mit etwas Gespür, sowohl für das technische als auch das psychologische Gewicht seiner Worte formulieren.

Der gütige Diktator muss *nicht* die besten technischen Fähigkeiten im Projekt haben. Er muss lediglich gut genug sein, um selber am Code zu arbeiten und einen Kommentar zu einer in Frage stehenden Änderung abgeben zu können, mehr aber nicht. Die Position des gütigen Diktators kann man sich weder durch einschüchternd gute Programmierfähigkeiten aneignen noch behalten. *Wichtig* ist vor allem Erfahrung und ein allgemeines Gespür für die Architektur von Quellcode – nicht unbedingt die Fähigkeit, auf Befehl guten Code zu produzieren, wohl aber die Fähigkeit, guten Code unabhängig von der Quelle zu erkennen.

Der gütige Diktator ist auch häufig der Gründer des Projekts, was sich aber eher so ergibt, als das es ein Grund wäre. Die Qualitäten die es ermöglichen, erfolgreich ein Projekt zu starten – technische Kompetenz, die Fähigkeit andere zur Beteiligung überzeugen zu können, usw. – sind genau die Qualitäten, die jeder gütige Diktator besitzen muss. Natürlich besitzen die Gründer auch automatisch eine gewisse Erfahrung mit dem Projekt, was ausreichen kann, allen Beteiligten eine gütige Diktatur als einfachsten Weg erscheinen zu lassen.

Bedenken Sie, dass ein Fork aus beiden Richtungen droht. Ein gütiger Diktator kann genau so wie jeder andere einen Fork anfangen, und das haben gelegentlich schon welche gemacht, sobald sie das Gefühl hatten, das sie das Projekt in eine andere Richtung führen sollten, als die Mehrheit der übrigen Entwickler. Da jeder die Möglichkeit zu einem Fork hat, ist es egal, ob der gütige Diktator vollen Zugriff auf die Server des Projekts hat oder nicht. Die Leute reden mitunter vom Zugriff auf den Server, als sei er die ultimative Quelle der Macht in einem Projekt, tatsächlich ist er aber bedeutungslos. Die Fähigkeit, Benutzer zur Versionsverwaltung hinzuzufügen oder zu entfernen, beeinflusst nur die Kopie des Projekts die auf dem Server liegt. Anhaltender Missbrauch dieser Macht, ob vom gütigen Diktator oder jemand anderem, würde einfach zum Umzug der Entwicklung auf einem anderen Server führen.

Ob Ihr Projekt einen gütigen Diktator haben sollte oder mit einem weniger zentralisierten System besser laufen würde, hängt größtenteils davon ab, wer die Rolle einnehmen könnte. Eine gute Faustregel ist: wenn für alle klar ist, wer der gütige Diktator sein sollte, dann ist das der richtige Weg. Wenn es aber keinen offensichtlichen Kandidaten gibt, sollte das Projekt wahrscheinlich ein dezentrales System für Entscheidungen benutzen, wie im nächsten Abschnitt beschrieben.

Konsensbasierte Demokratie

Mit zunehmenden Alter neigen Projekte dazu, vom Modell des gütigen Diktators zu offeneren demokratischen Systemen zu wechseln. Das ist nicht zwangsläufig auf Unzufriedenheit mit einem bestimmten gütigen Diktator zurückzuführen. Ein gemeinschaftliches Regierungssystem ist auf Dauer einfach stabiler. Immer wenn ein gütiger Diktator zurücktritt, oder versucht die Entscheidungsgewalt gleichmäßiger zu verteilen, bekommt die Gruppe eine Gelegenheit, ein neues, nicht-diktatorisches System einzuführen – sich sozusagen eine neue Verfassung zu geben. Die Gruppe wird diese Gelegenheit vielleicht nicht beim ersten oder zweiten Mal wahrnehmen, doch letztendlich wird sie es; sobald sie es tut, ist es unwahrscheinlich, dass das jemals rückgängig gemacht wird. Dieser Vorgang ist auch logisch: Wenn eine Gruppe bestehend aus N Personen, einer bestimmten Person Macht zuspräche, würde das bedeuten, dass N - 1 Personen damit einverstanden wären, ihre eigene Macht zu verringern. Menschen können sich auf so etwas im Allgemeinen nicht einigen. Selbst wenn, wäre das entstandene System auch nur bedingt eine Diktatur: Ein von der Gruppe erkorener gütiger Diktator kann genau so gut durch die Gruppe wieder abgesetzt werden. Sobald ein Projekt von der Führung durch eine einzelne charismatische Person zu einem formaleren gemeinschaftlichen System wechselt, wird es schwerlich einen Schritt zurück tun.

Die Einzelheiten dieser Systeme variieren zwar erheblich, es gibt aber zwei Gemeinsamkeiten: Erstens, die Gruppe arbeitet meist im Konsens; zweitens, es gibt eine formale Einrichtung zur Abstimmung, auf die man zurückgreifen kann, sobald kein Konsens erreicht werden kann.

Konsens bedeutet lediglich eine Vereinbarung, mit der jeder leben kann. Es ist kein unklarer Zustand: Eine Gruppe hat Konsens erreicht, wenn jemand behauptet, man habe sich geeinigt und keiner wider-

spricht. Die Person, die den Konsens konstatiert, sollte natürlich genau formulieren worauf man sich einigt, und welche konkreten Schritte sich daraus ergeben, falls diese nicht offensichtlich sind.

Die meisten Unterhaltungen in einem Projekt drehen sind um technische Themen, z.B. den besten Weg, einen Fehler zu beheben, ob eine neue Funktion hinzugefügt werden soll oder nicht, wie streng Schnittstellen dokumentiert werden sollen, usw. Konsens-basierte Entscheidungen funktionieren gut, weil sie nahtlos mit der technischen Diskussion verlaufen. Am Ende der Diskussion ist man sich meist über den richtigen Kurs einig. Oft schreibt jemand eine abschließende Zusammenfassung der Entscheidungen, und schlägt dadurch implizit den Konsens vor. So gibt es eine letzte Möglichkeit, Einspruch zu erheben, um das Thema gründlicher zu besprechen.

Bei kleinen, nicht kontroversen Entscheidungen ist der Konsensvorschlag implizit. Wenn ein Entwickler zum Beispiel spontan einen Commit macht, um einen Fehler zu beheben, ist das zugleich ein Konsensvorschlag: "Ich nehme an, dass wir alle darüber einstimmen, dass dieser Fehler behoben werden muss und dies der Weg dazu ist". Natürlich sagt das der Entwickler nicht tatsächlich; Er macht einfach den Commit, und die Anderen im Projekt machen sich nicht die Mühe, ihre Zustimmung zu geben, da Schweigen hier Zustimmung bedeutet. Wenn jemand einen Commit macht, bei dem sich herausstellt, dass es *keinen* Konsens gab, wird die Änderung so besprochen, als wäre sie noch gar nicht gemacht. Der Grund dafür ist Thema des nächsten Abschnitts.

Versionsverwaltung bedeutet Entspannung

Die Tatsache, dass der Quellcode des Projekts in der Versionsverwaltung gehalten wird, hat zur Folge, dass die meisten Entscheidungen leicht rückgängig gemacht werden können. Der häufigste Fall ist, dass jemand einen Commit in der falschen Annahme ausführt, dass alle einverstanden seien, und sich kurze Zeit später mit Einwänden konfrontiert zu sehen. Diese Einwände beginnen im Allgemeinen mit der obligatorischen Entschuldigung dafür, dass man die vorangegangene Diskussion verpasst habe, wobei man dies auslassen kann, falls in den Archiven der Mailingliste keine Aufzeichnung der Diskussion zu finden sind. So oder so sollte der Ton nach dem Commit nicht anders sein als davor. Jede Änderung kann rückgängig gemacht werden, zumindest solange, bis davon abhängige Änderungen eingeführt werden (wie neuer Code, der nicht mehr funktionieren würde, wenn man die Änderung plötzlich herausnehmen würde). Die Versionsverwaltung eröffnet dem Projekt die Möglichkeit, Auswirkungen schlechter oder übereilter Entscheidungen rückgängig zu machen. Das wiederum gibt den Leuten die Freiheit, sich in der Frage auf ihren Instinkten zu verlassen, wie viel Rücksprache eine Aktion erfordert.

Der Vorgang der Konsenssuche muss deshalb auch nicht sonderlich formal sein. In den meisten Projekten läuft das nach Gefühl. Kleine Änderungen können ohne Diskussion erledigt werden, oder nach minimaler Diskussion und beiläufigem Abnicken. Bei größeren Änderungen, besonders solchen die eine Menge Code instabil machen könnten, sollten die Beteiligten ein bis zwei Tage warten, bevor sie Konsens annehmen, denn bei einer so wichtigen Diskussion sollte niemand übergangen werden, nur weil er seine E-Mails nicht oft genug abgerufen hat.

Wenn sich also jemand sicher ist, was getan werden muss, sollte er es einfach durchziehen. Das gilt nicht nur für Fehlerbehebungen, sondern auch für Aktualisierungen der Website, Änderungen an der Dokumentation und alles andere, für das Kontroversen unwahrscheinlich sind. Es gibt selten Fälle, bei denen etwas rückgängig gemacht werden muss, und diese können jeder für sich behandelt werden. Natürlich sollte man Menschen nicht zu Eigensinnigkeit ermutigen. Es gibt immer noch einen psychologischen Unterschied zwischen einer Entscheidung, die zur Debatte steht und einer, die bereits getroffen wurde, selbst wenn sie technisch umkehrbar ist. Menschen haben das Gefühl, dass Bewegung mit Tatkraft einhergeht, und sind deshalb seltener bereit eine Änderung rückgängig zu machen, als sie von vornherein zu verhindern. Wenn ein Entwickler diese Tatsache missbraucht, indem er eine potentiell kontroverse Änderung zu schnell einpflegt, kann und sollte man sich beschweren und dem Entwickler engere Grenzen setzen, bis hier Besserung eintritt.

Wenn kein Konsens möglich ist, stimme ab!

Zwangsläufig wird es Debatten geben, bei denen man sich einfach nicht einig wird. Wenn alle anderen Möglichkeiten fehlschlagen, einen Stillstand aufzulösen, sollte man abstimmen. Bevor aber eine Abstimmung stattfinden kann, muss es eine klare Auswahl auf dem Stimmzettel geben. Auch hier läuft die technische Diskussion glücklicherweise mit den Entscheidungsabläufen des Projekts zusammen.
Die Art von Fragen, die zu einer Abstimmung führen, drehen sich oftmals um komplexe, mehrschichtige Angelegenheiten. Bei jeder dieser komplexen Diskussionen gibt es meistens ein oder zwei Personen mit der Rolle des *ehrlichen Vermittlers*: sie fassen periodisch die verschiedenen Argumente zusammen und behalten den Überblick über die zentralen Streitpunkte (und Übereinstimmungen). Diese Zusammenfassungen helfen allen, den Fortschritt einzuschätzen und erinnern daran, welche Punkte noch offen sind. Sie dienen auch als Prototypen für die Stimmzettel, sollte es zur Abstimmung kommen. Wenn die Vermittler ihre Arbeit gut machen, werden sie glaubhaft abstimmen können, sobald das nötig wird, und die Gruppe wird diese Stimmzettel akzeptieren. Die Vermittler können an der Debatte teilnehmen; es ist nicht nötig, dass sie sich aus dem Schlachtgetümmel heraushalten, solange sie die Ansichten Anderer verstehen und angemessen repräsentieren können, und die Meinungen ihrer Partei sie nicht daran hindert, den Stand der Diskussion auf eine neutrale Art zusammenzufassen.

Der tatsächliche Inhalt des Stimmzettels ist normalerweise nicht kontrovers. Bis es zu einer Abstimmung kommt, hat sich die Meinungsverschiedenheit auf ein paar Kernpunkte reduziert, mit erkennbaren Namen und kurzen Beschreibungen. Ab und zu wird ein Entwickler die Form des Stimmzettels beanstanden. Manchmal sind diese Bedenken gerechtfertigt, wenn beispielsweise ein wichtiger Punkt auf dem Stimmzettel ausgelassen oder nicht richtig beschrieben wurde. Zuweilen kann ein Entwickler aber auch lediglich die Absicht haben, das Unausweichliche hinauszuschieben, vielleicht mit dem Wissen, dass die Wahl nicht zu seinen Gunsten ausfallen wird. Siehe „Schwierige Leute“ im Kapitel Kapitel 6, *Kommunikation* für eine Beschreibung wie man mit solchen Quertreibern umgehen sollte.

Denken Sie daran, die Wahlform festzulegen, da es viele verschiedene gibt und es falsche Annahmen darüber geben könnte, welche Form benutzt wird. Meistens ist eine *Zustimmungs-Wahl* eine gute Entscheidung, bei der jeder Wähler für so viele der Kandidaten auf den Stimmzettel stimmen kann wie er möchte. Zustimmungs-Wahlen sind einfach zu erklären und auszuzählen, und im Gegensatz zu anderen Wahlsystemen erfordern sie nur einen Wahlgang. Siehe http://de.wikipedia.org/wiki/Wahl_durch_Zustimmung für weitere Details über Zustimmungs-Wahlen. Lassen Sie sich aber nicht zu einer Debatte verleiten, welches Wahlsystem benutzt werden soll (dann fänden Sie sich schnell in einer Debatte über ein Wahlsystem, um über ein Wahlsystem abzustimmen!). Zustimmungs-Wahlen sind auch gut, weil es sehr wenig an ihnen zu beanstanden gibt – sie sind so gerecht, wie ein Wahlsystem nur sein kann.

Und schließlich sollten die Wahlen öffentlich gehalten werden. Es gibt keinen Grund für Geheimhaltung oder Anonymität bei einer Wahl, bei der zuvor sowieso öffentlich debattiert wurde. Lassen Sie jeden Beteiligten seine Stimmen an die Mailingliste schicken, damit jeder Beobachter selber die Ergebnisse nachrechnen und überprüfen kann, und alles in den Archiven aufgezeichnet wird.

Wann sollte abgestimmt werden?

Das schwerste bei Wahlen ist zu bestimmen, wann es dafür Zeit ist. Im allgemeinen sollten Wahlen sehr selten sein – als letztes Mittel, wenn alle anderen fehlgeschlagen sind. Betrachten Sie Wahlen nicht als eine tolle Möglichkeit, Debatten zu klären. Das sind sie nicht. Sie beenden Diskussionen und dadurch auch kreative Überlegungen zu dem Problem. So lange die Diskussion weitergeht, kann jemand auf eine neue Lösung kommen, die allen gefällt. Das geschieht überraschend oft: Eine lebhafte Debatte kann zu einer neuen Art führen, das Problem anzugehen und schließlich zu einer für alle zufriedenstellenden Lösung. Selbst ohne solche Vorschläge ist es meist dennoch besser, einen Kompromiss auszuhandeln als eine Wahl abzuhalten. Nach einem Kompromiss ist keiner ganz glücklich, aber zumindest gibt es anders als bei Wahlen keinen der völlig unzufrieden ist. Politisch gesehen ist die erste Situation vorzuziehen:

Zumindest bekommt jeder das Gefühl, etwas für seine Unzufriedenheit bekommen zu haben. Er mag unzufrieden sein, aber das sind alle anderen auch.

Der Hauptvorteil einer Abstimmung ist die Lösung der Frage, damit alle endlich weitermachen können. Es wird aber durch das Zählen von Köpfen erledigt, anstatt durch einen vernünftigen Dialog, der alle zum selben Ergebnis führt. Die Bereitschaft, Fragen durch Wahlen zu lösen, scheint mir mit zunehmender Erfahrung der Teilnehmer bei Open-Source-Projekten abzunehmen. Stattdessen versuchen Sie, vorher nicht in Erwägung gezogene Lösungen zu erkunden, oder größere Kompromisse einzugehen, als sie ursprünglich geplant hatten. Es gibt verschiedene Möglichkeiten, um eine voreilige Wahl zu verhindern. Die offensichtlichste ist einfach zu sagen, "Ich denke nicht, dass wir schon bereit für eine Wahl sind" und zu erklären warum. Eine weitere ist, eine informelle (nicht bindende) Zählung durchzuführen. Wenn die Reaktion klar in die eine oder andere Richtung tendiert, wird es manche Teilnehmer plötzlich kompromissbereiter und damit eine formelle Wahl überflüssig machen. Die effektivste Lösung ist aber, einfach eine neue Lösung anzubieten, oder eine neue Sicht auf einen alten Vorschlag, damit Leute sich erneut mit dem Sachverhalt befassen, anstatt immer wieder die gleichen Argumente zu wiederholen.

Es gibt auch seltene Fälle, in denen alle sich einig sind, dass jede Kompromisslösung schlimmer ist als jeder der Nicht-Kompromisse. In einem solchen Fall gibt es weniger an einer Abstimmung auszusetzen, sowohl weil es wahrscheinlich zu einer besseren Lösung führt, als auch weil die Beteiligten nicht sonderlich unglücklich mit der Lösung sein werden, egal wie es ausgeht. Auch dann sollte man die Wahl nicht übereilen. Die Diskussion, die der Wahl vorausgeht, bildet die Wählerschaft auch weiter, also kann ein frühes Ende dieser Diskussion die Qualität des Ergebnisses schmälern.

(Denken Sie daran, dass diese Empfehlung zur Zurückhaltung beim Ausrufen von Wahlen nicht für Wahlen zur Aufnahme von Änderungen gilt, die in „Stabilisierung einer neuen Version“ im Kapitel Kapitel 7, *Paket-Erstellung, Veröffentlichung, und tägliche Entwicklung* beschrieben werden. Dort sind Wahlen eher ein Mittel zur Kommunikation, eine Möglichkeit seine Beteiligung an der Überprüfung einer Änderung kundzugeben, damit alle erkennen können ob eine Änderung ausreichend überprüft wurde).

Wahlberechtigung

Bei einem Wahlsystem stellt sich die Frage nach der Wählerschaft: Wer darf wählen? Das kann eine sensible Frage sein, da es das Projekt zwingt, bestimmte Personen offiziell für ihre Beteiligung anzuerkennen oder für ihr besseres Urteilsvermögen gegenüber anderen anzuerkennen.

Die beste Lösung ist einfach eine vorhandene Unterscheidung wie den Commit-Zugriff zu nehmen, und das Wahlrecht daran festzumachen. Bei Projekten mit mehreren Ebenen für den Zugriff, hängt die Frage ob Personen mit geringerem Zugriff wählen können, größtenteils von dem Ablauf ab mit dem die Zugriffsrechte gewährt werden. Wenn ein Projekt sie freizügig austeilt, um beispielsweise eine Vielzahl an Programmen von dritten Parteien im Projektarchiv zu pflegen, solle es klar gestellt werden, dass es bei eingeschränktem Commit-Zugriff wirklich um die Pflege der Software geht und nicht um die Wahlberechtigung. Die umgekehrte Implikation gilt natürlich auch: Da Personen mit vollem Commit-Zugriff Wahlberechtigte sind, müssen sie nicht nur als Programmierer, sondern als Mitglieder der Wählerschaft ausgesucht werden. Wenn jemand auf der Mailingliste dazu tendiert, Unruhe zu stiften oder Behinderungen zu verursachen, sollte die Gruppe sich gut überlegen ob er die Commit-Berechtigung bekommt, selbst wenn diese Person guten Code schreibt.

Das Wahlsystem selbst, sollte benutzt werden um neue Commiter zu wählen, sowohl Teil- als auch Vollberechtigte. Hier ist allerdings eine der wenigen Stellen an denen Geheimhaltung angemessen ist. Sie können keine Wahlen über Kandidaten auf der öffentlichen Mailingliste halten, da es die Gefühle (und den Ruf) des Kandidaten verletzen könnte. Der übliche Weg ist stattdessen, dass bereits bestehende Commiter an einen privaten Verteiler schreiben, ausschließlich im Kreise der anderen Committer, mit dem Vorschlag für den neuen Kandidaten. Die anderen Committer geben ihre ehrliche Meinung,

in dem Wissen, dass die Diskussion privat ist. Oft wird es keinen Einspruch geben und eine Wahl ist nicht notwendig. Nachdem man ein paar Tage gewartet hat, um allen Committern genügend Reaktionszeit zu geben, schreibt der Antragsteller dem neuen Kandidaten eine E-Mail indem ihm der Commit-Zugriff angeboten wird. Wenn es Einwände gibt, wird wie bei jeder anderen Frage eine Diskussion entstehen die möglicherweise mit einer Abstimmung endet. Damit diese Diskussion offen und ehrlich abläuft, sollte die bloße Tatsache, dass sie stattfindet geheim gehalten werden. Wenn die Person die in Betracht gezogen wird davon wüsste und niemals das Angebot bekäme, könnte er daraus schließen, dass er die Wahl verloren hat und wäre wahrscheinlich gekränkt. Wenn jemand natürlich explizit nach Commit-Zugriff fragt, dann gibt es keine andere Möglichkeit als den Vorschlag in Betracht zu ziehen und ihn entweder anzunehmen oder abzuweisen. Wenn letzteres der Fall ist, dann sollte es so höflich wie möglich verlaufen, mit einer eindeutigen Erklärung: "Wir mögen deine Patches, haben davon aber bisher nicht genug gesehen", oder "Wir mögen deine Patches, mussten aber wesentliche Anpassungen machen, damit sie angewandt werden konnten, also fühlen wir uns noch nicht Wohl dabei, dir derzeit Commit-Zugriff zu geben. Wir hoffen allerdings, dass sich das mit der Zeit ändert". Bedenken Sie, dass ihre Worte der Person einen Schlag versetzen könnte, je nachdem wie selbstsicher sie ist. Versuchen Sie aus der Sicht des Empfängers zu sehen wenn Sie die E-Mail schreiben.

Da es eine schwerwiegendere Entscheidung ist einen neuen Committer aufzunehmen als die meisten anderen einmaligen Entscheidungen, haben manche Projekte besondere Anforderungen für solche Abstimmungen. Es kann beispielsweise erforderlich sein, *n* Stimmen für und keine gegen den Kandidaten zu bekommen, oder die Zustimmung einer qualifizierte Mehrheit. Die genaue Grenze ist nicht wichtig; grunsätzlich geht es darum bei der Aufnahme Vorsicht walten zu lassen. Ähnliche, oder noch strengere, spezielle Anforderungen können bei Abstimmungen über das *Entfernen* eines Committers, obgleich das hoffentlich niemals nötig sein wird. Siehe „Committer“ im Kapitel Kapitel 8, *Leitung von Freiwilligen* für weiteres über Aspekte zu der Aufnahme und Entfernung neuer Committer, ohne Zusammenhang mit Wahlen.

Meinungsumfragen contra Abstimmung

Für bestimmte Abstimmungen, kann es sich lohnen den Wahlkreis auszuweiten. Wenn die Entwickler beispielsweise einfach nicht herausbekommen, ob eine bestimmte Schnittstelle zu der Art passt, in der Leute die Software tatsächlich benutzen, ist eine Lösung, auf der Mailingliste des Projekts darüber abstimmen zu lassen. Dies sind in Wirklichkeit eher *Meinungsumfragen* als Abstimmungen, aber die Entwickler können die Ergebnisse als bindend betrachten, wenn sie wollen. Wie bei jeder Meinungsumfrage erklären Sie den Beteiligten, dass es eine Möglichkeit gibt, Vorschläge zu machen: Wenn jemandem etwas besseres einfällt als bei der Umfrage angeboten wird, kann seine Reaktion zu dem wichtigsten Ergebnis der Umfrage werden.

Vetos

Manche Projekte erlauben besondere Stimmen, bekannt als *Veto*. Ein Veto ist ein Weg für einen Entwickler eine übereilte oder schlecht überlegte Änderung aufzuhalten, zumindest lange genug, um weiter darüber zu diskutieren. Sie können ein Veto irgendwo zwischen starken Einspruch und handfester Obstruktion einordnen. Seine genaue Bedeutung unterscheidet sich von einem Projekt zu anderen. Manche Projekte machen es sehr schwer ein Veto aufzuheben; andere erlauben ihre Aufhebung durch eine gewöhnliche Mehrheit, etwa nach einer erzwungenen Verzögerung für weitere Diskussion. Jedes Veto sollte von einer ausführliche Erklärung begleitet werden; ein Veto ohne Erklärung sollte gleich als ungültig erachtet werden.

Mit Vetos kommt das Problem des Missbrauchs. Manchmal sind Entwickler zu begierig den Einsatz zu erhöhen indem Sie ein Veto aussprechen. Sie können den Missbrauch von Vetos verhindern, indem Sie selber nur widerstreben darauf zurückgreifen, sowie indem Sie sanft darauf hinweisen wenn jemand anderes das Veto zu oft benutzt. Falls nötig, können Sie die Gruppe auch daran erinnern, dass Vetos nur so lange bindend sind, wie die Gruppe sich darüber einig ist – schließlich wird sich Änderung X durch-

setzten, wenn eine klare Mehrheit der Entwickler X machen will. Entweder wird das Veto zurückgezogen, oder die Gruppe wird sich entschließen seine Bedeutung zu verringern.

Sie werden Leute vielleicht "-1" schreiben sehen, um ein Veto auszudrücken. Diese Nutzung kommt von der Apache Software Foundation, die einen hochgradig strukturierten Ablauf für Abstimmungen und Vetos hat, beschrieben bei http://www.apache.org/foundation/voting.html. Die Apache-Normen haben sich auf andere Projekte übertragen und Sie werden ihre Konventionen in unterschiedlichem Maße an vielen Stellen in der Open-Source-Welt beobachten können. Technisch gesehen, bedeutet "-1" nicht immer ein formelles Veto, selbst nach den Apache-Normen. Informell wird es jedoch für gewöhnlich als solches betrachtet, oder zumindest als starken Einspruch.

Wie Stimmen bei einer Wahl, kann ein Veto auch im Nachhinein ausgesprochen werden. Es ist nicht in Ordnung einem Veto mit der Begründung zu widersprechen, dass die zur Debatte stehende Änderung bereits durchgeführt oder die Aktion bereits getätigt wurde (es sei denn es handelt sich um etwas unumkehrbares, wie eine Pressemitteilung). Andererseits wird und sollte ein Veto welches Wochen oder Monate später ankommt wahrscheinlich nicht sonderlich Ernst genommen werden.

Schriftliche Regeln

Irgendwann wird die Anzahl der Vereinbarungen und Übereinkünfte die in Ihrem Projekt umhergehen derart groß werden, dass sie irgendwo aufgezeichnet werden müssen. Damit ein solches Dokument rechtmäßig ist, sollten Sie klarstellen, dass es auf Diskussionen und Vereinbarungen auf den Mailinglisten beruht, die bereits in Kraft sind. Wenn Sie es zusammenstellen, verweisen Sie auf die relevanten Threads in den Archiven und immer wann Sie an einen Punkt erreichen, bei dem Sie sich nicht sicher sind, fragen Sie nochmal nach. Das Dokument sollte keine Überraschungen enthalten: Es ist keine Quelle für Vereinbarungen sondern lediglich ihre Beschreibung. Wenn es Erfolg hat, werden Leute natürlich anfangen es zu Zitieren, als eine Quelle für Autorität, was aber nur bedeutet, dass es den allgemeinen Wille der Gruppe zutreffend widerspiegelt.

Dieses ist das Dokument, welches in „Richtlinien für Entwickler“ im Kapitel Kapitel 2, *Der Einstieg* angespielt wird. Wenn das Projekt noch sehr jung ist, werden Sie selbstverständlich die Richtlinien auslegen müssen, ohne den Vorteil einer langen Historie im Projekt, worauf Sie sich beziehen können. Mit zunehmender Reife der Entwicklungergemeinschaft, können Sie die Sprache anpassen um widerzuspiegeln, welche Abläufe sich tatsächlich entwickelt haben.

Versuchen Sie nicht alles abzudecken. Kein Dokument kann alles umfassen, was Leute wissen müssen um an einem Projekt teilzunehmen. Viele der Konventionen die ein Projekt entwickelt werden nie ausgesprochen oder explizit erwähnt und würden von dem wichtigen aber nicht offensichtlichem Material ablenken. Es hat beispielsweise keinen Sinn, Richtlinien wie folgende zu schreiben "Seien Sie höflich und respektvoll zu anderen auf der Mailingliste und fangen Sie keine Flamewars an", oder "Schreiben Sie sauberen, lesbaren, bugfreien Code". Diese Sachen sind natürlich wünschenswert, da es allerdings kein denkbares Universum gibt, indem sie *nicht* wünschenswert wären, ist ihre Erwähnung der Mühe nicht wert. Wenn Leute auf der Mailingliste unhöflich sind oder fehlerhaften Code schreiben, dann werden sie nicht damit aufhören, nur weil es die Richtlinien des Projekts es vorschreiben. Solche Situationen müssen bei ihrem Auftreten behandelt werden, nicht durch pauschale Ermahnungen sich gut zu benehmen. Wenn das Projekt andererseits spezifische Richtlinien hat *wie* man guten Code schreibt, z.B. Regeln über die Dokumentation jeder API in einem bestimmten Format, dann sollten diese Richtlinien so vollständig wie möglich niedergeschrieben werden.

Der Inhalt des Dokuments lässt sich gut bestimmen, durch die die Fragen von Neulingen sowie anhand der Beschwerden, denen erfahrene Entwickler am häufigsten begegnen. Das bedeutet nicht zwangsläufig, dass es zu einer FAQ werden sollte – es braucht wahrscheinlich eine kohärentere erzählerische Struktur, als es eine FAQ bieten kann. Es sollte aber genauso auf tatsächlichen Fragen basieren, nicht auf solchen die Sie erwarten gestellt zu bekommen.

Ein Projekt mit einem gütigen Diktator, oder Mitglieder mit besonderen Vollmachten (Präsidenten, Vorsitzende, was auch immer), sollte das Dokument als gute Gelegenheit betrachten, den Ablauf für Nachfolger festzulegen. Manchmal kann das so einfach sein wie bestimmte Personen als Ersatz zu benennen, sollte der gütige Diktator das Projekt aus irgend einem Grund verlassen. Wenn es einen gütigen Diktator gibt, kann im allgemeinen nur der Diktator selber damit durchkommen einen Nachfolger zu benennen. Wenn es gewählte Vorstandsmitglieder hat, dann sollten die gleichen Abläufe die bei ihrer Wahl verwendet wurden in dem Dokument beschrieben werden. Wenn es ursprünglich keinen Ablauf gab, dann sollten Sie sich innerhalb des Projekts auf einen bestimmten Ablauf einigen *bevor* Sie darüber schreiben. Menschen können empfindlich sein, was hierarchische Strukturen angeht.

Das wichtigste ist vielleicht klarzustellen, dass die Regeln überdacht werden können. Sollten die Konventionen in dem Dokument anfangen das Projekt zu behindern, erinnern Sie alle daran, dass es ein lebendes Spiegelbild der Absichten der Gruppe sein soll, und keine Quelle für Frustration und Behinderung. Wenn jemand es sich zur Gewohnheit macht, unangebrachterweise immer gerade dann wenn ihm Regeln im Weg stehen, darum zu bitten, diese neu zu überdenken – ist Schweigen manchmal die beste Taktik. Wenn andere mit den Beschwerden übereinstimmen, werden sie auch das Wort ergreifen und es wird offensichtlich sein, dass sich etwas ändern muss. Wenn sonst keiner zustimmt, dann wird die Person nicht sonderlich viel Resonanz erhalten und die Regeln werden so stehenbleiben wie sie sind.

Zwei gute Beispiele für Projekt-Richtlinien sind der Subversion Community Guide unter http://subversion.apache.org/docs/community-guide/ und die Dokumente zur Organisation der Apache Software Foundation unter http://www.apache.org/foundation/how-it-works.html und http://www.apache.org/foundation/voting.html. Die ASF ist in Wirklichkeit eine Sammlung von Software-Projekten, rechtlich als eine Gemeinnützige Organisation aufgestellt, also tendieren ihre Dokumente stärker dazu, Leitungsabläufe zu beschreiben als Konventionen für Entwickler. Sie sind es trotzdem lesenswert, denn sie stellen die gesammelte Erfahrung einer Vielzahl von Open-Source-Projekten dar.

Kapitel 5. Geld

In diesem Kapitel untersuchen wir die Finanzierung freier Software. Es richtet sich nicht nur an die Entwickler, die für ihre Arbeit an freien Software-Projekte bezahlt werden wollen, sondern auch an Projektleiter, die ein Verständnis über die soziale Dynamik der Entwicklungsumgebung haben müssen. In den folgenden Abschnitten gehe ich davon aus, dass Sie entweder ein bezahlter Entwickler, oder ein Leiter solcher Entwickler sind. Die Ratschläge werden für beide oft die gleichen sein; wenn nicht, wird die angesprochene Gruppe aus dem Kontext klar ersichtlich sein.

Unternehmen finanzieren die Entwicklung freier Software schon seit langem. Viele Entwicklung wurden schon immer informell subventioniert. Wenn ein Systemadministrator ein Programm zur Netzwerkanalyse schreibt, um seine Arbeit zu erleichtern, stellt er es online und erhält Beiträge von anderen Administratoren, in Form von Bugfixes und neuen Funktionen, sodass sich sich ein neues Konsortium bildet. Die Finanzierung stammt aus den Gehältern der Administratoren, seine Bürofläche und Infrastruktur werden, wenn auch ohne ihr Wissen, von dem jeweiligen Arbeitgeber gespendet. Diese Organisationen profitieren natürlich von der Investition, auch wenn sie zunächst, aus institutioneller Sicht, nichts davon wissen.

Der Unterschied heute ist, dass viele dieser Anstrengungen formalisiert werden. Firmen sind sich der Vorteile von Open-Source-Software bewusst geworden und fangen an sich direkter an ihrer Entwicklung zu beteiligen. Entwickler erwarten mittlerweile, dass wirklich wichtige Projekte zumindest Spenden, oder sogar längerfristige Sponsoren anlocken. Geld hat zwar nicht sonderlich viel an der Dynamik der Entwicklung freier Software geändert, der Maßstab der Abläufe ist aber doch größer geworden, sowohl hinsichtlich der Anzahl der Entwickler, als auch der Zeit pro Entwickler. Es hat auch die Organisation der Projekte beeinflusst und wie die beteiligten Parteien miteinander umgehen. Es geht nicht nur darum, wie das Geld ausgegeben wird oder wie Rendite gemessen wird, sondern auch um Verwaltung und Ablauf: Wie kann die hierarchische Befehlsstruktur einer Firma, mit einer halb dezentralisierten Gemeinschaften von Freiwilligen im Einklang gebracht werden? Werden die Entwickler dieser beiden Gruppen sich überhaupt darauf einigen können, was "Produktivität" bedeutet?

Finanzielle Rückendeckung wird allgemein von Open-Source-Gemeinschaften gerne angenommen. Sie kann vor den Folgen vom Chaos schützen, die so viele Projekte wegfegen, bevor sie es wirklich schaffen, vom Boden abzuheben. Leute sind eher bereit, der Software eine Chance zu geben, wenn sie das Gefühl haben, ihre Zeit in etwas zu investieren, was auch noch in 6 Monaten da sein wird. Schließlich ist Glaubwürdigkeit zu einem gewissen Grad ansteckend. Wenn sagen wir IBM ein Projekt unterstützt, kann man davon ausgehen, dass ein Scheitern des Projekts nicht zugelassen wird, und ihre sich daraus ergebende Bereitschaft selbst Mühe aufzubringen kann zu einer zu einer selbst erfüllende Prophezeiung werden.

Finanzierung bringt jedoch auch ein Gefühl von Kontrolle. Ohne sorgfältige Handhabung kann Geld die Entwicklergemeinschaft in Lager spalten. Wenn die unbezahlten Entwickler das Gefühl bekommen, dass Entscheidungen über die Architektur oder neuen Funktionen, einfach eine Frage des Geldes sind, werden sie zu einem anderen Projekt wechseln, indem sie eher das Gefühl haben, dass Leistung das Ausschlaggebende ist. Unbezahlte Arbeit für Funktionen, die alleine im Interesse einer Firma sind, wird kein freiwilliger Entwickler machen. Sie werden sich vielleicht nicht auf der Mailingliste beschweren, mit der Zeit, wird es aber immer weniger von außen zu hören geben, während Freiwillige sich immer weniger bemühen ernst genommen zu werden. Das Surren der kleineren Aktivitäten, in Form von Bug-Reports und gelegentlichen kleinen Fixes wird weitergehen. Es wird aber von außen keine größeren Beiträge oder Beteiligung an wichtigen Diskussionen geben. Leute spüren, was man von ihnen erwartet (bzw. nicht erwartet), und werden diesen Erwartungen gerecht werden.

Geld muss zwar vorsichtig benutzt werden, Einfluss ist aber deswegen noch lange nicht käuflich. Der Haken ist, dass man ihn nicht direkt erkaufen kann. Bei einem einfachen kommerziellen Geschäft, tau-

schen Sie Geld gegen ein Gut. Wenn Sie eine zusätzliche Funktion benötigen, unterschreiben Sie einen Vertrag, zahlen dafür und es wird umgesetzt. In einem Open-Source-Projekt geht das nicht so einfach. Sie werden vielleicht mit ein paar Entwickler Verträge abschließen, aber diese würden sich – und Sie – täuschen, wenn sie garantieren, dass die von Ihnen finanzierte Arbeit von der Entwicklergemeinschaft angenommen wird. Nur weil Sie dafür bezahlen, mag die Funktion noch lange nicht zu der Vorstellung über die Zukunft der Software der Gemeinschaft passen. Die Arbeit kann nur im Rahmen seiner Leistung und wie gut sie sich in die Vision der Community für die Software einfügt angenommen werden. Sie werden dabei möglicherweise etwas zu der Vision zu sagen haben, Sie werden aber nicht die einzige sein.

Einfluss ist also nicht käuflich, Sachen die zu Einfluss *führen* sind es aber sehr wohl. Das offensichtlichste Beispiel sind Programmierer. Wenn gute Programmierer eingestellt werden, und sie lange genug bleiben um Erfahrung mit der Software und Glaubwürdigkeit in der Gemeinschaft zu sammeln, können Sie das Projekt gleichermaßen beeinflussen wie jedes andere Mitglied. Sie haben bei Wahlen eine Stimme, mehreren Entwickler geben Ihnen sogar ein Stimmblock. Wenn Sie in dem Projekt respektiert werden, bekommen sie Einfluss über ihre Stimmen hinaus. Bezahlte Entwickler müssen auch nicht ihre Motive versuchen zu verschleiern. Schließlich will jeder der eine Änderung macht es aus irgend einem Grund. Die Motive Ihrer Firma sind in keiner Weise weniger berechtigt, als die von irgendjemand anderem. Das Stimmgewicht ihrer Firma hängt jedoch von dem Status ihrer Stellvertreter im Projekt ab, nicht von ihrer Größe, ihrem Budget oder Geschäftsplan.

Arten der Beteiligung

Es gibt viele verschiedene Gründe Open-Source-Projekte zu Finanzieren. Die Punkte auf dieser Liste schließen sich nicht gegenseitig aus; oftmals wird die Finanzielle Rückendeckung das Resultat mehrerer oder all dieser Motivationen sein:

Geteilte Last
: Separate Organisationen mit überlappenden Softwarebedarf merken, dass sie die gleiche Arbeit machen, entweder schreiben sie redundant den gleichen Code in der selben Firma, oder sie kaufen ähnliche Produkte von proprietären Anbietern. Wenn sie bemerken, was vor sich geht, werden sie vielleicht ihre Ressourcen zusammenlegen und ein Open-Source-Projekt gründen (oder beitreten) welches an ihren Bedarf angepasst ist. Die Vorteile sind offensichtlich: Die Kosten der Entwicklung werden geteilt, die Vorteile kommen aber allen gleichermaßen zugute. Obwohl dieses Szenario die intuitivste für gemeinnützige Organisationen scheint, kann es auch für profitorientierte Konkurrenten Sinn machen.

  Beispiele: http://www.openadapter.org/, http://www.koha.org/

Verbesserung von Dienstleistungen
: Wenn eine Firma Dienstleistungen zu bestimmten Open-Source-Anwendungen verkauft oder diese durch Open-Source-Software attraktiver gemacht werden, liegt es natürlich im Interesse der Firma sicherzustellen, dass diese Anwendungen auch gepflegt werden.

  Beispiel: Die Unterstützung von http://subversion.tigris.org/ durch CollabNet [http://www.collab.net/] (Haftungsauschluss: Das ist meine tägliche Arbeit, es ist aber auch ein perfektes Beispiel für dieses Modell).

Unterstützung des Hardware-Absatzes
: Der Wert von Computern und Hardware ist direkt von der dafür zur Verfügung stehenden Software abhängig. Hardware-Verkäufer – nicht nur Verkäufer kompletter Maschinen, sondern auch die Hersteller von Peripheriegeräten und Mikrochips – haben herausgefunden, dass es ihren Kunden wichtig ist, hochwertige freie Software für ihre Hardware zu haben.

Untergrabung der Konkurrenz
: Manche Firmen unterstützen ein bestimmtes Open-Source-Projekt um ein Produkt der Konkurrenz zu untergraben, welches unter Umständen auch selber Open-Source sein kann. Den Marktanteil des Konkurrenten wegzufressen ist selten der einzige Grund ein Open-Source-Projekt zu unterstützen, kann aber eine Rolle spielen.

  Beispiel: http://www.openoffice.org/ (nein, das ist nicht der einzige Grund für die Existenz der Software, es ist aber zumindest teilweise eine Reaktion auf Microsoft Office).

Marketing
: Ihre Firma mit einer bekannten Open-Source-Anwendung zu assoziieren kann einfach gute Markenpflege sein.

Doppelte Lizenzierung
: *Doppelte Lizenzierung* bedeutet, Software unter einer traditionellen proprietären Lizenz für Kunden anzubieten, die es als Teil ihrer proprietären Anwendung verkaufen wollen. Gleichzeitig veröffentlicht man es unter einer freien Lizenz für diejenigen, die bereit sind, es unter den Open-Source-Bedingungen zu benutzen (siehe „Doppelte Lizenzierung“ im Kapitel Kapitel 9, *Lizenzen, Urheberrecht und Patente*). Wenn die Open-Source-Entwicklergemeinschaft aktiv ist, bekommt die Software die Vorteile vieler Anwender um die Entwicklung zu testen, die Firma bekommt dennoch Nutzungsgebühren um ihre Vollzeitentwickler zu unterstützen.

  Zwei bekannte Beispiele sind MySQL [http://www.mysql.com/], die Hersteller der gleichnamigen Datenbank-Software, und Sleepycat [http://www.sleepycat.com/], welche Distributionen und Support für die Berkeley-Datenbank anbietet. Beide sind nicht zufällig Datenbank-Firmen. Datenbank-Software neigt dazu eher in anderen Anwendungen integriert zu werden als direkt an Kunden vermarktet zu werden, also ist es sehr gut für das Modell der doppelten Lizenzierung geeignet.

Spenden
: Ein erfolgreiches Projekt kann manchmal wesentliche Beiträge, sowohl von Einzelpersonen als auch von Organisationen bekommen, allein durch einen Spendenknopf, oder den Verkauf von Waren mit ihre Marke wie Tassen, T-Shirts, Mousepads, usw. Hierbei sollte man vorsichtig sein: Wenn ihr Projekt Spenden annimmt, sollten Sie planen wie das Geld benutzt werden soll, *bevor* es auftaucht, und schreiben Sie ihre Pläne auf die Webseite des Projekts. Diskussionen über die Aufteilung von Geld verlaufen meistens friedlicher wenn es noch keins gibt; sollte es doch bedeutende Streitigkeiten über die Verwendung geben, ist es trotzdem besser es in einer eher akademischen Diskussion herauszufinden.

Das Geschäftsmodell eines Geldgebers ist nicht sein einziger Einfluss auf eine Open-Source-Gemeinschaft. Seine historische Beziehung spielt auch eine wesentliche Rolle: Hat die Firma das Projekt gegründet, oder tritt es einer bereits laufenden Entwicklung bei? So oder so muss sich der Geldgeber Glaubwürdigkeit verdienen, es ist deswegen auch nicht verwunderlich, dass letzteres etwas mehr Mühe erfordert. Die Organisation muss klare Ziele im Bezug auf das Projekt haben. Versucht sie eine Führungsposition zu behalten, oder will sie einfach eine Stimme in der Gemeinschaft, um diese zu führen aber nicht unbedingt die Richtung des Projekts vorzugeben? Vielleicht will die Firma auch einfach nur ein paar Entwickler parat haben, um Fehler für ihre Kunden beheben und Änderungen ohne große Umstände in die öffentliche Distribution einbinden zu können?

Behalten Sie diese Fragen beim lesen der nachfolgenden Richtlinien im Hinterkopf. Sie sollen für jede organisatorische Beteiligung an einem freien Software-Projekt gelten, da Sie es aber mit Menschen zu tun haben ist jedes Projekt einzigartig. Zu einen gewissen Grad werden Sie immer nach Gehör spielen müssen, die Entwicklung wird aber eher nach ihren Vorstellungen verlaufen, wenn Sie diese Prinzipien befolgen.

Langzeit-Entwickler

Behalten Sie Ihre Open-Source-Programmierer lange genug bei einem Projekt, um sich sowohl technische als auch politische Kompetenzen aneignen können – mindestens einige Jahre. Natürlich profitiert kein Projekt, ob Open-Source oder nicht vom häufigen Wechsel unter den Programmierern. Sich jedes mal neu einarbeiten zu müssen wäre in jeder Umgebung für Neulinge schwierig. Für Open-Source-Projekte ist die Strafe aber noch größer, da Entwickler, die ein Projekt verlassen nicht nur ihre Kenntnisse über den Code mitnehmen, sondern auch ihre Position in der Gemeinschaft sowie die dort aufgebauten menschlichen Beziehungen.

Die von einem Entwickler angeeignete Glaubwürdigkeit kann nicht übertragen werden. Das offensichtlichste Beispiel wäre wohl, dass Commit-Zugriff nicht von einem Entwickler zum Anderen vererbt werden kann (siehe „Liebe kann nicht mit Geld erkauft werden“ später in diesem Kapitel), wenn ein neuer Entwickler also noch keinen Commit-Zugriff hat, wird er bis dahin Patches einreichen müssen. Commit-Zugriff ist aber nur die messbarste Erscheinung für den verlorenen Einfluss. Ein langjähriger Entwickler kennt auch alle alten Streitigkeiten die immer wieder in Diskussionen in den Foren aufgeflammt sind. Ein neuer Entwickler ohne Erinnerung an solche Unterhaltungen, könnte versuchen diese Themen erneut anzusprechen, was zum Verlust der Glaubwürdigkeit Ihrer Organization führt; die anderen werden sich vielleicht wundern "ob die sich denn garn ichts behalten können"? Neue Entwickler werden auch kein politisches Gespür für die Persönlichkeiten im Projekt haben und werden nicht in der Lage sein die Richtung der Entwicklung so schnell oder reibungslos zu beeinflussen, wie ein alter Hase.

Lassen Sie Anfänger sich zur Einweisung, unter Beaufsichtigung, an dem Projekt beteiligen. Der neue Entwickler sollte vom ersten Tag sehr eng mit der öffentlichen Entwicklergemeinschaft arbeiten, angefangen mit Bugfixes und Aufräumarbeiten, um die Codebasis kennenzulernen und sich einen Ruf in der Gemeinschaft zu erarbeiten, er sollte jedoch keine langwierige und verstrickte Diskussionen über Codestruktur zünden. Währenddessen sollten ein Paar erfahrene Entwickler für Fragen zur bereitstehen, die jede Nachricht des Anfängers an die Verteiler, lesen sollten, sogar bei Threads denen sie sonst keine Beachtung schenken würden. Das wird der Gruppe helfen mögliche Steine auf dem Weg zu erkennen, bevor der Anfänger drüber stolpert. Private Anregung und Hinweise im Hintergrund können auch eine große Hilfe sein, besonders wenn der Anfänger es nicht gewohnt ist, dass sein Code der Kritik durch die Gemeinschaft unterworfen ist.

Wenn CollabNet einen neuen Entwickler einstellt, um an Subversion zu arbeiten, setzen wir uns zusammen und wählen ein paar offen stehende Fehler für die neue Person, um sich erst einmal die Krallen zu schärfen. Wir diskutieren den technischen Rahmen der Lösungen und weisen dann mindestens einen erfahrenen Entwickler an (öffentlich) den vom Neuling eingereichten Patch, kritisch unter die Lupe zu nehmen (alles für die Öffentlichkeit sichtbar). Normalerweise schauen wir uns den Patch nicht einmal an, bevor er auf dem zentralen Verteiler für Entwickler zu sehen ist, auch wenn wir es könnten, gäbe es dazu einen Grund. Das wichtige für den neuen Entwickler ist die öffentliche Überprüfung durchzulaufen, dabei den Codebase kennenzulernen und sich an Kritik von völlig Fremden zu gewöhnen. Wir versuchen es aber so abzustimmen, dass unsere Bewertung möglichst bald nach dem Patch kommt. Dadurch ist unsere Bewertung die erste auf dem Verteiler, was helfen kann, den Ton der nachfolgenden Bewertungen zu setzen. Es trägt auch zu der Idee bei, dass diese neue Person ernst genommen werden soll: Wenn andere sehen wie wir uns Zeit nehmen ausführliche Bewertungen zu schreiben, mit gründlichen Erklärungen und, wenn angemessen, Verweise auf die Archive, werden sie es als Schulungsform erkennen, was wahrscheinlich eine längerfristige Investition andeutet. Das kann sie positiver gegenüber dem Entwickler stimmen, zumindest soweit, dass sie sich ein wenig mehr Zeit nehmen, um Fragen zu beantworten und Patches zu bewerten.

Treten Sie als viele in Erscheinung

Ihre Entwickler sollten anstreben in den öffentlichen Foren als einzelne Beteiligte aufzutreten und nicht als monolithisches Firmenwesen. Das liegt nicht an irgend einer negativen Behaftung von monolithischen Unternehmen (naja, vielleicht schon ein bisschen, aber darum geht es nicht in diesem Buch). Einzelpersonen sind schlicht das einzige wofür Open-Source-Projekte strukturell gewappnet sind. Ein einzelner Beteiligter kann sich an Diskussionen beteiligen, Patches einreichen, sich Glaubwürdigkeit verschaffen, an Abstimmungen teilnehmen usw. Eine Firma kann das nicht.

Desweiteren verhindert man durch dezentralisiertes Verhalten die Bildung einer zentralisierten Opposition. Lassen Sie Ihre Entwickler sich auf der Mailingliste untereinander streiten. Ermutigen Sie häufige und öffentliche gegenseitig Überprüfung und Bewertung des Codes. Raten Sie davon ab, immer als Block abzustimmen, denn wenn sie das machen, werden andere das Gefühl bekommen allein aus Prinzip, einen Gegenpol organisieren zu müssen, um sie in Schach zu halten.

Es gibt einen Unterschied zwischen einer wirklich dezentralisierten Organisation und einfach als solches erscheinen zu wollen. In bestimmten Fällen kann es durchaus nützlich sein, ihre Entwickler an einem Strang ziehen zu lassen und sie sollten vorbereitet sein falls nötig, im Hintergrund zu koordinieren. Wenn beispielsweise ein Vorschlag gemacht wird, kann ihm durch frühzeitige Zustimmung anderer Entwickler auf die Sprünge geholfen werden, indem sie den Eindruck von zunehmenden Konsens vortäuschen. Andere spüren ein gewisses Momentum für den Vorschlag, der durch einen Einspruch ihrerseits aufgehalten würde. Deshalb werden Leute nur dann Einsprechen, wenn sie dazu einen guten Grund haben. Es gibt im Übrigen nichts verwerfliches daran, Zustimmung auf diese Art zu koordinieren, solange Einsprüche weiterhin ernst genommen werden. Wie sich diese private Übereinkunft öffentlich offenbart ist nicht unaufrichtig oder für das Projekt schädlich, nur weil sie vorher koordiniert wurde, so lange sie nicht dazu benutzt wird um Gegenargumente frühzeitig und abträglich auszustechen. Der Hintergrund ist lediglich bestimmte Personen aufzuhalten, die einfach nur Einwände machen um Fit zu bleiben; siehe „Je weicher das Thema, desto länger die Debatte“ im Kapitel Kapitel 6, *Kommunikation* für weiteres.

Seien Sie offen bezüglich Ihrer Absichten

Seien Sie so offen hinsichtlich Ihrer Motive wie Sie es können, ohne hierbei Geschäftsgeheimnisse offenzulegen. Wenn Sie in Ihrem Projekt eine bestimmte Funktion haben wollen, sagen wir, weil Ihre Kunden danach schreien, sprechen Sie auf den Mailinglisten offen darüber. Wenn die Kunden anonym bleiben wollen, was manchmal vorkommt, fragen Sie wenigstens, ob Sie sie als anonyme Beispiele anführen dürfen. Je mehr die öffentliche Entwicklergemeinschaft darüber weiß, *warum* Sie etwas wollen, desto eher werden sie Ihre Vorschlage akzeptieren.

Das widerspricht dem Instinkt – einfach anzueignen, aber schwer loszuwerden –, dass Wissen gleich Macht ist und Ihre Ziele anderen gegenüber bekannt zu geben, diesen mehr Kontrolle über Sie gibt. Der Instinkt wäre hier aber falsch. Indem Sie sich öffentlich für eine Funktion (oder ein Bugfix, oder was auch immer) aussprechen, haben Sie ja schon Ihre Karten auf den Tisch gelegt. Die einzige Frage ist, ob Sie die Gemeinschaft überzeugen können Ihr Ziel mit Ihnen zu teilen. Wenn Sie einfach sagen, dass Sie die Funktion haben wollen, aber keine konkreten Gründe dafür nennen wollen, stehen Sie auf schwachem Boden und Leute werden anfangen verborgene Absichten zu vermuten. Aber auch nur ein paar echte Szenarien zu nennen, die aufzeigen warum die vorgeschlagene Funktion wichtig ist, kann sich dramatisch auf die Debatte auswirken.

Um den Grund dafür zu erkennen, sollten Sie die Alternative bedenken. All zu oft sind Debatten über neue Funktionen oder Richtungen langwierig und ermüdend. Die Argumente die angebracht werden reduzieren sich meistens auf "Ich persönlich will X", oder das immer wieder beliebte "Meine langjähri-

gen Erfahrung als Software-Entwickler hat gezeigt, dass X für Nutzer äußerst wichtig ist / eine nutzlose Krause ist, die keinen zufrieden stellen wird". Solche Debatten werden dementsprechend durch Mangel an Daten über echte Nutzung weder verkürzt noch verlängert, sondern treiben immer weiter von der Praxis ab. Ohne irgend eine ausgleichende Kraft, wird das Endergebnis eher von dem Wortgewandtesten, dem Hartnäckigsten oder dem Ranghöchsten bestimmt.

Ihre Position als Organization mit reichlich Kundendaten, ermöglicht Ihnen genau solch eine ausgleichende Kraft bereitzustellen. Sie können ein Kanal für Informationen sein, die sonst keine Möglichkeit hätten zu der Gemeinschaft zu gelangen. Sie müssen sich nicht dafür schämen, dass diese Informationen Ihre Wünsche untermauern. Die meisten Entwickler haben für sich keine breite Erfahrung über die Nutzung Ihrer Software. Jeder Entwickler benutzt die Software auf seine eigene Art; über das Nutzungsverhalten anderer spekuliert er lediglich und verlässt sich auf seine Intuition. Tief im Inneren ist ihm das auch bewusst. Durch glaubwürdige Daten einer nicht unwesentlichen Anzahl an Nutzern geben Sie der Entwicklergemeinschaft etwas wie Sauerstoff. So lange Sie es richtig auslegen, wird sie es mit Begeisterung annehmen und der Wind wird zu Ihrem Gunsten wehen.

Der Schlüssel ist natürlich, es richtig zu präsentieren. Es reicht auf keinen Fall aus, nur weil Sie mit einer Menge Nutzern zu tun haben und weil diese eine gegebene Funktion brauchen (oder meinen zu brauchen), darauf zu bestehen Ihre Lösung zu implementieren. Statt dessen sollten sich Ihre ersten Nachrichten eher auf das gegebene Problem konzentrieren, als auf eine bestimmte Lösung. Beschreiben Sie sehr ausführlich die Erfahrungen Ihrer Kunden, bieten Sie die Ihnen bestmögliche Auswertung, sowie möglichst viele Lösungsansätze. Wenn Leute anfangen zu spekulieren, wie effektiv die verschiedenen Lösungen sind, können Sie mit Ihren Daten die verschiedenen Antworten untermauern oder entkräften. Sie werden vielleicht die ganze Zeit eine bestimmte Lösung im Hinterkopf haben, heben Sie es am Anfang aber nicht hervor. Das ist keine Täuschung, sondern das übliche Verhalten eines ehrlichen Vermittlers. Schließlich ist Ihr wahres Ziel die Lösung des Problems; eine Lösung ist lediglich das Mittel zum Zweck. Wenn Ihre bevorzugte Lösung wirklich die überlegene ist, werden andere Entwickler das irgendwann auch erkennen – und Sie von sich aus unterstützen, was viel besser ist, als die Implementierung durch Einschüchterung zu erzwingen. (Es besteht natürlich auch die Möglichkeit, dass jemand anderem eine bessere Lösung einfällt als Ihnen).

Das soll nicht heißen, dass Sie sich niemals zugunsten einer bestimmten Lösung äußern können. Sie müssen aber Geduld haben, damit die Analyse, die Sie bereits intern gemacht haben, auf dem öffentlichen Verteilern wiederholt wird. Schreiben Sie nicht etwas wie "Ja, wir haben das alles hier schon gehabt, aber das funktioniert so nicht und zwar aus diesen Gründen... Wenn man der Sache auf den Grund geht, dann ist die einzige Lösung..." Das Problem ist weniger der arrogante Ton, als vielmehr der Eindruck, dass Sie *bereits* eine unbestimmte (aber vermutlich große) Menge analytischer Ressourcen dem Problem hinter verschlossenen Türen gewidmet haben. Es lässt die Sache so erscheinen, als ob Anstrengungen unterwegs gewesen sind, und Entscheidungen vielleicht getroffen wurden, ohne die Einweihung der Öffentlichkeit, und das führt unweigerlich zu Verbitterung.

Sie wissen natürlich, wieviel Mühe Sie dem Problem intern gewidmet haben, und dieses Wissen ist in gewisser Hinsicht ein Nachteil. Ihre Entwickler haben dadurch eine etwas andere Einstellung als alle anderen auf den Mailinglisten, wodurch ihre Fähigkeit eingeschränkt wird, die Dinge aus der Sicht derjenigen zu sehen, die noch nicht über das Problem nachgedacht haben. Je früher Sie alle anderen dazu bringen können ihre Sicht einzunehmen, desto geringer werden die resultierende Auswirkungen sein. Diese Logik gilt nicht nur für einzelne technische Situationen, sondern auch für das breitere Mandat Ihre Ziele so klar wie möglich darzulegen. Das Unbekannte verursacht immer mehr Unruhe als das Bekannte. Wenn Menschen verstehen warum Sie etwas wollen, werden sie sich dabei wohl fühlen mit Ihnen zu reden, selbst wenn sie anderer Meinung sind als Sie. Wenn sie nicht herausbekommen was Sie bewegt, werden sie zumindest Zeitweise vom Schlimmsten ausgehen.

Sie werden natürlich nicht alles veröffentlichen können, und man wird es nicht von Ihnen erwarten. Alle Organisationen haben Geheimnisse; vielleicht haben Profitorientierte Organisationen mehr davon, aber

gemeinnützige haben sie auch. Wenn Sie einen bestimmten Kurs verfechten müssen, aber nichts über Ihre Gründe offenbaren können, dann bieten Sie einfach die Ihnen mit dieser Behinderung bestmöglichen Argumente. Finden Sie sich mit der Tatsache ab, dass Sie vielleicht nicht den gewünschten Grad an Einfluss bei der Diskussion haben. Das ist einer der Kompromisse die Sie eingehen, wenn die Entwicklergemeinschaft nicht auf Ihrer Gehaltsliste steht.

Liebe kann nicht mit Geld erkauft werden

Wenn Sie ein bezahlter Entwickler in einem Projekt sind, dann legen Sie frühzeitig die Richtlinien fest, was für Geld käuflich ist, und was nicht. Das bedeutet nicht, dass Sie es zweimal täglich in den Foren wiederholen müssen, um Ihre noble und unbestechliche Natur zu verdeutlichen. Es bedeutet lediglich, dass Sie auf Gelegenheiten achten sollten, um Spannungen zu entschärfen, die durch Geld entstehen *könnten*. Sie müssen nicht von vorhandenen Spannungen ausgehen; Sie müssen allerdings zeigen, dass es das Potential dazu gibt.

Ein perfektes Beispiel hierfür hat das Subversion-Projekt zu bieten. Subversion wurde im Jahr 2000 von CollabNet [http://www.collab.net/] gestartet, das von Anfang an als Arbeitgeber für mehrere Entwickler und der größte Geldgeber für das Projekt gewesen ist (Anmerkung: Ich bin einer davon). Bald nach Beginn des Projekts stellten wir einen weiteren Entwickler ein, Mike Pilato, um dem Projekt beizutreten. Wir hatten schon mit dem Programmieren angefangen und obwohl Subversion sicherlich noch in seiner Anfangsphase war, hatte es bereits eine Entwicklergemeinschaft mit wenigen einfachen Grundregeln.

Die Ankunft von Mike warf eine interessante Frage auf. Subversion hatte bereits ein Verfahren, nach dem neue Entwickler Commit-Zugriff bekamen. Zuerst reicht man ein paar Patches auf der Entwickler-Liste ein. Nachdem genügend Patches eingetroffen sind, damit anderen Committer sehen können, dass der neue Mitwirkende weiß was er macht, schlägt jemand vor, ihm den direkten Zugriff zu gewähren (dieser Vorschlag findet im Privaten statt, wie in „Committer“ beschrieben). Angenommen die Committer stimmen überein, schreibt einer von ihnen dem neuen Entwickler eine Mail in der er den direkten Commit-Zugriff auf das Projektarchiv des Projekts anbietet.

CollabNet hatte Mike ausdrücklich eingestellt, um an Subversion zu arbeiten. Bei denen die ihn kannten, zweifelte keiner an seinen Fähigkeiten als Programmierer oder seiner Bereitschaft, an dem Projekt zu arbeiten. Desweiteren hatten die freiwilligen Entwickler eine sehr gute Beziehung zu den Mitarbeitern von CollabNet und hätten wahrscheinlich keine Einwände, Mike einfach gleich am ersten Arbeitstag Commit-Zugriff zu geben. Wir wussten aber, dass dies einen Präzedenzfall sein würde. Wenn wir Mike von oben herab Zugriff auf das Projektarchiv gegeben hätten, käme das der Aussage gleich, dass CollabNet das Recht hätte, die Richtlinien des Projekts zu ignorieren, einfach nur weil es der größte Geldgeber war. Obwohl dieser Schaden nicht unbedingt gleich ersichtlich gewesen wäre, hätte es langsam dazu geführt, dass unbezahlte Entwickler sich ihrer Rechte beraubt fühlen: Andere müssen sich ihren Commit-Zugriff erarbeiten – CollabNet kann ihn sich kaufen.

Mike willigte also ein seine Arbeit bei CollabNet wie jeder andere freiwillige Entwickler anzufangen, ohne Commit-Zugriff. Er schickte Patches an die öffentlichen Foren, wo sie von allen überprüft und kritisch beurteilt werden konnten und wurden. Wir erklärten auch öffentlich unser explizites Vorgehen, um Missverständnisse zu vermeiden. Nach ein paar Wochen solider Aktivität von Mike, schlug jemand (ich kann mich nicht mehr erinnern ob es ein Entwickler von CollabNet war oder nicht) vor, ihm Commit-Zugriff zu gewähren und wurde wie von allen erwartet angenommen.

Diese konsequente Durchsetzung verschafft einem Glaubwürdigkeit, die man mit Geld nicht kaufen kann. Und Glaubwürdigkeit ist eine wertvolle Währung bei technischen Diskussionen: Es schützt einem vor spätere Angriffe auf seine Motive. Bei einer hitzigen Debatte, greifen Leute manchmal auf Angriffe ohne technischische Bedeutung zurück, um die Schlacht für sich zu entscheiden. Der Haupt-Geldgeber bietet durch seine tiefe Beteiligung und offensichtliche Sorgen über die Richtung die das Projekt nimmt

eine breiteres Ziel als die Meisten. Indem er gewissenhaft alle Richtlinien des Projekts von Anfang an wahrnimmt, stellt der Finanzier sich auf derselben Ebene wie alle anderen.

(Siehe auch den Blog von Danese Cooper bei http://blogs.sun.com/roller/page/DaneseCooper/20040916 für eine ähnliche Geschichte um Commit-Zugriff. Cooper war damals die "Open-Source-Diva" von Sun Microsystem's – ich glaube das war ihr offizieller Titel – und sie beschreibt in ihrem Blog, wie die Entwicklergemeinschaft von Tomcat Sun dazu brachte, die gleichen Richtlinien für Commit-Zugriff für seine eigenen Entwickler zu befolgen, wie für die Entwickler außerhalb von Sun.)

Weil Geldgeber nach den selben Regeln spielen müssen, wie alle anderen, lässt sich das Modell der gütigen Diktatur (siehe „Gütige Diktatoren“ im Kapitel Kapitel 4, *Soziale und politische Infrastruktur*) nur schwer anwenden, wenn Geld im Spiel ist, insbesondere wenn der Diktator für den Geldgeber arbeitet. Da eine Diktatur wenig Regeln hat, ist es schwierig für einen Geldgeber zu beweisen, dass er die Normen der Gemeinschaft befolgt, selbst wenn das der Fall ist. Es ist sicherlich nicht unmöglich, es erfordert aber einen Projektleiter, der in der Lage ist, die Dinge sowohl aus der Sicht der externen Entwickler als auch derer die für den Geldgeber arbeiten, zu sehen und sich entsprechend zu verhalten. Selbst dann, ist es wahrscheinlich eine gute Idee ein nicht diktatorisches Modell bereit zu halten, welches man bei Anzeichen von breiter Unzufriedenheit in der Gemeinschaft vorschlagen kann.

Auftragsarbeit

Mit Auftragsarbeit muss in freien Software-Projekten vorsichtig umgehen. Im Idealfall wollen Sie, dass die Arbeit des Auftragnehmers von der Gemeinschaft angenommen wird und in die veröffentlichte Version aufgenommen wird. Theoretisch würde es keinen Unterschied machen, wer der Auftragnehmer ist, solange er gute Arbeit macht und die Richtlinien des Projekts beachtet. Theorie und Praxis kann manchmal auch zusammenpassen: Ein komplett Fremder der mit einem guten Patch auftaucht, *wird* es im Allgemeinen schaffen, diese in die Software zu bekommen. Das Schwierige ist, dass es als komplett Fremder sehr schwer ist, einen guten Patch für eine nicht triviale Erweiterung oder Funktion zu produzieren; man muss es zuerst mit dem übrigen Teilnehmern diskutieren. Die Dauer dieser Diskussion, kann man nicht genau voraussehen. Wenn der Auftragnehmer nach Arbeitszeit bezahlt wird, werden Sie vielleicht mehr Bezahlen, als Sie erwartet haben; wenn er Pauschal bezahlt wird, kann es passieren, dass er länger arbeitet, als er es sich leisten kann.

Es gibt zwei Möglichkeiten das Problem zu umgehen. Man sollte vorzugsweise eine fundierte Vermutung über die Dauer der Diskussion machen, beruhend auf der vergangenen Erfahrung mit einem Puffer für Fehler, was dem Vertrag zugrunde liegt. Es hilft auch das Problem in möglichst viele Einzelteile zu spalten, um die einzelnen Brocken eher einschätzen zu können. Der andere Weg ist den Auftrag auf den eigentlichen Patch zu beschränken, unabhängig von seiner Aufnahme im Projekt. Es ist dadurch viel einfacher den Vertrag zu schreiben, hat aber den Nachteil, dass Ihnen dann der Klotz am Bein hängt, den Patch so lange zu pflegen, wie Sie auf die Software angewiesen sind, oder zumindest so lange bis Sie es schaffen eine gleichwertige Funktion in den Hauptzweig zu bekommen. Natürlich kann man selbst beim ersten Vorgehen nicht im Vertrag darauf bestehen, etwas über die Aufnahme im Projekt festzulegen, weil das den Verkauf von etwas bedeuten würden, was nicht zum Verkauf steht. (Was passiert wenn die Übrigen im Projekt sich entscheiden, die Funktion nicht zu unterstützen)? Der Vertrag kann allerdings eine gewisse angemessen und glaubwürdige Anstrengung beinhalten, um die Änderung von der Gemeinschaft angenommen zu bekommen, und sie danach im Projektarchiv einzuspielen. Wenn das Projekt Normen über Änderungen am Code festgelegt hat, kann der Vertrag auf diese verweisen und festlegen, das die Arbeit sich danach richten muss. In der Praxis verläuft das meistens positiv für alle Parteien.

Die beste Taktik für erfolgreiche Auftragsarbeit ist, einen der Entwickler des Projekts – vorzugsweise einen etablierten Beteiligten – als Auftragnehmer anzustellen. Das mag sich nach erkauftem Einfluss anhören, und nun ja, dass ist es auch. Es ist aber nicht so korrupt, wie es sich anhört. Der Einfluss den ein Entwickler innerhalb eines Projekts hat, hängt vorrangig von der Qualität seiner Arbeit, sowie sei-

nen Umgang mit anderen Entwicklern ab. Die Tatsache, dass er ein Vertrag hat, um bestimmte Sachen erledigt zu bekommen, hebt seinen Status in keiner Weise und es senkt ihn auch nicht, auch wenn Leute ihn vielleicht etwas vorsichtiger hinterfragen werden. Die meisten Entwickler würden ihre längerfristige Position in einem Projekt nicht riskieren um eine weitestgehend unbeliebte Funktion zu unterstützen. Tatsächlich sollte ein Teil von dem was Sie erhalten, wenn Sie einen solchen Entwickler einstellen, Ratschläge darüber sein, welche Änderungen wahrscheinlich von der Gemeinschaft angenommen werden. Sie bekommen auch eine gewisse Umstellung der Prioritäten im Projekt. Da die Prioritäten lediglich davon abhängen, wer Zeit hat an etwas zu arbeiten, sorgen Sie durch Entlohnung bestimmter Arbeiten dafür, dass diese eine etwas höhere Priorität bekommen. Unter erfahrenen Open-Source-Entwicklern ist diese Tatsache wohlbekannt, und zumindest manche werden der Arbeit eines Auftragnehmers Aufmerksamkeit widmen, einfach nur weil es den Anschein hat *fertig zu werden*, also wollen sie dabei helfen, dass es richtig gemacht wird. Sie schreiben dabei zwar vielleicht keinen Code, werden aber Entwürfe diskutieren und Kritik am Code üben, was alles sehr nützlich sein kann. Aus all diesen Gründen, sollte der Auftragnehmer am ehesten aus den Reihen der bereits am Projekt Beteiligten gezogen werden.

Dadurch stellen sich zwei Fragen: Sollten Verträge jemals privat sein? Und wenn sie es nicht sind, sollten Sie sich darüber sorgen machen, Spannungen in der Gemeinschaft zu verursachen indem Sie mit manchen der Entwickler Verträge geschlossen haben, und nicht mit anderen?

Verträge sollten wo möglich offen sein. Sonst kann das Verhalten des Beauftragten anderen in der Gemeinschaft komisch vorkommen – vielleicht gibt er bestimmten Funktionen plötzlich eine unerklärlich hohe Priorität, die ihn früher aber nicht interessierten. Wenn Leute ihn darauf ansprechen, wie soll er überzeugend antworten, wenn er nicht darüber reden kann, dass er beauftragt wurde, sie zu schreiben?

Gleichzeitig sollten weder Sie noch der Auftragnehmer so tun als ob andere ihre Vereinbarung als etwas besonderes behandeln sollten. All zu oft habe ich Auftragnehmer erlebt, die auf der Mailingliste mit der Einstellung auftreten, dass ihre Nachrichten ernster genommen werden sollten, einfach nur weil sie bezahlt werden. Diese Einstellung zeigt den anderen Teilnehmern, dass der Auftragnehmer den Vertrag an sich als das Wesentliche erachtet – im Gegensatz zu dem aus dem Vertrag *resultierenden* Code. Aus Sicht der anderen Entwickler ist der Code das einzig Wichtige. Technische Probleme sollten immer im Mittelpunkt stehen, nicht die Details darüber, wer von wem bezahlt wird. Einer der Entwickler in der Subversion-Gemeinschaft geht mit Auftragsarbeit besonders elegant um. Während er seine Änderungen am Code im IRC bespricht, macht er nebenbei Anmerkungen (oftmals in einer privaten Nachricht, also *privmsg* im IRC, an andere Entwickler), dass er für die Arbeit an diesem bestimmten Fehler oder Funktion bezahlt wird. Er gibt aber durchweg den Eindruck, dass er an dieser Änderung auch so arbeiten wolle, und dass er froh ist, dass das Geld es ihm ermöglicht. Ob er sagt für wen er arbeitet oder nicht, den Vertrag macht er nie zum eigentlichen Thema. Seine Anmerkungen darüber sind lediglich eine Zierde einer ansonsten technische Diskussion über die Lösung eines Problems.

Dieses Beispiel zeigt einen weiteren Grund offen über Verträge zu reden: es mag mehrere Organisationen geben, die bei einem Open-Source-Projekt Verträge beisteuern, und wenn jeder über die Arbeit des anderen bescheid weiß, können sie vielleicht ihre Ressourcen bündeln. Im obigen Fall, ist der größte Finanzier (CollabNet) in keinster Weise an diesen Einzelverträgen beteiligt. Aber mit dem Wissen, dass jemand anderes bestimmte Bugfixes fördert, wird CollabNet ermöglicht sich auf andere Fehler zu konzentrieren, und insgesamt die Effizienz im Projekt zu erhöhen.

Nehmen es andere Entwickler es den beauftragten Mitgliedern übel, dass sie für die Arbeit am Projekt bezahlt werden? Im allgemeinen nein, insbesondere wenn diejenigen, die bezahlt werden sowieso anerkannte, geachtete Mitglieder der Gemeinschaft sind. Keiner erwartet, dass Auftragsarbeit gleichmäßig auf die Beteiligten aufgeteilt wird. Leute verstehen wie wichtig dauerhafte Beziehungen sind: Die Ungewissheiten bei Auftragsarbeit sind derart, dass wenn man einen zuverlässigen Geschäftspartner gefunden hat, man nur widerwillig zu jemand anderen wechselt, nur der Gerechtigkeit halber. Sie können es sich so vorstellen: Wenn Sie das erste mal jemand wählen wird es bestimmt keine Beschwerden geben, denn schließlich mussten Sie ja *irgendjemand* wählen – Sie können schließlich nicht alle beauftragen. Wenn Sie später dieselbe Person ein zweites mal beauftragen, ist das nur vernünftig: Sie kennen ihn schon, der

letzte Auftrag verlief erfolgreich, warum sollten Sie ein unnötiges Risiko eingehen? Es ist deshalb ganz natürlich ein-zwei Leute in der Gemeinschaft zu haben an die man sich wenden kann, anstatt die Arbeit gleichmäßig aufzuteilen.

Kritik und Annahme von Änderungen

Die Gemeinschaft ist trotzdem wichtig für den Erfolg der Auftragsarbeit. Ihre Beteiligung beim Entwurf und der Beurteilung der Änderung darf nicht beiläufig geschehen. Sie müssen es als Teil der Arbeit auffassen und vollständig im Auftrag einbeziehen. Betrachten Sie die Kritik der Gemeinschaft nicht als ein Hindernis, das es zu überwinden gilt – sondern als kostenlose Platform für Entwürfe, Fragen und Antworten. Es ist ein Vorteil, den Sie entschlossen nutzen sollten, anstatt ihn lediglich hinzunehmen.

Fallbeispiel: Das CVS-Protokoll zur Passwort-Authentifizierung

1995 war ich an einer Partnerschaft für technische Unterstützung und Erweiterungen an CVS (das Concurrent Versions System; siehe http://www.cvshome.org/) beteiligt. Mein Partner Jim und ich waren damals informell für die Instandhaltung von CVS zuständig. Wir hatten aber nie sorgfältig darüber nachgedacht, wie wir mit der vorhandenen, größtenteils freiwilligen Entwicklergemeinschaft von CVS umgehen sollten. Unsere Erwartung war einfach Patches zu bekommen, die wir anwenden würden, und so war auch weitestgehend der Ablauf.

Damals konnte man CVS im Netzwerk nur über ein Fernzugriff Programm wie `rsh` betreiben. Man brauchte dasselbe Passwort für CVS wie für den Fernzugriff, was ein offensichtliches Sicherheitsrisiko darstellte und viele Organisationen abschreckte. Eine bedeutende Anlagebank beauftragte uns einen neuen Authentifizierungsmechanismus zu implementieren, damit sie ihr vernetztes CVS sicher mit ihren Außenstellen benutzen konnten.

Jim und ich nahmen den Vertrag an und fingen an ein Entwurf für das neue Authentifizierungssystem auszuarbeiten. Unser Entwurf war relativ einfach (die Vereinigten Staaten hatten damals Einschränkungen auf den Export von Kryptographischem Code, also hatte der Kunde Verständnis dafür, dass wir keine Starke Authentifizierung implementieren konnten). Da wir aber keine Erfahrung mit derartigen Protokollen hatten, machten wir dennoch einige grobe Fehler, die einem Experten sofort aufgefallen wären. Diese Ausrutscher wären mit Leichtigkeit erkannt worden, hätten wir uns die Zeit genommen hätten einen Vorschlag zu verfassen und den anderen Entwickler zur Überprüfung vorgestellt hätten. Uns kam es nie in den Sinn, die Entwickler auf dem Verteiler als Ressource zu betrachten und machten alleine weiter. Wir wussten, dass unsere Arbeit, egal wie sie aussah, wahrscheinlich angenommen werden würde, und – da wir nicht wussten was wir nicht wussten – machten wir uns nicht die Mühe, unsere Arbeit für alle sichtbar offenzulegen, d.h. häufig Patches abzuschicken, kleine, leicht verdauliche Änderungen an einem bestimmten Branch, usw. Das entstandene Authentifizierungsprotokoll war nicht sonderlich gut, und nach seiner Etablierung ließ es sich es natürlich aufgrund von Sorgen um die Kompatibilität nur sehr schwer verbessern.

Im Kern lag das Problem nicht an unserem Mangel an Erfahrung; wir hätten das Nötige mit Leichtigkeit lernen können. Das Problem lag an unserer Einstellung zur Entwicklergemeinschaft. Wir betrachteten die Annahme der Änderungen als eine Hürde die es zu überwinden galt, nicht als Ablauf um die Qualität der Änderungen zu verbessern. Wir waren zuversichtlich, dass jedwede Arbeit von uns angenommen würde (was auch geschah), und machten uns deshalb wenig Mühe um äußere Beteiligung.

Es ist offensichtlich, dass Sie bei der Suche nach einem Auftragnehmer auf die richtigen technischen Fähigkeiten und Erfahrung achten. Es ist aber auch wichtig jemanden zu wählen, der nachweislich mit den anderen Entwicklern in der Gemeinschaft eine konstruktive Zusammenarbeit pflegt. Sie bekommen dadurch mehr als nur eine Person; Sie bekommen einen Agenten, der in der Lage sein wird aus einem Netzwerk von Fachwissen zu schöpfen um robuste und wartbare Arbeit sicherzustellen.

Tätigkeiten neben dem Programmieren finanzieren

Programmieren ist nur ein Teil der Arbeit in einem Open-Source-Projekt, für Freiwillige im Projekt ist es der sichtbarste und glorreichste Teil. Leider können dadurch andere Tätigkeiten, wie die Dokumentation, formale Tests, usw. manchmal vernachlässigt werden, zumindest im Vergleich zu der Aufmerksamkeit die sie bei proprietärer Software oftmals erhalten. Unternehmen können das manchmal auszugleichen, indem Sie einen Teil ihrer internen Infrastruktur für die Software-Entwicklung den Open-Source-Projekten widmen.

Das Wesentliche für den Erfolg dieser Arbeit ist die Übersetzung zwischen den Abläufen in der Firma und denen in der Gemeinschaft. Sie ist nicht Mühelos: Oft passen beide nicht gut zusammen, und die Unterschiede können nur durch menschliche Eingriffe überwunden werden. Die Firma kann beispielsweise einen anderen Bugtracker verwenden, als das öffentliche Projekt. Selbst wenn beide die gleiche Software benutzen, werden die darin gespeicherten Daten sehr unterschiedlich sein, da die Anforderungen einer Firma am Bugtracking ganz andere sind, als die einer freien Software-Gemeinschaft. Eine Information die in einem Bugtracker anfängt, muss vielleicht auf den anderen übertragen werden, wobei vertrauliche Teile entfernt, oder in umgekehrter Richtung hinzugefügt werden müssen.

Die folgenden Abschnitte drehen sich um den Aufbau und die Instandhaltung solcher Brücken. Das Endergebnis sollte ein reibungsloser Betrieb im Open-Source-Projekt sein, die Investitionen der Firma sollten von der Gemeinschaft anerkannt werden, ohne dass das Gefühl entsteht unangemessen durch die Firma beeinflusst zu werden.

Qualitätssicherung

Bei der Entwicklung proprietärer Softwareist es üblich dass sich eine gesonderte Abteilung der Qualitätssicherung widmet: Nach Fehlern sucht, die Performance und Skalierbarkeit evaluiert, Schnittstellen, Dokumentation usw. überprüft. Diese Aktivitäten werden üblicherweise nicht so energisch von freiwilligen in einem freien Software-Projekt verfolgt. Das liegt teilweise an der Schwierigkeit Freiwillige für unrühmliche Tätigkeiten wie das Testen zu finden und zum Anderen daran dass man annimmt, dass eine große Nutzergemeinschaft auch eine gute Test-Abdeckung mit sich bringt. Performance und Skalierbarkeit sind sogar ein Gebiet wofür Freiwilligen oftmals sowieso nicht die nötige Hardware zur Verfügung steht.

Die Annahme, dass viele Nutzer auch viele Tester sind, ist nicht ganz ohne Grundlage. Es macht sicherlich wenig Sinn Personen einzustellen, die die Grundfunktionen der Software auf den üblichen Zielumgebungen überprüfen: Die dortigen Fehler werden ohnehin beim normalen Betrieb schnell von gewöhnlichen Nutzern gefunden. Da Nutzer aber lediglich versuchen ihre Arbeit zu erledigen, begegeben Sie sich nicht bewusst auf die Suche nach ungewöhnliche Grenzfällen in der Funktionalität der Software und werden wahrscheinlich bestimmte Fehler unentdeckt lassen. Bei einem Fehler der sich leicht umgehen lässt werden Sie sogar eher im Stillen diese Abhilfe implementieren ohne sich die Mühe zu machen den Fehler zu melden. Am heimtückischsten kann der Umgang Ihrer Kunden (die Quelle von *Ihrem* Interesse) mit der Software sein, die statistisch ganz anders aussehen kann als das Verhalten eines beliebig anderen Nutzers.

Eine professionelle Testgruppe kann solche Fehler genau so gut in freier, wie in proprietärer Software aufdecken. Die Herausforderung ist, die Ergebnisse der Tester der Öffentlichkeit in einer nützlichen Form mitzuteilen. Die Test-Abteilungen haben im Betrieb meistens ihre eigene Art Ergebnisse zu melden, mit firmenspezifischem Jargon, oder speziellem Fachwissen über bestimmte Kunden und ihre Datensätze. Solche Berichte wären auf einen öffentlichen Bugtracker unangemessen, sowohl wegen ihrer Form als auch aus Datenschutz Gründen. Selbst wenn die interne Bugtracking-Software Ihrer Fir-

ma die gleiche wäre wie im öffentlichen Projekt, kann es sein, dass die Betriebsverwaltung firmenspezifische Kommentare sowie Änderungen an den Metadaten der Vorfälle machen muss (zum Beispiel um die interne Priorität eines Vorfalls anzuheben, oder seine Lösung für einen bestimmten Kunden anzusetzen). Für gewöhnlich sind solche Anmerkungen vertraulich – manchmal werden sie nicht einmal dem Kunden gezeigt. Aber selbst wenn diese Daten nicht vertraulich sind, sind diese für das öffentliche Projekt uninteressant und deshalb sollte die Öffentlichkeit nicht von ihnen abgelenkt werden.

Die Meldung des *eigentlichen* Fehlers ist für die Öffentlichkeit dennoch wichtig. Tatsächlich ist eine Bug-Meldung von Ihrer Test-Abteilung in mancherlei Hinsicht wertvoller als die der üblichen Benutzer, da die Test-Abteilung nach Sachen Ausschau hält, die andere Nutzer nicht interessieren. Angesichts der Tatsache, dass Sie diese Fehler aus keiner anderen Quelle erfahren werden, sollten sie die Fehler unbedingt aufbewahren, und es dem öffentlichen Projekt zur Verfügung stellen.

Entweder kann die Abteilung zur Qualitätssicherung die Meldungen direkt in den öffentlichen Bugtracker eintragen, wenn sie sich dabei wohl fühlen, oder ein Vermittler (gewöhnlich einer der Entwickler) kann die internen Meldungen der Test-Abteilung zu dem Öffentlichen Tracker "übersetzen". Übersetzen bedeutet in diesem Zusammenhang den Bug so zu beschreiben, dass es keine Bezüge auf kundenspezifische Informationen hat (sofern der Kunde dem zustimmt, kann die Anleitung um den Fehler zu reproduzieren natürlich auch Kundendaten beinhalten).

Der Eintrag in den Tracker sollte vorzugsweise von der Abteilung zur Qualitätssicherung gemacht werden. So kann die Öffentlichkeit die Beteiligung Ihrer Firma besser sehen und würdigen: Nützliche Bug-Meldungen tragen ebenso zum guten Ruf Ihrer Organization bei wie jeder andere technische Beitrag. Es gibt freiwilligen Entwickler auch einen direkten Draht um mit der Test-Abteilung zu kommunizieren. Wenn diese Abteilung den Bugtracker beobachtet, kann ein Entwickler eine Änderung machen um z.B. einen Skalierbarkeits-Bug zu beheben (der Entwickler selber kann die Korrektur nicht überprüfen, da er nicht die nötigen Ressourcen hat), und anschließend dem Ticket eine Anmerkung anhängen, mit der Bitte an die Qualitätssicherung, zu überprüfen, ob es die gewünschte Wirkung hatte. Stellen Sie sich auf den Widerstand einiger Entwickler ein; Programmierer haben die Angewohnheit Qualitätssicherung allerhöchstens als ein notwendiges Übel zu erachten. Die Test-Abteilung kann das leicht überwinden, indem sie schwerwiegende Fehler findet und nachvollziehbare Tickets schreibt; wenn ihre Tickets andererseits nicht mindestens so gut sind, wie die von der übrigen Gemeinschaft, hat es keinen Sinn, dass sie direkt mit den Entwicklern zusammenwirken.

So oder so, sollte sobald es ein öffentliches Ticket gibt sollte das interne Ticket im Bezug auf technische Inhalte nur noch auf das öffentliche Verweisen. Der Betrieb kann weiterhin Anmerkungen firmenspezifischer Angelegenheiten bei Bedarf beifügen, sollte aber Informationen, die allen zur Verfügung stehen sollten, in das öffentliche Ticket schreiben.

Sie sollten sich bei diesem Verfahren auf einen höheren Aufwand einstellen. Die Pflege von zwei Tickets für einen Bug bedeutet natürlich mehr Arbeit als eins. Der Vorteil ist, dass viel mehr Programmierer das Ticket sehen werden und ihre Lösungen beitragen können.

Rechtliche Beratung und Schutz

Gesellschaften, ob profitorientiert oder nicht, sind fast die einzigen, die bei einem freien Software-Projekt für komplexe rechtliche Angelegenheiten irgendwelche Aufmerksamkeit aufbringen. Einzelne Entwickler verstehen oft die Nuancen verschiedener Open-Source-Lizenzen, haben im allgemeinen aber weder Zeit noch Ressourcen um Urheber-, Marken- und Patentrecht im Detail zu verfolgen. Wenn Ihre Firma eine Rechtsabteilung hat, kann sie einem Projekt helfen, indem sie den urheberrechtlichen Stand des Quellcodes überprüft und den Entwicklern hilft, mögliche Patent und markenrechtliche Angelegenheiten zu verstehen. Die genauen Ausprägungen, die diese Hilfe annehmen kann, wird in Kapitel 9, *Lizenzen, Urheberrecht und Patente* diskutiert. Die Hauptsache ist bei einer Kommunikation zwischen der Rechtsabteilung und der Entwicklergemeinschaft sicherzustellen, dass sie die äußerst unterschiedlichen Welten, aus denen beide Parteien kommen gegenseitig anerkennen. Gelegentlich können diese

beiden Gruppen aneinander vorbeireden, wenn sie von fachspezifischem Wissen ausgehen, welches die anderen Partei nicht hat. Es ist eine gute Strategie, eine Verbindungsperson zu haben (meistens ein Entwickler, oder ein Anwalt mit technische Fachkenntnisse) die so lange wie nötig zwischen beiden Seiten steht und übersetzt.

Dokumentation und Benutzerfreundlichkeit

Die Dokumentation und Benutzerfreundlichkeit sind beides wohlbekannte Schwachstellen in Open-Source-Projekten, obwohl ich denke, dass der Unterschied zu proprietärer Software in Bezug auf die Dokumentation oftmals hochgespielt wird. Die Erfahrung zeigt trotzdem, dass es Open-Source-Software meistens an einer erstklassigen Dokumentation, sowie Untersuchungen bezüglich ihrer Benutzerfreundlichkeit mangelt.

Wenn Ihre Firma helfen will, diese Lücken für ein Projekt zu füllen, dann sollten Sie wahrscheinlich am ehesten Leute einstellen, die üblicherweise *nicht* am Projekt mitentwickeln, aber dennoch in der Lage sein werden mit den Entwicklern produktiv zusammenzuarbeiten. Leute einzustellen die keine gewöhnlichen Entwickler sind, ist aus zwei Gründen gut: Erstens entziehen Sie dem Projekt dadurch keine Entwicklerzeit; zweitens sollten diejenigen die der Software am nächsten sind im Allgemeinen sowieso nicht die Dokumentation schreiben oder die Benutzerfreundlichkeit untersuchen, da sie Schwierigkeiten haben die Software aus der Sicht eines Außenstehenden zu betrachten.

Eine Kommunikation zwischen diesen beiden Parteien wird allerdings trotzdem immer nötig sein. Finden Sie Leute, die genügend technische Kenntnisse haben, um mit den Entwicklern zu kommunizieren, aber nicht so weit Experten mit der Software sind, dass sie kein Einfühlungsvermögen für gewöhnliche Benutzer haben.

Ein halbwegs erfahrener Benutzer ist wahrscheinlich die richtige Person um eine gute Dokumentation zu schreiben. Ich bekam nach der ersten Veröffentlichung dieses Buchs sogar eine E-Mail von einem Open-Source-Entwickler namens Dirk Reiners:

```
Eine Anmerkung im Bezug auf Geld::Dokumentation und
Benutzerfreundlichkeit: Als wir etwas Geld übrig hatten und uns
entschieden, dass eine Einleitung für Anfänger das wichtigste war,
stellten wir einen halbwegs erfahrenen Benutzer ein. Er war erst
kürzlich in das System eingewiesen worden, sodass er sich an seine
Probleme erinnern konnte, er hatte es aber daran vorbei geschafft und
konnte sie dementsprechend beschreiben. Er konnte dadurch etwas
schreiben, dass von den Hauptentwicklern nur wenige Korrekturen
bedurfte, bei Dingen die er nicht richtig aufgefasst hatte. Trotzdem
konnte er das 'Offensichtliche' abdeckten, dass die Entwicklern
übersehen hätten.

In seinem Fall war es sogar noch besser, da es seine Aufgabe gewesen
war einen Haufen anderer Personen (Studenten) in das System einzuführen,
also kombinierte er die Erfahrung vieler Personen, was etwas ist, dass
einfach nur ein glücklicher Zufall war und wahrscheinlich schwer in den
meisten Fällen zu erreichen ist.
```

Bereitstellung von Hosting/Bandbreite

Bei einem Projekt, welches nicht einer der freien Hosting-Pakete benutzt (siehe „Hosting-Pakete“ im Kapitel Kapitel 3, *Technische Infrastruktur*), kann die Bereitstellung von Server und Netzwerkverbindung – und besonders die Hilfe bei der System-Administration – eine wesentliche Unterstützung sein.

Selbst wenn das alles ist was Ihre Firma für das Projekt macht, kann es eine halbwegs effektive Art sein Karma in der Öffentlichkeit zu sammeln, auch wenn es Ihnen keinen Einfluss auf die Richtung des Projekts bringen wird.

Sie können wahrscheinlich einen Banner oder eine Anerkennung auf der Projekt-Seite erwarten, auf dem man Ihrer Firma für das Hosting dankt. Wenn Sie das Hosting so einrichten, dass die Webseite des Projekts auf Ihrer Domain ist, werden Sie alleine schon durch die URL eine etwas größere Assoziierung bekommen. Das wird die meisten Benutzer dazu bringen, zu denken, dass das Projekt *irgendetwas* mit Ihrer Firma zu tun hat, selbst wenn Sie gar nicht zu der Entwicklung beitragen. Das Problem ist, dass die Entwickler sich dieser Assoziation auch bewusst sind, und werden sich nicht sonderlich wohl dabei fühlen, das Projekt unter Ihrer Domain zu hosten, wenn Sie nicht mehr Ressourcen als lediglich Bandbreite zu dem Projekt beitragen. Schließlich gibt es heutzutage viele Orte für Hosting. Die Gemeinschaft mag irgendwann der Meinung sein, dass die angedeutete falsche Anerkennung nicht dem Komfort vom Hosting wert ist und das Projekt anderswo unterbringen. Wenn Sie also Hosting anbieten wollen, machen Sie es – planen Sie aber entweder sich eingehender zu beteiligen oder seien sie umsichtig darüber wieviel Beteiligung sie sich Zusprechen.

Marketing

Auch wenn die meisten Open-Source-Entwickler es nur ungern zugeben würden, funktioniert Marketing. Eine gute Marketingkampagne *kann* Aufmerksamkeit um ein Open-Source-Projekt entstehen lassen, selbst zu dem Grad, dass starrköpfige Entwickler unklare positive Gedanken über die Software haben, aus Gründen, die sie sich nicht ganz erklären können. Es ist nicht an mir die Dynamik des Wettrüstens beim Marketing um Allgemeinen zu untersuchen. Jede Gesellschaft die mit freier Software zu tun hat, wird irgendwann sich dabei vorfinden zu bedenken, wie sie sich, die Software, oder ihre Beziehung zu der Software vermarkten sollen. Die folgenden Ratschläge handeln darüber, wie Sie bei solch einer Bemühung häufige Stolpersteine vermeiden können; siehe auch „Öffentlichkeit“ im Kapitel Kapitel 6, *Kommunikation*.

Denken Sie daran, dass Sie beobachtet werden

Um die Entwicklergemeinschaft auf Ihre Seite zu behalten, ist es *sehr* wichtig nichts zu sagen, was nicht nachweislich wahr ist. Überprüfen Sie vorsichtig alle Behauptungen, bevor Sie diese in den Raum stellen, und geben Sie der Öffentlichkeit einen Weg Ihre Behauptungen zu überprüfen. Unabhängige Überprüfung von Tatsachen ist ein bedeutender Teil von Open Source, und es gilt für mehr als nur den Code.

Natürlich würde eh niemand Firmen raten, nicht überprüfbare Behauptungen aufzustellen. Bei Open-Source-Aktivitäten gibt es für gewöhnlich eine hohe Anzahl an Personen mit dem nötigen Wissen um Behauptungen zu verifizieren – Personen die wahrscheinlich auch einen breitbandigen Internetzugang haben und die richtigen sozialen Kontakte um Ihre Ergebnisse auf eine schädliche Art zu verbreiten, wenn Sie es wollen. Wenn Global Megacorp Chemical Industries einen Fluss verunreinigt, kann das nachgewiesen werden, aber nur von ausgebildeten Wissenschaftlern, die wiederum von den Wissenschaftlern von Global Megacorp angefochten werden können, wodurch die Öffentlichkeit verwirrt wird und sich nicht sicher ist, was sie denken soll. Im Gegensatz dazu ist Ihr Verhalten in der Open-Source-Welt nicht nur sichtbar und aufgezeichnet; es ist auch ein Leichtes für viele Leute es unabhängig voneinander zu überprüfen, ihre eigene Schlussfolgerungen zu ziehen und diese mittels Mundpropaganda zu verbreiten. Diese Kommunikationsnetzwerke sind bereits gelegt; sie sind das Wesentliche wodurch Open Source funktioniert, und sie können benutzt werden, um jede Information zu übertragen. Widerlegung ist für gewöhnlich schwer, wenn nicht sogar unmöglich, insbesondere wenn das was die Leute sagen war ist.

Es ist zum Beispiel in Ordnung von Ihrer Organization zu sagen, dass Sie "Projekt X gegründet" hat wenn das tatsächlich der Fall ist. Bezeichnen Sie sich aber nicht als die "Hersteller von X" wenn der

meiste Code von Fremden geschrieben wurde. Umgekehrt sollten Sie nicht behaupten, dass Sie eine zutiefst beteiligte Gemeinschaft freiwilliger Entwickler haben, wenn jeder in Ihr Projektarchiv hineinschauen und sehen kann, dass es wenige oder gar keine Änderungen gibt, die von außerhalb Ihrer Gesellschaft kommen.

Vor nicht all zu langer Zeit sah ich eine Meldung von einer sehr bekannten Computer-Firma, welche behauptete, dass Sie ein wichtiges Software-Paket unter einer Open-Source-Lizenz veröffentlichten. Als die ursprüngliche Ankündigung herausgegeben wurde, sah ich mir ihr nunmehr öffentliches Projektarchiv an und erkannte, dass es nur drei Revisionen beinhaltete. Mit anderen Worten, hatte sie den ersten Import des Quellcodes gemacht, seit dem war aber fast nichts geschehen. Das alleine für sich war nicht beunruhigend – schließlich hatten sie eben erst die Ankündigung gemacht. Es gab keinen Grund gleich vorne weg eine Menge Aktivität zu erwarten.

Etwas später machten Sie eine weitere Meldung. Dabei behaupteten sie folgendes (wobei der Name und die Versionsnummer durch Pseudonyme ausgetauscht wurden):

> *Wir freuen uns bekannt zu geben, dass nach rigorosen Tests durch die Singer-Gemeinschaft, Singer 5 für Linux und Windows für den Einsatz in produktiven Umgebungen bereit ist.*

Neugierig was die Community bei ihren "rigorosen Tests" aufgedeckt hatte, ging ich danach zurück zum Projektarchiv um seine Historie kürzlicher Änderung an zu sehen. Das Projekt war immer noch bei der Revision 3. Scheinbar hatten sie *keinen einzigen* Bug gefunden der es vor dem Release wert gewesen wäre behoben zu werden! In der Annahme, dass die Ergebnisse der Tests durch die Gemeinschaft anderswo aufgezeichnet worden sein müssen, untersuchte ich als nächstes den Bugtracker. Es gab genau sechs offene Meldungen, von denen vier bereits seit mehreren Monaten offen gewesen waren.

Das ist so natürlich kaum glaubwürdig. Wenn Tester auf einem großen und komplexen Stück Software für einige Zeit einhämmern, werden sie Fehler finden. Selbst wenn die Fixes für diese Bugs es nicht in die nächste Version schaffen, sollte man annehmen, dass durch das Testen irgendwelche Aktivität auf dem Versionsverwaltungssystem resultieren würde oder zumindest ein paar neue Bug-Meldungen. Allem Anschein nach war jedoch nichts seit der Ankündigung der Open-Source-Lizenzierung und der ersten Open-Source-Version passiert.

Die Sache ist nicht, dass die Firma über die Tests durch die Gemeinschaft gelogen hatte. Ich habe keine Ahnung ob dies der Fall war oder nicht. Sie waren aber völlig nachlässig darüber, wie sehr es danach *aussah* als ob sie am Lügen waren. Da weder das Versionsverwaltungssystem noch der Bugtracker irgend eine Andeutung über die angeblichen rigorosen Tests beinhaltete, hätte die Firma entweder von vorn herein die Behauptung nicht machen sollen oder einen klaren Verweis auf irgendein greifbares Ergebnis dieser Tests ("Wir haben 278 Bugs gefunden; klicken Sie hier für weitere Details") geben sollen. Letzteres hätte jedem die Möglichkeit gegeben, sich sehr schnell ein Bild über die Aktivität der Gemeinschaft zu machen. So wie es war, brauchte ich nur ein paar Minuten um heraus zu finden, dass was auch immer diese Gemeinschaft am testen war, es hatte keine Spuren an den üblichen Stellen zurück gelassen. Das ist keine große Mühe, und ich bin mir sicher, dass ich nicht der einzige war der sich diese gemacht hat.

Transparenz und Überprüfbarkeit sind natürlich auch ein wichtiger Teil der rechten Würdigung. Siehe „Anerkennung“ im Kapitel Kapitel 8, *Leitung von Freiwilligen* für weiteres hierüber.

Machen Sie konkurrierende Open-Source-Produkte nicht schlecht

Unterlassen Sie es, negative Ansichten über konkurrierende Open-Source-Software zu verkünden. Es ist völlig in Ordnung, negative *Tatsachen* zu benennen – also leicht überprüfbare Behauptungen von

der Art, wie man sie oft in Vergleichstabellen sieht. Negative Beschreibungen, von weniger rigoroser Natur sollte man aus zwei Gründen lieber vermeiden. Erstens machen Sie sich damit schuldig Flamewars anzufangen, welche von produktiven Diskussionen ablenken. Zweitens und noch wichtiger, kann es sich herausstellen, dass einige der freiwilligen Entwickler in *Ihrem* Project, auch an dem konkurrierenden Projekt arbeiten. Das ist wahrscheinlicher als es zunächst den Anschein haben mag: Die Projekte sind bereits in dem selben Anwendungs und Geschäftsbereich (deshalb stehen sie zu einander in Konkurrenz). So kann es passieren, dass die Entwickler die Fachwissen auf diesem Gebiet haben überall dort Beiträge leisten wo es anwendbar ist. Selbst wenn es keine direkte Überlappung bei den Entwicklern gibt, ist es wahrscheinlich, dass Entwickler in Ihrem Projekt zumindest mit denen aus verwandten Projekten vertraut sind. Ihre Möglichkeiten konstruktive persönliche Bekanntschaften zu pflegen könnten durch übermäßig negative Marketing-Botschaften eingeschränkt werden.

Auf konkurrierenden proprietären Produkten herumzuhacken scheint eher in der Open-Source-Welt akzeptiert zu sein, insbesondere wenn diese Produkte von Microsoft stammen. Ich persönlich verabscheue diese Neigung (obwohl es auch in diesem Fall nichts gegen sachliche Vergleiche auszusetzen gibt), nicht nur weil es unhöflich ist, sondern weil es gefährlich für ein Projekt ist, anzufangen ihren eigenen Hype zu glauben und dadurch zu ignorieren, inwiefern die Konkurrenz vielleicht tatsächlich überlegen sein mag. Im allgemeinen sollten Sie darauf aufpassen, welche Auswirkungen Marketing-Botschaften auf Ihre eigene Entwicklergemeinschaft haben mag. Manche mögen derart aufgeregt darüber sein, durch Marketing Rückenwind zu bekommen, dass sie ihre Objektivität über die wahren Stärken und Schwächen ihrer Software verlieren. Es ist normal, und sogar zu erwarten, dass die Entwickler einer Firma zu einem gewissen Grad Unberührtheit von Marketing-Behauptungen ausdrücken, selbst in den öffentlichen Foren. Sie sollten ganz klar diesen Marketing-Botschaften nicht direkt widersprechen (es sei denn sie sind tatsächlich falsch, obwohl man hoffen mag, dass soetwas vorher abgefangen werden sollte). Von Zeit zu Zeit kann es aber passieren, dass sie sich darüber lustig machen, um die restliche Entwicklergemeinschaft wieder auf den Teppich zu bringen.

Kapitel 6. Kommunikation

Die Fähigkeit klar zu schreiben ist vielleicht die Wichtigste, die man in einer Open-Source-Umgebung haben kann. Auf lange Sicht ist sie sogar wichtiger als Programmierbegabung. Ein sehr guter Programmierer mit schlechten Kommunikationsfähigkeiten kann nur eines nach dem anderen erledigen, und hat selbst dann vielleicht Schwierigkeiten, andere zu überzeugen. Ein schlechter Programmierer, der aber gut kommuniziert, kann viele Leute koordinieren und überzeugen, viele verschiedene Dinge zu machen , und hat dadurch einen wesentlichen Einfluss auf die Richtung und Dynamik des Projekts.

Es scheint keinen großen Zusammenhang zu geben, in die eine oder andere Richtung, zwischen der Fähigkeit guten Code zu schreiben und der Fähigkeit mit seinen Mitmenschen zu kommunizieren. Es gibt einen gewissen Zusammenhang zwischen der Fähigkeit zu Programmieren und technische Angelegenheiten gut zu beschreiben, aber das ist nur ein winziger Teil der Kommunikation in einem Projekt. Viel wichtiger ist die Fähigkeit mit seinem Publikum einfühlsam umzugehen, seine Nachrichten und Kommentare aus der Sicht anderer zu sehen, und andere dazu zu bringen ihre eigene Nachrichten mit einer ähnlichen Objektivität zu sehen. Gleichermaßen ist es wichtig zu bemerken, wenn ein bestimmtes Nachrichtenmedium oder eine Kommunikationsmethode nicht mehr gut funktioniert, vielleicht weil es nicht mit einer zunehmenden Anzahl an Nutzern skaliert, und sich dann die Zeit zu nehmen dagegen etwas zu tun.

Hiervon ist alles in der Theorie offensichtlich – was in der Praxis schwierig macht, ist dass in Umgebungen der Entwicklung freier Software eine verblüffend viele verschiedene Mechanismen gibt, sowohl was das Publikum angeht als auch bei der Kommunikation. Soll eine gegebener Gedanke in einer E-Mail an den Verteiler verfasst werden, als Anmerkung in dem Bugtracker oder als Kommentar in dem Quellcode? Wenn man eine Frage in einem öffentlichem Forum beantwortet, wie viel Wissen kann man von "dem Lesenden" erwarten, vor dem Hintergrund das derjenige der die Frage gestellt hat nicht der einzige ist der die Rückmeldung lesen könnte? Wie können die Entwickler eine konstruktive Verbindung mit den Nutzern aufrecht erhalten, ohne von Anfragen für Funktionen, fälschliche Bug-Meldungen, und allgemeinem Geschwätz überschwemmt zu werden? Woran kann man erkennen, wann ein Medium die Grenzen seiner Kapazität erreicht hat, und was man dagegen machen kann?

Lösungen zu diesen Problemen sind für gewöhnlich nur Teillösungen, da jede bestimmte Lösung letztendlich durch Wachstum oder Änderungen an der Struktur des Projekts obsolet gemacht werden kann. Sie sind oftmals auch *ad hoc*, da sie improvisierte Reaktionen auf dynamische Situationen sind. Alle Beteiligten müssen sich darüber im klaren sein, wann und wie Kommunikationen festgefahren werden können, und an Lösungen beteiligt sein. Leuten zu helfen das zu erreichen ist ein großer Teil der Verwaltung eines Open-Source-Projekts. Die folgenden Abschnitte handeln davon wie Sie Ihre eigene Kommunikation abwickeln, als auch wie Sie die Aufrechterhaltung der Kommunikationsmittel zur Priorität für alle im Projekt machen können.[1]

Du bist was du schreibst

Bedenken Sie folgendes: Das einzige was irgend jemand über Sie im Internet weiß, kommt von dem was Sie schreiben, oder was andere über Sie schreiben. Es mag sein, dass Sie als Person geistreich, scharfsinnig und charismatisch sind – wenn Ihre E-Mails aber ausschweifend und unstrukturiert sind, wird man von Ihnen annehmen, dass Sie wirklich so sind. Oder vielleicht sind Sie wirklich persönlich wirklich ausschweifend und unstrukturiert, aber keiner muss das je erfahren, wenn Ihre Nachrichten deutlich und informativ sind.

[1] Es hat im Bezug auf dieses Thema viele akademische Untersuchungen gegeben; siehe zum Beispiel *Group Awareness in Distributed Software Development* von Gutwin, Penner, und Schneider (diese waren ehemals online verfügbar, scheinen aber mittlerweile zumindest zeitweise verschwunden zu sein; benutzen Sie eine Suchmaschine um es zu finden).

Ihrem Schreiben Mühe zu widmen kann sich in großem Maße auszahlen. Der langjährige Hacker an freier Software Jim Blandy erzählt folgende Geschichte:

> Damals 1993, arbeitete ich für die Free Software Foundation, und wir machten gerade Beta-Tests der Version 19 von GNU Emacs. Wir machten ungefähr jede Woche einen Release, und Leute probierten es aus und reichten Bug-Meldungen bei uns ein. Es gab einen Kerl den keiner von uns vorher persönlich getroffen hatte, der aber tolle Arbeit leistete: Seine Meldungen waren immer klar und brachten uns direkt zum Problem, und wenn er selber einen Fix anbot, war er fast immer richtig. Er war einfach erstklassig.
>
> Bevor die FSF Code, der von jemand anderem geschrieben wurde benutzen kann, lassen wir sie etwas Papierkram erledigen, um ihre urheberrechtlichen Interessen an dem Code der FSF zuzuweisen. Einfach den Code von komplett Fremden einzusetzen lädt geradezu auf eine rechtliche Katastrophe ein.
>
> Also sandte ich dem Kerl die Formulare per E-Mail zu und sagte ihm: "Hier ist ein bisschen Papierkram den wir brauchen, es hat folgendes zu bedeuten, Du unterschreibst hier, dein Arbeitgeber den hier, und dann können wir anfangen deine Fixes einzusetzen. Vielen Dank."
>
> Er sickte mir eine Antwort zurück, indem er sagte, "Ich habe keinen Arbeitgeber."
>
> Also sagte ich, "In Ordnung, das macht nichts, lass es einfach von deiner Universität stempeln und schicke es zurück."
>
> Nach einer gewissen Zeit, schrieb er mir wieder zurück und sagte, "Naja, eigentlich... bin ich dreizehn Jahre alt und lebe noch bei meinen Eltern".

Da der Junge nicht wie ein Dreizehnjähriger schrieb, wusste keiner, dass er einer war. Im Folgenden sind ein paar Möglichkeiten, mit denen Ihr Schriftverkehr auch einen guten Eindruck hinterlassen kann.

Struktur und Formatierung

Tappen Sie nicht in die Falle alles so zu schreiben, als wäre es eine SMS. Schreiben Sie vollständige Sätze mit Punkt und Komma, und nutzen Sie wenn nötig einen Absatzwechsel. Am wichtigsten ist das bei E-Mails und anderen ausformulierten Nachrichten. Im IRC oder anderen ähnliche flüchtigen Foren, ist es im allgemeinen in Ordnung wenn man nicht auf Groß- und Kleinschreibung achtet, sich komprimiert ausdrückt, usw. Achten Sie nur darauf, dass Sie diese Gewohnheiten nicht auf formalere, langlebigere Plattformen übertragen. E-Mails, Dokumentation, Bug-Meldungen und andere Schriftstücke, für die eine längere Lebensdauer vorgesehen ist, sollten mit normaler Grammatik, Rechtschreibung und einer verständlichen Erzählstruktur geschrieben werden. Das heißt nicht, dass es an und für sich gut wäre, irgendwelche Regeln zu befolgen, sondern dass diese Regeln eben *nicht* willkürlich sind: Sie haben sich zu ihrer heutigen Form entwickelt, weil sie Text lesbarer machen, und Sie sollten sich aus diesem Grunde daran halten. Lesbarkeit ist nicht nur deshalb erwünscht, weil es mehr Menschen ermöglicht, Ihre Texte zu lesen, sondern weil Sie sich damit als eine Person darstellen, die sich die Zeit nimmt, auf eine klare Art zu kommunizieren: sprich, jemand der seinerseits Aufmerksamkeit verdient.

Insbesondere für E-Mail, haben sich erfahrene Open-Source-Entwickler auf folgende Konventionen geeinigt:

Schicken Sie E-Mails ausschließlich im Klartext, kein HTML, RichText, oder andere Formaten die nicht lesbar wären für Mail-Programme die Klartext verarbeiten. Formatieren Sie Ihre Zeilen so, dass sie um die 72 Spalten breit sind. Überschreiten Sie nicht 80 Spalten, was *de facto* zur Standardbreite für Terminals geworden ist (d.h. dass manche breitere Terminals benutzen werden, aber niemand schmalere).

Indem Sie Ihre Zeilen auf etwas *weniger* als 80 Spalten fassen, lassen Sie Platz für die Einfügung einige Stufen von Zitatkennzeichnungen in den Rückmeldungen anderer, ohne damit Zeilenumbrüche in Ihrem Text zu erzwingen.

Benutzen Sie echte Zeilenumbrüche. Manche Mail-Programme machen eine Art von scheinbarem Umbruch, bei dem Sie als Schreiber der E-Mail Zeilenumbrüche angezeigt bekommen, die nicht wirklich existieren. Wenn die E-Mail abgeschickt wird, kann es sein, dass sie nicht die Zeilenumbrüche aufweist, die Sie erwarten, und Ihr Text wird auf manchen Bildschirmen an den unmöglichsten Stellen umgebrochen werden. Wenn Ihr Mail-Programm solche Schein-Umbrüche bietet, suchen Sie nach der Einstellung, bei der Sie die echten Zeilenumbrüche sehen, während Sie E-Mails verfassen.

Wenn Sie Bildschirm-Ausgaben, Code-Abschnitte, oder anderen vorformatierten Text mit einbeziehen, setzen Sie diese eindeutig von dem Rest Ihres Textes ab, sodass selbst ein träges Auge leicht die Grenzen zwischen dem was Sie schreiben und dem was Sie zitieren erkennen können. (Ich hätte nie erwartet, je diesen Ratschlag zu schreiben, als ich mit diesem Buch anfing, ich habe aber eine gewisse Anzahl an Open-Source-Malinglisten gesehen, auf denen Leute Texte aus verschiedenen Quellen miteinander vermischen, ohne klarzustellen, was was ist. Die Auswirkungen sind sehr frustrierend. Es macht ihre Nachrichten wesentlich schwerer zu verstehen, und offen gesagt lässt es diese Personen auch ein wenig schlecht organisiert aussehen.)

Wenn Sie E-Mails von anderen zitieren, fügen Sie Ihre Stellungnahmen dort ein, wo sie am ehesten angemessen sind, an verschiedenen Stellen falls nötig, und schneiden Sie die Teile heraus, die Sie nicht verwendet haben. Wenn Sie einen kurzen Kommentar schreiben, welcher sich auf die ganze vorherige E-Mail bezieht, ist es in Ordnung einen *top-post* zu machen (also Ihren Kommentar über den zitierten Text zu stellen); ansonsten sollten Sie die relevanten Stellen des ursprünglichen Texts zitieren, gefolgt von Ihrer Rückmeldung.

Verfassen Sie die Betreff-Zeilen neuer E-Mails mit Sorgfalt. Es ist die wichtigste Zeile in Ihrer E-Mail, da es jeder anderen Person im Projekt erlaubt, zu entscheiden, ob sie mehr lesen soll oder nicht. Moderne Mailprogramme ordnen zusammenhängende E-Mails in Threads, die nicht nur durch die Überschrift definiert sein können, sondern auch durch andere E-Mail-Header (welche manchmal nicht angezeigt werden). Daraus folgt, dass wenn das Thema des Threads zu sehr abschweift, Sie die Überschriften Ihrer E-Mails entsprechend anpassen können – und sollten – wenn Sie antworten. Der Thread wird durch die anderen Header intakt bleiben, die neue Überschrift wird Leuten, die auf die Übersicht des Threads schauen, aber helfen zu erkennen, dass das Thema sich geändert hat. Genauso sollten Sie, wenn Sie wirklich ein neues Thema anreißen wollen, eine frische E-Mail schreiben, und nicht auf eine bereits vorliegende antworten, indem Sie die Überschrift ändern. Ansonsten würde Ihre neue E-Mail in denselben Thread einsortiert werden, aus dem heraus Sie antworten, und dadurch Leuten vormachen, es ginge um etwas, um das es tatsächlich nicht geht. Wieder wäre die Strafe nicht allein eine Verschwendung ihrer Zeit, sondern auch ein kleiner Kratzer in Ihrer Glaubwürdigkeit als jemand, der sicher im Umgang mit Kommunikationsmitteln ist.

Inhalt

Wohlformatierte E-Mails locken Leser an, aber erst Inhalte fesseln sie. Natürlich kann kein Satz festgelegter Richtlinien für guten Inhalt garantieren, es gibt aber ein paar Prinzipien die ihn etwas wahrscheinlicher machen.

Machen Sie Ihren Lesern die Sache leicht. Es gibt Unmengen an Information die in jedem beliebigen aktiven Open-Source-Projekt herumschwirren, Sie können nicht erwarten dass Ihre Lesern mit den meisten davon vertraut wären – tatsächlich können Sie von Ihren Lesern auch nicht erwarten, dass sie wüssten, wie sie sich über diese Dingen informieren können. Wo immer möglich, sollten Ihre Nachrichten Informationen so bereit stellen, wie es für die Leser am bequemsten ist. Wenn Sie zwei zusätzliche Minuten damit verbringen müssen, die URL zu einem bestimmten Thread aus den Archiven der Mailingliste heraus zu graben, um es den Lesern Ihrer E-Mail zu ersparen, ist es das wert. Wenn Sie 5 bis 10

zusätzliche Minuten damit verbringen, die Ergebnisse eines komplexen Threads zusammen zu fassen, um Leuten einen Kontext zu geben, in dem sie Ihre Nachricht verstehen können, dann tun Sie das. Sehen Sie es einmal so: Je erfolgreicher ein Projekt ist, desto höher ist das Leser/Autoren-Verhältnis, egal auf welcher Plattform. Wenn jede Nachricht von Ihnen *n* Personen gelesen wird, dann wird bei zunehmendem *n* der Wert zunehmen, sich dem zusätzlichen Zeitaufwand auszusetzen, um ihn anderen zu ersparen. Und wenn die Leute sehen, dass Sie sich selbst diese Regeln auferlegen, werden sie ihre eigene Kommunikationen dem anpassen. Das Ergebnis ist im Idealfall eine Zunahme der allgemeinen Effizienz des Projekts: Wenn es die Wahl gibt zwischen dem Aufwand von *n* Personen und dem einer Person, wird das Projekt letzteren vorziehen.

Meiden sie Übertreibungen. Hochspielungen in Online-Nachrichten ist ein klassischer Fall von Wettrüsten. Zum Beispiel könnte eine Person, die einen Bug meldet, glauben, dass die Entwickler diesem nicht genügend Aufmerksamkeit aufbringen werden, und so wird sie den Bug als schwerwiegenden Fehler beschreiben, ein Problem welches sie (und all ihre Freunde/Mitarbeiter/Verwandte) daran hindert die Software produktiv zu nutzen, obwohl es in Wirklichkeit nur ein kleines Ärgernis ist. Übertreibungen beschränken sich aber nicht nur auf die Nutzer – Programmierer machen bei technischen Diskussionen oft das Gleiche, insbesondere wenn sich die Auseinandersetzung mehr um eine Geschmackssache dreht als um Korrektheit:

> "Das zu machen, würde den Code völlig unlesbar machen. Ihn zu warten, würde zu einem Albtraum, im Vergleich dem Vorschlag von H. Mustermann..."

Der gleiche Gedanke wird sogar *stärker* wenn man ihn weniger scharf formuliert:

> "Das geht, aber ich denke es ist bezüglich Lesbarkeit und Wartbarkeit nicht so optimal. Der Vorschlag von H. Mustermann vermeidet diese Probleme, da hier..."

Sie werden Übertreibungen nicht völlig vermeiden können, und im Allgemeinen ist das auch nicht nötig. Im Vergleich zu anderen Formen der Fehlkommunikation, ist Übertreiben nicht für die Allgemeinheit schädlich – es schadet hauptsächlich den Ausübenden. Die Empfänger können kompensieren, es ist nur dass der Sender mit jeder Nachricht ein wenig an Glaubwürdigkeit verliert. Sie sollten deshalb im Interesse von Ihrem Einfluss im Projekt versuchen in die moderatere Richtung zu gehen. Auf diese Art werden Leute Sie ernst nehmen, wenn Sie *wirklich* einen wichtigen Hinweis machen müssen.

Bearbeiten Sie zwei mal. Bei jeder Nachricht, die länger ist als ein mittellanger Absatz, sollten Sie, nachdem Sie der Meinung sind, dass sie fertig ist, diese von oben bis unten erneut durchlesen, bevor Sie sie versenden. Dieser Ratschlag wird jedem bekannt vorkommen, der einmal an einem Schreibkurs teilgenommen hat, ist aber bei Online-Diskussionen besonders wichtig. Da das Verfassen von Online-Nachrichten von ständigen Unterbrechungen begleitet wird (während Sie schreiben, müssen Sie vielleicht in andere E-Mails nachschauen, bestimmte Webseiten besuchen, einen Befehl ausführen, um Diagnosmeldungen abzugreifen, usw.), passiert es besonders schnell, dass man den erzählerischen Faden verliert. Nachrichten, die mit Unterbrechungen verfasst und vor dem Versenden nicht überprüft wurden, sind, zum Verdruss der Autoren (möchte man zumindest hoffen), oft als solche erkennen. Nehmen Sie sich die Zeit zu überprüfen, was Sie abschicken. Je besser der strukturelle Zusammenhalt Ihrer Nachrichten, desto mehr werden Sie gelesen.

Tonfall

Nachdem Sie tausende Nachrichten geschrieben haben, werden Sie bemerken, dass Ihr Schreibstil zum knappen neigt. Das scheint in den meisten technischen Foren der Normalfall zu sein, und an und gibt es daran nichts falsches. Ein Grad an Knappheit, welcher im normalen sozialen Umgang unzumutbar wäre ist für Hacker freier Software einfach der Standard. Hier ist eine vollständig zitierte Reaktion, welche ich einmal von einem Mailverteiler über freie Content-Management-Software erhielt:

```
Können Sie möglicherweise etwas näher auf die Probleme auf die Sie
gestoßen sind, usw. eingehen?

Desweiteren:

Welche Version von Slash benutzen Sie? Das konnte ich aus Ihrer
ursprünglichen Nachricht nicht erkennen.

Wie genau haben Sie den Build des apache/mod_perl-Quellcode ausgeführt?

Haben Sie den Apache-2.0-Patch ausprobiert, über den auf slashcode.com
berichtet wurde?

  Shane
```

Das ist jetzt mal knapp! Keine Begrüßung, abgesehen vom Namen keine Abmeldung, und die Nachricht selbst ist lediglich eine Aneinanderreihung von Fragen die so kompakt wie möglich formuliert sind. Sein einziger Satz mit einer Aussage war eine implizite Kritik an meine ursprüngliche Nachricht. Trotzdem war ich glücklich darüber die Nachricht von Shane zu sehen, und ich fasste die Knappheit seiner Antwort nicht als ein Anzeichen für irgend etwas anderes als das er eine beschäftigte Person ist. Alleine die Tatsache, dass er Fragen stellte, anstatt meine Nachricht zu ignorieren bedeutete, dass er bereit war etwas Zeit für meinem Problem aufzubringen.

Werden alle Leser auf diesen Schreibstil positiv reagieren? Nicht unbedingt; es kommt auf die Person und den Kontext an. Wenn Jemand zum Beispiel eben erst geschrieben hat das sie einen Fehler gemacht hat (vielleicht hat sie einen Bug geschrieben), und Sie aus vergangener Erfahrung wissen, dass diese Person dazu neigt etwas Unsicher zu sein, auch wenn Sie trotzdem noch eine knappe Antwort schreiben, sollten Sie es durch etwas aufwiegen was ihre Gefühle anerkennt. Der größte Teil Ihrer Antwort mag kurz gehalten sein, eine Analyse der Situation aus Sicht eines Ingenieurs, so kurz wie Sie wollen. Melden Sie sich aber zum Schluss mit etwas ab, dass es nicht als Kälte aufgefasst werden soll. Wenn Sie zum Beispiel der Person eben erst eine Unmenge an Ratschläge gegeben haben wie sie einen Bug beheben soll, dann melden Sie sich mit "Viel Glück, <hier Ihr Name>" um anzudeuten, dass Sie ihnen gutes wünschen und nicht Sauer sind. Ein strategisch platzierter Smiley oder ein anderer Hinweis auf die Gefühlslage kann oft auch ausreichen um den Gesprächspartner zu beruhigen.

Es mag komisch erscheinen derart auf die Gefühle des Beteiligten zu achten, als darauf was sie sagen, aber um es kahl zu sagen, beeinflussen Gefühle die Produktivität. Gefühle sind auch aus anderen Gründen wichtig, selbst wenn wir uns aber ausschließlich auf nutzungsbezogene Gebiete zu beschränken, können wir anmerken, dass unglückliche Menschen schlechtere Software schreiben, und weniger davon. Angesichts der begrenzten Natur der meisten elektronischen Medien, wird es jedoch keinen offensichtlichen Hinweis darauf geben, in welcher Stimmung sich die Person befindet. Sie werden eine wohlbegründete Vermutung darüber anstellen müssen basierend auf a) wie die meisten Menschen sich in einer solchen Situation fühlen, und b) was Sie über diese Person aus vergangenen Dialogen wissen. Manche Menschen bevorzugen eine eher unpersönliche Einstellung, und behandeln all gleichermaßen nach ihrer oberflächlichen Erscheinung, wobei die Idee die dahintersteckt ist, dass wenn die Beteiligte nicht offen sagt, dass sie sich irgendwie fühlt, hat man kein Recht sie so zu behandeln als ob das der Fall wäre. Ich mag diese Herangehensweise aus mehreren Gründen nicht. Erstens, verhalten sich Menschen im echten Leben nicht so, also warum sollten Sie es online machen? Zweitens, da die meisten Interaktionen in öffentlichen Foren stattfinden, neigen Leute dazu sich noch mehr zurück zu halten als das im privaten der Fall wäre. Um es genauer zu sagen, sie sind oft bereit auf andere gerichtete Gefühle auszudrücken, wie Dankbarkeit oder Unmut, nicht jedoch nach innen gerichtete Gefühle wie Unsicherheit oder Stolz. Trotzdem arbeiten die meisten Menschen besser, wenn sie wissen, dass andere über ihre Verfassung im klaren sind. Indem Sie auf kleine Hinweise achten, können Sie die meiste Zeit für gewöhnlich richtig raten, und andere Personen motivieren weiterhin in größerem Maße beteiligt zu bleiben als es sonst der Fall wäre.

Damit meine ich natürlich nicht, dass Sie die Rolle des Gruppentherapeuten einnehmen sollen, der andauernd jeden dabei helfen soll, sich über seine Gefühle im klaren zu sein. Indem Sie aber sorgfältig auf Muster im langfristigen Verhalten von Personen achten, werden Sie ein Gespür für sie als Individuen bekommen, selbst wenn Sie nie von Angesicht zu Angesicht treffen. Und indem Sie auf den Ton Ihrer Nachrichten achten, können Sie einen überraschenden Einfluss darauf haben wie sich andere fühlen, was letztendlich dem Projekt zugute kommt.

Unhöflichkeiten erkennen

Eine der bestimmenden Eigenschaften der Open-Source-Kultur ist ihre ausgeprägte Auffassung darüber, was unhöflich ist und was nicht. Obwohl die unten beschriebenen Grundsätze nicht alleine für die Entwicklung freier Software gelten, oder auch für Software im allgemeinen – sie wären jedem bekannt der im Bereich der Mathematik, der technischen Wissenschaften oder im Ingenieurswesen arbeitet – ist freie Software, mit seinen offenen Grenzen und ständigen Fluss an Einwanderern, eine Umgebung in der diese Grundsätze besonders häufig von Personen begegnet wird, die mit Ihnen nicht vertraut sind.

Lass uns damit anfangen, was *nicht* unhöflich ist:

Technische Kritik, selbst direkte und ungepolsterte ist nicht unhöflich. Es kann sogar eine Art von Kompliment sein: Der Kritiker sagt implizit, dass die Zielperson es wert ist ernst genommen zu werden und Zeit mit auf zu bringen. Das heißt, je attraktiver es gewesen wäre die Nachricht von jemand zu ignorieren, desto eher ist es ein Kompliment sich die Zeit zu nehmen es zu kritisieren (natürlich ausgeschlossen ist wenn die Kritik zu einem persönlichen Angriff oder einer anderen Form von Unhöflichkeit verfällt).

Knappe, ungeschminkte Fragen, wie die in der vorhin zitierten E-Mail von Shane an mich, sind auch nicht unhöflich. Fragen die in einem anderen Kontext kalt oder rhetorisch, selbst verspottend erscheinen könnten, sind oft ernst gemeint, und haben keinen Hintergedanken außer die Informationen so schnell wie möglich zu herauszulocken. Die berühmte Frage vom technischen Support "Ist Ihr Computer angeschlossen?" ist ein klassisches Beispiel hierfür. Die Person von Support muss wirklich wissen, ob Ihr Computer angeschlossen ist, und nach den ersten paar Tagen bei dieser Arbeit, ist sie müde geworden höflichen Schmeicheleien ihren Fragen voranzustellen ("Entschuldigen Sie, ich möchte lediglich vorher ein paar einfache Fragen stellen, um ein paar Möglichkeiten aus dem Weg zu räumen. Manche davon mögen sich ziemlich einfach anhören, haben Sie aber bitte Nachsicht..."). Mittlerweile macht sie sich aber nicht mehr die Mühe mit der Höflichkeit, sie fragt einfach gerade heraus: Ist es angeschlossen oder nicht? Die gleichen Fragen werden andauernd auf den Mailverteilern freier Software gestellt. Die Absicht ist nicht, den Empfänger zu beleidigen, sondern schnell einige der offensichtlichsten (und möglicherweise häufigsten) Erklärungen auszuschließen. Empfänger die das verstehen und entsprechend reagieren gewinnen Pluspunkte eine aufgeschlossene Sicht ohne Widerrede eingenommen zu haben. Es ist einfach ein Aufeinandertreffen verschiedener Kulturen, und nicht die Schuld von irgendjemandem. Erklären Sie freundlich, dass Ihre Frage (oder Kritik) keine versteckte Bedeutung hatte; es war lediglich gedacht die Information so schnell und effizient wie möglich zu bekommen (oder übertragen) und sonst nichts.

Was ist als Unhöflich?

Nach dem gleichen Prinzip mit der detaillierte technische Kritik als Kompliment aufgefasst werden kann, kann das Weglassen von hochwertiger Kritik eine Art von Beleidigung bedeuten. Ich meine nicht nur die Arbeit von jemandem zu ignorieren, sei es ein Vorschlag, eine Änderung am Code oder ein die Meldung von einem Issue oder sonst etwas. Wenn Sie nicht vorher explizit eine detaillierte Antwort versprochen haben, ist es gewöhnlich in Ordnung einfach überhaupt nicht zu reagieren. Man wird annehmen, dass Sie einfach keine Zeit hatten etwas zu sagen. Wenn Sie aber *doch* eine Antwort geben, sollten Sie nicht knausern: nehmen Sie sich die Zeit die Sachen wirklich zu untersuchen, geben Sie konkrete Beispiele an wo angemessen, wühlen Sie in den Archiven herum um verwandte Nachrichten zu finden, usw. Oder wenn Sie nicht die Zeit haben eine solche Mühe aufzubringen, aber trotzdem irgend eine kur-

ze Antwort geben müssen, dann erklären Sie den Defizit offen in Ihrer Nachricht ("Entschuldigung, ich denke es gibt hierzu eine Meldung, hatte aber leider nicht die Zeit um danach zu suchen"). Die Hauptsache ist die kulturelle Norm anzuerkennen, entweder indem man sie erfüllt, oder offen zuzugeben, dass man ihr dieses mal nicht gerecht geworden ist. In beiden Fällen wird die Norm verstärkt. Der Norm aber nicht gerecht zu werden und gleichzeitig nicht zu erklären *warum* , Sie es nicht geschafft haben Ihnen gerecht zu werden, ist das gleiche, als ob Sie sagen würden, dass das Thema (und die daran Beteiligten) nicht viel Ihrer Zeit wert war. Es ist besser zu zeigen, dass Ihre Zeit wertvoll ist, indem Sie sich kurz halten als indem Sie faul sind.

Es gibt viele andere Formen der Unhöflichkeit, die nicht nur auf Open Source Entwicklung zutrifft und der gesunde Menschenverstand hilft in der Regel sie zu vermeiden. Siehe auch „Unhöflichkeit im Keim ersticken“ im Kapitel Kapitel 2, *Der Einstieg*, wenn Sie es noch nicht gemacht haben.

Gesicht zeigen

Es gibt einen Bereich im menschlichen Gehirn, der speziell der Erkennung von Gesichtern gewidmet ist. Es wird als "fusiformes Gesichtsareal" (en. fusiform face area) bezeichnet und seine Fähigkeiten sind größtenteils angeboren und nicht angelernt. Es hat sich herausgestellt, dass die Erkennung von Gesichtern eine derart wichtige Fähigkeit ist um zu überleben, dass wir spezielle Hardware dafür entwickelt haben.

Internet basierende Zusammenarbeit ist deshalb psychologisch etwas merkwürdig, da es eine enge Mitarbeit zwischen Menschen erfordert, die fast nie die Gelegenheit bekommen sich gegenseitig mit den intuitivsten Methoden zu identifizieren: erstens durch Gesichter, aber auch durch Stimme, Haltung, usw. Um das zu kompensieren, sollten Sie versuchen, überall den selben *Bildschirm-Namen* verwenden. Es sollte der vordere Teil Ihrer E-Mail-Adresse sein (der Teil vor dem @-Zeichen), Ihr Nutzername im IRC, der Name Ihres Kontos im Projektarchiv, im Bug Tracker usw. Dieser Name ist Ihr online "Gesicht": Ein Kürzel um Sie zu identifizieren, welches manche der gleichen Funktionen erfüllt wie Ihr Gesicht, auch wenn es leider nicht die eingebaute Hardware im Gehirn anregt.

Der Bildschirm-Name sollte eine intuitive Permutation Ihres echten Namens sein (meiner, zum Beispiel ist "kfogel"). In manchen Situationen wird es sowieso von Ihrem kompletten Namen begleitet, zum Beispiel im Kopf einer E-Mail:

```
Von: "Karl Fogel" <kfogel@irgendeinedomain.com>
```

Tatsächlich gibt es zwei Sachen in dem Beispiel auf die man achten sollte. Wie vorhin erwähnt, gleicht der Bildschirm-Name dem echten auf eine intuitive Art. Der echte Name ist aber auch *echt*. Also kein erfundener wie:

```
Von: "Super Hacker" <superhacker@irgendeinedomain.com>
```

Es gibt einen berühmten Cartoon von Paul Steiner, in der Ausgabe vom 5 Juli 1993 der Zeitung *The New Yorker*, welches einen Hund an einem Terminal zeigt, der zu einem anderen verschwörerisch herunterschaut und sagt: "Im Internet weiß keiner, dass du ein Hund bist". Diese Denkart ist es wahrscheinlich, die hinter einer Vielzahl an selbstverherrlichenden soll-wohl-cool-sein Online-Identitäten liegen, die Leute sich selber geben – als ob die Leute glauben würden man wäre *tatsächlich* ein toller Hacker wenn man sich "Super Hacker" nennt. Tatsache bleibt aber: Selbst wenn keiner weiß das Sie ein Hund sind, sind Sie immer noch ein Hund. Eine glorreiche Online-Identität beeindruckt nie die Leser. Statt dessen gibt es ihnen den Eindruck, als würden Sie eher auf das Erscheinungsbild achten als auf den Inhalt, oder dass Sie einfach nur unsicher sind. Benutzen Sie Ihren echten Namen für alle Dialoge, oder wenn Sie aus irgend einem Grund anonym bleiben müssen, dann erfinden Sie einen Namen der sich wie ein gewöhnlicher anhört und bleiben Sie ab da bei ihm.

Es gibt noch ein paar weitere Sachen die Sie machen können um Ihr Online-Gesicht attraktiver zu machen, abgesehen davon, dass Sie es konsistent halten. Wenn Sie einen Titel haben (wie "Doktor" oder "Professor"), sollten Sie nicht mit Ihm herum stolzieren, oder auch nur erwähnen, außer wenn es direkt für die Diskussion bedeutend ist. Im Allgemeinen neigt die Kultur von Hacker und der freien Software dazu, das vorzeigen von Titel als etwas ausschließendes und ein Zeichen von Unsicherheit zu betrachten. Es ist in Ordnung wenn Ihr Titel als Teil Ihrer Signatur am ende jeder E-Mail die Sie abschicken erscheint, benutzen Sie es jedoch niemals als um Ihre Position in einer Diskussion zu verstärken – der Versuch wird garantiert zurückschlagen. Sie wollen, dass man die Person respektiert, nicht den Titel.

Wo wir gerade von Signaturen sprechen: halten Sie sie kurz und geschmackvoll, oder lassen Sie sie besser gleich weg. Vermeiden Sie lange rechtliche Klausel für den Haftungsausschluss die an jede E-Mail angehängt werden, insbesondere wenn Sie eine Stimmung ausdrücken die nicht mit der Beteiligung an einem freien Software-Projekt vereinbar sind. Der folgende Klassiker dieser Gattung erscheint zum Beispiel am Ende jeder Nachricht von einem bestimmten Nutzer auf einem öffentlichen Verteiler bei dem ich angemeldet bin:

```
WICHTIGER HINWEIS

Wenn Sie diese E-Mail fehlerhafterweise erhalten haben oder den
Haftungsausschluss unserer E-Mails lesen wollen, wenden Sie sich bitte
an die folgende Erklärung under nehmen Sie mit dem Absender Kontakt
auf.

This communication is from Deloitte & Touche LLP.  Deloitte &
Touche LLP is a limited liability partnership registered in England
and Wales with registered number OC303675.  A list of members' names
is available for inspection at Stonecutter Court, 1 Stonecutter
Street, London EC4A 4TR, United Kingdom, the firm's principal place of
business and registered office.  Deloitte & Touche LLP is
authorised and regulated by the Financial Services Authority.

This communication and any attachments contain information which is
confidential and may also be privileged.  It is for the exclusive use
of the intended recipient(s).  If you are not the intended
recipient(s) please note that any form of disclosure, distribution,
copying or use of this communication or the information in it or in
any attachments is strictly prohibited and may be unlawful.  If you
have received this communication in error, please return it with the
title "received in error" to IT.SECURITY.UK@deloitte.co.uk then delete
the email and destroy any copies of it.

E-mail communications cannot be guaranteed to be secure or error free,
as information could be intercepted, corrupted, amended, lost,
destroyed, arrive late or incomplete, or contain viruses.  We do not
accept liability for any such matters or their consequences.  Anyone
who communicates with us by e-mail is taken to accept the risks in
doing so.

When addressed to our clients, any opinions or advice contained in
this e-mail and any attachments are subject to the terms and
conditions expressed in the governing Deloitte & Touche LLP client
engagement letter.
```

```
Opinions, conclusions and other information in this e-mail and any
attachments which do not relate to the official business of the firm
are neither given nor endorsed by it.
```

Für jemanden der lediglich ab und an ein paar Fragen stellen möchte, erscheint dieser riesige Haftungsausschluss etwas albern aber verursacht wahrscheinlich keinen dauerhaften Schaden. Wenn diese Person sich jedoch aktiv an dem Projekt beteiligen wollte, würde dieser Rechtliche Textblock eine heimtückischere Wirkung haben. Es würde mindestens zwei möglicherweise schädliche Signal aussenden: Erstens, dass diese Person nicht die komplette Kontrolle über seine Werkzeuge hat – er ist in irgend einem E-Mail-System von einem Unternehmen, welches an jede E-Mail eine nervige Botschaft anhängt, und er hat keine Möglichkeit es zu umgehen – und zweitens, dass er wenig oder keine Unterstützung von seiner Organisation für seine Aktivitäten bei freier Software hat. Zugegeben, die Organisation hat ihm ganz klar nicht direkt verboten an öffentliche Verteiler zu schreiben, aber sie lässt seine Nachrichten eindeutig unfreundlich aussehen, als ob das Risiko vertrauliche Informationen herauszulassen über allens anderen Prioritäten steht.

Wenn Sie für eine Organisation arbeiten, welche darauf besteht, solche Signaturen an alle abgehenden E-Mails anzuhängen, dann sollten Sie in Betracht ziehen, sich eine kostenlose E-Mail-Adresse anzulegen, zum Beispiel bei gmail.google.com, www.hotmail.com, oder www.yahoo.com, und benutzen Sie diese Adresse für das Projekt.

Vermeidung häufiger Fallstricke

Schreiben Sie nicht ohne Veranlassung

Eine häufiger Fehler bei der Beteiligung an einem Online-Projekt ist zu denken, dass Sie auf alles reagieren müssen. Das müssen Sie nicht. Erstens wird es für gewöhnlich mehr Threads geben, über die Sie den Überblick behalten können, zumindest nachdem das Projekt die ersten paar Monate überschritten hat. Zweitens wird das meiste in den Theras bei denen Sie sich entschieden haben, sie zu verfolgen, keiner Antwort erfordern. Gerade Entwicklerforen neigen dazu von drei Arten von Nachrichten beherrscht zu werden:

1. Nachrichten die etwas nicht triviales vorschlagen
2. Nachrichten die Unterstützung oder Widerstand für oder gegen etwas ausdrücken was jemand anderes gesagt hat
3. Zusammenfassende Nachrichten

Von diesen erfordert keine *an und für sich* eine Rückmeldung, insbesondere wenn Sie sich auf der Grundlage von dem was Sie bisher in dem Thread gesehen haben, sicher sein können, dass jemand anderes wahrscheinlich sowieso das sagen wird, was Sie eh gesagt hätten. (Machen Sie sich keine Sorgen, dass Sie in einer Schleife gefangen werden, bei dem jeder auf den anderen wartet, weil jeder die gleiche Taktik verfolgt; es gibt fast immer *Irgendjemanden* der das Bedürfnis hat sich in die Schlacht zu stürzen. Fragen Sie sich erstens: Wissen Sie was Sie erreichen wollen? Und zweitens: Wird dieses Ziel nicht erreicht werden, ohne dass Sie etwas sagen?

Zwei gute Gründe Ihre Stimme bei einem Thread einzulegen, sind a) wenn Sie einen Fehler in einem Vorschlag sehen, und vermuten, dass Sie die einzige sind, der es sieht, und b) wenn Sie sehen, dass in der Kommunikation etwas schief läuft und Sie wissen, dass Sie es richten können indem Sie eine klärende Nachricht schreiben. Es ist im allgemeinen auch in Ordnung eine Nachricht zu schicken, nur um jemanden für etwas zu danken, oder um "Ich auch!" zu sagen, da ein Leser bei solchen Nachrichten sofort erkennen kann, dass sie keiner weiteren Antwort oder Handlung bedürfen und deshalb endet die geistige Anstrengung die sie erfordern sauber mit der letzten Zeile der E-Mail. Selbst dann sollten Sie zwei mal darüber nachdenken, bevor Sie etwas sagen; es ist immer besser Menschen wünschen zu las-

sen, dass Sie mehr schreiben, als dass die weniger schreiben. (Siehe den zweiten Teil von Anhang C, *Warum sollte es mich kümmern, welche Farbe der Fahrradschuppen hat?* für weiteres darüber wie man sich auf betriebsamen Mailverteilern verhalten soll.)

Produktive kontra unproduktive Threads

Auf einem betriebsamen Mailverteiler haben Sie zwei Pflichten. Eines ist herauszufinden, worauf Sie achten müssen und was Sie ignorieren können. Das andere ist, sich derart zu verhalten, dass Sie es vermeiden Rauschen zu *erzeugen*: Sie wollen nicht nur, dass Ihre eigene Nachrichten ein hohes Signal/Rausch-Verhältnis haben, sondern, dass es die Art von Nachrichten sind, die *andere* dazu anregen entweder Nachrichten mit einem ähnlich hohen Signal/Rausch-Verhältnis zu schreiben, oder überhaupt nicht zu schreiben.

Um zu sehen wie Sie das erreichen können, lassen Sie uns den Kontext bedenken in dem so etwas passiert. Was sind einige der Ecksteine von unproduktiven Threads?

- Argumente die bereits aufgeworfen wurden fangen an wiederholt zu werden, als ob der Absender denkt, dass keiner sie beim erstem mal gehört hat.
- Eine Zunahme an Übertreibung und Beteiligung während die der Einsatz immer kleiner wird.
- Eine Mehrheit der Kommentare stammt von Personen die wenig oder gar nichts machen, während diejenigen die dazu neigen Sachen erledigt zu bekommen ruhig sind.
- Viele Ideen werden diskutiert, ohne dass klare Vorschläge gemacht werden. (Natürlich fängt jede interessante Idee als eine ungenaue Vision an; die wichtige Frage ist in welche Richtung sie sich von da weiterentwickelt. Erscheint es so als ob der Thread die Vision in etwas konkreteres verwandelt, oder schweift es ab in unter Unter-Visionen, Neben-Visionen und Auseinandersetzungen über Grundsatzfragen?)

Nur weil ein Thread anfangs nicht produktiv ist, bedeutet es nicht, das es Zeitverschwendung ist. Es kann sich um ein wichtiges Thema drehen, bei dem die Tatsache, dass es nicht vorankommt um so beunruhigender ist.

Einen Thread in eine nützliche Richtung zu führen, ohne aggressiv zu sein, ist eine Kunst. Es wird nicht funktionieren, Leute einfach abzumahnen, ihre Zeit nicht zu verschwenden, oder Sie darum zu bitten nicht zu schreiben, es sei denn Sie haben etwas konstruktives beizutragen. Es mag natürlich sein, dass Sie das privat denken, aber wenn Sie es laut sagen, werden Sie beleidigend sein. Statt dessen müssen Sie Bedingungen für den weiteren Fortschritt vorschlagen – geben Sie den Leuten eine Route, einen Weg welcher zu den Ergebnissen führt, die Sie haben wollen, jedoch ohne dass Sie sich anhören, als würden Sie das Verhalten diktieren. Der Unterschied liegt größtenteils im Ton. Folgendes ist zum Beispiel schlecht:

> *Diese Diskussion führt nirgendwo hin. Können wir bitte dieses Thema so lange fallen lassen, bis jemand einen Patch hat, um einen der Vorschläge zu implementieren? Es hat keinen Sinn immer wieder die gleichen Sachen zu sagen. Code spricht lauter als Worte, Leute.*

Wohingegen folgendes gut ist:

> *Verschiedene Vorschläge wurden in diesem Thread gemacht, bei keinem wurden aber die Details ausgearbeitet, zumindest nicht so weit um darüber abzustimmen. Trotzdem sagen wir derzeit nichts neues; wir wiederholen einfach nur was vorher schon gesagt wurde. Derzeit wäre es also wahrscheinlich das beste, wenn die folgenden Nachrichten entweder ausgearbeitete Vorschläge beinhalten, oder einen Patch. Dann hätten wir wenigstens etwas festes worauf wir reagieren könnten (d.H. Konsens über den Vorschlag erreichen oder den Patch anwenden und Testen).*

Vergleichen Sie die zweite Herangehensweise mit der ersten. Die Zweite zieht keinen Strich zwischen Ihnen und die anderen, noch beschuldigt es die Teilnehmer die Diskussion zu verzetteln. Es ist die Rede von "wir", was wichtig ist, ob Sie vorher tatsächlich an dem Thread beteiligt waren oder nicht, da es alle daran erinnert, dass selbst diejenigen die bisher ruhig geblieben sind, trotzdem noch einen Anteil an seinem Ausgang haben. Die Nachricht beschreibt warum der Thread nirgendwo hinführt, macht es aber ohne Abwertungen oder Verurteilungen – es hält lediglich leidenschaftslos einige Tatsachen fest. Am wichtigsten ist, dass es eine positive Vorgehensweise anbietet, anstatt dass man das Gefühl bekommt, als würde die Diskussion abgebrochen werden (eine Maßnahme die dazu verleiteten dürfte zu rebellieren), man wird es als eine Möglichkeit ansehen, wird die Unterhaltung auf eine konstruktivere Ebene zu führen. Dies ist ein Normwert den man natürlich erfüllen will.

Sie werden nicht immer wollen, dass ein Thread es auf die nächste höhere konstruktive Ebene schafft – manchmal werden Sie einfach nur wollen das er verschwindet. Der Sinn von Ihrer Nachricht ist dann, das eine oder das andere herbeizuführen. Wenn Sie schon an der Art wie der Thread bisher verlaufen ist, erkennen können, dass keiner die von Ihnen vorgeschlagenen Maßnahmen wirklich *machen* wird, haben Sie den Thread effektiv beendet ohne dass es danach aussieht. Es gibt natürlich keinen narrensicheren Weg einen Thread zu beenden, und selbst wenn es den gäbe, würden Sie ihn nicht einsetzen wollen. Die Beteiligten aber darum zu bitten, entweder sichtbaren Fortschritt zu machen, oder aufzuhören Nachrichten zu schicken, ist wenn Sie es diplomatisch anstellen, ohne weiteres vertretbar. Hüten Sie sich jedoch davor Threads voreilig zu schließen. Eine gewisse Menge an spekulativem Gerede kann, je nach Thema, produktiv sein, und darum zu bitten, dass man es zu schnell klärt, wird den kreativen Ablauf ersticken, und Sie zusätzlich als ungeduldig erscheinen lassen.

Erwarten Sie von keinem Thread, dass er sofort aufhört. Es wird wahrscheinlich immer noch ein paar Nachrichten nach Ihrer geben, entweder weil Sie gleichzeitig mit jemand anderem gepostet haben, oder weil Leute immer das letzte Wort haben wollen. Das ist nichts, worüber Sie sich Sorgen machen müssten und Sie müssen nicht erneut ein Schreiben schicken. Lassen Sie den Thread einfach auslaufen, oder nicht auslaufen wie immer der Fall auch sein mag. Sie können nicht die absolute Kontrolle haben; andererseits, können Sie über viele Threads gesehen, eine statistisch signifikante Wirkung zu haben.

Je weicher das Thema, desto länger die Debatte

Obwohl Diskussionen bei jedem Thema mäandrieren können, geht die Wahrscheinlichkeit dafür hoch sobald die technische Schwierigkeit runter geht. Schließlich ist die Anzahl der Teilnehmer die dem Thema folgen können um so kleiner, je schwieriger das Thema ist. Diejenigen die dann folgen können, sind wahrscheinlich die erfahrensten Entwickler, die bereits tausende Male vorher an solchen Diskussionen teilgenommen haben, und wissen, welches Verhalten wahrscheinlich zu einem Konsens führt, mit dem alle leben können.

Konsens ist deshalb am schwersten zu erreichen bei technischen Fragen, die einfach zu verstehen sind und bei denen man leicht eine Meinung haben kann, sowie bei "weichen" Themen, wie Organisation, Öffentlichkeitsarbeit, Finanzierung, usw. Menschen können sich ewig über solche Themen unterhalten, da es keine nötige Qualifikation dafür gibt, keine klaren Möglichkeiten um zu entscheiden (selbst im Nachhinein) ob eine Entscheidung richtig war oder falsch, und weil es manchmal eine erfolgreiche Taktik ist einfach länger zu warten als andere Diskussionsteilnehmer.

Das Prinzip, dass die Menge an Diskussion umgekehrt proportional dazu ist, wie komplex das Thema ist, hat es schon lange gegeben, und ist allgemein unter dem Begriff *Bikeshed Effect* (de. Fahrradschuppen-Effekt) bekannt. Hier ist die Erklärung von Poul-Henning Kamp davon, aus einer nunmehr berühmten E-Mail an BSD-Entwickler:

> Es ist eine lange Geschichte, bzw. eher eine alte Geschichte, aber sie ist auf jeden Fall ziemlich kurz. C. Northcote Parkinson schrieb Anfang der 1960'er ein Buch namens "Parkinsons Gesetz", welches eine Menge Einblicke in die Dynamik von Management beinhaltet.

> [...]
>
> Bei dem spezifischen Beispiel welches mit einem Fahrradschuppen zu tun hat, ist die andere entscheidende Komponente ein Atomkraftwerk, was schätze ich die Zeit in der das Buch geschrieben wurde widerspiegelt.
>
> Parkinson zeigt, wie Sie zu einem Vorstand gehen können und Zustimmung für den Bau einer multi-millionen oder gar Milliarden Euro teuren Atomkraftwerk bekommen können, wenn Sie aber einen Fahrradschuppen bauen wollen, werden Sie sich in endlosen Diskussionen verzetteln.
>
> Parkinson erklärt, dass das daran liegt, dass ein Atomkraftwerk so gewaltig, so kostspielig und so kompliziert ist, das Menschen es nicht begreifen können, und eher als es zu versuchen, fallen Sie auf die Annahme zurück, dass jemand anderes bereits alle Details überprüft hat vor es so weit gekommen ist. Richard P. Feynmann gibt in seinen Büchern ein paar interessante und treffende Beispiele die mit Los Alamos zu tun haben.
>
> Einen Fahrradschuppen andererseits kann jeder übers Wochenende bauen und trotzdem noch genügend Zeit übrig haben um das Spiel im Fernsehen zu schauen. Egal wie gut Sie sich vorbereiten, egal wie vernünftig Sie bei Ihrem Vorschlag also sind, irgend jemand wird die Chance ergreifen, um Ihnen zu zeigen, dass er seine Arbeit macht, dass er aufpasst, dass er *da* ist.
>
> In Dänemark nennen wir das "seinen Fingerabdruck hinterlassen". Es geht um den persönlichen Stolz und Ansehen, es geht darum irgendwo drauf zeigen zu können und zu sagen "Da! Das habe *ich* gemacht". Es ist ein starker Wesenszug bei Politikern, aber auch in den meisten Menschen vorhanden, wenn sie dazu die Gelegenheit bekommen. Denken Sie einfach an Fußabdrücke im nassen Zement.

(Seine komplette Nachricht ist auch sehr lesenswert. Siehe Anhang C, *Warum sollte es mich kümmern, welche Farbe der Fahrradschuppen hat?*; siehe auch http://bikeshed.com.)

Jeder der jemals an der Entscheidungsfindung in einer Gruppe beteiligt war, wird erkennen worüber Kamp redet. Es ist jedoch für gewöhnlich unmöglich *alle* zu überreden es zu vermeiden Fahrradschuppen anzumalen. Das beste was Sie machen können, ist darauf hinzuweisen, dass das Problem existiert sobald es auftaucht, und Ihre Senior-Entwickler – die Personen deren Nachrichten am meisten Gewicht tragen – dazu überreden, frühzeitig Ihre Pinsel nieder zu legen, damit zumindest sie nicht zum Rauschen beitragen. Fahrradschuppen-Anmal-Partys werden nie komplett verschwinden, aber Sie können sie verkürzen und weniger häufig machen, indem Sie das Bewusstsein für sie in der Projektkultur verinnerlichen.

Vermeiden Sie Heilige Kriege

Ein *heiliger Krieg* ist eine Debatte, die oft, aber nicht immer über eine relativ unbedeutende Angelegenheit geführt wird, welche nicht anhand der Vorzüge verschiedener Argumente zu klären ist, bei dem aber Leute leidenschaftlich genug sind um trotzdem weiter darüber zu argumentieren, in der Hoffnung das ihre Seite sich durchsetzen wird. Heilige Kriege sind nicht ganz das selbe wie das Anmalen von Fahrradschuppen. Leute die Fahrradschuppen anmalen sprechen für gewöhnlich schnell Ihre Meinung (weil sie es können), fühlen sich aber nicht sonderlich stark an diese Meinung gebunden, und werden manchmal sogar andere, nicht kompatible Meinungen äußern, um zu zeigen, dass Sie alle Seiten der Angelegenheit verstehen. Bei einem heiligen Krieg hingegen wird Verständnis für andere Seiten als Schwäche aufgefasst. Bei einem heiligen Krieg weiß jeder, dass es eine richtige Antwort gibt; sie sind sich nur nicht darüber einig welche es ist.

Wenn ein heiliger Krieg erst einmal angefangen hat, kann es im allgemeinen nicht zur Zufriedenheit von allen aufgelöst werden. Es nützt nichts während einem heiligen Krieg darauf hinzuweisen, dass man sich in einem heiligen Krieg befindet. Jeder weiß das schon. Ein leider häufiges Merkmal von heiligen Kriegen sind Meinungsverschiedenheiten über die Frage *ob* die Debatte sich überhaupt durch weitere Diskussion auflösen lässt. Von außen betrachtet, ist es klar, dass keine Seite die Meinung der anderen ändert. Von innen betrachtet, benimmt sich die andere Seite stumpfsinnig und denkt nicht ganz klar, sie kommt aber vielleicht zur Besinnung, wenn man sie nur genügend einschüchtert. Ich sage jetzt *nicht*, dass es nie eine richtige Seite bei einem heiligen Krieg gibt. Manchmal gibt es eine – und natürlich ist es bei denen an den ich bisher teilgenommen habe immer meine gewesen. Das macht aber keinen Unterschied, weil es keinen Algorithmus gibt, um überzeugend zu demonstrieren, dass die eine oder andere Seite richtig ist.

Eine verbreitete, aber nicht zufriedenstellende Art, wie Leute versuchen heilige Kriege zu lösen, ist zu sagen "Wir haben bereits viel mehr Zeit und Energie bei der Diskussion hiervon verbraucht, als es wert ist! Können wir es bitte einfach fallen lassen"? Es gibt dabei zwei Probleme. Erstens wurde diese Zeit und Energie bereits aufgebracht, und sie kann nie wieder zurückgewonnen werden – die einzige Frage die jetzt noch übrig bleibt ist, wieviel *mehr* man investieren muss? Wenn einige Personen der Meinung sind, dass nur ein wenig mehr Diskussion das Thema zum Abschluss bringen wird, dann macht es immer noch Sinn (aus ihrer Sicht) weiter zu machen.

Bei der Bitte das Thema fallen zu lassen ist das andere Problem, dass der Status Quo es einer Seite erlauben würde, den Sieg durch Untätigkeit zu erklären. Und in manchen Fällen ist der Status Quo sowieso nicht akzeptabel: Alle sind sich darüber einig, dass irgendeine Entscheidung getroffen, irgendeine Maßnahme ergriffen werden muss. Das Thema fallen zu lassen, wäre schlimmer für alle als den Streit aufzugeben. Da das Dilemma aber für alle gleichermaßen gilt, ist es immer noch möglich ewig darüber zu streiten, was getan werden soll.

Wie sollten Sie als heilige Kriege handhaben?

Die erste Antwort ist, die Dinge so einzurichten, dass sie gar nicht erst passieren. Das ist nicht so hoffnungslos wie es sich anhört:

Sie können einige immer wiederkehrende heilige Kriege vorausahnen: Sie tendieren zu Themen wie Programmiersprachen, Lizenzen (siehe „Die GPL und Lizenz-Kompatibilität“ im Kapitel Kapitel 9, *Lizenzen, Urheberrecht und Patente*), Änderung des reply-to Feldes (siehe „Die große "reply-to"-Debatte“ im Kapitel Kapitel 3, *Technische Infrastruktur*) sowie ein paar andere Themen. Jedes Projekt hat auch ein oder zwei ganz eigene heilige Kriege, womit langjährige Entwickler schnell vertraut werden. Die Techniken um heilige Kriege aufzuhalten, oder zumindest ihren Schaden zu begrenzen, sind überall ziemlich die gleichen. Selbst wenn Sie sich sicher sind, dass Ihre Seite recht hat, versuchen Sie *irgendeine* Möglichkeit zu finden um Mitgefühl und Verständnis für die Argumente die die anderen Seite macht auszudrücken. Oftmals ist das Problem bei einem heiligen Krieg, dass jede Seite ihre Mauern so hoch wie möglich gebaut hat, und klar gemacht hat, dass jede andere Meinung schlichtweg albern ist, wird Kapitulation oder die Änderung seiner Meinung psychologisch unerträglich: Es wäre nicht nur ein Geständnis, dass man falsch lag, sondern sich *sicher* gewesen zu sein und trotzdem falsch. Sie können dieses Geständnis für die andere Seite schmackhaft machen, ist indem Sie selber Ungewissheit zeigen – gerade indem Sie zeigen, dass Sie die Argumente die sie machen verstehen und sie zumindest vernünftig finden, wenn auch nicht ganz überzeugend. Zeigen Sie eine Geste, welche Raum lässt, für eine gegenseitige Geste lässt, und die Situation wird sich gewöhnlich verbessern. Es ist weder wahrscheinlicher oder unwahrscheinlicher, dass Sie das Ergebnis, das Sie ursprünglich wollten erreichen, zumindest können Sie dadurch aber unnötigen Kollateralschaden an der Moral des Projekts vermeiden.

Wenn ein heiliger Krieg nicht vermieden werden kann, entscheiden Sie sich frühzeitig wie sehr sie die Sache kümmert, und seien Sie bereit öffentlich aufzugeben. Wenn Sie das tun, können Sie sagen, dass Sie aussteigen, weil ein heiliger Krieg es nicht wert ist, drücken Sie dabei aber keine Bitterkeit aus und nutzen Sie die Gelegenheit *nicht* als eine letzte Gelegenheit um gegen die Argumente der Gegenseite zu argumentieren. Aufzugeben ist nur effektiv wenn es taktvoll gemacht wird.

Heilige Kriege über Programmiersprachen sind ein Spezialfall, da dazu neigen sehr technisch zu sein, dennoch fühlen sich viele Leute qualifiziert an Ihnen teil zu nehmen, und der Einsatz ist sehr hoch, da das Resultat bestimmen kann, in welcher Sprache ein Großteil des Codes vom Projekt geschrieben wird. Die beste Lösung ist es, die Sprache frühzeitig zu wählen, mit Unterstützung durch einflussreiche Entwickler, und es dann auf der Grundlage zu verteidigen, dass sie sich alle wohl fühlen in dieser Sprache zu schreiben, *nicht* auf der Grundlage, dass es besser ist als irgend eine andere Sprache, die man sich statt dessen hätte aussuchen können. Lassen Sie die Unterhaltung nie zu einem akademischen Vergleich verschiedener Programmiersprachen verfallen (das scheint, aus irgendeinem Grund, besonders of zu passieren, wenn jemand Perl aufbringt); das ist einfach ein Thema des Todes in welches Sie sich weigern müssen hineingezogen zu werden.

Für ein eher historischen Hintergrund über heilige Kriege, siehe http://catb.org/~esr/jargon/html/H/holy-wars.html, und die Veröffentlichung von Danny Cohen welches den Begriff populär http://www.ietf.org/rfc/ien/ien137.txt.

Der "Laute Minderheit"-Effekt

Bei jedem Mailverteiler ist es leicht, für eine kleine Minderheit den Eindruck zu erwecken, dass eine Menge Dissens gibt, indem Sie den Verteiler mit einer Menge langer E-Mails überfluten. Es ist ein eine art Obstruktionspolitik, mit dem Unterschied, dass der Eindruck von ausgedehntem Dissens noch stärker ist, da es auf eine beliebige Anzahl einzelner Nachrichten aufgeteilt ist und die meisten Leute werden sich nicht die Mühe machen mitzuverfolgen wer was wann gesagt hat. Sie werden nur den instinktiven Eindruck haben, dass das Thema sehr kontrovers ist, und warten, bis die Aufregung sich gelegt hat.

Sehr klar zu Belegen, wie klein die tatsächliche Anzahl der Dissidenten ist im Vergleich zu denen die zustimmen ist die beste Art gegen diesen Effekt anzukommen. Um die Ungleichheit zu vergrößern, sollten Sie im privaten Leute ansprechen die größtenteils ruhig geblieben sind, von denen Sie aber vermuten, dass Sie der Mehrheit zustimmen. Sagen Sie nichts, was darauf hindeutet, dass die Dissidenten absichtlich versucht haben den Eindruck den sie hinterlassen zu verstärken. Wahrscheinlich war das nicht der Fall, und selbst wenn, gäbe es keinen strategischen Vorteil darauf hinzuweisen. Alles was Sie tun müssen, ist eine Gegenüberstellung der echten Zahlen, und Leute werden erkennen, dass ihr intuitiver Eindruck der Situation nicht der Wirklichkeit entspricht.

Dieser Ratschlag gilt nicht nur für Angelegenheiten mit klaren Positionen für oder gegen etwas. Sondern für jede Diskussion bei der viel Lärm gemacht wird, aber nicht klar ist, ob die meisten die Angelegenheit als ein echtes Problem ansehen. Nach einer gewissen Zeit, wenn Sie darüber einstimmen, dass die Meldung keiner Reaktion wert ist, und sehen können, dass sie es nicht geschafft hat an Zug zu gewinnen (selbst wenn es eine Menge E-Mails hervorgebracht hat), können Sie einfach öffentlich feststellen, dass es keinen Zug gewinnt. Wenn der "Laute Minderheit"-Effekt aufgetreten war, wird Ihre Nachricht wie eine Atemzug frischer Luft wirken. Der Eindruck den die meisten Leute bisher von der Diskussion hatten, wird etwas düster gewesen sein: "Hmm, es scheint eine große Sache zu sein, weil es wirklich eine Menge Nachrichten gibt, aber ich sehe keinen echten Fortschritt". Indem Sie erklären, wie die Art der Diskussion es hat scheinen lassen, als wäre es stürmischer als in Wirklichkeit, werden Sie es im Nachhinein eine neue Gestalt geben, wodurch Leute ihr Verständnis von dem was passiert ist neu gestalten können.

Schwierige Leute

Schwierige Leute sind im Umgang in elektronischen Foren nicht einfacher als im echten Leben. Mit "schwierig" meine ich nicht "unhöflich". Unhöfliche Leute nerven, aber sie sind nicht unbedingt schwierig. Dieses Buch hat bereits behandelt, wie man mit denen umgeht: Machen Sie eine Bemerkung beim ersten mal, und ab dann, sollten Sie sie entweder ignorieren oder sie wie alle anderen behandeln. Wenn sie weiterhin unhöflich sind, werden sie sich meistens derart unbeliebt machen, dass sie keinen Einfluss auf andere im Projekt haben, also sind sie ein in sich abgeschlossenes Problem.

Die wirklich schwierigen Fälle sind Leute die nicht sonderlich unhöflich sind, die aber die Arbeitsabläufe im Projekt derart manipulieren oder missbrauchen, dass es die Zeit und Energie anderer auffrisst, dem Projekt jedoch keinen Nutzen bringt. [2].

Solche Menschen suchen oft nach Lücken in den Abläufen des Projekts, um sich selber mehr Einfluss zu verschaffen als es sonst der Fall wäre. Das ist viel heimtückischer als schlichte Unhöflichkeit, weil weder das Verhalten noch der Schaden für einen beiläufigen Beobachter offensichtlich ist. Ein klassisches Beispiel ist die Verschleppungstaktik bei der jemand (der sich natürlich immer so vernünftig wie möglich anhört) immer wieder behauptet, dass die Angelegenheit die zur Debatte steht, nicht für eine Klärung bereit ist und weitere mögliche Lösungen anbietet, oder neue Sichten auf alte Lösungen, wenn in wirklich derjenige merkt, dass ein Konsens oder eine Wahl sich langsam anbahnt, und ihm gefällt die Richtung nicht in der sie wahrscheinlich geht. Ein weiteres Beispiel ist, wenn eine Debatte die sich keinem Konsens annähert, die Gruppe aber versucht wenigstens die Eckpunkte der Meinungsverschiedenheit klar zu stellen und eine Zusammenfassung für alle zu produzieren, auf der sich alle von da an beziehen können. Der Quertreiber, der weiß, dass die Zusammenfassung zu einem Ergebnis führen könnte, welches ihm nicht gefällt, wird oft versuchen selbst die Zusammenfassung hinauszuzögern, indem er schonungslos die Fragen verkompliziert, was darin beinhaltet sein soll, entweder indem er Einspruch erhebt oder indem er unerwartete neue Punkte vorstellt.

Handhabung schwieriger Leute

Um solchem Verhalten entgegenzuwirken hilft es, die Mentalität derjenigen zu verstehen, die es ergreifen. Menschen machen es im Allgemeinen nicht bewusst. Keiner wacht morgens auf und sagt sich: "Heute werde ich Verwaltungsverfahren zynisch manipulieren, um ein irritierender Quertreiber zu sein". Statt dessen geht solches Verhalten oftmals ein halb paranoides Gefühl voraus, von den Interaktion und den Entscheidungen in der Gruppe ausgeschlossen zu werden. Die Person fühlt sich nicht ernst genommen, oder (in schwerwiegenderen Fällen), dass es fast schon eine Verschwörung gegen ihn ist – dass die anderen Mitglieder des Projekts sich entschieden haben, einen exklusiven Club zu bilden, in dem er kein Mitglied ist. Das rechtfertigt dann, in seinen Augen, die Regeln wörtlich zu nehmen und anzufangen die Abläufe des Projekts formal zu manipulieren, um alle anderen zu *zwingen* Ihn ernst zu nehmen. Im Extremfall, kann die Person sogar glauben, dass er einen einsamen Kampf ausficht, um das Projekt vor sich selbst zu retten.

Es liegt in der Natur solch eines Angriffs von innen, dass nicht jeder es zur gleichen Zeit bemerken wird, und manche werden es überhaupt nicht sehen, bis man ihnen aussagekräftige Beweise vorlegt. Das bedeutet, dass es eine Menge Arbeit sein kann es zu neutralisieren. Es reicht nicht sich selbst zu überzeugen, dass es passiert; Sie müssen genügend Beweise sammeln, um andere auch zu überzeugen, und dann müssen Sie diese Beweise wohl überlegt verbreiten.

Wenn man bedenkt wie viel Arbeit es ist dagegen anzukämpfen, ist es oft besser, es einfach eine Weile lang zu tolerieren. Betrachten Sie es als eine parasitäre aber leichte Krankheit: Wenn es nicht zu sehr das Projekt behindert, kann das Projekt es sich leisten infiziert zu bleiben, und Medizin könnte schädliche Nebenwirkungen haben. Wenn es jedoch zu schädlich wird um es zu tolerieren, dann ist es Zeit etwas dagegen zu machen. Fangen Sie an Notizen und Muster zu sammeln, die Sie erkennen. Stellen Sie sicher Verweise auf die öffentlichen Archive mit einzubeziehen – das ist eines der Gründe warum das Projekt alles protokolliert, also können Sie sie genau so gut auch nutzen. Sobald Sie einen guten Fall aufgebaut haben, fangen Sie an private Unterhaltungen mit den anderen Beteiligten am Projekt zu führen. Sagen Sie ihnen nicht was Sie beobachtet haben; statt dessen, fragen Sie zuerst, was sie beobachtet haben. Das kann Ihre letzte Gelegenheit sein, ungetrübte Rückmeldung zu erhalten, wie andere das Verhalten des Störenfrieds sehen; wenn Sie erst einmal angefangen haben öffentlich darüber zu reden, wer-

[2]Für eine ausführliche Diskussion einer Subspezies dieser Personen, siehe:*Help Vampires: A Spotter's Guide [http://slash7.com/2006/12/22/vampires/]*. Um Hoy zu zitieren: "Es ist so normal, dass man seine Uhr danach stellen mann. Der Zerfall einer Community ist genauso vorhersagbar wie der Zerfall von nuklearen Isotopen. Sobald ein Open Source Projekt, Sprache usw. seine Halb- wertszeit erreicht, kommen sie und nehmen das Leben aus der Community. *Sie* sind die Vampire der Hilfe. Und ich bin hier um sie zu stoppen…"

den die Meinungen polarisiert werden und keiner wird sich daran erinnern können wie er vorher darüber gedacht hat.

Wenn private Diskussionen darauf hindeuteten, dass zumindest ein paar andere das Problem sehen, dann ist es Zeit etwas zu unternehmen. Ab dann müssen Sie *wirklich* vorsichtig sein, denn es ist sehr leicht für diese Person es danach aussehen zu lassen als würden Sie unfairerweise auf ihr herum hacken. Was immer Sie auch tun, beschuldigen Sie sie nicht, die Abläufe im Projekt arglistig missbraucht zu haben, paranoid zu sein oder im Allgemeinen, alles was Sie vermuten, dass es wahrscheinlich wahr ist. Ihre Strategie sollte es sein, sowohl vernünftiger und eher besorgt um das Wohl des Projekts zu sein, mit dem Ziel entweder das Verhalten der Person zu reformieren, oder Sie dazu zu bringen permanent zu verschwinden. Abhängig davon, ob andere Entwickler, und Ihr Verhältnis mit ihnen, kann es vorteilhaft sein, zuerst im privaten Verbündete zu sammeln. Oder auch nicht; dass kann auch nur Feindseligkeit hinter den Vorhängen erzeugen, wenn Leute denken, dass Sie eine nicht angemessene flüster-Kampagne ergreifen.

Denken Sie daran, dass obwohl die andere Person vielleicht derjenige ist, die sich zerstörerisch verhält, werden *Sie* diejenige sein, die zerstörerisch erscheint, wenn Sie eine öffentliche Beschuldigung machen, welche Sie nicht untermauern können, und es so sanft wie möglich sagen, aber trotzdem noch direkt sind. Sie werden die betreffende Person vielleicht nicht überzeugen können, aber das ist in Ordnung, so lange Sie alle anderen Überzeugen können.

Fallbeispiel

Ich erinnere mich nur an eine Situation, in mehr als 10 Jahren Arbeit an freier Software, wo die Sachen derart schlimm wurden, dass wir tatsächlich jemanden darum bitten mussten, komplett seine Nachrichten einzustellen. Wie es so oft der Fall ist, war er nicht unhöflich, und wollte aufrichtig hilfreich sein. Er wusste nur nicht wann er schreiben sollte und wann er es lieber lassen sollte. Unsere Mailinglisten waren für die Öffentlichkeit frei zugänglich, und er schrieb so oft Nachrichten, und stellte bei so vielen verschiedenen Themen Fragen, dass es für die Gemeinschaft zu einem Problem im Geräuschpegel führte. Wir hatten bereits versucht, ihn nett darum zu bitten etwas mehr vor seinen Nachrichten nach Antworten zu recherchieren, aber es zeigte keine Wirkung.

Die Strategie die letztendlich funktionierte, ist ein perfektes Beispiel, wie man einen starken Fall aufbaut, basierend auf neutralen und messbaren Daten. Einer unserer Entwickler schürfte etwas in den Archiven herum, und schickte dann im privaten folgende Nachricht an ein paar Entwickler. Der Übeltäter (der dritte Name in der nachfolgenden Liste, hier als "H. Mustermann" aufgeführt) hatte eine sehr kurze Geschichte mit dem Projekt und hatte weder Code noch Dokumentation beigetragen. Trotzdem war er die drittaktivste Person auf den Mailinglisten:

```
Von: "Brian W. Fitzpatrick" <fitz@collab.net>
An: [... Liste der Empfänger wegen Datenschutz ausgespart...]
Betreff: Der Energieabfall in Subversion
Datum: Mit, 12 Nov 2003 23:37:47 -0600

In den letzten 25 Tagen, waren die folgenden 6 Personen auf den svn
[dev|users] Verteilern am aktivsten:

    294  kfogel@collab.net
    236  "C. Michael Pilato" <cmpilato@collab.net>
    220  "H. Mustermann" <hmustermann@problemposter.com>
    176  Branko #ibej <brane@xbc.nu>
    130  Philip Martin <philip@codematters.co.uk>
    126  Ben Collins-Sussman <sussman@collab.net>
```

```
Ich würde sagen, dass fünf dieser Personen etwas zu Subversion
beitragen, welches bald Version 1.0 erreichen wird.

Ich würde auch sagen, dass einer dieser Personen beständig Zeit und
Energie von den anderen 5 wegnimmt, gar nicht erst von dem Verteiler
insgesamt zu sprechen, und dadurch (wenn auch unbeabsichtigt) die
Entwicklung von Subversion ausbremst. Ich hab keine Analyse der
Threads gemacht, aber ein vgrep meiner Subversion-Mails sagt mit, dass
auf jede E-Mail von dieser Person, mindestens ein mal von mindestens zwei
der anderen fünf Entwickler geantwortet wurde.

Ich denke das irgendein radikaler Eingriff hier nötig ist, selbst wenn
wir die angesprochene Person verschrecken. Nettigkeiten und
Freundlichkeit haben bereits keine Wirkung gezeigt.

dev@subversion ist ein Mailverteiler um die Entwicklung eines
Versionsverwaltungsystems zu unterstützen, keine Gruppentherapie.

-Fitz, im Versuch durch drei Tage an SVN-Mails durchzuarbeiten, welche
 er angehäuft hat
```

Obwohl es zuerst nicht so aussehen mag, war das Verhalten von H. Mustermann ein klassisches Beispiel für den Missbrauch von Abläufen im Projekt. Er hat nichts offensichtliches gemacht, wie eine Abstimmung hinauszuzögern, sondern machte sich die Richtlinie des Verteilers zunutze, dass Mitglieder selber die Moderatoren sind. Wir überließen es dem Urteilsvermögen von jedem einzelnen, wann man zu welchen Themen etwas schreibt. Wir hatten deshalb nichts worauf wir zurückgreifen konnten, bei jemand der ein solches Urteilsvermögen entweder nicht hatte, oder nicht benutzte. Es gab keine Regel auf die wir hätten verweisen konnten und sagen konnten, dass dieser Kerl es gebrochen hat, jeder wusste jedoch, dass seine häufigen Nachrichten zu einem echten Problem wurden.

Die Strategie von Fitz war, im Nachhinein, meisterhaft. Er sammelte eindeutige und messbare Beweise, verteilte sie dann aber diskret, indem er sie zuerst an ein paar Leute sandte, deren Unterstützung bei jeder drastischen Maßnahme entscheidend wäre. Sie stimmten überein, dass irgend eine Maßnahme nötig war, und schließlich riefen wir J. Random telefonisch an, beschrieben ihm direkt das Problem, und baten ihn darum, seine Nachrichten einfach einzustellen. Er hat niemals wirklich verstanden warum; wenn er dazu in der Lage gewesen wäre, dann hätte er wahrscheinlich schon am Anfang ein angemessenes Urteilsvermögen aufgebracht. Er willigte aber ein, seine Nachrichten einzustellen, und die Mailinglisten wurden wieder brauchbar. Dass diese Strategie funktionierte, lag wohl zum Teil auch an der impliziten Drohung, dass wir seine Nachrichten auch durch die Moderations-Software hätten eingrenzen können, die normalerweise dazu benutzt wird, Spam zu verhindern (siehe „Schutz vor Spam“ im Kapitel Kapitel 3, *Technische Infrastruktur*). Der Grund warum wir diese Option als Rücklage hatten, war das Fitz die nötige Unterstützung zuerst von entscheidenden Personen gesammelt hatte.

Handhabung von Wachstum

Der Preis vom Erfolg ist bei groß in der Open-Source-Welt. Während Ihre Software beliebter wird, nimmt die Anzahl der Menschen zu, die auftauchen um nach Informationen zu suchen, dramatisch zu, während die Anzahl der Menschen die in der Lage sind Informationen bereitzustellen sehr viel langsamer zunimmt. Desweiteren, gäbe es selbst wenn das Verhältnis ausgeglichen wäre immer noch das Problem, mit der Art wie die meisten Open-Source-Projekte Kommunikation handhaben. Betrachten Sie zum Beispiel Mailverteiler. Die meisten Projekte haben Mailverteiler für allgemeine Fragen von Nutzern – manchmal heißt der Verteiler "users", "discuss", "help" oder irgend etwas anderes. Unabhängig von Namen, ist der Sinn des Verteilers immer der gleiche: Einen Ort zur Verfügung zu stellen, wo Leu-

te auf ihre Fragen Antworten bekommen können, während andere zuschauen und (vermutlich) aus der Beobachtung dieses Austauschs Wissen aufsaugen.

Diese Mailverteiler funktionieren sehr gut, bis zu ein paar Tausend Nutzern, und/oder ein paar hundert Nachrichten am Tag. Irgendwo danach fängt das System jedoch an zusammenzubrechen, weil jeder der angemeldet ist jede Nachricht sieht; wenn die Anzahl der Nachrichten an einem Verteiler über das hinaus geht, was eine einzige Person am Tag verarbeiten kann, wird der Verteiler zu einer Last für seine Mitglieder. Stellen Sie sich zum Beispiel vor, wenn Microsoft einen solchen Verteiler für Windows XP hätte. Windows XP hat Hunderte von Millionen von Nutzern; wenn auch nur ein Promille von denen in einer vierundzwanzigstündigen Zeitspanne Fragen hätte, dann bekäme dieser Verteiler theoretisch Hunderte von Tausende von Nachrichten am Tag! Solch einen Verteiler könnte es natürlich niemals geben, denn keiner bliebe bei ihm angemeldet. Dieses Problem beschränkt sich nicht auf Mailverteiler; die gleiche Logik lässt sich anwenden, auf IRC, Online-Diskussionsforen, sogar jedes System in dem eine Gruppe die Fragen von Einzelnen hört. Die Implikationen sind bedenklich: Das gewöhnliche Open-Source-Modell von massiv paralleler technischer Unterstützung skaliert schlicht und einfach nicht auf der Ebene, die für Weltherrschaft nötig ist.

Es wird keine Explosion geben, wenn die Foren anfangen überzulaufen. Es wird nur einen leisen Folge negativer Rückmeldungen geben: Leute melden sich von den Verteilern ab, oder verlassen das IRC, oder hören zumindest auf sich die Mühe zu machen Fragen zu stellen, da sie sehen können, dass man Sie in dem ganzen Lärm nicht gehört werden. Sowie immer mehr Leute diese im hohen Maße vernünftige Wahl treffen, wird die Aktivität auf dem Forum aussehen als wäre es auf einer annehmbaren Stufe. Es es bleibt genau deshalb annehmbar, weil die Vernünftigen (oder zumindest erfahrenen) Leute anfangen an anderen Orten nach Informationen zu suchen – während die unerfahrenen zurück bleiben und weiterhin Nachrichten schreiben. Mit anderen Worten, während das Projekt wächst, ist eine weitere Nebenwirkung von nicht-skalierbaren Kommunikationsmodellen, dass die durchschnittliche Qualität sowohl der Fragen, als auch der Antworten dazu neigt schlechter zu werden, was es danach aussehen lässt, als wären die neuen Nutzer blöder als es ehemals der Fall war, obwohl das wahrscheinlich nicht der Fall ist. Es ist nur, dass das Kosten/Nutzen-Verhältnis dieser Foren mit einer hohen Bevölkerung weniger wird, also fangen diejenigen mit mehr Erfahrung natürlich zuerst an woanders nach Antworten zu suchen. Die Kommunikationsmechanismen anzupassen, um mit dem Wachstum des Projekts umzugehen, beinhaltet deshalb zwei verwandte Strategien:

1. Zu erkennen, wann bestimmte Teile eines Forums *nicht* unter unbegrenztem Wachstum leiden, selbst wenn das für das Forum insgesamt der Fall ist, und diese Teile in neue, speziellere Foren zu Trennen (d.H. lassen Sie das Gute nicht vom Schlechten herunterziehen).

2. Stellen Sie sicher, dass viele automatisierte Informationsquellen zur Verfügung stehen, und dass Sie organisiert, aktuell, und leicht auffindbar gehalten werden.

Strategie (1) ist gewöhnlich nicht all zu schwer. Die meisten Projekte fangen mit einem Hauptforum an: Ein Mailverteiler für allgemeine Diskussionen, auf dem Ideen für neue Funktionen, Fragen zum Aufbau der Software, und programmier-Probleme zerlegt werden können. Jeder der an dem Projekt beteiligt ist, ist auf dem Verteiler. Nach einer Weile, wird es für gewöhnlich offensichtlich, dass der Verteiler sich in mehrere unterschiedliche Unterverteiler, auf bestimmten Themen basierend, aufgeteilt hat. Zum Beispiel geht es in manchen Threads eindeutig um Entwicklung und Aufbau des Codes; andere sind eher Benutzerfragen, von der Art "Wie mache ich X?"; es gibt vielleicht eine dritte Familie die sich um die Verarbeitung von Bug-Meldungen und Erweiterungs-Wünsche dreht; und so weiter. Eine beliebige Person mag sich natürlich an vielen verschiedenen Thread-Arten beteiligen, das Wichtige ist aber, dass nicht viel Überlappung zwischen den verschiedenen Arten gibt. Sie könnten in separate Verteiler aufgeteilt werden ohne irgend eine schädliche Balkanisierung zu verursachen, da die Threads selten mehrere Themen überspannen.

Diese Aufteilung wirklich zu machen, ist ein Vorgang mit zwei Schritten. Sie werden einen neuen Verteiler (oder einen IRC-Kanal, oder was immer es auch sein soll), und dann müssen Sie welche Zeit auch

immer nötig ist, um Leute zu nerven und erinnern die neuen Foren entsprechend zu *nutzen*. Letzteres kann Wochen andauern, letztendlich werden es die Leute aber kapieren. Sie müssen es sich nur zum Prinzip machen, dem Absender immer zu sagen, wenn er an das falsche Ziel geschickt hat, und es auf eine sichtbare Art tun, damit andere dazu ermutigt werden, beim umleiten mitzuhelfen. Es ist auch nützlich, eine Webseite mit einem Wegweiser zu allen Verteilern bereit zu haben; Ihre Rückmeldungen können einfach auf diese Seite verweisen und als Bonus, mag der Empfänger etwas darüber lernen, nach Richtlinien zu suchen, vor er eine Nachricht schreibt.

Strategie (2) ist ein anhaltender Vorgang, welcher die ganze Lebensdauer des Projekts überdauern kann und mit vielen Beteiligten zu tun haben kann. Natürlich handelt es sich dabei zum Teil auch um die Frage eine aktuelle Dokumentation zu haben (siehe „Dokumentation“ im Kapitel Kapitel 2, *Der Einstieg*) und weisen Sie Leute darauf hin. Es ist aber auch viel mehr als das; die Abschnitte die folgen, behandeln diese Strategie im detail.

Auffällige Nutzung der Archive

Typischerweise, werden alle Kommunikationen in einem Open-Source-Projekt (außer manchmal IRC-Unterhaltungen) archiviert. Die Archive sind öffentlich, können durchsucht werden und referenzen darauf bleiben Stabil: Dass heißt, sobald ein Stück Information unter einer bestimmten Adresse aufgezeichnent wird, bleibt es ewig an dieser Adresse.

Benutzen Sie diese Archive so viel wie möglich, und so auffällig wie möglich. Selbst wenn Sie die Antwort auf eine Frage aus dem Kopf wissen, wenn Sie denken, dass es eine Referenz auf die gleiche Frage in den Archiven gibt, welche die Antwort beinhaltet, nehmen Sie sich die Zeit es auszugraben und zu präsentieren. Jedes mal wenn Sie das auf eine öffentlich sichtbare Art tun, lernen manche Leute zum ersten mal, dass es die Archive gibt, und sie zu durchsuchen, Antworten hervorbringen kann. Indem Sie auf die Archive referenzieren, bestärken Sie auch die soziale Norm, Informationen nicht zu duplizieren. Warum die gleiche Antwort an zwei verschiedenen Stellen haben? Wenn die Anzahl der Stellen an der es gefunden werden kann, auf ein minimum beschränkt wird, werden Leute die es vorher gefunden haben sich eher daran erinnern wonach sie suchen müssen um es wieder zu finden. Gut platzierte Referenzen tragen auch allgemein zu der Qualität der Suchergebnisse bei, da Sie den Rang der referenzierten Ressourcen in Suchmachinen erhöhen.

Es gibt Zeiten, wann die Verdopplung von Informationen jedoch Sinn macht. Nehmen wir zum Beispiel an, es gäbe bereits eine Rückmeldung in den Archiven, nicht von Ihnen, welche sagt:

```
Es scheint, dass Ihre Scanley-Indexe verfrobbt wurden. Führen Sie diese
Schritte aus, um sie zu entfrobben:

1. Schalten Sie den Scanley-Server aus.
2. Lassen Sie das 'Entfrobb'-Programm, welches mit Scanley
ausgeliefert wird, aus.
3. Starten Sie den Server.
```

Monate später, sehen Sie dann eine weitere Nachricht, welche andeutet, dass die Indexe von jemand verfrobbt wurden. Sie suchen in den Archiven und finden obige Rückmeldung, sehen aber, dass ein paar Schritte Fehlen (vielleicht aus versehen, oder weil die Software sich, seitdem die Nachricht geschrieben wurde, geändert hat). Die eleganteste Art das zu handhaben, ist es eine neue vollständigere Nachricht zu schreiben, und die alte Information explizit als obsolet zu markieren, indem Sie es erwähnen:

```
Es scheint, dass Ihre Scanley Indexe verfrobbt wurden. Wir haben
dieses Problem im July gesehen und H. Mustermann schrieb eine
Lösung bei http://blahblahblah/blah. Unten ist eine vollständigere
```

```
Beschreibung wie Sie Ihre Indexe entfrobben können, basierend auf
der Anleitung von H. Mustermann aber ein wenig erweitert:

1. Schalten Sie den Scanley-Server aus.
2. Wechseln Sie zum Benutzer, unter dem Scanley normalerweise läuft.
3. Lassen Sie mit diesem Benutzer das 'Entfrobb'-Programm über die
Indexe laufen.
4. Lassen Sie Scanley per Hand laufen, um zu sehen, ob die Indexe jetzt
funktionieren.
5. Starten Sie den Server neu.
```

(In einer idealen Welt wäre es möglich, eine Anmerkung an die alte Nachricht anzuhängen, welche sagt, dass eine neuere Version der Information zur Verfügung steht, und auf die neue Nachicht hinzuweisen. Mir ist allerdings keine Archivierungssoftware bewusst, welche eine "obsolet durch" Funktion anbietet, vielleicht wäre es ein wenig schwierig, auf eine Art zu implementieren, welche die Vorgabe eines Archivs, eine wortgetreue Aufzeichnung zu sein, nicht verletzen würde. Das ist ein weiterer Grund, warum es eine gute Idee ist, Webseiten zu erstellen, die der Beantwortung von häufigen Fragen gewidmet sind.)

Archive werden wahrscheinlich am häufigsten nach Antworten zu technischen Fragen durchsucht, ihre Bedeutung für das Projekt geht jedoch weit darüber hinaus. Wenn die formalen Richtlinien eines Projekts seine in einer Satzung festgelegten Gesetze sind, sind die Archive seine allgemeinen Gesetze: Eine Aufzeichnung aller Diskussionen die geführt wurden, und wie sie erreicht wurden. Bei jeder wiederkehrenden Diskussion, ist es heutzutage quasi bindend, mit einer Suchen in den Archiven anzufangen. Das erlaubt es Ihnen, die Diskussion mit einer Zusammenfassung des derzeitigen Standes anzufangen, Einsprüche vorauszuahnen, Konterargumente vorzubereiten, und möglicherweise Sichtweisen zu entdecken, an der Sie noch nicht gedacht hatten. Die anderen Beteiligten, werden auch *erwarten* , dass Sie eine Durchsuchung des Archivs gemacht haben. Selbst wenn vorangehende Diskussionen nirgendwo hingeführt haben, sollten Sie Verweise darauf einbeziehen, wenn Sie das Thema wieder aufgreifen, damit Leute sich selber davon überzeugen können a) dass sie nirgends hingeführt haben, und b) dass Sie Ihre Hausaufgaben gemacht haben, und desshalb wahrscheinlich etwas sagen, was noch keiner zuvor gesagt hat.

Behandeln Sie alle Ressourcen wie Archive

Alle vorhergehenden Ratschläge gelden für mehr als nur die Archive von Mailverteilern. Bestimmte Informationsstücke an einer stabilen, leicht auffindbaren Adresse zu haben, sollte ein Organisationsprinzip alle Informationen des Projekts sein. Lass uns die FAQ des Projekts als Fallstudie betrachten.

Wie nutzen Leute eine FAQ?

1. Sie wollen es nach bestimmten Worten und Begriffen durchsuchen.
2. Sie wollen darin stöbern, Informationen aufsaugen, ohne zwangsläufig nach Antworten zu bestimmten Fragen zu suchen.
3. Sie erwarten, dass Suchmachinene wie Googel über den Inhalt der FAQ bescheid wissen, damit Suchergebnisse Einträge der FAQ beinhalten können.
4. Sie wollen Leute direkt auf bestimmte Einträge in der FAQ verweisen können.
5. Sie wollen neues Material zu der FAQ hinzufügen können, bedenken Sie aber, dass das sehr viel weniger geschiedt, als die Antworten aufgerufen werden – in einer FAQ wird sehr viel öfter geschieben als gelesen.

Punkt 1 impliziert, dass die FAQ in irgend einem textuellen Format zur verfügung stehen sollte. Punkte 2 und 3 implizieren, dass die FAQ als HTML Seite zur verfügung stehen sollte, zusätzlich impliziert

Punkt 2, dass diese HTML Datei auf eine lesbare Art aufgebaut sein soll, (d.H. Sie werden etwas Einfluss auf sein Aussehen), und sollte ein Inhaltsverzeichnis haben. Punkt 4 bedeutet, dass jeder einzelne Eintrag in der FAQ einen HTML *benannten Verweis (en. "anchor")* zugewiesen haben sollte, ein tag welches es Leute erlaubt, eine bestimmte Stelle in der Seite auf eine bequeme Art zu erreichen (siehe „Versioniere alles“ im Kapitel Kapitel 3, *Technische Infrastruktur*), in einem Format welches leicht zu bearbeiten ist.

Benannte Verweise und ID-Attribute

Es gibt zwei Möglichkeiten einen Browser dazu zu bringen an einer bestimmte Stelle in einer Webseite zu springen: benannte Verweise und id Attribute.

Ein *benannter Verweis* is lediglich ein normaler HTML Verweis (`<a>...</a>`), aber mit ein "name" Attribut:

```
<a name="meinEtikett">...</a>
```

Neuere Versionen von HTML unterstützen eine generisches *id-Attribut*, welches an jedem HTML-Element angehängt werden kann, nicht nur an `<a>`. Zum Beispiel:

```
<p id="meinEtikett">...</p>
```

Sowohl benannte Verweise als auch id-Attribute werden auf die gleiche Art benutzt. Eines hängt eine Raute und den Bezeichner an eine URL, um den Browser dazu zu bringen direkt zu der Stelle in der Seite zu springen:

```
http://meinprojekt.beispiel.de/faq.html#meinEtikett
```

Praktisch alle Browser unterstützen benannte Verweise; die meisten modernen Browser unterstützen das id-Attribut. Um sicher zu gehen, würde ich Ihnen empfehelen, entweder nur benannte Verweise zu benutzen, oder benannte Verweise *und* id-Attribute zusammen (natürlich mit dem gleichen Bezeichner für beide bei einem gegebenen Paar). Benannte Verweise können sich nicht nicht selber Abschließen – selbst wenn es keinen Text in dem Element gibt, müssen Sie es trotzdem in der vollständigen form Ausschreiben:

```
<a name="meinEtikett"></a>
```

...obwohl es normalerweise etwas Text, wie den Titel von dem Abschnitt geben würde.

Ob Sie einen benannten Verweis benutzen oder ein id-Attribut oder beides, denken Sie daran, dass der Bezeichner nicht für jemandem sichtbar ist, der zu der Stelle blättert, ohne den Bezeichner zu benutzen. Solch eine Person mag aber vielleicht den Bezeichner für eine bestimmte Stelle herausfinden wollen, um zum Beispiel die URL an einem Freund als Antwort zu schicken. Um ihnen dabei zu helfen, fügen Sie einen *title-Attribut* an dasselbe Element bei dem Sie das "name" und/oder id-Attribut hizugefügt haben:

```
<a name="meinEtikett"
  title="#meinEtikett">...</a>
```

Wenn der Mauszeiger über den Text innerhalb des Elements platziert wird, auf den das title-Attribut angewandt wurde, werden die meisten Browser einen kleinen Kasten aufklappen, in dem der Titel angezeigt wird. Ich füge normalerweise das Raute-Zeichen hinzu, um die Leute daran zu erinnern, es am Ende der URL zu platzieren um beim nächsten Mal direkt an diese Stelle zu springen.

Die FAQ zu formatieren, ist ledigleich eines der Beispiele, wie Sie eine Ressource vorzeigbar machen können. Die gleichen Eigenschaften – direkte Durchsuchbarkeit, Verfügbarkeit für größere Suchmaschi-

nen, stabile Referenzen, und (wo möglich) Bearbeitbarkeit – gelten für andere Webseiten, den Quellcode-Baum, den Bugtracker usw. Es ist zufällig so, dass die meiste Software zur Archivierung von Mailinglisten schon lange erkannt haben, wie wichtig diese Eigenschaften sind, weshalb Mailverteiler dazu neigen, diese Funktionen von sich aus eingebaut zu haben, während andere Formate unter Umständen etwas zusätzliche Mühe seitens des wartenden (Kapitel 8, *Leitung von Freiwilligen* behandelt, wie Sie diese Bürde der Wartung über viele Entwickler verteilen können).

Festschreiben von Traditionen

Während das Projekt Geschichte und Komplexität ansammelt, nimmt die Menge an Daten zu, die ein neuer Teilnehmer sich aneignen muss. Solche die schon seit langen bei dem Projekt sind, konnten die Konventionen des Projekts im laufe der Zeit lernen und erfinden. Sie werden sich oft nicht der riesigen Menge an Traditionen die sich angesammelt hat bewusst sein, und mögen überrascht sein, wie viele Fehlschritte neue Mitglieder scheinbar machen. Natürlich, liegt das nicht daran, dass die Neuen von einer schlechteren Qualität sind als vorher; es ist lediglich, dass Sie einer größeren bürde der kulturellen Anpassung als in der Vergangenheit gegenüberstehen.

Die Traditionen, die ein Projekt ansammelt haben genau soviel damit zu tun, wie man komunizieren und Informationen bewahren, wie es um die Festhaltung von Standards und andere technische Kleinigkeiten geht. Wir haben uns bereits beide Standards im Kapitel „Entwickler-Dokumentation“ sowie Kapitel 2, *Der Einstieg* und „Schriftliche Regeln“ im Kapitel Kapitel 4, *Soziale und politische Infrastruktur* jeweils, und mit Beispielen angeschaut. In diesem Abschnitt, geht es darum, wie man solche Richtlinien aktuell hält, während das Projekt sich weiter entwickelt, insbesondere Richtlinien darüber, wie Kommunikation gehandhabt wird, denn das sind diejenigen die sich am meisten ändern wärend das Projekt an Größe und Komplexität zunimmt.

Erstens, halten Sie ausschau danach, wie Leute verwirrt werden. Wenn Sie die gleichen Situationen immer wieder auftauchen sehen, insbesondere bei neue Beteiligten, ist die Wahrscheinlichteit groß, dass eine Richtlinie dokumentiert werden muss, es aber noch nicht ist. Zweitens, werden Sie nicht müde die gleichen Sachen immer wieder zu sagen, und *klingen* Sie nicht, als würden Sie langsam davon müde werden. Sie und die anderen Verteranen im Projekt werden sich oft wiederholen müssen; es ist ein unausweichliche Nebenwirkung von dem Ankommen von Neuankömmlinge.

Jede Webseite, jede Nachrich im Mailverteiler, und jeder IRC-Kanal sollte als eine Werbefläche betrachtet werden – nicht für kommerzielle Werbung, sondern für Werbung über die Ressourcen Ihres eigenen Projekts. Was Sie auf dieser Fläche anbringen, hängt ab von der Demographie von denen die es wahrscheinlich lesen werden. Ein IRC-Kanal für Benutzerfragen zum Beispiel, wird wahrscheinlich Leute anlocken, die noch nie zuvor mit dem Projekt zu tun hatten – oftmals jemand der eben erst die Software installiert hat, und eine Frage hat, die er gleich beantwortet haben möchte (schließlich hätte er es statt dessen an den Mailverteiler schicken können, wenn es hätte warten können, was wahrscheinlich weniger von seiner Zeit insgesamt in anspruch genommen hätte, obwohl es länger gedauert hätte, bis er eine Antwort erhalten hätte). Leute machen für gewöhnlich keine permanente Investition in einem IRC-Kanal; sie tauchen auf, stellen ihre Frage und gehen wieder.

Desshalb, sollte das Thema des Kanals auf Leute abgesehen sein, die technische Fragen *genau jetzt*über die Software haben, eher als sagen wir, Leute die sich vielleicht auf längere Sicht an dem Projekt beteiligen wollen und für die Richtlinien für den Umgang innerhalb der Gemeinschaft eher angebracht wären. Ein wirklich betriebsammer Kanal handhabt es folgendermaßen (Vergleichen Sie es, mit dem früheren Beispiel in „IRC / Echtzeit-Nachrichtendienste“ im Kapitel Kapitel 3, *Technische Infrastruktur*):

```
Sie reden jetzt in #linuxhelp

Das Thema für #linuxhelp ist Bitte LESEN Sie
http://www.catb.org/~esr/faqs/smart-questions.html &&
```

```
http://www.tldp.org/docs.html#howto BEVOR Sie Fragen stellen | Regeln
für den Kanal finden Sie bei http://www.nerdfest.org/lh_rules.html |
Bitte schauen Sie sich http://kerneltrap.org/node/view/799 bevor Sie
Fragen über einen Upgrade nach Kernel 2.6.x stellen | Speicher Abfrage
möglich: http://tinyurl.com/4s6mc -> update nach 2.6.8.1 oder
2.4.27 | hash algo Katastrophe: http://tinyurl.com/6w8rf
| reiser4 weg
```

Bei Mailverteilern ist die "Werbefläche" eine kleine Fußnote, die an jeder Nachricht angehängt wird. Die meisten Projekte schreiben Anweisungen zur Anmeldung/Abmeldung dort, und vielleicht einen Hinweis auf die Webseite des Projekts oder auch von der FAQ. Sie denken vielleicht, das jeder der an dem Veriteiler angemeldet ist, wissen würde, wie man diese Informationen findet, und das ist wahrscheinlich auch der Fall – aber viel mehr Leute als nur die angemeldeten sehen diese Nachrichten des Verteilers. Auf eine archivierte Nachricht, kann von vielen Stellen aus verwiesen werden; tatsächlich werden manche Nachrichten derart bekannt, dass sie letztendlich mehr Leser erhält, als der gesamte Verteiler angemeldete Nutzer hat.

Formatierung kann einen großen Unterschied machen. Im Subverison-Projekt zum Beispiel hatten wir eingeschränkten Erfolg bei der Verwendung der Methode zur Filterung von Bugs, beschrieben in „Vor-Filterung des Bugtrackers“ im Kapite Kapitel 3, *Technische Infrastruktur*. Viele falsche Bug-Meldungen wurden immer noch von unerfahrenen Personen eingereicht, und jedes mal, als das geschah, musste der Absender auf genau die gleiche Art aufgeklärt werden, wie die 500 Personen vor ihm. Eines tages, nahdem bei einem unserer Entwickler schließlich der Faden gerissen war, und irgend einen armen Nutzer zugeflamed hatte, der die Richtlinien für den Bugtracker nicht sorgfältig genug gelesen hatte, entschied sich ein anderer Entwickler, dass dieses Muster schon viel zu lange gelaufen war. Er schlug vor, dass wir die erste Seite des Bugtrackers neu formatieren sollten, so dass der wichtigste Teil, die Vorgabe den Bug zuerst auf dem Mailverteiler oder im IRC zu diskutieren, vor man einen Bug meldet, in riesigen, fetten, roten Buchstaben auf einem gelben Hintergrund, in der Mitte der Seite über alles andere stehen würde. Das machten wir (Sie können das Ergebnis bei folgender Seite sehen http://subversion.tigris.org/project_issues.html), und das Ergebnis, war ein merklicher Abfall in der Anzahl der falschen Bug-Meldungen. Wir bekommen sie natürlich immer noch – das wird immer der Fall sein – aber die Menge ist wesentlich weniger geworden, selbst wärend die Anahl der Nutzer zunimmt. Die Folge ist, dass die Bug-Datenbank nicht nur weniger Müll enthält, sondern, dass diejenigen die auf diese Meldungen reagieren, besser gelaunt bleiben und eher freundlich bleiben, wenn sie auf einer der nunmehr seltenen Falschmeldungen reagieren. Das verbessert sowohl das Erscheinungsbild des Projekts als auch die psychische Verfassung seiner Freiwilligen.

Die Lektion die wir hierraus ziehen konnten war, dass es nicht ausreichte, die Richtlinien einfach nur nieder zu schreiben. Wir mussten sie auch dort platzieren wo sie von denjenigen gefunden werden, die sie am meisten brauchen, und sie so formatieren, dass ihr status als Einführungsmaterial sofort für Personen klar wird, die noch nicht mit dem Projekt vertraut sind.

Statische Webseiten sind nicht die einzige Ort um für die Bräuche des Projekts zu werben. Eine bestimmte Menge an interaktiver Kontrolle (im sinne von freundlichen Hinweisen, nicht grober Zurechtweisung) ist auch notwendig. Jede Überprüfungm auch die Überprüfung von Commits, beschrieben in „Code Review“ im Kapitel Kapitel 2, *Der Einstieg*, sollten auch eine Evaluierung in wiefern es mit den Normen des Projekts übereinstimmt oder nicht, insbesondere im Bezug auf Konventionen über die Kommunikation.

Ein weiteres Beispiel aus dem Subversion-Projekt: Wie einigten uns auf die Konvention wonach "r12908" für "Revision 12908 des Projektarchivs der Versionsverwaltung." Das klein geschriebene vorangehende "r" ist einfach zu schreiben und da es lediglich die halbe höhe der Zahlen ist, ergibt es einen leicht zu ekennnenden Textblock, wenn man es mit den Zahlen kombiniert. Sich auf diese Konventien zu einigen, bedeutet natürlich nicht, dass jeder gleich anfangen wird, sie gleich zu benutzen. Deshalb, wenn eine Commit-E-Mail mit folgendem Kommentar eintrifft:

```
------------------------------------------------------------------------
r12908 | qsimon | 02-02-2005 14:15:06 -0600 (Mit, 02 Feb 2005) | 4 Zeilen

Patch vom Freiwilligen H. Mustermann <hmustermann@gmail.com>

* trunk/contrib/client-side/psvn/psvn.el:
  Einige Rechtschreibfehler in revision 12828 behoben.
------------------------------------------------------------------------
```

...gehört es zur Überprüfung des Commits, zu sagen "Übrigens, benutze bitte die 'r12828', nicht die 'revision 12828' schreibweise um auf vergangene Änderungen zu verweisen." Das ist nicht einfach nur pedantisch; es ist genau so wichtig um es automatisch parsen zu können, wie für die menschliche Leserschaft.

Indem Sie das allgemeine Prinzip verfolgen, dass es kanonische Methoden geben sollte, um auf häufige Entitäten zu verweisen und dass man diese verweis Methoden überall auf eine konsistente Art verwenden sollte, exportiert das Projekt in Wirklicheit gewisse Standards. Diese Standards ermöglichen es andere, Werkzeuge zu schreiben, welche die Kommunikationen des Projekts auf andere brauchbarere Arten zu presentieren – zum Beispiel könnte eine Revision mit dem Format "r12828" in einen Link zu der Seite des Projektarchivs verwandelt werden. Das wäre schwieriger, wenn die Revision als "revision 12828" geschrieben würde, sowohl weil diese Form durch einen Zeilenumbruch getrennt werden könnte, und weil es weniger eindeutig ist (das Wort "revision" wird häufig alleine erscheinen, und gruppen von Zahlen erscheinen heufig alleine, wohingegen die Kombination "r12828 sich nur auf eine Revisionsnummer beziehen kann). Ähnliche Bedenken gelten für die Meldungsnummern im Bugtracker, FAQ Einträge (tipp: Benutzen Sie eine URL mit einem benannten Verweis, wie in Benannte Verweise und ID-Attribute beschrieben), usw.

Selbst bei Einträgen, wo es keinen offensichtlichen kurze, kanonische Form gibt, sollten Leute dazu ermutigt werden, wesentliche Informationen einheitlich anzugeben. Wenn man zum Beispiel auf eine Nachricht im Mailverteiler verweist, geben Sie nicht nur den Absender und die Überschrift an; geben Sie auch die URL im Archiv *und* den Message-ID-Header an. Letzteres erlaubt es Leute die ihre eigene Kopie des Mailverteilers haben, (manche Leute halten offline Kopieen, zum Beispiel auf einem Laptop für die Nutzung auf Reisen) eindeutig die richige Nachricht zu identifizieren, selbst wenn sie keinen Zugriff auf die Archive haben. Der Absender und die Überschrift würden dazu nicht ausreichen, weil dieselbe Person vielleicht mehrere Nachrichten im selben Thread schreibt, sogar am selben Tag.

Je mehr ein Projekt wächst, desto wichtiger wird diese Art von Konsistenz. Konsistent bedeutet, dass überall wo Leute hinschauen, sie die gleichen Muster in verwendung sehen, also wissen sie auch sie selber zu befolgen. Das wiederum, verringert die Anzahl der Fragen die sie stellen müssen. Die Bürde millionen Leser zu haben ist nicht größer, als die einen zu haben; Probleme der Skalierbarkeit fangen nur dann aufzutreten, wann ein gewisser Prozentsatz dieser Leser Fragen stellen. Wärend ein Projekt wächst, muss es desshalb diesen Prozentsatz verringern, indem es die Dichte und Verfügbarkeit von Informationen erhöht, damit jede beliebige Person eher das findet, wonach sie sucht, ohne fragen zu müssen.

Keine Unterhaltungen im Bugtracker

Bei jedem Projekt, welches aktiv seinen Bugtracker benutzt, gibt es immer die Gefahr, dass dieser Tracker sich selbst in ein Diskussionsforum verwandelt, obwohl der Mailverteiler in Wirklichkeit besser wäre. Für gewöhnlich, fängt es ganz unschuldig an: Jemand hängt einen Kommentar an eine Meldung, sagen wir mit einem Lösungsvorschlag, oder einem unvolständigen Patch. Jemand anderes sieht das und bemerkt, dass es Probleme mit der Lösung gibt, und hängt einen weiteren Kommentar an um sie zu schildern. Die erste Person antwortet wiederrum, auch wieder mit einem Kommentar an der Meldung... und so geht es weiter.

Das Problem dabei ist erstens, dass der Bugtracker ein ziemlich unbequemer Ort ist, um eine Unterhaltung zu führen, und zweitens, dass andere Leute vielleicht nicht zuschauen – schließlich erwarten Sie, dass Diskussionen über die Entwicklung auf dem Entwickler Verteiler stattfinden, also ist das der Ort an dem sie suchen. Es mag sein, dass sie überhaupt nicht bei dem Verteiler für Änderungen im Bugtracker angemeldet sind, und wenn sie es sind, kann es sein, dass sie es nicht sonderlich gründlich verfolgen.

Aber wo genau bei dem Ablauf ging etwas schief? War es als die ursprüngliche Person ihre Lösung an die Meldung anhängte – hätte sie lieber eine Nachricht an den Verteiler schreiben sollen? Oder war es als die zweite Person in der Meldung reagierte, anstatt an den Verteiler?

Es gibt keine eine richtige Antwort, aber es gibt eine allgemeine Regel: Wenn Sie lediglich Daten an einer Meldung anhängen, dann machen Sie es auf dem Tracker, wenn Sie aber eine *Unterhaltung* anfangen, dann machen Sie es auf dem Mailverteiler. Es mag sein, dass Sie nicht immer erkennen können, was der Fall ist, nutzen Sie aber einfach Ihre Urteilsvermögen. Wenn Sie zum Beispiel, einen Patch anhängen, welches eine möglicherweise kontroverse Lösung beinhaltet, kann es sein, dass Sie vorrausahnen können, das Leute Fragen darüber haben werden. Obwohl Sie also normalerweise den Patch an die Meldung anhängen würden (angenommen Sie wollen nicht oder lönnen nicht direkt einen Commit der Änderung machen), wäre es in diesem Fall vieleicht eher angemessen statt dessen, eine Nachricht an den Mailverteiler zu schicken. In jedem Fall wird es aber einen Punkt geben, an dem die eine oder andere Partei erkennen kann, dass es gleich von dem einfachen Anhängen von Informationen zu einer echten Unterhaltung übergeht – in dem Beispiel mit dem dieser Abschnitt anfing, wäre das die zweite Person, welche in dem Moment, wo er bemerkt, dass es Probleme mit dem Patch gibt, hätte vorrausahnen können, dass eine echte Diskussion sich gleich entwickelt, und desshalb in dem angemessenen Medium gehalten werden sollte.

Um einen Vergleich zur Mathematik zu führen, wenn die Information danach aussieht, als würde es schnell konvergieren, dann stellen Sie es gleich in den Bugtracker; wenn es danach aussieht als würde es divergieren, dann wäre ein Mailverteiler oder ein IRC-Kanal der bessere Ort.

Das bedeutet nicht, dass es niemals Austausche auf dem Bug Tracker geben sollte. Dem Meldenden nach weiteren Details einer Anleitung zur Reproduktion zu fragen, neigt beispielsweise dazu ein im hohen maße konvergenter Ablauf zu sein. Die Rückmeldung wird wahrscheinlich keine neuen Probleme hervorbringen, es wird lediglich die Informationen erweitern, die bereits erfasst wurden. Es gibt keinen Grund den Mailverteiler mit diesem Vorgang abzulenken; Sie können es durchaus in den Kommentaren des Bugtrackers erledigen. Gleichermaßen gilt, wenn Sie sich ziemlich sicher sind, dass der Bug fälschlicherweise gemeldet wurde (d.h. kein Bug ist), dann können Sie es gleich in der Meldung sagen. Selbst auf ein kleines Problem bei einer vorgeschlagenen Lösung hinzuweisen ist in Ordnung, angenommen es ist kein Problem welches die Lösung gänzlich verhindert.

Wenn Sie andererseits philosophische Angelegenheiten über den Ramen der Meldung oder das Verhalten der Anwendung aufbringen, können Sie sich sicher sein, dass andere Entwickler beteiligt sein wollen. Die Diskussion wird wahrscheinlich eine weile lang divergieren, bevor es konvergiert, also schreiben Sie an den Mailverteiler.

Verlinken Sie immer von der Meldung auf den Thread im Mailverteiler. Es ist immer noch wichtig für jemand der die Meldung verfolgt, in der Lage zu sein, die Diskussion zu erreichen, selbst wenn die Meldung selber nicht das Forum der Diskussion ist. Die Person die den Thread anfängt, mag das ein wenig aufwendig finden, aber in Open-Source-Kultur hat im allgemeinen der Autor die Verantwortung: Es ist viel wichtiger Sachen für duntzende oder hunderte Leute die den Bug möglicherweise lesen als für die drei oder fünf die darüber schreiben.

Es ist in Ordnung, wichtige Schlussfolgerungen oder Zusammenfassungen aus der Listen-Diskussion in die Meldung zu übertragen, wenn das es für die Leser einfacher macht. Ein häufiger Ablauf ist es, eine E-Mail-Diskussion anzufangen, einen Link an den Thread in der Meldung zu schreiben, und dann sobald die Diskussion zu Ende ist, die finale Zusammenfassung in die Meldung zu kopieren (zusammen mit

einem Link zu der Nachricht welche die Zusammenfassung beinhaltet), damit jemand die sich die Meldung anschaut, leicht erkennen kann, welche Lösung beschlossen wurde, ohne irgendwo anders hinclicken zu müssen. Man merke an, dass die gewöhnliche Problematik doppelter Informationen hier nicht gibtm da sowohl Archive als auch Kommentare in einer Meldung, im allgemeinen sowieso statische, unveränderliche Daten sind.

Öffentlichkeit

Bei freier Software gibt es einen relativ glatten Übergang zwischen rein internen Diskussionen und öffentlichen Bekanntmachungen. Das liegt zum Teil daran, dass die Zielgruppe immer schlecht bestimmbar ist: Angesichts, dass die meisten oder alle Nachrichten öffentlich zugänglich sind, hat das Projekt nicht die volle Kontrolle darüber, welchen Eindruck die Öffentlichkeit bekommt. Jemand – sagen wir ein heise.de Mitarbeiter – kann die Aufmerksamkeit von Millionen Lesern auf eine Nachricht richten, von der keiner vermutet hätte, dass sie jemals außerhalb des Projekts gesehen werden würde. Das ist eine Tatsache des Lebens, mit der alle Open-Source-Projekte leben müssen, in der Praxis aber, ist das Risiko für gewöhnlich klein. Im algemeinen, sind die Bekanntgaben die das Projekt am ehesten öffentlich verbreiten will, diejenigen die auch am ehesten bekannt werden, angenommen Sie nutzen die richtigen Mechanismen darauf hinzuweisen, wie berichtenswert es ist für die Öffentlichkeit ist.

Für großere Bekanntgaben, gibt es allgemein vier Hauptkanäle der Verbreitung, auf den Melungen so gleichzeitig wie möglich gemacht werden sollten:

1. Die erste Seite Ihres Projekts wird wahrscheinlich von mehr leuten gesehen, als irgend ein anderer Teil des Projekts. Wenn Sie eine wirklich große Ankündigung haben, schreiben Sie einen paar kurzen Sätze dorthin. Sie sollten eine kurze Zusammenfassung sein die auf die Pressemitteilung (siehe unten) für weitere Informationen verweist.

2. Gleichzeitig, sollten Sie auch einen "Nachrichten" oder "Pressemitteilungen" Bereich auf der Webseite haben wo die Bekanntgabe im detail erläutert werden kann. Der Sinn von einer Pressemitteilung liegt zum Teil darin, ein einziges kanonisches "Bekanntgabe-Objekt" zu haben, auf das andere Seite verlinken können, also stellen Sie sicher, dass es entsprechend strukturiert ist: entweder als eine Webseite für jede neue Version, als diskreten Eintrag in einem Blog, oder als irgend eine andere Art Entität auf die verlinkt werden kann, und die trotzdem eindeutig von anderen Veröffentlichungen imselben Bereich zu unterscheiden ist.

3. Wenn Ihr Projekt einen RSS-Feed hat, stellen Sie sicher, dass die Ankündigung auch dort veröffentlicht wird. Das mag automatisch passieren wenn Sie die Pressemitteilung erstellen, abhängig davon, wie Ihre Webseite eingerichtet ist. (*RSS* ist eine Methode, um Nachrichten-Zusammenfassungen mit einer Menge Metadaten an "Abonenten" zu verteilen, zu verbreiten, also Leute die ein Interesse gezeigt haben, diese Zusammenfassungen zu bekommen. Siehe http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html für weitere Informationen über RSS.)

4. Wenn es in der Ankündigung um eine neue Version der Software geht, dann aktualisieren Sie den Eintrag Ihres Projekts bei http://freshmeat.net/ (siehe „Bekanntgabe“ in dem es darum geht diesen Eintrag überhaupt erst anzulegen). Jedes mal, wenn Sie einen Freshmeat-Eintrag aktualisieren, geht dieser Eintrag auf die Änderungsliste von Freashmeat für den Tag. Die Änderungsliste wird nicht nur auf Freashmeat selber aktualisiert, sondern auf verschiedenen anderen Portalen (inklusive slashdot.org) welches erwartungsvoll von einer Horde Menschen beobachtet wird. Freshmeat bietet auch die gleichen Daten mittels eines RSS-Feed an, damit Leute die den RSS-Feed von Ihrem projekt aboniert haben, die Ankündigung trotzdem durch den von Freshmeat sehen.

5. Schreiben Sie eine E-Mail an den Verteiler für Ankündigungen (en. announcement) ihres Projekts. Der Name von diesem Verteiler sollte wirklich "announce" sein, d.h. `announce@ihrprojekt.org`, weil das mittlerweile so ziemlich zum Standard geworden ist, und Ihr Projekt sollte klarstellen, dass es einen sehr geringen Betrieb hat, ausschließlich für Projekt Ankündigunen. Die meisten

dieser Ankündigunen werden sich um neue Versionen der Software drehen, gelegentlich aber andere Ereignisse, wie eine Spendaktion, die Entdeckung einer Sicherheitslücke (siehe „Bekanntgabe von Sicherheitslücken“) später in diesem Kapitel, oder eine größere Änderung in der Richtung des Projekts können dort auch gemeldet werden. Da es ein Verteiler mit geringem Betrieb ist und nur für wichige Sachen benutzt wird, hat der announce Verteiler typischerweise die höchste Anzahl an Anmeldungen von allen Verteilern in Ihrem Projekt (das bedeutet natürlich, dass Sie es nicht missbrauchen sollten – überlegen Sie genau vor Sie etwas schreiben). Um zu vermeiden, dass jeder beliebige, oder schlimmer noch, Spam durchkommt, muss der announce Verteiler immmer moderiert werden.

Versuchen Sie die Ankündigunen an all diesen Stellen zur gleichen Zeit zu machen, so nahe wie möglich. Manche werden vielleicht verwirrt, wenn Sie eine Ankündigung auf dem Verteiler sehen, aber keinen entsprechenden Eintrag auf der Webseite des Projekts oder seinen Pressemitteilungen, welches dem entspricht. Wenn Sie die die verschiedenen Änderungen (E-Mails, Bearbeitung der Webseite, usw.) aufreihen und alle in einem Zug abschicken, können Sie das Fenster der inkonsistenz sehr klein halten.

Bei einem weniger wichtigen Ereignis, können Sie manche oder alle obigen Kanäle ausschließen. Das Ereignis wird immer noch von der Öffentlichkeit entsprechend seiner Wichtigkeit bemerkt werden. Während zum Beispiel eine neue Version der Software ein großes Ereignis ist, ist lediglich das Datum für eine neue Version festzulegen, wenn auch einigermaßen Berichtenswert, nich annähernd so wichtig wie die neue Version selbst. Ein Datum festzulegen ist eine E-Mail an die täglichen Mailverteiler wert (nicht an den "announce" Verteiler), und eine aktualisierung der Zeitplan oder der Status-Seite, aber nicht mehr.

Sie werden vielleicht trotzdem dieses Datum in anderen Diskussionen im Internet auftauchen sehen, überall dort, wo Leute an dem Projekt interesiert sind. Leute die auf Ihren Verteilern herumschleichen, lediglich horchen und nie etwas sagen, sind nicht unbedingt an andern Stellen still. Mundpropaganda sorgt für eine sehr weite Verbreitung; Sie sollten darauf zählen, und selbst kleine Ankündigunen sorgfältig ausarbeiten, derart, dass eine korrekte übertragung der Informationen ermutigt wird. Insbesondere Nachrichten, von denen Sie erwarten, dass sie zitiert werden, sollten einen klaren für die Zitierung gedachten Anteil haben, genau so, als ob Sie eine formale Pressemitteilung schreiben würden. Zum Beispiel:

> *Nur eine Information über den aktuellen Stand: Wir planen die Version 2.0 von Scanley mitte August 2005 zu veröffentlichen. Sie können immer http://www.scanley.org/status.html auf Aktualisierungen überprüfen. Die große neue Funtion wird die Suche mit Regulären Ausdrücken sein.*
>
> *Andere neue Funktionen sind unter anderem: ... Es wird auch verschiedene Bugfixes geben, unter anderem: ...*

Der erste Absatz ist kurz, gibt die zwei wichtigsten Informationen (Datum der Veröffentlichung und die wichtige neue Funktion), und eine URL die man für weitere Nachrichten besuchen kann. Wenn dieser Absatz das einzige ist, was über den Bildschirm von jemanden läuft, sind Sie immer noch auf einem guten Weg. Die übrige E-Mail könnte verloren gehen ohne das wesentlich am Ihnalt zu beeinflussen. Manchmal werden Leute sowieso auf die ganze E-Mail verlinken, aber genau so oft, werden Sie nur einen kleinen Teil zitieren. Wenn man letztere Möglichkeit bedenkt, können Sie es ihnen auch genau so gut einfach machen, und bei dem Geschäft etwas Einfuss darüber haben, was zitiert wird.

Bekanntgabe von Sicherheitslücken

Eine Sicherheitslücke zu handhaben ist anders, als jede andere Art von Bug-Meldung zu handhaben. Bei freier Software, ist es fast schon eine religiöse Überzeugung alles offen zu machen. Jeder Schritt der im Ablauf der normalen Bug handhabung ist für alle sichtbar, denen es interesiert: Das Eintreffen einer ersten Meldung, die darauf folgende Diskussion, und die letztendliche Behebung.

Sicherheitslücken sind anders. Sie können die Daten und möglicherweise die Rechner von Nutzer kompromitieren. Solche Probleme offen zu diskutieren käme dem weltweiten Bewerben ihrer Existenz gleich – inklusive gegenüber allen Parteien, die vielleicht einen böswilligen Nutzen aus dem Bug ziehen könnten. Selbst den Fix zu commiten, kündigt effektiv die Existenz des Bugs an (es gibt potentielle Angreifer, welche die Commit-Kommentare öffentlicher Projekte verfolgen, systematisch auf der Suche nach Änderungen, die auf Sicherheitsprobleme in dem noch nicht geänderten Code hinweisen). Die meisten Projekte haben sich auf ungefär die gleichen Schritte festgelegt, um diesen Konflikt zwischen Offenheit und Geheimhaltung zu handhaben, basierend auf folgende Richtlinien:

1. Reden Sie nicht öffentlich über den Bug bis ein Fix verfügbar ist; stellen Sie dann den Fix zu genau der selben Zeit zur Verfügung wie Sie den Bug bekanntgeben.

2. Denken Sie sich diesen Fix so schnell wie möglich aus – insbesondere wenn jemand von außerhalb des Projekts den Bug gemeldet hat, denn dann wissen Sie, dass es zumindest eine Person außerhalb des Projekts gibt, der in der Lage ist die Lücke auszunutzen.

In der Praxis, führen diese Prinzipien zu einer ziemlich standardisierte reihe an Schritten, welche in den Abschnitten weiter unten beschrieben werden.

Empfang der Meldung

Ein Projekt muss offensichtlich die Möglichkeit habem Meldungen über Sicherheitslücken von jedem zu bekommen. Die gewöhnlichen Adressen für Bug-Meldungen reichen aber nicht aus, da sie auch von jedem beobachtet werden können. Führen Sie desshalb einen getrennten Mailverteiler um Bug-Meldungen über Sicherheitslücken zu bekommen. Dieser Verteiler darf keine öffentlich zugänglichen Archive haben, und seine Anmeldungen müssen strengstens überwacht werden – nur langjährige, vertrauenswürdige Entwickler dürfen auf den Verteiler sein. Wenn Sie eine formale Definition darüber haben wollen was "vertrauenswürdig" ist, können Sie "jeder der seit zwei oder mehr Jahren commit Zugriff hat" verwenden, um Bevorzugung zu vermeiden. Das ist die Gruppe die Sicherheitslücken handhaben wird.

Im Idealfall sollte der Sicherheits-Verteiler nicht vor Spam geschützt oder moderiert werden, da Sie nicht wollen, dass eine wichtige Meldung gefiltert oder verzögert wird, nur weil an dem Wochenende keine Moderatoren online waren. Wenn Sie doch Software zum automatischen Schutz vor Spam benutzen, versuchen Sie es mit einer hohen Toleranz zu konfigurieren; es ist besser ein paar Spam-Mails durchkommen zu lassen, als eine Meldung zu verpassen. Damit der Verteiler effektiv ist, müssen Sie natürlich seine Adresse bewerben; in anbetracht, dass er nicht moderiert wird, wenn überhaupt, nur leicht vor Spam geschützt wird, versuchen Sie nie seine Adresse ohne irgend eine Art verschleierung der Adresse zu verbreiten, wie in „Verschleierung von Adressen im Archiv“ im Kapitel Kapitel 3, *Technische Infrastruktur* beschrieben. Glücklicherweise, muss die Verschleierung der Adresse sie nicht unbedingt unleserlich machen; siehe http://subversion.tigris.org/security.html, und schauen Sie sich den HTML Quellcode für die Seite als Beispiel an.

Entwickeln Sie den Fix im stillen

Was macht der sicherheits Verteiler also wenn es eine Meldung bekommt? Die erste Aufgabe ist zu evaluieren wie schwerwiegend und dringend es ist:

1. Wie ernst ist die Lücke? Erlaubt es einen böswilligen Angreifer den Computer von jemanden der Ihre Software benutzt zu übernehmen? Oder gibt es lediglich Informationen darüber, wie groß manche ihrer Dateien sind?

2. Wie leicht ist es die Lücke auszunutzen? Kann ein Angriff automatisiert werden, oder erfordert es Wissen über bestimmte Umstände, wohlbegründete Vermutungen und Glück?

3. *Wer* hat Ihnen das Problem gemeldet? Die Antwort auf diese Frage ändert natürlich nichts an der Natur der Lücke, aber es gibt Ihnen eine Idee davon, wie viele andere darüber bescheid wissen könn-

ten. Wenn die Meldung von einem der eigenen Mitglieder des Projekts kommt, können Sie es etwas gelassener nehmen (aber nur ein bisschen), da Sie ihnen Vertrauen können, dass sie es keinen anderen gesagt haben. Wenn es andereseits in einer E-Mail von `anonymer14@globalehacker.net` kam, dann sollten Sie lieber so schnell wie möglich etwas unternehmen. Die Person hat Ihnen einen Gefallen getan, indem sie Ihnen überhaupt über das Problem informiert hat, aber Sie haben keine Ahnung wie viele weitere sie darüber bescheid gesagt hat, oder wie lange sie warten wird, vor sie die schwachstelle in einem laufenden System ausnutzen wird.

Man merke an, dass die Unteschiede von denen wir hier reden oft nur eine enge Spanne zwischen *dringend* und *extrem dringend* ist. Selbst wenn die Meldung von einer bekannten, wohl gesinnten Quelle kommt, könnte es immer noch andere im Netz geben, die diesen Bug schon vor langem entdeckt haben, und es nur nicht gemeldet haben. Der einzige Fall bei dem es nicht dringen ist, ist wenn der Bug von sich aus die Sicherheit nicht ernsthaft Kompromitiert.

Das "`anonyer@globalehacker.net`" Beispiel ist im übrigen nicht schertzhaft gemeint. Es kann durchaus sein, dass Sie Bug-Meldungen von anonymen Personen bekommen, die mit Ihren Worten nie ganz klar stellen ob sie auf Ihrer Seite sind oder nicht. Es macht keinen Unterschied: Wenn Sie eine Sicherheitslücke an Sie gemeldet haben, werden sie das gefühl haben Ihenen eine Gefallen getan zu haben, und Sie sollten entsprechend antworten. Danken Sie ihnen für die Meldung, geben Sie ein Datum an vor dem Sie vorhaben einen öffentlichen Fix freizugeben, und halten Sie sie den Kontakt. Manchmal werden sie *Ihnen* ein Datum geben – das heißt, eine implizite Drohung den Bug an einem bestimmten Datum bekannt zu geben, ob Sie bereit sind oder nicht. Das mag sich nach einschüchterndem Machtspiel anfühlen, aber es ist wahrscheinlich eher eine vorsorgliche Maßnahme die sich aus vergangener Enttäschung mit nicht reagierenden Software-Produzenten, welche die Sicherheitsmeldungen nicht ernst genug nahmen. In jedem Fall, können Sie es sich nicht leisten, dieser Person auf die Nerven zu gehen. Wenn der Bug ernst ist, hat er schließlich Wissen, welches Ihren Nutzern ernste Probleme bereiten könnte. Behandeln Sie solche Personen gut, und hoffen Sie darauf, dass sie auch gut behandelt werden.

Eine weitere Person die häufig Bugs meldet, ist der Sicherheitsprofi, jemand der davon lebt, Code zu überprüfen und sich auf dem Laufenden über die neusten Software-Schwachstellen hält. Diese Leute haben für gewöhnlich Erfahrung mit beiden Seiten des Zauns – sie haben sowohl Meldungen gemacht als auch bekommen, wahrscheinlich mehr als die meisten in Ihrem Projekt. Auch sie werden wahrscheinlich ein Frist setzen, vor dem eine Lücke geschlossen werden muss, vor sie an die öffentlichkeit gehen. Die Frist mag etwas verhandelbar sei, das hängt aber vom Meldenden ab; Fristen sind unter Sicherheitsexperten so ziemlich als die einzige Möglichkeit anerkannt worden, Organisationen zuverlässig dazu zu bringen, sich um Sicherheitsprobleme zeitnah zu kümmern. Betrachten Sie die Frist also nicht als unhöflich, sie ist eine altehrwürdige Tradition und es gibt gute Gründe dafür.

Sobald Sie wissen wie schwerwiegend und dringend eine Lücke ist, können Sie sich an dem Fix zu schaffen machen. Es gibt manchmal einen Kompromiss zwischen einen eleganten Fix und es schnell zu machen; das ist der Grund warum Sie sich vorher über die Dringlichkeit einigen müssen vor Sie anfangen. Halten Sie die Diskussion über den Fix natürlich auf die Mitglieder des Sicherheitsverteilers beschränkt, sowie auf den ursprünglichen Meldenden (wenn die beteiligt sein möchte) und solche Entwickler die aus technischen Gründen herangezogen werden müssen.

Committen Sie den Fix nicht zum Projektarchiv. Halten Sie ihn als Patch bereit bis zum Datum der Bekanntgabe. Wenn Sie es committen würden, selbst mit einem unscheinbaren Kommentar, könnte jemand die Änderung bemerken und verstehen. Sie wissen nie, wer Ihr Projektarchiv im Auge behält und warum sie vielleicht interesiert sein könnten. Commit-E-Mails abzuschalten würde nichts bringen; erstens weil sich die Lücke in der Reihe der E-Mails verdächtig aussehen würde, und die Daten wären immer noch im Projektarchiv. Machen Sie einfach die ganze Entwicklung in einem Patch und halten Sie ihn an einem privaten Ort, vielleicht ein getrenntes privates Projektarchiv, das nur denjenigen bekannt ist, die schon über den Bug Bescheid wissen. (Wenn Sie eine dezentralisierte Versionsverwaltung benut-

zen wie Arch oder SVK, können Sie die Arbeit mit kompletter Versionsverwaltung machen und dieses Projektarchiv einfach vorm Zugriff für Ausenstehende schützen.)

CAN/CVE-Nummer

Sie haben vielleicht eine *CAN-Nummer* oder eine *CVE-Nummer* im Zusammenhang mit Sicherheitsproblemen gesehen. Diese Zahlen sehen für gewöhnlich zum Beispiel so "CAN-2004-0397" oder so "CVE-2002-0092" aus.

Beide Zahlenarten repräsentieren den gleichen Entitätstyp: Ein Eintrag in der Liste von "Common Vulnerabilities and Exposures" (de. "Verbreitete Schwachstellen und Aufdeckungen") die bei http://cve.mitre.org/ gepflegt wird. Der Sinn dieser Liste ist es standardisierte Namen für alle bekannte Sicherheitsprobleme zur verfügung zu stellen, damit jeder einen eindeutigen, kanonischen Namen bei der Diskussion eines solchen hat, und einen zentralen Ort zu haben, um mehr Informationen zu bekommen. Der einzige Unterschied zwischen einer "CAN"-Nummer und einer "CVE"-Nummer ist, dass erstere einen Kandidaten-Eintrag repräsentiert, welches noch nichtbestätigt wurde, und letzteres einen bestätigten Eintrag repräsentiert. Beide Typen sind jedoch für die Öffentlichkeit sichtbar, und die Nummer für einen Eintrag ändert sich nicht, wenn Sie bestätigt wird – der "CAN"-Prefix wird einfach mit "CVE" ersetzt.

Ein CAN/CVE-Eintrag entält selber nicht eine vollständige Beschreibung von dem Bug und wie Sie sich vor ihm schützen. Stattdessen, enthält es eine kurze Beschreibung und eine Liste von Verweisen auf externe Ressourcen (wie die Archive des Verteilers) wo Leute detalliertere Infomationen nachschlagen können. Der wirkliche Sinn von http://cve.mitre.org/ ist einen wohl organisierten Ort zu haben, in dem jede Schwachstelle einen Namen haben und eine klare Route zu weiteren Daten haben kann. Siehe http://cve.mitre.org/cgi-bin/cvename.cgi?name=2002-0092 als Beispiel für einen Eintrag. Beachten Sie, dass Verweise sehr kurz gehalten sein können, mit Quellen die als kryptische Abkürzungen erscheinen. Ein Schlüssel zu diesen Abkürzungen befindet sich bei http://cve.mitre.org/cve/refs/refkey.html.

Wenn Ihre Schwachstelle die Kriterien für ein CVE erfüllt, kann es sein, dass Sie dafür eine CAN-Nummer bekommen wollen. Der Prozess dafür ist absichtlich umständlich gehalten: Im Prinzip müssen Sie jemanden kennen, oder jemanden kennen der jemand anderen kennt. Das ist nicht so verrückt wie es sich anhören mag. Damit die CVE-Redaktion es vermeiden kann von unechten oder schlecht geschriebenen Vorschlägen überwältigt wird, nehmen Sie nur Vorschläge von Quellen, die sie bereits kennen und denen Sie vertrauen. Damit Ihre Lücke in der Liste aufgenommen wird, müssen Sie desshalb einen Pfad von Bekanntschaften von Ihrem Projekt hin zu der CVE-Redaktion finden. Fragen Sie bei Ihren Entwicklern nach; einer von Ihnen wird wahrscheinlich jemand anderen kennen der bereits durch den CAN-Prozess gelaufen ist, oder der jemand kennt der es gemacht hat, usw. Der Vorteil dieser Methode ist auch, das irgendwo in der Kette jemand sein kann, der genug weiß, um Ihnen sagen zu können, dass a) es nicht als Lücke oder Aufdeckung nach den Kriterien von MITRE zählen würde, es also keinen Sinn hat es vorzuschlagen, oder b) dass die Lücke bereits eine CAN- oder CVE-Nummer *hat*. Letzteres kann passieren, wenn der Bug bereits auf einer andern Liste für Sicherheitsberatung veröffentlicht wurde, zum Beispiel bei http://www.cert.org/ oder auf dem BugTraq-Mailverteiler bei http://www.securityfocus.com/. (Wenn das passiert ist, ohne das Ihr Projekt davon erfahren hat, dann sollten Sie sich sorgen machen, was noch möglicherweise abläuft worüber Sie nichts wissen.)

Wenn Sie überhaupt eine CAN/CVE-Nummer bekommen, werden Sie sie wahrscheinlich im frühen Stadium Ihrer Bug-Untersuchung bekommen, damit alle weitere Kommunikation sich auf diese Nummer beziehen kann. CAN-Einträge stehen unter einem Embargo bis zu dem Veröffentlichungsdatum; Der Eintrag wird als leerer Platzhalter existieren (damit Sie den Namen nicht verlieren), es wird aber keine Informationen über die Lücke enthüllen, bist zu dem Datum an dem Sie den Bug und den Fix bekanntgeben.

Weitere Informatione über den CAN/CVE-Prozess kann bei http://cve.mitre.org/about/candidates.html gefunden werden, und eine besonders klare Darstellung von dem Nutzen eines Open-Source-Projekts der CAN/CVE-Nummern befindet sich auf http://www.debian.org/security/cve-compatibility.

Vorankündigung

Sobald Ihr Sicherheitsteam (also die Entwickler die auf dem sicherheits Verteiler sind, oder die zu einer bestimmten Meldung herangezogen wurden) eine Fix bereit hat, müssen Sie entscheiden wie Sie es Verbreiten wollen.

Wenn Sie den Fix einfach zu Ihrem Projektarchiv committen, oder sonstwie der Welt bekanntgeben, zwingen Sie effektiv alle die Ihre Software benutzen sofort zu aktualisieren oder das Risiko einzugehen, gehackt zu werden. Es ist deshalb manchmal angemessen, für bestimmte Nutzer eine *Vorankündigung* zu machen. Das gilt insbesondere bei Client-Server-Software, bei dem es wenige wohl bekannte Server geben kann, die verlockende Ziele für Angreifer sind. Die Administratoren dieser Server wären dankbar darüber einen zusetzlichen Tag oder zwei für die Aktualisierung zu haben, damit Sie bereits geschützt sind, wenn die Lücke öffentlich bekannt wird.

Vorankündigung bedeutet lediglich eine E-Mail an diese Administratoren zu senden, vor dem Veröffentlichungsdatum, in dem Sie ihnen über die Lücke bescheid geben und wie sie es beheben können. Sie sollten eine Vorankündigung nur an Personen schicken, die Sie diese diskrete Information anvertrauen können. Die Qualifikation um eine Vorankündigung zu erhalten hat also zwei Bedingungen: Der Empfänger muss eine große Installation in Betrieb haben, wichtige Server bei dem eine Kompromitierung etwas ernstes wäre, *und* der Empfänger muss jemanden sein, der sich über das Sicherheitsproblem vor dem Veröffentlichungsdatum verplappert.

Senden Sie jede E-Mail mit einer Vorankündigung einzeln (eines nach dem anderen) an jedem Empfänger. Senden Sie *nicht* an die komplette Liste von Empfängern auf ein mal, da sie dann ihre gegenseitigen Namen sehen würden – was zur Folge hat, dass Sie im Grunde genommen jedem Empfänger über die Tatsache aufmerksam machen, dass jeder *andere* Empfänger möglicherweise die Sicherheitslücke auf seinem Empfänger hat. Es an alle mittels einem blinden CC (BCC) zu schicken ist auch keine gute Lösung, da manche Administratoren ihre Mail-Konten mit Spam-Filtern schützen, die entweder BCC-Mails blockieren, oder ihre Priorität reduzieren, da heutzutage sehr viel Spam mit BCC verschickt wird.

Hier ist ein Beispiel E-Mail für eine Vorankündigung:

```
Von: Ihr Name hier
An: admin@grosser-bekannter-server.de
Antwort-an: Ihr Name hier (nicht die Adresse des Sicherheitsverteilers)
Betreff: Vertrauliche Meldung einer Scanly-Sicherheitslücke.


Diese E-Mail ist eine vertrauliche Vorankündigung einer
Sicherheitsmeldung zum Scanley-Server.

Bitte leiten Sie keinen Teil dieser E-Mail an irgend jemand weiter. Die
öffentliche Bekanntgabe wird nicht vor dem 19. Mai stattfinden, und
wir würden die Information gerne bis dann unter Verschluss halten.

Sie erhalten diese E-Mail, da (wir denken, dass) Sie einen Scanley-Server
betreiben, und wir ihn lieber gepatcht hätten, bevor die Sicherheitslücke
am 19ten Mai bekannt gegeben wird.

Verweise:
===========

   CAN-2004-1771: Scanley Speicherüberlauf in Abfragen
```

```
Sicherheitslücke:
==============

   Der Server kann dazu gebracht werden, beliebige Befehle auszuführen,
   wenn die Locale des Servers falsch konfiguriert ist und der Client
   eine fehlerhafte Abfrage macht.

Sicherheitsgrad:
=========

   Sehr ernst, kann die Ausführung von beliebigen Code auf dem Server
   zur Folge haben.

Provisorische Lösungen:
============

   Die 'natural-language-processing'-Einstellung in der
   scanley.conf auf 'off' zu stellen, schließt die Sicherheitslücke.

Patch:
======

   Der folgende Patch gilt für Scanley 3.0, 3.1, und 3.2.

   Eine neu Version (Scanley 3.2.1) wird am oder kurz vor dem 19. Mai
   veröffentlicht werden, damit es zur gleichen Zeit verfügbar istm wie
   die Bekanntgabe der Lücke. Sie können dem Patch jetzt anwenden oder
   einfach auf die neue Version warten. Der einzige Unterschied
   zwischen 3.2 und 3.2.1 wird diese Patch sein.

[...Patch folgt hier...]
```

Wenn Sie eine CAN-Nummer haben, geben Sie sie bei der Vorankündigung mit an (wie oben gezeigt), selbst wenn die Information unter Verschluss steht und die MITRE Seite desshalb leer sein wird. Die CAN-Nummer erlaubt es dem Empfänger mit sicherheit zu wissen, dass der Bug über den sie eine Vorankündigung bekommen haben, der gleiche ist von dem sie später durch öffentliche Kanäle erfahren, also müssen sie sich keine Sorgen darüber machen, ob weitere Schritte notwendig sind, was genau der Sinn einer CAN/CVE-Nummer ist.

Verteilen Sie den Fix öffentlich

Der letzte Schritt bei der Handhabung einer Sicherheitslücke, ist den fix öffentlich zu verteilen. Sie sollten in einer einzigen, umfangreichen Meldung, das Problem beschreiben, die CAN/CVE-Nummer angeben, falls vorhanden, beschreiben wie das Problem zu umgehen ist, und wie es permanent 'gefixed' werden kann. 'Fix' bedeutet für gewöhnlich auf eine neuere Version der Software zu aktualisieren, obwohl es manchmal die Anwendung von einem Patch bedeuten kann, insbesondere wenn die Software sowieso meistens vom Quellcode-Format aus betrieben wird. Wenn Sie doch eine neue Version machen, sollte es sich von einer existierenden Version durch genau diesen sicherheits-Patch unterscheiden. So können konservative Administratoren, aktualisieren, ohne sich sorgen zu machen, was sie noch beeinflussen könnten; sie müssen sich also keine Sorgen über zukünftige Upgrades machen, da der sicherheits Patch als Folge in allen zukünftigen Versionen enthalten sein wird. (Details der Abläufe für neue Versionen werden in „Sicherheitsupdates“ im Kapitel Kapitel 7, *Paket-Erstellung, Veröffentlichung, und tägliche Entwicklung* behandelt.)

Ob der öffentliche Fix eine neue Version zur Folge hat oder nicht, machen Sie die Meldung mit ungefähr der der gleichen Priorität wie Sie eine neue Version machen würden: Senden Sie eine E-Mail an den announce Verteiler des Projekts, machen Sie eine neue Pressemitteilung, aktualisieren Sie den Freshmeat-Eintrag, usw. Während Sie nie versuchen sollten, die Existenz einer Sicherheitslücke herunterzuspielen, aus Sorge um den Ruf des Projekts, können Sie durchaus den Ton und die Gewichtung der Sicherheitsmeldung anpassen, um den Ernst des Problems zu entsprechen. Wenn die Sicherheitslücke nur eine kleine Offenlegung von Informationen ist, nicht eine Lücke die es erlaubt den kompletten Computer des Anwenders zu übernehmen, mag es keinen all zu großen Wirbel gerechtfertigen. Sie können sich sogar Entscheiden den announce Verteiler damit abzulenken. Wenn das Projekt schließlich jedes mal Wolf schreit, werden sich die Nutzer denken, dass die Software weniger sicher ist, als das tatsächlich der Fall ist, und es möglicherweise nicht glauben wenn Sie wirklich ein riesiges Problem ankündigen müssen. Siehe http://cve.mitre.org/about/terminology.html als eine gute Einführung zu dem Problem der Beurteilung wie schwerwiegend eine Sicherheitslücke ist.

Im Allgemeinen, wenn Sie sich nicht sicher sind, wie Sie ein Sicherheitsproblem behandeln sollen, suchen Sie sich jemand Erfahrenen und reden Sie darüber. Lücken zu handhaben und zu bewerten, ist größtenteils eine angelernte Fähigkeit und es ist leicht, die ersten paar Male Fehler zu machen.

Zum Umgang mit Sicherheitslücken siehe auch die Richtlinien der Apache Software Foundation unter http://www.apache.org/security/committers.html. Sie bieten eine ausgezeichnete Checkliste, anhand derer Sie prüfen können, ob Sie alle Punkte sorgfältig bedacht haben.

Kapitel 7. Paket-Erstellung, Veröffentlichung, und tägliche Entwicklung

In diesem Kapitel geht es darum, wie freie Software-Projekte ihre Software-Pakete erstellen und veröffentlichen, und wie sich allgemeine Entwicklungsmuster um diese Ziele herum organisieren.

Ein großer Unterschied zwischen Open Source und proprietären Projekten ist der Mangel einer zentralen Kontrolle über das Entwicklerteam. Wenn eine neue Version vorbereitet wird, ist der Unterschied besonders gravierend: Ein Unternehmen kann das gesamte Entwicklerteam darum bitten, sich auf die bevorstehenden Version zu konzentrieren und die Entwicklung neuer Funktionen und die Behebung unkritischer Bugs ruhen zu lassen, bis die neue Version fertig ist. Gruppen von Freiwilligen sind nicht derart monolitisch. Sie arbeiten aus allen möglichen Gründen an dem Projekt, und diejenigen, die nicht daran interessiert sind, bei einer bestimmten Version zu helfen, wollen immer noch ihre normale Entwicklung weiterführen, während die neue Version vorbereitet wird. Da die Entwicklung niemals aufhört, neigen der Herausgabe-Prozesse in Open-Source-Projekten dazu, länger zu dauern, verursachen aber weniger Unruhe, als die kommerziellen Herausgabe-Prozesse. Es ist ein wenig wie die Reparatur einer Autobahn. Es gibt zwei Möglichkeiten, eine Straße zu reparieren: Sie können sie komplett absperren, damit die Reparaturmannschaft in voller Kapazität ausschwärmen kann, bis das Problem gelöst ist, oder Sie können jeweils nur ein paar Spuren gleichzeitig bearbeiten, während die anderen für den Verkehr offen bleiben. Die erste Methode ist sehr effizient *für die Reparaturmannschaft*, aber für niemanden sonst – die Straße ist komplett blockiert, bis die Arbeit erledigt ist. Die zweite Methode verursacht mehr Arbeitszeit und Kopfschmerzen für die Reparaturmannschaft (jetzt müssen sie mit weniger Leuten und Mitteln, unter eingeengten Bedingungen arbeiten, mit Schildern um den Verkehr zu verlangsamen und zu lenken, usw.), allerdings bleibt die Straße weiterhin benutzbar, wenn auch nicht mit der vollen Kapazität.

Open-Source-Projekte tendieren dazu nach der zweiten Methode zu arbeiten. Bei einer ausgereiften Software ist es sogar so, dass mehrere verschiedene Versionslinien gleichzeitig gepflegt werden, das Projekt befindet sich in einer Art ständigen Straßenreparatur. Es sind immer ein paar Spuren geschlossen; beständige aber geringe Umstände im Hintergrund werden die ganze Zeit über von der Entwicklergemeinschaft toleriert, damit die neuen Version plangemäß fertig werden.

Das Modell, das dies ermöglicht, lässt sich auf mehr als nur neue Versionen verallgemeinern. Es ist das Prinzip, Aufgaben zu parallelisieren, die nicht von einander abhängen – ein Prinzip das natürlich keinesfalls auf Open-Source-Entwicklung beschränkt ist, das jedoch von Open-Source-Projekten auf eine jeweils eigene bestimmte Art umgesetzt wird. Sie können es sich nicht leisten, ihre Straßenbau-Truppe oder den gewöhnlichen Verkehr zu sehr zu nerven, aber sie können es sich auch nicht leisten Leute dafür abzustellen, bei den orangen Kegeln zu stehen und den Verkehr zu dirigieren. Sie streben deshalb eher zu Abläufe die einen flachen, konstanten Grad an Mehraufwand haben, als Höhen und Tiefen. Freiwillige sind im Allgemeinen eher dazu bereit mit kleinen gleichbleibenden Mengen an Unbequemlichkeiten zu arbeiten; die Berechenbarkeit erlaubt es ihnen zu kommen und zu gehen, ohne sich darüber Sorgen zu machen, ob ihr Terminkalender mit dem was im Projekt passiert kollidiert. Wenn das Projekt aber eine einem Produktionsplan unterliegen würde, wäre das Ergebnis, eine Menge Entwickler die die meiste Zeit untätig herumsitzen – was nicht nur ineffizient wäre, sondern auch langweilig und dadurch gefährlich, insofern, dass ein gelangweilter Entwickler wahrscheinlich bald ein Ex-Entwickler sein wird.

Arbeit an neuen Versionen ist für gewöhnlich die am ehesten bemerkbare Aufgabe die nicht zur Entwicklung gehört, welche neben der Entwicklung her läuft, also sind die Methoden die in den folgenden Abschnitten folgen meistens darauf ausgelegt neue Versionen zu ermöglichen. Beachten Sie jedoch,

dass sie auch für andere Aufgaben gelten, die sich parallelisieren lassen, wie Übersetzungen und Lokalisierung, weitgreifende Änderungen an den Schnittstellen die nach und nach über den gesamten Quellcode gemacht werden, usw.

Versionszählung

Vor wir uns darüber unterhalten, wie man eine neue Version macht, lasst uns anschauen, wie man diese Versionen benennt, wozu wir wissen müssen was eine neue Version für die Benutzer tatsächlich bedeutet. Eine neue Version bedeutet, dass:

- Alte Bugs behoben wurden. Das ist wahrscheinlich eines der Sachen auf den sich die Benutzer für jede neue Version verlassen können.
- Neue Bugs wurden hinzugefügt. Darauf kann man sich für gewöhnlich auch verlassen, außer in manchen Fällen, bei der Behebung von Sicherheitslücken oder andere Einmalige Änderungen (siehe „Sicherheitsupdates“ später in diesem Kapitel).
- Neue Funktionen können hinzugefügt worden sein.
- Neue Konfigurationseinstellung können hinzugefügt worden sein, oder die Bedeutung alter Einstellungen sich ein klein wenig geändert haben. Die Installationsabläufe können sich auch seit der letzten Version leicht geändert haben, auch wenn man immer hofft, dass das nicht der Fall ist.
- Nicht kompatible Änderungen können eingeführt worden sein, wie die Formatierung der Daten welche von älterer Software benutzt werden nicht weiter ohne irgend einer (möglicherweise händischen) Einweg-Konvertierung benutzt werden können.

Wie sie sehen, ist nicht alles davon etwas gutes. Deshalb gehen erfahrene Benutzer immer mit ein wenig Angst an neue Versionen heran, ganz besonders, wenn die Software ausgereift ist und vorher schon zum größten Teil das gemacht hat, was sie wollten (oder dachten das sie wollten). Selbst der Einbau neuer Funktionen hat insofern nicht nur Vorteile, dass es bedeuten könnte, dass sich die Software jetzt unerwartet verhält.

Der Sinn einer Versionsnummer ist deshalb zweifältig: Offensichtlich sollte die Nummer unzweideutig die Reihenfolge der Versionen mitteilen (d.h. wenn man sich zwei Versionsnummern anschaut, kann man unterscheiden, welches die später kam), sie sollten aber auch so kompakt wie möglich den Grad und Art der Änderungen in der Version andeuten.

Das alles in einer Zahl? Nun, im Grunde genommen, ja. Die Strategien für Versionsnummern sind eine der ältesten Fahrradschuppen-Diskussionen, die es gibt (siehe „Je weicher das Thema, desto länger die Debatte“ im Kapitel Kapitel 6, *Kommunikation*), und ist unwahrscheinlich, dass sich die Welt auf einen, vollständigen Standard in der irgendwann in der nächsten Zeit einigt. Ein paar gute Strategien sind aber entstanden, zusammen mit dem universell anerkannten Prinzip: *konsistent zu sein*. Wählen Sie ein Nummerierungsschema und bleiben Sie dabei. Ihre Nutzer werden es Ihnen danken.

Die Komponenten der Versionsnummer

Dieser Abschnitt beschreibt im Detail die formalen Konventionen der Nummerierung von Versionen, und geht von sehr wenig Vorwissen aus. Es ist hauptsächlich als eine Referenz gedacht. Wenn Sie bereits mit diesen Konventionen vertraut sind, können Sie diesen Abschnitt überspringen.

Eine Versionsnummer ist eine Gruppe von Zahlen die durch punkte getrennt sind:

Scanley 2.3
Singer 5.11.4

...und so weiter. Die Punkte sind *keine* Dezimalzeichen, sie sind lediglich Trennzeichen; nach "5.3.9" käme "5.3.10". Ein paar wenige Projekte haben ab und zu auf was anderes hingedeutet, am bekanntesten der Linux-Kernel mit seiner "0.95", "0.96"... "0.99" Reihe die zu Linux 1.0 hinführte, die Konvention, dass die Punkte keine Dezimalzeichen sind, ist jetzt aber fest etabliert und sollte als Standard betrachtet werden. Es gibt keine Grenze bei der Anzahl der Komponenten (Ziffernsequenzen, die keine Punkte enthalten), aber die meisten Projekte gehen nicht über drei oder vier hinaus. Die Gründe dafür werden später klar.

Zusätzlich zu den numerischen Komponenten, hängen Projekte manchmal erläuternde Kennschrift wie "Alpha" oder "Beta" an (siehe Alpha und Beta), als Beispiel:

Scanley 2.3.0 (Alpha)
Singer 5.11.4 (Beta)

Ein Alpha- oder Beta-Vermerk bedeutet, dass diese Version einer zukünftigen *vorausgeht* welches die selbe Zahl haben wird, jedoch ohne den Vermerk. Deshalb führt "2.3.0 (Alpha)" letztendlich zu "2.3.0". Um mehrere solche Kandidaten in einer Folge zu ermöglichen, können die Vermerke selber Meta-Vermerke haben. Hier ist als Beispiel eine Serie von Versionen, in der Reihenfolge mit der sie veröffentlicht werden würden:

Scanley 2.3.0 (Alpha 1)
Scanley 2.3.0 (Alpha 2)
Scanley 2.3.0 (Beta 1)
Scanley 2.3.0 (Beta 2)
Scanley 2.3.0 (Beta 3)
Scanley 2.3.0

Beachten Sie, dass es mit dem "Alpha" Vermerk, Scanley "2.3" als "2.3.0" geschrieben wird. Die beiden Zahlen sind gleich – folgende Nullanteile können der Kürze halber immer weggelassen werden – wenn ein Vermerk aber vorhanden ist, ist die Kürze eh schon nicht gegeben, also kann man genau so gut vollständig ausschreiben.

Andere Vermerke die mehr oder weniger oft benutzt werden sind "Stable" (de. stabil), "Unstable" (de. nicht stabil), "Development" (de. Entwicklerversion), und "RC" (für "Release Candidate") (de. Kandidat für eine finale Version). Die am weitesten verbreiteten sind immer noch "Alpha" und "Beta", mit "RC" nahe an dritter Stelle, beachten Sie aber, dass zu "RC" immer einen Meta-Vermerk gehört. Sie veröffentlichen als nicht "Scanley 2.3.0 (RC)", sondern "Scanley 2.3.0 (RC 1)", gefolgt von RC2, usw.

Diese drei Vermerke, "Alpha", "Beta", und "RC", sind mittlerweile relativ gut bekannt, und ich empfehle nicht irgend welche andere zu benutzen, auch wenn andere auf dem ersten Blick passender erscheinen da sie normale Wörter sind, und kein Jargon. Diejenigen die neue Software-Versionen installieren, sind aber bereits mit den großen dreien vertraut, und es gibt keinen Grund die Sache grundlos anders von allen anderen zu machen.

Obwohl die Punkte in Versionsnummer keine Dezimalzeichen sind, deuten sie doch einen Stellenwert an. Alle "0.X.Y" Versionen gehen "1.0" voraus (was natürlich gleichbedeutend ist mit "1.0.0"). "3.14.158" geht direkt "3.14.159" voraus, und indirekt "3.14.160", sowie "3.15.irgendwas" und ähnliches vorausgeht.

Eine konsistente Richtlinie für die Numerierung der Versionen ermöglicht es den Benutzer zwei Versionsnummern der selben Software anzuschauen und alleine durch die Zahlen, die wichtigen Unterschiede zwischen ihnen zu unterscheiden. Bei einer typischen System aus drei Komponenten, ist die erste Komponente die *major Nummer*, die zweite ist die *minor Nummer*, und die dritte ist die *micro Nummer*. Version "2.10.17" ist zum Beispiel die siebzehnte Micro-Version in der zehnten Minor-Reihe innerhalb der zweiten Major-Versions-Serie. Die Worte "Reihe" (en. line) und "Serie" (en. series) werden hier infor-

mell verwendet, sie bedeuten aber was man erwarten würde. Eine Major-Serie ist lediglich alle Versionen die die gleich major Nummer teilen und eine minor Serie (oder Reihe) besteht aus allen Versionen welche die gleiche minor *und* major Nummer teilen. "2.4.0" und "3.4.1" sind also nicht in der selben minor Reihe, obwohl sie beide "4" als minor Nummer haben; andererseits sind, "2.4.0" und "2.4.2" in der selben Reihe, obwohl sie nicht auf einander folgen, wenn "2.4.1" zwischendurch veröffentlicht wurde.

Die Bedeutung dieser Zahlen ist genau das, was man erwarten würde: Eine Erhöhung der major Nummer deutet auf große Änderungen hin; eine Erhöhung der minor Nummer deutet auf kleine Änderungen hin; und eine Erhöhung der micro Nummer deutet auf wirklich triviale Änderungen. Manche Projekte fügen eine vierte Komponente hinzu, gewöhnlich als *patch Nummer* bezeichnet, für besonders fein granulierte Kontrolle über die Unterschiede zwischen ihren Versionen (verwirrenderweise, benutzen andere Projekte "patch" als synonym für "micro" bei einem System mit drei Komponenten). Es gibt auch Projekte, welche die letzte Komponente als *build Nummer* verwenden, welches jedes mal hochgezählt wird, wenn ein neuer Build der Software gemacht wird und keine andere Änderung außer diesen Build repräsentiert. Das hilft dem Projekt jede Bug-Meldung mit einem bestimmten Build in Verbindung zu bringen, und ist wahrscheinlich am nützlichsten wenn binären Pakete die übliche Methode der Verteilung ist.

Obwohl es viele verschiedene Konventionen gibt, wieviele Komponenten man verwenden soll, und was die Komponenten bedeutet, neigen die Unterschiede dazu unwesentlich zu sein – Sie haben ein wenig Spielraum aber nicht viel. Die nächsten beiden Abschnitte besprechen einige der am meisten benutzten Konventionen.

Die einfache Strategie

Die meisten Projekte haben Regeln darüber, welche Arten von Änderungen bei einer neuen Version erlaubt sind wenn man nur eine micro Nummer erhöht, andere wenn man die minor Nummer erhöht und wieder andere bei der major Nummer. Es gibt noch keinen Satz von Normen für diese Regeln, ich werde hier aber eine Regelung beschreiben, die schon bei mehreren Projekten erfolgreich verwendet worden ist. Sie können diese Regelung für Ihr Projekt einfach übernehmen, aber selbst wenn Sie das nicht tun, ist es trotzdem ein gutes Beispiel für die Art von Informationen die Versionsnummern ausdrücken sollten. Diese Regelung basiert auf dem Nummernsystem, das vom APR-Projekt benutzt wird, siehe http://apr.apache.org/versioning.html.

1. Änderungen an der micro Nummer (d.h Änderungen innerhalb der selben minor Reihe) müssen sowohl aufwärts, als auch abwärtskompatibel sein. D.h. die Änderungen sollten nur Bugfixes sein, oder sehr kleine Verbesserungen an bestehenden Funktionen. Neue Funktionen sollten bei micro Versionen nicht eingeführt werden.

2. Änderungen an der minor Nummer (d.h, innerhalb der selben major Reihe) müssen abwärtskompatibel sein, aber nicht unbedingt aufwärtskompatibel. Es ist normal neue Funktionen in einer minor Version einzuführen, für gewöhnlich aber nicht zu viele auf einmal.

3. Änderungen an der major Nummer kennzeichnen Grenzen der Kompatibilität. Eine neue major Version kann zu vorhergehenden und folgenden Versionen inkompatibel sein. Von einer major Version werden neue Funktionen erwartet, und sogar ganze Sammlungen von Funktionen.

Was *abwärtskompatibel* und *aufwärtskompatibel* genau bedeuten, hängt davon ab, was Ihre Software macht, im Kontext sind sie aber für gewöhnlich nicht sonderlich frei interpretierbar. Wenn Ihr Projekt zum Beispiel eine Client-Server-Anwendung ist, dann bedeutet "abwärtskompatibel", dass eine Aktualisierung auf 2.6.0 keine existierende 2.5.4 Clients Funktionen verlieren oder sich anders als vorher verhalten (natürlich abgesehen von den Bugs die behoben wurden). Eine Aktualisierung einer dieser Clients auf 2.6.0 zu aktualisieren könnte ihm andererseits *neue* Funktionen zur Verfügung stellen, bei dem die 2.5.4 Clients nicht wissen wie sie genutzt werden sollen. Wenn das passiert, dann ist die Aktualisierung

nicht "aufwärtskompatibel": Sie können diesen Client nicht zurück auf 2.5.4 setzen und alle Funktionen aus 2.6.0 behalten, da manche dieser Funktionen bei 2.6.0 neu waren.

Deshalb sind micro Versionen im wesentlichen ausschließlich zur Behebung von Fehler. Sie müssen in beiden Richtungen kompatibel bleiben: Wenn Sie von 2.5.3 auf 2.5.4 aktualisieren, Ihre Meinung dann ändern und wieder zurück auf 2.5.3 wechseln, sollten keine Funktionen verloren gehen. Die Fehler, die durch 2.5.4 behoben wurden, würden natürlich wieder auftauchen, aber Sie würden keine Funktionen verlieren, außer insofern, dass die wiederhergestellten Fehler die Nutzung existierender Funktionen verhindert.

Client-Server-Protokolle sind nur eines vieler möglicher Kompatibilitätsbereiche. Ein weiteres sind Datenformate: Schreibt die Software auf ein permanentes Speichermedium? Wenn ja, müssen die Formate die es liest und schreibt die Kompatibilitätsrichtlinien die von den durch die Versionsnumerierung versprochenen Richtlinien befolgen. Version 2.6.0 muss in der Lage sein die Dateien die von 2.5.4 geschrieben wurden zu lesen, kann aber im Stillen das Format auf etwas erweitern, welches 2.5.4 nicht lesen kann, da die Fähigkeit auf eine vorherige Version zu wechseln, nicht über die Grenze einer major Nummer erforderlich ist. Wenn Ihr Projekt Code-Bibliotheken, für die Nutzung durch andere Anwendungen, vertreibt, dann sind die Schnittstellen auch Bereich indem die Kompatibilität gewahrt werden muss: Sie müssen dafür sorgen, dass die Regeln für die Kompatibilität von Quellcode und Binärdateien, derart formuliert sind, dass ein informierter Nutzer sich niemals fragen muss, ob es sicher ist zu aktualisieren. Er wird in der Lage sein die Zahlen anzusehen, und es sofort zu wissen.

Bei diesem System, bekommen Sie keine Gelegenheit für einen sauberen Neuanfang, bis sie die major Nummer hochzählen. Das kann oft lästig sein: Es mag Funktionen geben, die Sie hinzufügen wollen, oder Protokolle, die Sie neu entwerfen wollen, was einfach nicht gemacht werden können, während Sie die Kompatibilität wahren. Es gibt hierfür keine magische Lösung, außer zu versuchen alles von vorn herein so zu entwerfen, dass es sich leicht erweitern lässt (ein Thema welches man leicht ein ganzes Buch widmen könnte, und sicherlich außerhalb des Rahmens von diesem). Die Kompatibilitätsrichtlinien zu veröffentlichen und sich an ihnen zu halten, ist ein unausweichlicher Teil beim Vertrieb von Software. Eine böse Überraschung kann viele Nutzer abschrecken. Die eben beschriebene Richtlinie ist zum Teil deshalb gut, da sie bereits ziemlich weit verbreitet ist, aber auch weil sie leicht zu erklären und zu behalten ist, selbst solche, die noch nicht damit vertraut sind.

Ist ist allgemein anerkannt, das diese Regeln nicht für Versionen vor 1.0 gelten (obwohl Ihre Versionsrichtlinien, um Missverständnisse zu vermeiden, das trotzdem explizit sagen sollte). Ein Projekt, welches sich noch in der frühen Entwicklung befindet, kann 0.1, 0.2, 0.3, usw. in einer Reihe veröffentlichen, bis es für 1.0 bereit ist, und die Unterschiede zwischen den Versionen können beliebig groß sein. Micro-Nummern in einer pre-1.0 Version sind optional. Abhängig von der Natur Ihres Projekts, und die Unterschiede zwischen den Versionen, werden Sie es nützlich finden, 0.1.0, 0.1.1, usw. zu haben oder nicht. Konventionen für pre-1.0 Versionsnummern sind relativ locker, hauptsächlich deshalb, weil Leute verstehen, dass strenge Einschränkungen für Kompatibilität die frühe Entwicklung zu sehr behindern würde, und weil Personen die ein Produkte in ihren frühen Phasen benutzen, sowieso etwas nachsichtig sind.

Denken Sie daran, dass all diese Vorschriften nur für dieses bestimmte Drei-Komponenten-System gelten. Ihr Projekt kann sich ganz leicht ein anderes Drei-Komponenten-System ausdenken, oder sich gar entscheiden, dass es keine derart feine Granularität benötigt und statt dessen ein Zwei-Komponenten-System verwenden. Das wichtige ist, sich frühzeitig zu entscheiden, genau zu veröffentlichen, was die Komponenten bedeuten, und dabei zu bleiben.

Die Gerade/Ungerade-Strategie

Manche Projekte benutzen die Parität der Minor-Nummer, um auf die Stabilität der Software zu deuten: gerade bedeutet stabil, ungerade bedeutet instabil. Das gilt nur für die Minor-Nummer, nicht für die Major- oder Micro-Nummern. Eine Erhöhung in der Micro-Nummer deutet immer noch auf Fehlerbe-

hebungen hin (keine neuen Funktionen), und eine Erhöhung der Major-Nummer deutet auf große Änderungen, neue Funktionen, usw. hin.

Der Vorteil des Gerade/Ungerade-Systems, welches unter anderem vom Linux-Kernel-Projekt verwendet wurde ist, dass es eine Möglichkeit bietet, neue Funktionen zum Test anzubieten, ohne die Nutzer in Produktivsystemen möglicherweise instabilen Code auszusetzen. Die Leute können durch die Zahlen erkennen, dass es sicher ist, "2.4.21" auf ihren im Einsatz befindliche Webserver zu installieren, aber dass "2.5.1" besser für Experimente auf dem heimischen Rechner beschränkt bleiben sollten. Das Entwicklerteam befasst sich mit den Bug-Meldungen die aus der instabilen (ungerade numerierten) Reihe kommen, und wenn sich die Sache nach ein paar Micro-Versionen langsam legt, erhöhen sie die Minor-Nummer (wodurch sie gerade gemacht wird), setzen die Micro-Nummer wieder auf "0", und veröffentlichen ein vermutlich stabiles Paket.

Dieses System bewahrt die vorhin erwähnten Kompatibilitsrichtlinien, oder tritt zumindest nicht im Konflikt ihnen. Es fügt der Minor-Nummer lediglich ein paar Informationen hinzu. Das zwingt die Minor-Nummer dazu ungefähr doppelt so oft erhöht zu werden, als das sonst der Fall wäre, was aber kein großer schaden ist. Das Gerade/Ungerade-System ist wahrscheinlich am besten für Projekte geeignet, die sehr lange Entwicklungszyklen haben, und von Natur aus einen großen Anteil konservativer Nutzer haben, die mehr Wert auf Stabilität als auf neue Funktionen setzen. Es ist jedoch nicht die einzige Möglichkeit, neue Funktionen im Feld getestet zu bekommen. „Stabilisierung einer neuen Version“ später in diesem Kapitel beschreibt eine weitere, vielleicht gebräuchlichere Art, potentiell instabilen Code zu veröffentlichen, der in einer Weise gekennzeichnet ist, die den Leuten bereits aus dem Namen der Version eine Vorstellung von dem mit ihr verbundenen Risiko/Nutzen-Kompromiss vermittelt.

Versionszweige

Aus der Sicht eines Entwicklers, ist ein freies Software-Projekt, in einem ständigen Zustand der Veröffentlichung. Entwickler betreiben für gewöhnlich zu jeder Zeit die aktuellste Version, da sie Fehler entdecken wollen, und da sie das Projekt dicht genug verfolgen, um von derzeit instabilen Bereichen des Funktionsumfangs fern zu bleiben. Sie aktualisieren ihre Kopie der Software für gewöhnlich jeden tag, manchmal öfter, und wenn Sie eine Änderung abschicken, können sie halbwegs erwarten, dass jeder andere Entwickler es innerhalb von 24 Stunden haben wird.

Wie sollte das Projekt dann eine formale Veröffentlichung machen? Sollte es einfach eine Momentaufnahme vom derzeitigen Zustand des Quellcode-Baums machen, und es der Welt als, sagen wir, "3.5.0" überreichen? Die Vernunft sagt nein. Erstens gäbe es möglicherweise keinen einzelnen Zeitpunkt, bei dem der ganze Quellcode-Baum bereit für eine neue Version ist. Neu angefangene Funktionen könnten an verschiedenen Stellen, in verschiedenen Zuständen der Fertigstellung herumliegen. Jemand könnte eine riesige Änderung committed haben, um einen Fehler zu beheben, die Änderung könnte aber kontrovers sein und zu der Zeit debattiert werden indem die Momentaufnahme gemacht wird. Wenn das der Fall ist, würde es nicht so einfach funktionieren, die Momentaufnahme zu verzögern, bis die Debatte beendet ist, da eine andere, nicht im Zusammenhang stehende Debatte in der Zwischenzeit anfangen könnte, und dann müssten Sie wieder *darauf* warten. Man kann nicht garantieren, dass dieser Ablauf aufhören wird.

In jedem Fall, würde die Nutzung von Momentaufnahmen des kompletten Quellcode-Baums als eine neue Version, die fortlaufenden Entwicklung behindern, selbst wenn der Quellcode in einem für die Veröffentlichung geeigneten Zustand gebracht werden könnte. Sagen wir, dass diese Momentaufnahme "3.5.0" sein wird; wäre die nächste Version vermutlich Bugfixes beinhalten, die in 3.5.0 gefunden wurden. Wenn beide Momentaufnahmen aber vom gleichen Baum sein sollen, was sollen die Entwickler in der Zeit zwischen den beiden Versionen machen? Sie können keine neuen Funktionen hinzufügen; dass verbieten die Kompatibilitätsrichtlinien. Aber nicht alle werden enthusiastisch darüber sein Fehler in dem Code von 3.5.0 zu beheben. Manche werden neue Funktionen haben, die sie versuchen fertigzustellen, und würden wütend werden, wenn sie dazu gezwungen wären zu wählen zwischen, untätig

herum zu sitzen oder an Sachen zu arbeiten, die sie nicht interessieren, nur weil der Ablauf des Projekts beim Erstellen einer neuen Version erfordert, dass die Entwicklung an dem Codebaum unnatürlich ruhig bleibt.

Die Lösung für diese Probleme ist, immer einen *Versionszweig* (en. release branch) zu benutzen. Ein Versionszweig ist lediglich ein Zweig in dem Versionsverwaltungssystem, (siehe *Branch*[9]), bei dem der Code, welcher für diese Release bestimmt ist, von der Hauptentwicklung getrennt werden kann. Diese Zweige sind sicherlich kein originäres Konzept freier Software; viele kommerzielle Unternehmen, die Software entwickeln, benutzen sie auch. In kommerziellen Umgebungen werden Versionszweige jedoch manchmal als ein Luxus betrachtet – eine Art formales "optimales Verfahren" (en. "best practice") welches unter dem Druck eines wichtigen Termins entfallen kann, während alle Entwickler sich dabei beeilen, den Hauptast stabil zu bekommen.

Versionszweige sind jedoch bei Open-Source-Projekten so ziemlich eine Notwendigkeit. Ich habe Projekte gesehen, die neue Versionen ohne sie machen, das Ergebnis war jedoch immer, dass einige Entwickler untätig herumsaßen, während einige andere – für gewöhnlich eine Minderheit – daran arbeiten die Version fertigzustellen. Dieses Ergebnis ist gewöhnlicherweise auf verschieden Arten schlecht. Erstens, verlangsamt sich die Geschwindigkeit der Entwicklung insgesamt. Zweitens, ist die neue Version von einer schlechteren Qualität als es hätte sein müssen, da nur wenige daran gearbeitet haben, und die hatten es eilig, damit alle anderen wieder an die Arbeit gehen konnten. Drittens, spaltet es psychologisch die Entwicklermannschaft, indem es eine Situation aufbaut, bei dem verschiedene Arten der Arbeit sich gegenseitig unnötig stören. Die Entwickler die untätig herumsitzen, wären wahrscheinlich froh *etwas* von ihrer Aufmerksamkeit zu dem Versionszweig beizutragen, solange es eine Wahl bleibt, die sie abhängig von ihren Interessen und Zeitplänen treffen könnten. Ohne den Zweig, beschränkt sich ihre Auswahl jedoch auf "Nehme ich heute an dem Projekt teil oder nicht?" anstatt "Arbeite ich heute an der neuen Version, oder an der neuen Funktion in dem Hauptzweig, dass ich Entwickele?".

Mechanik von Versionszweigen

Die genaue Mechanik der Erstellung eines Versionszweiges hängt natürlich von Ihrem Versionsverwaltungssystem ab, die allgemeinen Konzepte sind aber für die meisten Systemen gleich. Ein Zweig sprießt für gewöhnlich aus einem anderen Zweig under dem Stamm. Traditionell, wird die Entwicklung in dem Stamm betrieben, ungestört von den Einschränkungen einer neuen Version. Der erste Zweig, welcher zu der Version "1.0" führt, sprießt vom Stamm ab. Bei CVS sähe der Befehl für einen Zweig ungefähr wie folgt aus:

```
$ cd trunk-working-copy
$ cvs tag -b VERSION_1_0_X
```

Oder in Subversion, wie folgt:

```
$ svn copy http://.../repos/trunk http://.../repos/branches/1.0.x
```

(All diese Beispiele gehen von dem Drei-Komponenten-System für die Nummerierung der Version aus. Obwohl ich nicht die genauen Befehle für jedes Versionsverwaltungssystem geben kann, sind hier Beispiel für CVS und Subversion und ich hoffe, dass die entsprechenden Befehle von den beiden abgeleitet werden können.)

Beachten Sie, dass wir den Zweig "1.0.x" (mit dem Buchstaben "x") anstatt "1.0.0" erzeugt haben. Das liegt daran, dass die selbe Minor-Reihe – d.h. der selbe Ast – für alle Micro-Versionen in der Reihe benutzt werden wird. Der eigentliche Vorgang, den Zweig für die Veröffentlichung stabil zu machen,

[9] de. Zweig/Ast

wird in „Stabilisierung einer neuen Version“ später in diesem Kapitel behandelt. Hier geht es nur um die Wechselwirkung zwischen dem Versionsverwaltungssystem und dem Ablauf beim Erstellen der neuen Version. Sobald der Versionszweig in einen stabilen Zustand gebracht wurde und bereit ist, wird es Zeit, eine Momentaufnahme von dem Zweig zu machen:

```
$ cd VERSION_1_0_X-working-copy
$ cvs tag VERSION_1_0_0
```

or

```
$ svn copy http://.../repos/branches/1.0.x http://.../repos/tags/1.0.0
```

Dieses Tag repräsentiert jetzt genau den Zustand, in dem der Quellcode des Projekts war bei der Version 1.0.0 (das ist nützlich, falls Sie je eine ältere Version brauchen, nachdem die veröffentlichten Pakete nicht mehr zur Verfügung stehen). Die nächste Micro-Version in der selben Reihe, wird gleichermaßen in dem 1.0.x Zweig vorbereitet, und wenn es bereit ist, wird ein Tag für 1.0.1. gemacht. Gleiches gilt für 1.0.2, usw. Wenn es Zeit wird, über eine neue 1.1.x-Version nachzudenken, erzeugen Sie einen neuen Zweig vom Stamm:

```
$ cd trunk-working-copy
$ cvs tag -b VERSION_1_1_X
```

oder

```
$ svn copy http://.../repos/trunk http://.../repos/branches/1.1.x
```

Die Wartung kann parallel dazu in den Versionen 1.0.x und 1.1.x, weitergehen, und neue Versionen können in beiden Reihen unabhängig von einander veröffentlicht werden. Es ist sogar nicht einmal ungewöhnlich zwei Versionen aus verschiedenen Reihen zur gleichen Zeit zu veröffentlichen. Die ältere Reihe wird für die konservativeren Server Administratoren empfohlen, die vielleicht nicht den großen Sprung nach (sagen wir) 1.1 machen wollen, ohne sorgfältige Vorbereitung. In der Zwischenzeit, nehmen die eher abenteuerlustigen Personen die neuste Version der höchsten Reihe, um sicher zu sein, dass sie all die neusten Funktionen bekommen, selbst mit dem Risiko einer höheren Instabilität.

Das ist natürlich nicht die einzige Strategie für neue Versionszweige. Unter manchen Umständen mag es nicht einmal die beste sein, obwohl es für die Projekte mit denen ich zu tun hatte ziemlich gut funktioniert hat. Benutzen Sie irgend eine Strategie die zu funktionieren scheint, bedenken Sie aber die wichtigsten Punkte: Der Sinn eines Versionszweiges ist, die Arbeit an der neuen Version von den Schwankungen der täglichen Entwicklung zu trennen, und dem Projekt eine Physikalische Entität zu geben, um den es die Abläufe bei der Veröffentlichung organisieren kann. Diese Abläufe werden im nächsten Abschnitt detailliert beschrieben.

Stabilisierung einer neuen Version

Stabilisierung ist der Vorgang, den Zweig einer neue Version in einem Zustand zu bringen, der für die Veröffentlichung geeignet ist; mit anderen Worten, zu entscheiden, welche Änderungen in der Version sein werden, welche nicht, und den Inhalt vom Zweig entsprechend zu gestalten.

Es gibt eine menge möglichen Kummer in diesem Wort "entscheiden". Der Ansturm neuer Funktionen in der letzten Minute ist ein bekanntes Phänomen bei gemeinschaftlichen Software-Projekten: Sobald Entwickler sehen, dass eine neue Version gleich gemacht wird, beeilen sie sich ihre derzeitigen Änderungen fertig zu bekommen, um den Zug nicht zu verpassen. Das ist natürlich genau das Gegenteil,

von dem was Sie zum Zeitpunkt einer neuen Version haben wollen. Es wäre viel besser, wenn Leute in einem gemütlichen Tempo an neue Funktionen arbeiten, und sich nicht all zu viele sorgen darüber machen ob Ihre Änderungen es in diese Version oder die nächste schaffen. Je mehr Änderungen man versucht in eine neue Version in der letzten Minute zu quetschen, desto mehr Code wird instabil gemacht, und (für gewöhnlich) desto mehr neue Bugs werden erzeugt.

Die meisten Programmierer sind sich theoretisch über grobe Kriterien einig, welche Änderungen in eine Versionsreihe hereingelassen werden sollten, währen der Stabilisierungsphase. Offensichtlich, können Fixes für schwerwiegende Bugs reingehen, insbesondere wenn es keine Möglichkeit gibt sie zu umgehen. Aktualisierungen der Dokumentation sind in Ordnung, genau so wie der Text von Fehlermeldungen (außer wenn man sie als einen Teil der Schnittstelle betrachtet und stabil bleiben müssen). Viele Projekte erlauben also bestimmte Arten risikoarmer nicht grundsätzlicher Änderungen während der Stabilisierungsphase einzubinden, und können formale Richtlinien darüber haben, um das Risiko einzustufen. Keine Maß der Formalisierung kann jedoch die Notwendigkeit menschlichen Urteils obsolet machen. Es wird immer Fälle geben, bei dem das Projekt eine Entscheidung treffen muss, ob eine bestimmte Änderung in eine neue Version gehen kann. Die Gefahr ist, dass da jede Person ihre eigen Lieblingsängerungen aufgenommen haben will, wird es eine Menge Leute geben, die motiviert sind Änderungen zu erlauben, und nicht genügend die motiviert sind, sie zu verhindern.

Der Ablauf, eine neue Version zu stabilisieren dreht sich deshalb größtenteils darum, "nein" zu sagen. Der Trick insbesondere für Open-Source-Projekte ist, sich Wege auszudenken "nein" zu sagen, die nicht mit allzu vielen verletzten Gefühlen oder enttäuschten Entwicklern enden und die es dennoch ermöglichen, dass Änderungen in neue Versionen aufgenommen werden, die es verdienen. Es gibt viele verschiedene Wege das zu erreichen. Es ist ziemlich leicht Systeme zu entwerfen, welche diesen Kriterien gerecht werden, nachdem sich das Team auf sie als die wichtigsten Kriterien konzentriert hat. Ich werde hier knapp zwei der verbreitetsten Systeme beschreiben, die an den extremen Enden des Spektrums liegen, lassen Sie Ihr Projekt hiervon aber nicht abhalten, kreativ zu sein. Eine Menge anderer Reglungen sind möglich; diese sind lediglich zwei von denen ich gesehen habe, dass sie in der Praxis funktionieren.

Diktatur durch den Versionsherrn

Die Gruppe einigt sich darauf, dass eine Person der *Versionsherr* (en. release owner) sein wird. Diese Person hat das letzte Wort darüber, welche Änderungen es in die neue Version schaffen. Es ist natürlich normal und wird erwartet, dass es Diskussionen und Auseinandersetzungen gibt, letztendlich muss die Gruppe der Person die nötige Autorität geben um endgültige Entscheidungen zu treffen. Damit dieses System funktioniert, ist es nötig, jemand zu wählen, der die nötige technische Kompetenz hat um alle Änderungen zu verstehen, und den sozialen Stand und Fähigkeiten im Umgang mit Menschen um die Diskussionen zu führen, die einer neuen Version vorausgehen, ohne zu viele verletzte Gefühle zu verursachen.

Ein häufiges Muster ist, dass der Versionsherr sagt "Ich denke nicht, dass es irgend etwas an dieser Änderung auszusetzen gibt, aber wir haben noch nicht genug Zeit gehabt, um es zu testen, also sollte es nicht in diese Version gehen". Es hilft eine Menge wenn der Versionsherr ein breites technisches Wissen über das Projekt hat, und Gründe angeben kann, warum eine Änderung möglicherweise destabilisierend sein könnte (zum Beispiel seine Wechselwirkung mit andern Teilen der Software oder Bedenken im Bezug auf die Kompatibilität). Manche werden nach einer Rechtfertigung für solche Entscheidungen fragen, oder argumentieren, dass eine Änderung nicht so gefährlich ist wie sie aussieht. Diese Unterhaltungen müssen nicht konfrontierend sein, so lange der Versionsherr alle Argumente objektiv in Erwägung ziehen kann, und nicht aus reflex auf seinen Standpunkt beharrt.

Beachten Sie, dass der Versionsherr nicht die gleiche Person sein muss, wie der Projektleiter (bei solchen Fällen wo es überhaupt einen Projektleiter gibt; siehe „Gütige Diktatoren“ im Kapitel Kapitel 4, *Soziale und politische Infrastruktur*). Manchmal ist es sogar gut, wenn sie *nicht* ein und die selbe Person sind. Die Fähigkeiten die einen gute Projektleiter ausmachen, sind nicht unbedingt die gleichen, die

einen guten Versionsherr ausmachen. Bei etwas derart wichtigem wie die Veröffentlichung neuer Versionen, kann es klug sein, jemanden zu haben, der ein Gegengewicht zu dem Urteil vom Projektleiter bildet.

Vergleichen Sie die Rolle des Versionsverwalters mit der weniger diktatorischen Rolle beschrieben in „Release-Verwalter“ später in diesem Kapitel.

Abstimmung über Änderungen

Bei dem anderen extrem, von der Diktatur durch den Versionsherr, können Entwickler einfach darüber abstimmen, welche Änderungen in einer neun Version aufgenommen werden. Da die wichtigste Aufgabe bei der stabilisierung einer neun Version der *Ausschluss* von Änderungen ist, ist es wichtig, das Wahlsystem so zu gestalten, dass eine Änderung in eine Version zu bekommen die Zustimmung mehrerer Entwickler bedarf. Eine Änderung aufzunehmen, sollte mehr als eine einfache Mehrheit erfordern (siehe „Wahlberechtigung“ im Kapitel Kapitel 4, *Soziale und politische Infrastruktur*). Andererseits würde eine Stimme für eine Änderung und keine dagegen ausreichen, es in die Version zu bekommen und eine unglückliche Dynamik würde einsetzen, bei dem jeder Entwickler für seine eigenen Änderungen stimmen würde, jedoch widerwillig gegen die Änderungen anderer stimmen würde, aus Angst vor einer möglichen Vergeltung. Um das zu vermeiden, sollte das System derart ausgelegt sein das Untergruppen von Entwickler zusammen agieren müssen um irgend eine Änderung in eine Version zu bekommen. Das bedeutet nicht nur, dass mehr Personen jede Änderung überprüfen, es macht auch jeden Entwickler weniger zögerlich gegen eine Änderung zu stimmen, da kein bestimmter Entwicklern der dafür gestimmt hat, seine Gegenstimme es als einen persönlichen Angriff betrachten würde. Je mehr Leute beteiligt sind, desto mehr dreht sich die Entscheidung um die Änderung und weniger um die Individuen.

Das System welches wir beim Subversion-Projekt benutzen, scheint ein gutes Gleichgewicht getroffen zu haben, also werde ich es hier empfehlen. Damit eine Änderung auf einen Versionszweig angewandt werden kann, müssen mindestens drei Entwickler dafür stimmen, und keiner dagegen. Eine einzige Gegenstimme reicht aus, um zu verhindern, dass eine Änderung angewandt wird; in dem Kontext einer neuen Version heißt das also, dass eine Gegenstimme gleichbedeutend mit einem Veto ist (siehe „Vetos“). Jede solche Gegenstimme muss natürlich von einer Rechtfertigung begleitet werden, und theoretisch kann das Veto aufgehoben werden, wenn genügend der Meinung sind, dass unvernünftig ist und eine besondere Abstimmung darüber erzwingen. In der Praxis, ist das nie vorgekommen und ich erwarte nicht, dass es das jemals wird. Bei einer neuen Version sind sowieso alle schon konservativ eingestellt, und wenn jemand es als dringend genug erachtet, ein Veto einzulegen, dann gibt es für gewöhnlich einen guten Grund dafür.

Da da der Ablauf eine neue Version zu erstellen, absichtlich eine konservative Voreingenommenheit hat, sind die Rechtfertigungen die bei Vetos angeboten werden manchmal eher formal als technisch orientiert. Eine Person kann zum Beispiel das Gefühl haben, dass eine Änderung gut geschrieben ist und wahrscheinlich keine neuen Fehler verursachen wird, aber gegen die Aufnahme der Änderung in einer micro Version stimmt, einfach weil es zu groß ist – vielleicht fügt es eine neue Funktion hinzu, oder auf irgend eine subtile Art, die Kompatibilitätsrichtlinien nicht befolgt. Ich habe gelegentlich sogar Entwickler gesehen, die ein Veto eingelegt haben, einfach nur aus einem Bauchgefühl heraus, dass die Änderung mehr getestet werden musste, auch wenn sie bei der Überprüfung keine Fehler entdecken konnten. Leute murrten ein wenig, aber die Vetos blieben in Kraft, und die Änderung wurde bei der Version nicht aufgenommen (ich kann mich allerdings nicht daran erinnern ob bei den weiteren Tests irgend welche Fehler gefunden wurden oder nicht).

Gemeinschaftlichen Stabilisierung neuer Versionen

Wenn Ihr Projekt sich für ein System entscheidet bei dem über Änderungen abgestimmt wird, ist es unabdingbar, dass die physikalischen Mechanismen, die Wahl zusammen zu stellen und Stimmen, so bequem wie möglich ist. Obwohl es eine menge Open-Source-Software für elektronische Wahlen zur

Verfügung steht, ist in der Praxis, am einfachsten, eine Textdatei im Versionszweig zu erstellen, mit dem Namen `STATUS` oder `VOTES` oder etwas ähnliches. Diese Datei listet jede vorgeschlagene Änderung auf – alle Entwickler können eine neue Änderung für die Aufnahme vorschlagen – zusammen mit allen Stimmen dafür oder dagegen, mit zusätzlichen Anmerkungen oder Kommentare. (Eine Änderung vorzuschlagen bedeutet im übrigen nicht unbedingt dafür zu stimmen, auch wenn die beiden oftmals hand in hand gehen.) Ein Eintrag in einer solchen Datei, könnte wie folgt aussehen:

```
* r2401 (Meldung #49)
  Verhindern, dass der client/server handshake zwei mal durchgeführt
  wird.

  Rechtfertigung:
    Vermeidet überflüssigen Netzwerkverkehr; kleine Änderung und leicht
    zu überprüfen.
  Anmerkungen:
    Wurde in http://.../mailing-lists/message-7777.html und anderen
    Nachrichten in dem Thread besprochen.
  Stimmen:
    +1: jsmith, kimf
    -1: tmartin (Bricht die Kompatibilität mit manchen pre-1.0 Server;
                 zugegeben, diese Server sind Buggy, aber warum sollen
                 wird inkompatibel sein, wenn es nicht sein muss?)
```

In diesem Fall, erhielt die Änderung zwei positive Stimmen, wurde aber durch den Veto von tmartin aufgehalten, der den Grund für seinen Einspruch in Klammern angegeben hat. Das genau Format von diesem Eintrag macht keinen Unterschied; egal worauf sich Ihr Projekt einigt, es ist in Ordnung – vielleicht sollte die Erklärung von tmartin für seinen Einspruch in den "Anmerkungen:" Bereich gehen, oder vielleicht sollte die Beschreibung der Änderung eine "Beschreibung:" Überschrift bekommen um den anderen Abschnitten zu entsprechen. Dass wichtige ist, dass alle Information, die nötig ist, um die Änderung zu evaluieren zur Verfügung stehen und dass die Mechanismen um Stimmen abzugeben, so einfach wie möglich sind. Die vorgeschlagene Änderung wird anhand seiner Revisionsnummer im Projektarchiv gekennzeichnet (in diesem Fall eine einzige Revision, r2401, obwohl eine vorgeschlagene Änderung genau so gut aus mehreren Änderungen bestehen könnte). Von der Revision wird angenommen, dass sie sich auf den Stamm bezieht; wenn die Änderung bereits im Versionszweig wäre, gäbe es keinen Grund darüber abzustimmen. Wenn Ihr Versionsverwaltungssystem keine offensichtliche Syntax hat, um auf einzelne Revisionen zu verweisen, sollte das Projekt eins erfinden. Damit Abstimmungen durchgeführt werden können, müssen Änderungen die in Frage stehen, eindeutig zu identifizieren sein.

Diejenigen die einen Vorschlag einreichen, oder dafür abstimmen, sind dafür verantwortlich, dass es sauber auf den Versionszweig angewandt werden kann, d.h. ohne Konflikte (siehe *Konflikt*) angewandt werden kann. Wenn es Konflikte gibt, sollte der Eintrag entweder auf einen angepassten Zweig verweisen, auf dem sich der Patch sauber anwenden lässt, oder auf einen temporären Zweig welcher eine angepasste Version der Änderung beinhaltet, zum Beispiel:

```
* r13222, r13223, r13232
  Überarbeitung vom auto-merge Algorithmus in libsvn_fs_fs
  Rechtfertigung:
    Nicht akzeptable Performance (>50 Minuter für einen kleinen
    commit) bei einem Projektarchiv mit 300,000 Revisionen
  Zweig:
    1.1.x-r13222@13517
  Stimmen:
    +1: epg, ghudson
```

Dieses Beispiel stammt aus dem echten Leben; es stammt aus der STATUS Datei für Version 1.1.4 von Subversion. Beachten Sie, wie es die ursprünglichen Revisionen als kanonische Verweise auf die Änderung benutzt obwohl es auch einen Zweig gibt, indem die aufgrund von Konflikten angepasste Version der Änderung enthalten ist (der Zweig kombiniert auch die drei Revisionen im Stamm zu einer, r13517, um dem Merge zum Zweig der neuen Version zu erleichtern, sollte ihm zugestimmt werden). Die ursprünglichen Revisionen werden angegeben, da sie trotzdem noch am einfachsten zu überprüfen sind, da sie die ursprünglichen commit Kommentare enthalten. Der vorübergehende Zweig hätte diese Kommentar nicht; um eine Verdopplung der Informationen zu vermeiden (siehe „Eindeutigkeit von Informationen“ im Kapitel Kapitel 3, *Technische Infrastruktur*), der Commit Kommentar für r13517 einfach sagen "Anpassung von r13222, r13223, und r13232 für die Portierung auf den 1.1.x Zweig." Alle anderen Information über die Änderung kann bei ihren ursprünglichen Revisionen gefunden werden.

Release-Verwalter

Der eigentliche Merge-Vorgang (siehe *Merge*[11]) angenommener Änderungen in den Versionszweig kann von jedem Entwickler durchgeführt werden. Es muss keine bestimmte Person geben, deren Aufgabe es ist, die Änderungen zu mergen; wenn es eine große Menge von Änderungen gibt, kann es besser sein, die Bürde zu verteilen.

Obwohl sowohl Wahlen als auch die Merges auf eine dezentralisierte Art passieren, gibt es in der Praxis ein oder zwei Personen welche die Veröffentlichung vorantreiben. Diese Rolle wird manchmal als *Releaseverwalter* (en. release manager) bezeichnet, sie ist aber sehr verschieden von einem Versionsherr (siehe „Diktatur durch den Versionsherrn“ früher in diesem Kapitel) der das letzte Wort über die Änderungen hat. Versionsverwalter halten einen Überblick darüber, wie viele Änderungen derzeit in Betracht gezogen werden, wie vielen zugestimmt wurde, und wie vielen wahrscheinlich zugestimmt wird, usw. Wenn sie das Gefühl bekommen, dass wichtige Änderungen nicht die nötige Aufmerksamkeit bekommen, und aus einer Version gelassen werden könnten, aufgrund von Stimmenmangel, werden sie sanft andere Entwickler nerven, damit sie die Änderungen überprüfen und darüber abzustimmen. Wenn ein Satz von Änderungen Zustimmung erhalten haben, werden diese Personen es oft auf sich nehmen, sie in den Versionszweig aufzunehmen; es ist in Ordnung wenn andere ihnen die Aufgabe überlassen, so lange jeder versteht, dass sie keiner Obligation unterliegen, die ganze Arbeit zu übernehmen, es sei denn sie haben sich explizit dazu verpflichtet. Wenn die Zeit kommt, die neue Version zu veröffentlichen (siehe „Tests und Veröffentlichung“ später in diesem Kapitel), kümmern sich die Versionsverwalter auch um die logistische Arbeit, die Erstellung der Pakete, das sammeln der digitalen Signaturen, das Hochladen der Pakete, und die öffentliche Bekanntgabe.

Erstellung der Pakete

Die übliche Art, freie Software zu verteilen, ist in Form von Quellcode. Das gilt unabhängig davon, ob die Software normalerweise mit dem Quellcode betrieben wird (d.h. interpretiert werden kann, wie Perl, Python, PHP, usw.) oder vorher kompiliert werden muss (wie C, C++, Java, usw.). Bei kompilierter Software, werden die meisten Nutzer nicht unbedingt selbst die Quellen kompilieren, sondern vorkompilierte binären Paketen installieren wollen (siehe „Binäre Pakete“ später in diesem Kapitel). Diese binären Pakete leiten sich jedoch trotzdem von der ursprünglichen Quellcode-Distribution ab. Der Sinn der Quellpakete ist, die neue Version eindeutig zu definieren. Wenn das Projekt "Scanley 2.5.0" veröffentlicht, bedeutet das genau genommen, "Die Quellcode-Dateien, die wenn sie kompiliert (falls nötig) und installiert wurden, Scanley 2.5.0 erzeugen".

Es gibt einen ziemlich strikten Standard darüber, wie Quellcode-Distributionen auszusehen haben. Man wird ab und zu Abweichungen davon sehen, sie sind aber die Ausnahme, nicht die Regel. Wenn es keinen zwingenden Grund gibt es anders zu machen, sollte sich Ihr Projekt auch an diesen Standard halten.

[11] im engl. auch "port" de. Zusammenführung/Portierung

Formate

Der Quellcode sollte in den Standardformaten für die Übertragung von Verzeichnissen verteilt werden. Bei Unix und Unix-ähnlichen Betriebssystemen, ist die Konvention, das TAR Format zu benutzen, komprimiert mit **compress**, **gzip**, **bzip** oder **bzip2**. Bei MS Windows, ist die Standard-Methode für die Veröffentlichung von Verzeichnissen das *zip* Format, welches zufällig auch komprimiert, also gibt es keinen Grund es zu komprimieren, nach der Erstellung.

> **TAR-Dateien**
>
> *TAR* steht für "Tape ARchive", da das TAR Format ein Verzeichnis als einen linearen Datenstrom repräsentiert, ist es ideal für die Speicherung auf Datenbändern. Die gleiche Eigenschaft, macht es zum Standard für die Verteilung von Verzeichnissen als eine einzige Datei. Komprimierte TAR Dateien (auch*tarballs*) ist relativ einfach. Auf manchen Systemen, kann der **tar** Befehl von sich aus ein komprimiertes Archiv erstellen; auf anderen, wird eine separate Anwendung benutzt.

Name und Aufbau

Der Name des Pakets sollte aus dem Namen der Software bestehen, gefolgt von der Versionsnummer und den Dateiendungen für die entsprechenden Archiv typen. Scanley 2.5.0, als Paket für Unix mit der GNU Zip (gzip) Kompression, würde zum Beispiel so aussehen:

scanley-2.5.0.tar.gz

oder für Windows mit zip Kompression:

scanley-2.5.0.zip

Beide dieser Archive erzeugen, wenn man sie entpackt, eine einziges neues Verzeichnis namens `scanley-2.5.0` in dem derzeitigen Ordner. Im neuen Verzeichnis, sollten die Dateien des Quellcodes so ausgelegt sein, welches für die Kompilierung (wenn die Kompilierung nötig ist) und installation bereit ist. In der obersten Verzeichnisebene, sollte es eine Klartext-Datei mit dem Namen `README` geben, welches erklärt, was die Software macht, was diese Paket ist, und hinweise Auf andere Ressourcen, wie die Webseite des Projekts, andere Dateien die von Interesse sind, usw. Unter diesen anderen Dateien sollte eine `INSTALL` Datei sein, welches Anweisungen gibt, wie man einen Build der Software machen kann, und sie installieren kann, für alle Betriebssysteme die es unterstützt. Wie in „Eine Lizenz für Ihre Software“ im Kapitel Kapitel 2, *Der Einstieg* erwähnt, sollte es auch eine `COPYING` oder `LICENSE` Datei geben, welche die Bedingungen enthält, unter denen die Software vertrieben werden kann.

Es sollte auch eine `CHANGES`-Datei geben (manchmal `NEWS`) genannt, welches erklärt, was in dieser Version neu ist. Die `CHANGES`-Datei listet die Änderungen aller Versionen auf, in antichronologischer Richtung, sodass die Liste für diese Version in der Datei oben erscheint. Diese Liste zu vervollständigen, ist üblicherweise das Letzte, was auf einem stabilisierenden Versionszweig gemacht wird; manche Projekte schreiben diese Liste in Stücken während sie entwickeln, andere bevorzugen es, alles bis zum Schluss aufzusparen, und es dann von einer Person schreiben zu lassen, indem Informationen aus den Commit-Kommentaren der Versionsverwaltung gesammelt werden. Die Liste sieht in etwa so aus:

```
Version 2.5.0
(20 Dezember 2004, von /branches/2.5.x)
http://svn.scanley.org/repos/svn/tags/2.5.0/

 Neue Funktionen, Erweiterungen:
    * Neue Abfragen mit regulären Ausdrücken (Meldung #53)
```

```
    * Unterstützung für UTF-8- und UTF-16-Dokumente
    * Documentation in Polnisch, Russisch, Malaysisch
    * ...

Bugfixes:
    * Bugs in Neuindexierung behoben (Meldung #945)
    * Einige Abfragefehler behoben (Meldung #815, #1007, #1008)
    * ...
```

Die Liste kann so lang wie nötig sein, Sie brauchen sich aber nicht darum zu kümmern, dass jeder kleiner Bugfix und jede kleine Funktioserweiterung aufgelistet wird. Ihr Sinn ist es einfach, den Nutzern einen Überblick zu geben, inwiefern sie davon profitieren würden, auf die neue Version zu aktualisieren. Die Änderungsliste wird üblicherweise in der Ankündigungs-Mail aufgenommen (siehe „Tests und Veröffentlichung“ später in diesem Kapitel), schreiben Sie sie also mit dem Publikum im Hinterkopf.

CHANGES kontra ChangeLog

Traditionell, listet eine Datei namens *ChangeLog* jede Änderung die je an einem Projekt gemach wurde auf, – das heißt, jeden Commit der in das Versionsverwaltungssystem aufgenommen wurde. Es gibt verschiedene Formate für ChangeLog Dateien; die Details der Formate sind an dieser Stelle nicht wichtig, da sie alle die gleichen Informationen beinhalten: Das Datum der Änderung, sein Autor, und eine kurze Zusammenfassung (oder einfach nur den Commit-Kommentar für diese Änderung).

Eine `CHANGES` Datei ist anders. Es ist auch eine Liste von Änderungen, aber nur solche, von denen man denkt, dass sie wichtig genug sind, dass ein bestimmtes Publikum sie sieht, und ohne mit Metadaten wie das genau Datum und der Autor. Um Verwirrung zu vermeiden, vertauschen Sie die Begriffe nicht. Manche Projekte benutzen "NEWS" anstatt von "CHANGES"; obwohl das die mögliche Verwirrung mit "ChangeLog" vermeidet, ist es etwas falsch bezeichnet, da die CHANGES Datei die Änderungs Informationen über alle Versionen enthält, und deshalb eine Menge alte Informationen zusätzlich zu denen ganz oben enthält.

ChangeLog Dateien mögen sowieso langsam am verschwinden sein. Sie waren zu einer Zeit hilfreich, als CVS die einzige Wahl für die Versionsverwaltung war, weil Daten über Änderungen nicht leicht aus CVS zu bekommen waren. Bei neueren Systemen können die Daten die ehemals in dem ChangeLog geschrieben wurden, jedoch jederzeit von dem Versionsverwaltungssystem angefordert werden, was es Sinnlos für das Projekt macht, eine Statistik-Datei darüber zu halten – es ist sogar mehr als sinnlos, da die ChangeLog-Datei lediglich die Meldungen, die bereits im Projektarchiv gespeichert sind, duplizieren würde.

Der tatsächliche Aufbau der Quellcode-Dateien sollte der gleiche sein, oder zumindest so ähnlich wie möglich zu dem sein, den man aus der Versionsverwaltung herunterladen kann. Für gewöhnlich gibt es ein paar Unterschiede, zum Beispiel weil das Paket ein paar generierte Dateien enthält, die für die Konfiguration und Kompilierung benötigt werden (siehe „Kompilierung und Installation“ später in diesem Kapitel), oder weil es Software von dritten Parteien enthält, welches nicht von dem Projekt gepflegt wird, welches aber benötigt wird und die Benutzer wahrscheinlich nicht haben. Aber selbst wenn das Veröffentlichte Verzeichnis nicht genau dem Verzeichnis eines Entwicklungszweiges in der Versionsverwaltung entspricht, sollte die Distribution selbst, eine Arbeitskopie (siehe *Arbeitkopie*[5]) sein. Die Veröffentlichung soll einen statischen Referenzpunkt repräsentieren – eine bestimmte, unveränderliche Konfiguration von Quellcode-Dateien. Wenn es eine Arbeitskopie wäre, gäbe es die Gefahr, dass der Nutzer es aktualisieren würde, und nachher denken, dass er die neue Version hat, während er tatsächlich etwas anderes hat.

[5] engl. "working copy"

Denken Sie daran, dass das Paket das gleiche ist, unabhängig von seiner Verpackung. Die Version – also genau die Entität, die gemeint ist, wenn jemand "Scanley 2.5.0" – sagt, ist das Verzeichnis, welches erstellt wird wenn die zip Datei oder der Tarball entpackt wird. Das Projekt könnte also all diese zum Herunterladen anbieten:

scanley-2.5.0.tar.bz2
scanley-2.5.0.tar.gz
scanley-2.5.0.zip

...aber der Quellcode-Baum, der beim entpacken entsteht, muss der gleiche sein. Dieser Quellcode-Baum ist die veröffentlichte Version; die Form in der es heruntergeladen wird, ist lediglich eine Sache der Bequemlichkeit. Bestimmte unwesentliche Unterschiede zwischen den Quellcode-Paketen sind zulässig: Zum Beispiel sollten in dem Windows Paket die Textdateien mit CRLF (Carriage Return und Line Feed) enden, während Unix-Pakete nur LF verwenden sollten. Die Verzeichnisse können auch leicht Unterschiedlich zwischen den verschiedenen Betriebssystemen angeordnet sein, wenn diese Betriebssysteme verschiedene Anordnungen für die Kompilierung erfordern. Das sind jedoch alles im wesentlichen triviale Transformationen. Die wesentlichen Quellcode-Dateien sollten über alle Pakete für eine Version die gleichen sein.

Großschreibung - ja oder nein

Wenn man einen bestimmten Projektnamen nennt, schreiben ihn Leute im allgemeinen groß, wie bei einem Nomen [1] und schreiben Abkürzungen groß wenn solche vorhanden sind: "MySQL 5.0", "Scanley 2.5.0", usw. Ob das in dem Paketnamen mit übernommen wird, ist dem Projekt überlassen. Sowohl `Scanley-2.5.0.tar.gz` als auch `scanley-2.5.0.tar.gz` wären zum Beispiel in Ordnung (Ich persönlich bevorzuge letzteres, da ich es nicht mag, Leute zu zwingen die Umschalttaste zu drücken, aber eine Menge Projekte veröffentlichen Pakete mit Großschreibung). Das Wichtige ist, dass das Verzeichnis welches beim Entpacken des Tarballs erstellt wird, die gleiche Schreibweise verwendet. Es sollte keine Überraschungen geben: Der Nutzer muss den Namen des Verzeichnisses welches erstellt werden wird, wenn er eine Veröffentlichung entpackt, mit absoluter Genauigkeit vorhersagen können.

Vorveröffentlichungen

Wenn Sie eine Vorveröffentlichung machen, ist der Vermerk ein echter Bestandteil der Versionsnummer, nehmen Sie es also mit in den Namen des Pakets auf. Die geordnete Folge von Alpha- und Beta-Versionen, die vorher in diesem Kapitel in „Die Komponenten der Versionsnummer“ verwendet wurde, würde zum Beispiel folgende Paket namen haben:

scanley-2.3.0-alpha1.tar.gz
scanley-2.3.0-alpha2.tar.gz
scanley-2.3.0-beta1.tar.gz
scanley-2.3.0-beta2.tar.gz
scanley-2.3.0-beta3.tar.gz
scanley-2.3.0.tar.gz

Das erste würde zum Beispiel zu einem Verzeichnis namens `scanley-2.3.0-alpha1` entpacken, dass zweite nach `scanley-2.3.0-alpha2`, usw.

Kompilierung und Installation

Bei Software, die eine Kompilierung oder Installation aus den Quellen erfordert, gibt es für gewöhnlich Standardverfahren, welche erfahrene Nutzer erwarten anwenden zu können. Bei Programmen die in C,

[1] Anmk. des Übersetzers: Im Englischen werden Eigennamen (en. proper noun) auch groß geschrieben

C++, oder bestimmte anderen kompilierten Sprachen geschrieben sind, ist es in Unix-ähnlichen Systemen zum Beispiel für den Nutzer üblich, folgendes einzugeben:

```
$ ./configure
$ make
# make install
```

Der erste Befehl erkennt automatisch, soviel von der Umgebung wie möglich, und bereitet den Build-Vorgang vor, der zweite Befehl kompiliert die Software im derzeitigen Pfad (installiert es jedoch nicht), und der letzte Befehl installiert es auf das System. Die ersten beiden Befehle werden als ganz normalen Nutzer ausgeführt, der dritte als root. Für weitere Details über die Einrichtung dieses Systems, siehe das ausgezeichnete *GNU Autoconf, Automake, und Libtool* Buch von Vaughan, Elliston, Tromey, und Taylor. Es ist als Treeware durch New Riders veröffentlicht worden, und sein Inhalt ist unter http://sources.redhat.com/autobook/. frei zugänglich.

Das ist nicht der einzige Standard, obwohl es einer der am weitesten verbreitetsten ist. Das Build-System Ant (http://ant.apache.org/) nimmt an Beliebtheit zu, insbesondere bei Projekten, die in Java geschrieben sind, und hat seine eigenen Abläufe für den Build und die Installation. Desweiteren, empfehlen bestimmte Programmiersprachen, wie Perl und Python, dass die gleiche Methode für die meisten Programme die in dieser Sprache geschrieben werden, benutzt werden (Perl-Module benutzten zum Beispiel den **perl Makefile.PL** Befehl). Wenn es für Sie nicht offensichtlich ist, welche Standards für Ihr Projekt gelten, fragen Sie einen erfahrenen Entwickler; Sie können mit Sicherheit davon ausgehen, dass *irgend ein* Standard gilt, selbst wenn Sie zuerst nicht wissen was es ist.

Was immer die entsprechenden Standards für Ihr Projekt auch sein mögen, weichen Sie nicht von Ihnen ab, es sei denn es ist zwingend notwendig. Standardabläufe bei der Installation sind mittlerweile praktisch Reflexe im Rückenmark vieler Systemadministratoren. Wenn sie bekannte Anweisungen in der `INSTALL` Datei von Ihrem Projekt erkennen, hebt es gleich ihr vertrauen darin, dass Ihr Projekt sich allgemein über Konventionen im klaren ist, und dass es wahrscheinlich ist, dass Sie auch andere Sachen richtig gemacht haben. Wie in „Downloads“ im Kapitel Kapitel 2, *Der Einstieg* beschrieben, erfreute es auch mögliche zukünftige Entwickler, wenn Sie standard Build-Verfahren benutzen.

Unter Windows, sind die Standards um einen Build zu machen und zu installieren weniger festgelegt. Bei Projekten die eine Kompilation erfordern, scheint die allgemeine Konvention zu sein, ein Verzeichnis zu Veröffentlichen, welches in dem Standardmodell für Arbeitsumgebungen/Projekte von den Entwicklungsumgebungen von Microsoft (Developer Studio, Visual Studio, VS.NET, MSVC++, usw.). Abhängig von der Natur Ihrer Software kann es möglich sein, eine Unix-ähnliche Build-Option mittels der Umgebung Cygwin (http://www.cygwin.com/) auf Windows anzubieten. Und wenn Sie natürlich eine Sprache oder einen Programmier-Framework verwenden, welches seine eigenen Konventionen für die Installation hat, – z.B, Perl oder Python – sollte Sie einfach die Standardmethode für dieses Framework benutzen, ob auf Windows, Unix, Mac OS X oder jedem anderen Betriebssystem.

Seien Sie bereit, zusätzliche Mühen auf sich zu nehmen, um Ihr Projekt konform zu den relevanten Build- und Installationsstandards zu machen. Build und Installation sind die Einstiegspunkte: Es ist in Ordnung, wenn Dinge nach diesem Einstieg schwieriger werden, falls denn überhaupt, es wäre aber schändlich, Benutzern oder Entwicklern gleich bei ihrer allererste Begegnung mit der Software unerwartete Schritte abzunötigen.

Binäre Pakete

Auch wenn die formal veröffentlichte Version ein Quellcode-Paket ist, werden die meisten Nutzer von binären Paketen installieren, die entweder von dem Installationsprogramm ihres Betriebssystems angeboten wird, händisch von der Webseite Ihres Projekts besorgt wurden oder von irgend einer anderen

dritten Partei. "Binär" bedeutet in diesem Zusammenhang nicht unbedingt "kompiliert"; es bedeutet lediglich irgend eine vorkonfigurierte Form des Pakets, welches es dem Nutzer erlaubt es auf seinen Computer zu installieren, ohne durch die üblichen Build und Installations-Abläufe der Quellcode basierenden Pakete. Auf RedHat GNU/Linux, ist es das RPM System; auf Debian GNU/Linux, ist es das APT (`.deb`) System; unter MS Windows, sind es üblicherweise `.MSI` Dateien oder selbst installierende `.exe` Dateien.

Ob diese binären Pakete von Personen die nahe an dem Projekt sind, zusammengestellt werden, oder entfernte dritte Parteien, die Nutzer werden sie als gleichwertig zu den offiziellen Veröffentlichungen des Projekts *behandeln*, und werden Meldungen auf den Bugtracker einreichen, auf der Grundlage des Verhaltens dieser binärpakete. Deshalb ist es im Interesse des Projekts denjenigen die Pakete schnüren, klare Richtlinien zu geben, und eng mit ihnen zusammen zu arbeiten, um dafür zu sorgen, dass was sie produzieren, die Software ordentlich und genau repräsentiert.

Die Hauptsache, die Ersteller von Paketen wissen müssen, ist, dass sie ihre binären Pakete immer aus offiziellen Quellcode-Pakete herleiten sollten. Manchmal sind sie versucht, eine spätere Version des Quellcodes aus dem Projektarchiv zu nehmen, oder ausgewählte Änderungen, die nach der Veröffentlichung committet wurden, zu integrieren, um den Nutzern bestimmte Bugfixes oder andere Verbesserungen anzubieten. Der Ersteller des Pakets denkt, dass er seinen Nutzern einen Gefallen tut, indem er ihnen neueren Code gibt, tatsächlich kann dieses Verfahren eine Menge Verwirrung verursachen. Projekte sind darauf vorbereitet, Bug-Meldungen zu bekommen, die in den neun Versionen gefunden werden, und Bugs die in dem derzeitigen Stamm und den Quellen des Hauptastes (das heißt, solche die von Personen gefunden werden, die den allerneusten Code verwenden). Wenn eine Bug-Meldung aus diesen Quellen kommt, wird der Antwortende oft in der Lage sein zu bestätigen, dass dieser Bug in der Momentaufnahme vorhanden ist, und vielleicht, dass es seitdem behoben wurde und dass der Nutzer eine Aktualisierung durchführen soll, oder auf die nächste Version warten soll. Wenn es ein vorher nicht bekannter Bug ist, machte es die Reproduktion und die Einordnung des Fehlers in dem Tracker einfacher, wenn man die genaue Version weiß.

Projekte sind jedoch nicht darauf vorbereitet, Meldungen von Bugs zu erhalten, die auf nicht spezifizierte oder hybrid Versionen basieren. Solche Bugs können schwer zu reproduzieren sein; sie können auch auf unerwartete Wechselwirkungen, zwischen einzelnen Änderungen zurück zu führen sein, die aus der späteren Entwicklung herausgezogen wurden, und deshalb Fehlverhalten verursachen, für die die Entwicklergemeinde die Schuld nicht tragen müssen sollte. Ich habe bestürzend viele Mengen an Zeit daran vergeudet gesehen, weil ein Bug *gefehlt* hat, wenn es hätte da sein sollen: jemand hatte eine leicht aktualisierte Version betrieben, die auf einer offiziellen (aber nicht identische) Version basierte, und als der vorhergesagte Bug nicht auftrat, mussten alle herumsuchen um herauszufinden, warum.

Trotzdem wird es manchmal Umstände geben, unter denen der Ersteller der Pakete darauf besteht, dass Änderungen an der Quellcode-Version nötig sind. Paket-Ersteller sollten dazu ermutigt werden, das bei den Entwicklern des Projekts zur Sprache zu bringen, und ihre Pläne zu beschreiben. Es mag sein, dass sie die Freigabe bekommen, wenn aber nicht, werden sie zumindest das Projekt über ihre Vorhaben informiert haben, damit das Projekt für die üblichen Bug-Meldungen Ausschau halten kann. Die Entwickler mögen reagieren, indem sie eine Ausschlussklausel auf ihre Webseite stellen, und den Paket-Ersteller darum bitten, das gleiche bei sich an angemessener Stelle zu machen, damit die Nutzer dieser binären Pakete wissen, dass was sie bekommen, nicht genau das gleiche ist, wie die offizielle Version. Es muss in einer solchen Situation keine Bitterkeit geben, obwohl es leider oft der Fall ist. Es ist nur, dass die Paket-Ersteller leicht unterschiedliche Ziele im Vergleich zu den Entwicklern verfolgen. Sie wollen hauptsächlich für ihre Nutzer die bestmögliche Erfahrung. Die Entwickler wollen das natürlich auch, sie müssen aber auch sicherstellen, dass sie wissen, welche Versionen der Software im Umlauf sind, damit sie verständliche Bug-Meldungen bekommen können und Garantien über die Kompatibilität machen können. Manchmal stehen diese Ziele im Konflikt zueinander. Wenn das der Fall ist, sollte man sich daran erinnern, dass das Projekt keine Kontrolle über die Paket-Ersteller hat, und dass das Schuldverhältnis in beiden Richtungen läuft. Es stimmt zwar, dass das Projekt den Paket-Erstellern einen Gefallen tut, alleine schon indem sie die Software erstellt. Die Paket-Ersteller tun dem Projekt

aber auch einen Gefallen, indem sie die größtenteils unrühmliche Arbeit auf sich nehmen, die Software einer breiteren Masse zur Verfügung zu stellen, oftmals um mehrere Größenordnungen. Es ist in Ordnung, mit den Paket-Erstellern Meinungsverschiedenheiten zu haben, aber beleidigen Sie sie nicht; versuchen Sie einfach, sich mit ihnen so gut wie möglich zu verständigen.

Tests und Veröffentlichung

Wenn der Quellcode tarball von dem stabilisierten Zweig produziert wurde, beginnt der Ablauf sie zu veröffentlichen. Vor der tarball der allgemeinen Öffentlichkeit aber zur Verfügung gestellt wird, sollte es von einer minimalen Menge von Entwicklern getestet und bewilligt werden, üblicherweise drei oder mehr. Die Bewilligung, dreht sich nicht nur darum, die neue Version auf offensichtliche Mängel zu untersuchen, die Entwickler laden den Tarball herunter, machen einen Build und installieren es auf ein sauberes System, lassen die Folge der Regressionstests laufen (siehe „Automatisiertes Testen“) im Kapitel Kapitel 8, *Leitung von Freiwilligen*, und machen ein paar händische Tests. Angenommen, es besteht diese Überprüfungen, sowie irgend welche andere Kriterien für die Veröffentlichung, die das Projekt haben mag, machen die Entwickler eine digitale Signatur von dem tarball mit GnuPG (http://www.gnupg.org/), PGP (http://www.pgpi.org/), oder irgend einem anderen Programm welches mit PGP kompatible Signaturen erstellen kann.

In den meisten Projekten, benutzen die Entwickler einfach ihre eigenen digitalen Signaturen, anstatt des geteilten Schlüssel vom Projekt und so viele Entwickler die wollen, können es signieren (d.h. es gibt eine Minimum an Entwickler, aber kein Maximum). Je mehr Entwickler signieren, desto mehr Tests unterläuft die neue Version, und desto größer ist die Wahrscheinlichkeit, dass ein sicherheitsbewusster Benutzer ein Pfad des Vertrauens von sich bis zu dem tarball finden kann.

Sobald sie bewilligt sind, sollte die neue Version (also, alle tarballs, zip Dateien, sowie alle anderen Formate die veröffentlicht werden) in dem Download-Bereich des Projekts platziert werden, zusammen mit den Signaturen, und MD5/SHA1 Prüfsummen (siehe http://en.wikipedia.org/wiki/Cryptographic_hash_function). Es gibt verschiedene Standards um das zu machen. Eines ist es, jedes Paket von einer Datei begleiten zu lassen, mit den zugehörigen digitalen Signaturen, und eine weitere Datei mit der Prüfsumme. Wenn das veröffentlichte Paket also `scanley-2.5.0.tar.gz` ist, platzieren Sie in dem selben Verzeichnis eine Datei `scanley-2.5.0.tar.gz.asc` mit der digitalen Signatur für diesen tarball, eine weitere Datei `scanley-2.5.0.tar.gz.md5` mit seiner MD5 Prüfsumme und optional eine weitere Datei `scanley-2.5.0.tar.gz.sha1`, mit der SHA1 Prüfsumme. Eine andere Art die Überprüfung zu ermöglichen ist alle Signaturen der veröffentlichten Pakete in eine einzelne Datei zu sammeln `scanley-2.5.0.sigs`; das gleiche kann mit den Prüfsummen gemacht werden.

Es macht nicht wirklich einen Unterschied wie Sie es machen. Halten Sie sich nur an ein einfaches Schema, beschreiben Sie es klar, und seien einheitlich Sie über mehrere Versionen. Der Sinn von all diesen Signaturen und Prüfsummen ist, Nutzern eine Möglichkeit zu geben, zu verifizieren, dass die Kopie die sie erhalten haben nicht böswillig verändert wurde. Die Nutzer werden gleich den Code auf ihren Rechnern laufen lassen – wenn der Code manipuliert wurde, hätte ein Angreifer plötzlich Zugriff auf all ihre Daten. Siehe „Sicherheitsupdates“ später in diesem Kapitel für weitere Informationen über Paranoia.

Release Candidate

Bei wichtigen Veröffentlichungen die viele Änderungen beinhalten, bevorzugen es viele Projekte zuerst *release candidates* zu veröffentlichen, z.B., `scanley-2.5.0-beta1` vor `scanley-2.5.0`. Der Sinn eines solchen Kandidaten ist den Code weite Tests zu unterwerfen, von es als eine offiziell Version gesegnet wird. Wenn Probleme gefunden werden, werden sie in dem Versionszweig behoben und eine neuer Kandidat wird veröffentlicht (`scanley-2.5.0-beta2`). Dieser Kreislauf wird so lange weitergeführt, bis keine untragbaren Fehler mehr vorhanden sind, worauf der letzte veröffentlichte Kandidat

zur offiziellen Version wird – d.h. der einzige Unterschied zwischen dem letzten Release Candidate und der echten neuen version ist die Entfernung der Kennzeichnung von der Versionsnummer.

In den meisten anderen Beziehungen, sollten die Kandidaten wie eine echte finale Version behandelt werde. Die *alpha* , *beta*, oder *rc* Kennzeichnung reicht, um die konservativen Nutzer zu warnen, dass sie auf die echte finale Version warten sollen, und natürlich sollten die E-Mails zur Bekanntgabe darauf hinweisen, dass ihr Sinn das Einholen von Rückmeldungen ist. Davon abgesehen, geben Sie den Kandidaten die gleiche Menge an Sorgfalt wie gewöhnliche neue Versionen. Schließlich wollen Sie, dass Leute sie benutzen, da echte Nutzung die beste Möglichkeit ist, um Fehler zu entdecken, und weil Sie nie wissen, ob ein Release Candidate zur offiziellen Version wird.

Bekanntgabe neuer Versionen

Eine neue Version bekannt zu geben, ist wie die ankündigung von jedem anderen Ereignis, und sollte die Verfahren die in „Öffentlichkeit“ im Kapitel Kapitel 6, *Kommunikation* beschrieben sind benutzen. Es gibt allerdings ein paar spezifische Sachen bei neuen Versionen.

Immer wenn Sie die URL zu dem Tarball einer neuen Version herausgeben, geben Sie unbedingt auch die MD5/SHA1 Prüfsummen an und weisen Sie auf die Signatur-Datei hin. Da die Bekanntgabe in meheren Foren stattfindet (Mailverteiler, Ihre Webseite, usw.), können Benutzer die Prüfsummen aus mehreren Quellen bekommen, was die sicherheitsbewussten unter ihnen die zusätzliche Sicherheit gibt, dass die Prüfsummen selber nicht manipuliert wurden. Die Verweise auf die Signaturen macht diese nicht unbedingt sicherer, aber es gibt ihnen (insbesondere denjenigen die das Projekt nicht so nahe verfolgen) die Gewissheit, dass das Projekt die Sicherheit ernst nimmt.

In der E-Mail mit der Bekanntgabe, und den Webseiten, die mehr als nur eine kurzen Text über die neue Version enthalten, sollten Sie auch den relevanten Abschnitt aus der CHANGES Datei angeben, damit Leute sehen können, warum es in ihrem Interesse sein könnte eine Aktualisierung durchzuführen. Das ist so wichtig, bei dien Veröffentlichungs Kandidaten wie bei den finalen Versionen; die Existenz von Bugfixes und neuen Funktionen ist wichtig um Leute einen Reitz zu geben, einen neuen Kandidaten auszuprobieren.

Schließlich, vergessen Sie es nicht die Entwicklermannschaft zu danken, sowie die Tester und alle anderen die sich die Zeit genommen haben um gute Bug-Meldungen einzureichen. Heben Sie jedoch keinen besonders beim Namen heraus, es sei denn es ist einer für individuell für einen riesigen Anteil Arbeit verantwortlich ist, dessen Wert weithin von jedem in dem Projekt anerkannt ist. Seien Sie nur vorsichtig vor dem rutschigen Abhang der Inflation von Anerkennung (siehe „Anerkennung“ im Kapitel Kapitel 8, *Leitung von Freiwilligen*).

Wartung mehrerer Versionszweige

Die meisten ausgereiften Projekte pflegen mehrere Versionszweige nebeneinander. Nachdem 1.0.0 erscheint, sollte diese Reihe mit micro Versionen (Fehlerbehebung) 1.0.1, 1.0.2, usw., so lange weiter geführt werden, bis das Projekt sich explizit entscheidet, die Reihe zu beenden. Beachten Sie, dass die Veröffentlichung der 1.1.0 kein ausreichender Grund ist, die 1.0.x Reihe zu beenden. Manche Nutzer aktualisieren zum Beispiel prinzipiell nicht auf die erste Version einer minor oder major Reihe – sie lassen andere die Fehler herausfinden; sagen wir bei der Version 1.1.0 warten sie bis 1.1.1. Das ist nicht zwangsläufig egoistisch (bedenken Sie, dass sie damit auch auf Bugfixes und neue Funktionen verzichten); sie haben sich nur entschieden – aus welchen Gründen auch immer –, bei Aktualisierungen sehr vorsichtig zu sein. Wenn das Projekt von einem ernsthaften Fehler in 1.0.3 erfährt, kurz vor der Veröffentlichung von 1.1.0, wäre es entsprechend hart, den Bugfix nur in 1.1.0 zu stecken, und allen Nutzern der alten 1.0.x Reihe zu sagen, dass Sie aktualisieren sollen. Warum nicht sowohl 1.1.0 als auch 1.0.4 veröffentlichen, damit alle glücklich sind?

Nachdem die 1.1.x Reihe gut läuft, können Sie für 1.0.x erklären, dass es am Ende seines Lebenszykluses (en. *end of life*) ist. Das sollte offiziell bekannt gegeben werden. Diese Bekanntgabe kann für sich erfolgen, oder im Rahmen der Ankündigung einer neuen 1.1.x Version; wie auch immer Sie sich entscheiden: die Nutzer müssen wissen, dass die alte Reihe ausläuft, damit sie entsprechende Entscheidungen im Bezug auf Aktualisierungen treffen können.

Manche Projekte legen einen Fenster fest, indem sie versprechen die vorhergehende Versionsreihe zu pflegen. Im Kontext von Open Source, bedeutet "pflege" (en. "support") die annahme von Bug-Meldungen für diese Reihe, und neue Versionen zu veröffentlichen, wenn bedeutende Fehler gefunden werden. Andere Projekte geben keine definitive Zeitangabe, halten aber ein Auge auf eintreffende Bug-Meldugen um einen Überblick zu behalten, wie viele Personen die alte Reihe noch verwenden. Wenn der Prozentsatz unter einem gewissen Anteil sinkt, erklären sie für die Reihe, dass es am ende seines Lebens ist und hören auf es zu pflegen.

Sorgen Sie bei jeder neuen Version dafür, dass ein *target version* (de. Zielversion) oder *target milestone* (de. Ziel-Meilenstein) in dem Bugtracker zur Verfügung steht, damit Personen die Bugs finden, ihre Meldungen unter der richtig Version einordnen können. Vergessen Sie nicht auch einen Ziel namens "development" (de. Entwicklung) oder "latest" (de. aktuelle) für den neusten Quellcode zu haben, da manche – nicht nur die aktiven Entwickler – oft den offiziellen Versionen voraus bleiben werden.

Sicherheitsupdates

Die meisten Details über die Handhabung von Sicherheitslücken wurden in „Bekanntgabe von Sicherheitslücken“ im Kapitel Kapitel 6, *Kommunikation* behandelt, es gibt aber ein paar besondere Details, die es für Sicherheits-Updates zu besprechen gibt.

Ein *Sicherheitsupdate* ist eine Version die nur gemacht wird, um eine Sicherheitslücke zu schließen. Der Code, welcher den Bug behebt, kann nicht veröffentlicht werden, bis die neue Version zur Verfügung steht, was nicht nur bedeutet, dass die Fixes nicht zum Projektarchiv committet werden können, bis zum Tag der Veröffentlichung, sondern dass die Version nicht öffentlich getestet weden kann, bis sie freigegeben wird. Die Entwickler können den Fix offensichtlich unter sich untersuchen, und die neue Version privat testen, aber breites Testen unter realen Bedingungen ist nicht möglich.

Aufgrund dieses Mangel an Tests, sollte ein Sicherheits-Update immer aus einer bereits herausgegebenen Version bestehen, ergänzt duch die Fixes für die Sicherheitslücke *ohne sonstige Änderungen*. Denn je mehr Änderungen Sie in einer Version veröffentlichen, desto wahrscheinlicher ist es, dass eine von ihnen einen neuen Fehler verursachen wird, vieleicht sogar eine neue Sicherheitslücke! In dieser Hinsicht konservativ zu sein, kommt auch den Administratoren entgegen, die das Sicherheitsupdate einspielen werden, deren Richtlinien im Bezug auf Aktualisierungen es aber nicht zulassen, zur gleichen Zeit sonstige Änderungen einspielen.

Ein Sicherheitsupdate zu erstellen, ist manchmal auch mit ein wenig Täuschung verbunden. Das Projekt könnte zum Beispiel an der Version 1.1.3 arbeiten, für die bestimmte Bugfixes an 1.1.2 bereits öffentlich in Aussicht gestellt wurden, und in diesem Moment trifft die Sicherheitsmeldung ein. Die Entwickler können natürlich nicht über das Sicherheitsproblem reden, bis sie den Fix zur Verfügung stellen; bis dahin müssen sie weiter öffentlich so reden, als wäre 1.1.3 das, was die ganze Zeit geplant war. Wenn 1.1.3 aber wirklich erscheint, wird es sich von 1.1.2 nur durch die Sicherheitsfixes unterscheiden, und alles andere wird auf die Version 1.1.4 aufgeschoben (die jetzt natürlich *auch* den Sicherheitsfix beinhalten wird, genau so wie alle zukünftigen Versionen).

Sie könnten eine weitere Komponente zu der bestehenden Version hinzufügen, welche darauf hindeutet, dass nur Sicherheitsänderungen enthalten sind. Zum Beispiel könnte man alleine an den Zahlen erkennen, dass 1.1.2.1 ein Sicherheitsupdate zu 1.1.2 ist, und es würde deutlich, dass jede "höhere" Version (d.h. 1.1.3, 1.2.0, usw.) die gleichen Sicherheitsfixes beinhaltet. Für diejenigen die es wissen, teilt dieses System eine Menge Informationen mit. Andererseits kann es für diejenigen die das Projekt nicht so nah

verfolgenein, ein wenig verwirrend sein, die meiste Zeit ein Drei-Komponenten-System zu sehen, in das gelegentlich vierstellige Versionsnummen eingestreut sind. Die meisten Projekte die ich mir angeschaut habe, bevorzugen Konsistenz und benutzen einfach die nächste geplante Versionsnummer für Sicherheitsupdates, selbst wenn das bedeutet, andere geplante Versionen um eins zu verschieben.

Neue Versionen und tägliche Entwicklung

Parallele Versionen gleichzeitig zu pflegen, hat Auswirkungen darauf, wie die tägliche Entwicklung vonstatten geht. Es macht insbesondere etwas praktisch zur Pflicht, was sowieso empfohlen wird: Jeder Commit sollte eine einzige logische Änderung sein, und nicht zusammenhängende Änderungen sollten niemals in dem gleichen Commit gemacht werden. Wenn eine Änderung zu groß ist oder zu störend ist, um sie in einem Commit zu machen, verteilen Sie es über N Commits, wobei jeder Commit eine wohl aufgeteilte Untermenge der gesamten Änderung ist, und nichts beinhaltet, das keinen Bezug zu der Gesamtänderung hat.

Hier ist ein Beispiel eines schlecht überdachten Commit:

```
------------------------------------------------------------------------
r6228 | hmustermann | 2004-06-30 22:13:07 -0500 (Wed, 30 Jun 2004) | 8 Zeilen

Fehler #1729 behoben: Sorge dafür, dass die Indexierung dem Nutzer
elegant eine Warnung gibt, wenn eine Datei die sich ändert, indexiert
wird.

* ui/repl.py
  (ChangingFile): Neue Ausnahme-Klasse.
  (DoIndex): Verarbeite die neue Ausnahme.

* indexer/index.py
  (FollowStream): Werfen einer neuen Ausnahme, wenn eine Datei sich
  wärend der Indexierung ändert.
  (BuildDir): In einem anderen Zusammenhang, Entfernung von veralteten
  Komentaren, Neuformatierung vom Code, und Behebung der Fehlerprüfung
  wenn ein Verzeichnis erzeugt wird.

Andere nicht verwandte Aufräumarbeiten:

* www/index.html: Ein paar Vertipper behoben, das Datum der nächsten
Version gesetzt.
------------------------------------------------------------------------
```

Das Problem dabei wird offensichtlich, wenn jemand die Änderung der Fehlerprüfung in `BuildDir` für einen bald anstehende Micro-Version in einen anderen Zweig portieren muss. Der Portierende will nicht irgendwelche der anderen Änderungen – vielleicht wurde der Fix für die Meldung #1729 überhaupt nicht für den Stabilen Zweig genemigt, und die Verbesserungen an `index.html` wären dort einfach irrelevant. Mittels der Merge-Funktion kann er aber die Änderung an `BuildDir` nicht einfach übernehmen, weil dem Versionsverwaltungssytem gesagt wurde, dass diese Änderung mit den anderen nicht zusammenhängenden Dingen logisch gruppiert ist. Tatsächlich würde das Problem sogar vor dem Merge offensichtlich werden. Alleine schon die Auflistung der Änderung zur Abstimmung würde problematisch werden: statt nur die Revisionsnummer anzugeben, müsste der Antragsteller einen besonderen Patch oder Änderungszweig erstellen, nur um den Anteil der Änderung, welcher vorgeschlagen wird, zu isolieren. Das wäre eine Menge Arbeit, unter der die Anderen leiden müssten, und alles nur weil der ursprüngliche Committer keine Lust hatte, die Änderungen logisch zu gruppieren.

Dieser Commit hätte in Wirklichkeit sogar aus *vier* einzelnen Commits bestehen sollen: Einen um den Fehler #1729 zu beheben, einen weiteren um die veralteten Komentare zu entfernen und den Code in `BuildDir` neu zu formatieren, noch einen für die Fehlerprüfung in `BuildDir` , und zuletzt einen, um die Datei `index.html` zu überarbeiten.

Die Stabilisierung der neuen Version ist natürlich nicht der einzige Grund für den Wunsch danach, dass jeder Commit eine einzige logische Änderung ist. Psychologisch ist ein semantisch eindeitiger Commit leichter zu überprüfen und leichter rückgängig zu machen falls nötig (in manchen Versionsverwaltungssystemen ist sowieso eine besondere Art von Merge, wenn man etwas rückgängig macht). Etwas vorausschauende Disziplin der Beteiligten kann dem Projekt später eine Menge Kopfschmerzen ersparen.

Planung neuer Versionen

Ein Bereich, in dem Open-Source-Projekte sich historisch von proprietären Projekten differenziert haben, ist die Planung neuer Versionen. Proprietäre Projekte haben gewöhnlich strengere Fristen. Manchmal weil einem Kunden zu einem bestimmten Datum ein Upgrade versprochen wurde, da die neue Version mit einer anderen Absicht aus Marketing-Gründen koordiniert werden muss, oder weil die Risikokapitalgeber die in die ganze Sache investiert haben, Ergebnisse sehen müssen, bevor sie weitere Finanzierungsmittel hineinstecken. Freie Software-Projekte hingegen waren bis zuletzt meistens durch Amateurhaftigkeit im wörtlichsten Sinne motiviert: Sie wurden aus Liebe geschrieben. Keiner hatte ein Drang, etwas zu veröffentlichen, bevor alle Funktionen fertig waren, und warum sollten sie auch? Es war ja nicht so, dass hier der Arbeitsplatz von jemandem auf den Spiel stand.

Heutzutage werden viele Open-Source-Projekte durch Firmen finanziert, und werden entsprechen mehr und mehr durch fristbewusste Unternehmenskulturen beeinflusst. Das mag in vielerlei Hinsicht etwas Gutes sein, kann aber auch Konflikte verursachen zwischen den Prioritäten der Entwickler, die bezahlt werden, von denen der Entwickler, die ihre Zeit freiwillig investieren. Diese Konflikte kommen oft bei der Frage auf, wann und wie die neuen Versionen geplant werden. Die angestellten Entwickler, die unter Druck stehen, werden natürlich einfach irgendein Datum wählen wollen, an dem die Veröffentlichung stattfinden soll, und alle anderen Aktivitäten entsprechend einordnen. Die freiwilligen Entwickler können jedoch andere Pläne haben – vieleicht Funktionen, die sie fertig stellen wollen, oder Tests, die sie gemacht haben wollen – worauf ihrer Meinung nach die Veröffentlichung warten soll.

Es gibt keine allgemeine Lösung für dieses Problem, außer natürlich zu diskutieren und Kompromisse zu schließen. Sie können jedoch den Grad und die Menge an Reibung die verursacht wird minimieren, indem Sie die *Existenz* der vorgeschlagenen Version von dem Datum entkoppeln, an dem sie veröffentlicht werden soll. Das heißt, versuchen Sie die Diskussion in Richtung des Themas zu lenken, welche Versionen das Projekt in der nächsten bis mittelfristigen Zeit machen wird, und welche Funktionen sie haben werden, zunächst ohne über Termine zu sprechen, abgesehen von groben Schätzungen mit einer Menge Spielraum. [2] Indem Sie frühzeitig den Funktionsumfang festlegen, reduzieren Sie die Komplexität der Diskussion, die sich um irgend eine bestimmte Version dreht, und verbessern dadurch die Berechenbarkeit. Das verursacht auch eine Art träger Voreinstellung, auf Leute, die vorschlagen die Definition einer Version mit neuen Funktionen oder andere Komplikationen zu erweitern. Wenn der Inhalt der Version relativ gut definiert ist, obliegt die Rechtfertigungspflicht für Erweiterungen dem Vorschlagenden, auch wenn das Datum der Version noch nicht festgelegt wurde.

In seiner mehrbändigen Thomas-Jefferson-Biographie, *Jefferson and His Time*, erzählt Dumas Malone die Geschichte, wie Jefferson das erste Treffen handhabte, das abgehalten wurde, um über die Organization der zukunftigen Universität von Virginia zu entscheiden. Die Universität war von Anfang an die Idee von Jefferson gewesen, aber (wie es überall der Fall ist, nicht nur in Open-Source-Projekten)

[2] Einen alternativen Ansatz können Sie in Martin Michlmayrs Dr. phil. Dissertation nachlesen *Quality Improvement in Volunteer Free and Open Source Software Projects: Exploring the Impact of Release Management* (http://www.cyrius.com/publications/michlmayr-phd.html). Sie behandelt einen zeitbasierten im Gegensatz zu einem Feature-basierten Herausgabe-Prozess für umfangreiche free Software-Projekte. Michlmayr hielt auch einen Vortrag bei Google zu diesem Thema, verfügbar als Video unter http://video.google.com/videoplay?docid=-5503858974016723264.

waren sehr schnell viele andere Parteien an Bord, jede mit eigenen Interessen und Anliegen. Als sie sich zu diesem ersten Treffen versammelten, um alles auszuarbeiten, erschien Jefferson mit minuziös vorbereiteten Bauplänen, detaillierten Budgets für die Konstruktion und den Betrieb, einem Lehrplanentwurf, und den Namen der einzelnen Fakultäten die er aus Europa importieren wollte. Kein anderer im Raum war nur annähernd so gut vorbereitet; die Gruppe musste im wesentlichen vor der Vision von Jefferson kapitulieren und die Universität wurde letztlich mehr oder weniger entsprechend seinen Plänen gegründet. Die Tatsachen, dass die Konstruktion weit über sein Budget ging, und viele seiner Ideen aus verschieden Gründen am Ende nicht funktionierten, waren Dinge, von denen Jefferson wahrscheinlich anfangs genau wusste, dass sie passieren würden. Sein Vorhaben war strategisch: Bei der Versammlung mit etwas derart Stichhaltigem aufzutauchen, dass jeder ander in die Rolle verfallen müsste, lediglich Änderungen daran vorzuschlagen, damit die allgemeine Gestalt, und dadurch der Terminplan des Projekts, ungefähr so bliebe, wie er es wollte.

Im Falle eines freien Software-Projekts, gibt es kein einzelnes "Treffen", sondern stattdessen eine Reihe kleiner Vorschläge die meistens durch den Bugtracker gemacht werden. Wenn Sie aber von Anfang an etwas Ansehen im Projekt haben, und anfangen verschiedene Funktionen, Verbesserungen, und Fehler für bestimmte Versionen im Bugtracker festzulegen, entsprechend irgend einem erklärten Gesamtplan, werden die Leute meistens mitmachen. Wenn Sie erst alles mehr oder weniger so ausgelegt haben, wie Sie es wollen, werden die Unterhaltungen über echte *Termine* für neue Versionen sehr viel glatter verlaufen.

Es ist natürlich äußerst wichtig, dass Sie niemals irgend eine einzelne Entscheidung als in Stein gemeißelt präsentieren. Zeigen Sie in den Kommentaren anlässlich der Zuordnung von Bugs zu bestimmten Versionen wenn möglich stets Bereitschaft zu Diskussionen, Meinungsverschiedenheiten und die allgemeine Bereitschaft, überredet zu werden. Üben Sie niemals Kontrolle alleine um ihrer Ausübung willen: Je mehr andere sich an der Planung einer neuen Version beteiligen (siehe „Teilen sie sowohl Verwaltungsaufgaben als auch technische Aufgaben“ im Kapitel Kapitel 8, *Leitung von Freiwilligen*), desto leichter wird es sein sie zu überreden, Ihre Prioritäten bei Angelegenheiten zu teilen, die Ihnen wirklich wichtig sind.

Die andere Möglichkeit, Spannungen bei der Planung neuer Versionen des Projekts zu veringern ist, relativ häufig zu veröffentlichen. Wenn zwischen den Veröffentlichungen eine lange Zeit liegt, wird die Bedeutung von jeder einzenen in den Köpfen allern vergrößert; die Leute sind um so mehr betrübt, wenn ihr Code es nicht hinein schafft, weil sie wissen wie lange es dauern könnte, bis die nächste Gelegenheit kommt. Abhängig von der Komplexität des Ablaufs bei einer neuen Version und der Natur Ihres Projekts, liegt die richtige Zeit zwischen den einzelnen Veröffentlichungen gewöhnlich irgendwo zwischen drei und sechs Monaten, obwohl in den Stabilen Zweigen Micro-Veröffentlichungen ein wenig schneller geschehen können, wenn dafür Bedarf besteht.

Kapitel 8. Leitung von Freiwilligen

Leute dazu zu bringen, sich darauf zu einigen, was das Projekt benötigt, und zusammen zu arbeiten, um es zu erreichen, erfordert mehr als nur eine angenehme Atmosphäre und das Vermeiden der gröbsten Fehler. Es erfordert jemanden oder mehrere, die alle beteiligten Personen bewusst leiten. Die Leitung von Freiwilligen mag kein technisches Handwerk sein, vergleichbar etwa mit dem Programmieren, es ist aber dennoch ein Handwerk im dem Sinne, dass es durch lernen und üben verbessert werden kann.

Dieses Kapitel ist ein lose Sammlung spezifischer Techniken für die Leitung von Freiwilligen. Es greift vielleicht mehr als vorherige Kapitel auf das Subversion-Projekt als Fallstudie zurück, zum Teil da ich an dem Projekt gearbeitet habe, als ich das hier schrieb und alle primären Quellen griffbereit hatte, und zum Teil weil es akzeptabler ist, wenn man die Steine der Kritik ins eigene Glashaus wirft, als in das anderer. Ich habe aber auch in verschiedenen anderen Projekten die Vorteile der Anwendung der folgenden Emfehlungen kennengelernt – und die Konseqenzen der Nichtanwendung –; wenn es mir politisch korrekt erscheint, Beispiele aus einigen dieser Projekte zu geben, werde ich das tun.

Da wir schon einmal über Politik reden, sollten wir dieses äußerst unheilvolle Wort einmal genauer unter die Lupe nehmen. Viele Ingenieure meinen Politik sei etwas, mit dem sich besser andere beschäftigen. "*Ich* spreche nur für den besten Kurs für das Projekt, aber *sie* erheben Einsprüche aus politischen Gründen." Ich denke dieser Ekel vor Politik (oder dem was als Politik verstanden wird) ist in Ingenieuren besonders stark, da sie sich die Idee angeeignet haben, dass manche Lösungen objektiv besser sind als andere. Wenn es also scheint, dass jemand sich auf eine Art benimmt, als wäre er durch äußere Überlegungen motiviert – sagen wir der Erhaltung ihrer einflussreichen Position, die Verminderung des Einflusses von jemand anderem, gar offener Kuhhandel oder die Meidung, die Gefühle von jemand zu verletzen – könnten andere Mitwirkende im Projekt genervt sein. Das hindert sie natürlich nicht daran, sich auf die gleiche Art zu benehmen, wenn ihre lebenswichtigen Interessen auf dem Spiel stehen.

Wenn Sie "Politik" für ein schmutziges Wort halten, und hoffen Ihr Projekt davon frei zu halten, geben Sie es besser gleich auf. Politik ist unvermeidlich, wann immer Menschen im Verwalten gemeinschaftlicher Ressourcen kooperieren müssen. Es ist völlig vernünftig, dass eine der Überlegungen, die in der Entscheidungsfindung von jemandem einfließt, die Frage ist, wie ein Vorgang sich auf den eigenen zukünftigen Einfluss im Projekt, auswirken wird. Wenn Sie schließlich Ihrem eigenen Urteilsvermögen und Ihren Fähigkeiten vertrauen, wie es bei den meisten Programmierern der Fall ist, dann muss der mögliche Verlust von zukünftigem Einfluss, im gewissen Sinne als techniches Ergebnis in Betracht gezogen werden. Ähnliche Schlussfolgerungen gelten für anderes Verhalten, welche vielleicht oberflächlich betrachtet, nach "reiner" Politik aussieht. Tatsächlich gibt es so etwas wie reine Politik nicht: Es ist genau aus dem Grund, dass Vorgänge mehrere Konsequenzen in der echten Welt habe, dass Menschen überhaupt ein politisches Bewusstsein bekommen. Politik ist, letztendlich, einfach eine Anerkennung, dass *alle* Konzequenzen von Entscheidungen, in betracht gezogen werden müssen. Wenn eine bestimmte Entscheidung zu einem Ergebinis führt, welches die meisten Mitwirkenden technich zufriedenstellend finden, welches aber mit einer Änderung in den macht Beziehungen zusammenhängt, dass entscheidende Personen ein Gefühl der Isolation gibt, ist letzteres ein genau so wichtiges Ergebnis wie ersteres. Es zu ignorieren, wäre nicht hoch geistig, sondern kurz sichtig.

Wärend Sie also die nachfolgenden Ratschläge lesen, und wärend Sie an Ihrem eigenen Projekt arbeiten, denken Sie daran, dass es *keinen* gibt, der über die Politik steht. Zu erscheinen, als wäre man über die Politik, ist lediglich eine bestimmte politische Strategie, und manchmal ein sehr nützliches, es entspricht aber niemals der Wirklichkeit. Politik ist einfach was passiert, wenn Menschen Meinungsverschiedenheiten haben, und erfolgreiche Projekte sind solche die politische Mechanismen entwickeln um diese Streitigkeiten konstruktiv zu verwalten.

Das meiste aus Freiwilligen herausholen

Warum arbeiten Freiwillige an freien Softwareprojekten? [1]

Wenn man sie fragt, behaupten biele, dass sie es machen, weil sie gute Software produzieren wollen, oder persönlich daran beteiligt sein wollen, die Fehler zu beheben, die ihnen wichtig sind. Aber diese Gründe sind gewöhnlich nicht die ganze Geschichte. Könnten Sie sich schließlich einen Freiwilligen vorstellen, der bei einem Projekt bleibt, selbst wenn keiner je ein Wort der Anerkennung über seine Arbeit sagen würde, oder keiner ihm in Diskussionen zuhören würde? Natürlich nicht. Menschen verbringen ganz klar Zeit an freien Software-Projekten, aus Gründen die über das Bedürfnis. guten Code zu produzieren, hinaus gehen. Die wirkliche Motivation von Freiwilligen zu verstehen, wird Ihnen helfen, es so einzurichten, dass sie angelockt werden und dabei bleiben. Der Wunsch, gute Software zu produzieren, mag eine dieser Motivationen sein, zusammen mit der Herausforderung und dem Bildungswert an schwierign Problemen zu arbeiten. Menschen haben aber auch ein eingebautes Bedürfnis, mit anderen zusammen zu arbeiten, und Respekt durch Zusammenarbeit zu geben und verdienen. Gruppen die gemeinschaftlich aktiv sind, müssen Normen entwickeln, die es erlauben, Leistungen einzelner die den Zielen der Gruppe dienen, durch Erwerb und Erhalt von Ansehen zu belohnen.

Diese Normen werden nicht immer von allein auftauchen. Bei manchen Projekten – erfahrene Open-Source-Entwickler können wahrscheinlich spontan mehere nennen – haben Leute zum Beispiel anscheinend das Gefühl, dass Ansehen durch häufige und ausführliche Nachrichten angeeignet wird. Sie kommen nicht aus versehen zu diesem Schluss; sie kommen darauf, weil sie für lange, komplexe Argument, mit Respekt belohnt werden, ob das dem Projekt hilft oder nicht. Nachfolgend sind einige Techniken um eine Atmosphäre zu erzeugen, in dem Aktivitäten die Ansehen verschaffen auch konstruktive Aktivitäten sind.

Delegierung

Deligierung ist nicht nur eine möglichkeit um die Arbeit zu verteilen; es ist auch ein politisches und soziales Werkzeug. Betrachten Sie alle Auswirkungen, wenn Sie jemand darum bitten, etwas zu machen. Die offensichtlichste Folge ist, falls er annimmt, dass er die Aufgabe erledigt, und Sie nicht. Eine weitere Folge ist aber, dass ihm bewusst gemacht wird, dass Sie ihm zugetraut haben die Aufgabe zu erledigen. Desweiteren, wenn Sie die Anfrage in einem öffentlichen Forum gemacht haben, weiß er auch, dass ander in der Gruppe auch über dieses Vertrauen unterrichtet wurden. Er mag auch etwas Druck verspüren anzunehmen, was bedeutet, dass Sie derart Fragen müssen, dass es ihm erlaubt taktvoll abzulehnen, falls er die Aufgabe nicht wirklich haben will. Wenn die aufgabe Koordination mit anderen im Projekt erfordert, schlagen Sie effektiv vor, dass er sich mehr beteiligt, Verbindungen aufbaut die vielleicht sonst nicht entstanden wären, und möglicherweise zu einer Autorität auf in einem Teilgebiet des Projekts wird. Die zusätzliche Beteiligung mag einschüchternd sein, oder ihn dazu führen, sich auch auf andere Arten zu engagieren, durch ein erweitertes Gefühl seiner gesamten Verpflichtung.

Aufgrund all dieser Auswirkungen, macht es oft Sinn, jemand anderes darum zu bitten, etwas zu machen, selbst wenn Sie wissen, dass Sie es selber schneller oder besser machen könnten. Es gibt natürlich hierfür manchmal sowieso ein gänzlich wirtschaftliches Argument: Die Kosten der Möglichkeit es selber zu machen können zu hoch sein – es mag etwas noch wichtigeres geben, was Sie mit der Zeit machen könnten. Aber selbst wenn das Argument über die Kosten der Möglichkeit nicht gilt, kann es *trotzdem* in Ihrem Interesse sein, jemand anderen darum zu bitten die Aufgabe an sich zu nehmen, denn auf lange Sicht wollen Sie diese Person tiefer in das Projekt einbeziehen, selbst wenn es etwas mehr Zeit bedeutet, zunächst ein wenig mehr auf sie aufzupassen. Die umgekehrte technig gilt auch: Wenn

[1] Diese Frage wurde im Detail untersucht, mit interesanten Ergebnissen, in einer Veröffentlichung von Karim Lakhani und Robert G. Wolf, mit dem Titel *Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects* (de. Warum Hacker tun was sie tun: Verständnis der Motivation und Bestrebungen in Freien/Open Source Software Projekten). Siehe http://freesoftware.mit.edu/papers/lakhaniwolf.pdf.

Sie sich gelegentlich für Arbeit melden, welches jemand anderes nicht machen will oder wofür er nicht die nötig Zeit hat, werden Sie sein Wohlwollen und Respekt ernten. Deligierung und Ersetzung drehen sich nicht nur darum einzelne Aufgaben erledigt zu bekommen; es geht auch darum Leute in eine engere Verpflichung an das Projekt zu bringen.

Unterscheiden Sie eindeutig zwischen Anfrage und Anweisung

Manchmal kann man erwarten, dass eine Person eine bestimmte Aufgabe annehmen wird. Wenn jemand zum Beispiel eine Bug im Code schreibt, oder Code committed, welche auf irgend eine offensichtliche Art, nicht den Richtlinien des Projekts entspricht, dann reicht es aus auf das Problem hinzuweisen, und sich danach zu verhalten, als würden Sie annehmen, dass sich die Person darum kümmern wird. Es gibt aber andere Situationen bei denen es keineswegs klar ist, dass Sie ein Recht darauf haben, eine Reaktion zu erwarten. Die Person kann machen worum Sie ihn bitten, oder auch nicht. Da keiner es mag, als selbstverständlich erachtet zu werden, müssen Sie für den Unterschied zwischen diesen beiden Arten von Situationen, einfühlsam sein, und Ihre Anfragen entsprechend anpassen.

Eine Sache bei dem Leuten fast immer sofort verärgert werden, ist wenn sie auf eine Art gefragt werden etwas zu machen, welches impliziert, dass Sie der Meinung sind, dass es ganz klar ihre Verantwortung ist, wenn Sie anderer Meinung sind. Zum Beispiel ist die Zuweisung eintreffender Meldungen eine besonders reichhaltiger Boden für diese Art von Verärgerung. Die Beteiligten in einem Projekt wissen für gewöhnlich, wer Exerte in welchen Bereichen ist, wenn also eine Bug-Meldung eintrifft, wird es oft ein oder zwei Personen geben, von denen jeder weiß, dass sie das Problem schnell beheben könnten. Wenn Sie jedoch die Meldung einer dieser Personen zuweisen, ohne vorherige Zustimmung, kann er das Gefühl bekommen, in eine unbequeme Lage gebracht zu werden. Er spürt den Druck der erwartung, aber auch, dass er effektiv für seine Kentnisse bestraft wird. Schließlich erlangt man diese Kenntnisse, indem man Bugs behebt, also sollte vielleicht jemand anderes diesen übernehmen! (Beachten Sie, dass Ticket-Systeme die Tickets automatisch bestimmten Personen zuweisen, aufgrund von Informationen in dem Ticket, wahrscheinlich nicht so sehr beleidigend sind, da jeder weiß, dass die Zuweisung automatisch durchgeführt wurde, und keine Andeutung von menschlichen Erwartungen ist.)

Obwohl es nett wäre die Last so gleichmäßig wie möglich zu verteilen, gibt es bestimmte Zeiten, wenn Sie einfach die Person die einen Fehler am schnellsten beheben kann, dazu ermutigen wollen. Angesichts dessen, dass Sie es sich den Kommunikationsaufwand bei jeder solchen Zuweisung nicht leisten können ("Wärst du bereit dir diesen Bug anzuschauen? "Ja." "Alles klar, dann weise ich dir den Ticket zu." "Ok."), sollten Sie die Zuweisung einfach in der Form einer Anfrage machen, welche keinen Druck vermittelt. Praktisch alle Ticket-Systeme erlauben, dass ein Kommentar an eine Zuweisung angehängt wird. In diesem Kommentar, können Sie soetwas sagen:

> Ich weise dir das mal zu, hmustermann, weil du dich am besten in diesem Code auskennst. Wenn du keine Zeit hast es dir anzuschauen, kannst du es ruhig zurückgeben. (Und sag mir bescheid, wenn du in Zukunft solch Anfragen lieber nicht bekommen würdest.)

Das unterscheidet eindeutig zwischen der *Anfrage* für eine Aufgabe und *Annahme* dieser Aufgabe seitens des Empfängers. Das Publikum hier ist nicht nur der Zugewiesene, es sind alle: Die ganze Gruppe sieht eine öffentliche bestätigung der Fähigkeiten des Zugewiesenen, die Nachricht macht aber auch klar, dass es ihm frei steht, die Verantwortung dafür anzunehmen oder abzulehnen.

Bleiben Sie nach dem Deligieren auf dem Laufenden

Wenn Sie jemanden darum bitten, etwas zu machen, behalten Sie es im Hinterkopf, und schließen Sie mit ihm danach an, egal was passiert. Die meisten Anfragen werden in öffentlichen Foren gemacht, und sind ungefähr von der Art "Kannst du dich um X kümmern? Sag uns bescheid, egal wie du dich entscheidest; kein Problem wenn du keine Zeit hast, wir müssens nur wissen". Sie können darauf eine Ant-

wort bekommen oder auch nicht. Wenn ja, und die Antwort negativ war, ist der Kreis geschlossen – Sie werden eine andere Strategie versuchen müssen um mit X um zu gehen. Wenn es eine positive Antwort ist, da behalten Sie den Verlauf des Tickets im Auge, und machen Sie Anmerkungen zu den Fortschritten die Sie sehen oder nicht sehen (jeder arbeitet besser, wenn er weiß, dass jemand anderes seine Arbeit zu schätzen weiß). Wenn es nach ein paar Tagen keine Antwort gibt, fragen Sie nochmal nach, oder schreiben Sie, dass Sie keine Antwort bekommen haben und jetzt nach jemand anderem suchen der es erledigt. Oder machen Sie es einfach selber, aber vergewissern Sie sich, dass Sie keine Antwort auf die ursprüngliche Anfrage bekommen haben.

Öffentlich anzumerken, dass es keine Antwort gab, verfolgt *nicht* den Zweck, die Person zu beschämen, und Ihre Anmerkungen sollten dementsprechend formuliert sein. Der Sinn ist einfach zu zeigen, dass an Antworten auf Ihre Nachfragen Sie Anfragen gemacht haben, und dass Sie die Antworten die Sie bekommen tatsächlich interessiert sind und sie wahrnehmen. Das erhöht die Wahrscheinlichkeit, dass Leute beim nächsten mal ja sagen, da Sie (wenn auch nur unbewusst) feststellen, dass Sie sicherlich bemerken werden, wenn sie irgend welche Arbeit erledigen, angesichts Ihrer Aufmerksamkeit gegenüber dem viel weniger sichtbaren Ereignis, dass jemand nicht antwortet.

Achten Sie darauf, wofür Leute sich interessieren

Eine weitere Sache die Leute glücklich macht, ist wenn man ihre Interessen bemerkt – allgemein wird jemand umso gemütlicher sein, je mehr Aspekte seiner Persönlichkeit Sie bemerken und erinnern, und umso eher wird er in Gruppen arbeiten wollen, an denen Sie beteiligt sind.

Es gab zum Beispiel eine scharfe Unterscheidung im Subversion-Projekt zwischen Personen die eine endgültige 1.0-Version erreichen wollten (was wir letztendlich schafften), und solchen, die hauptsächlich neue Funktionen hinzufügen sowie an interessante Probleme bearbeiten wollten, die es aber nicht sonderlich kümmerte, wann 1.0 erscheinen würde. Keine dieser Positionen ist besser oder schlechter als die andere; sie sind lediglich zwei unterschiedliche Arten von Entwicklern, und beide erledigen eine Menge Arbeit im Projekt. Wir lernten aber schnell, dass es wichtig war *nicht* anzunehmen, dass die Aufregung um den Schub auf 1.0 von jedem geteilt wurde. Elektronische Medien können sehr trügerisch sein: Sie können eine Atmosphäre einer gemeinsammen Bestimmung spüren, wenn in Wirklichkeit es nur bon den Personen geteilt wird, mit denen Sie zufällig geredet haben, wären andere völlig unterschiedliche Prioritäten haben.

Je ehere Sie sich bewusst sind, was Leute von dem Projekt wollen, desto effektiver können Sie an ihnen Anfragen stellen. Selbst ein Verständnis dafür zu zeigen, was sie wollen, ohne eine zugehörige Anfrage, ist in sofern nützlich, dass es jeder Person bestätigt, dass sie nicht nur eine weiteres Teilchien in einer einheitlichen Masse ist.

Lob und Kritik

Lob und Kritik sind keine Gegensätze: In vielerlei Hinsicht, sind sie sich sehr ähnlich. Beide sind vor allem Formen von Aufmerksamkeit, und sind am effektivsten, wenn sie eher spezifisch sind als allgemein. Beide sollten, mit konkreten Zielen im Blick, angewandt werden. Beide können durch Iinflation geschwächt werden: Zuviel oder zu oft zu loben, wird den Wert des Lobs mindern; das gleiche gilt für Kritik, obwohl in der Praxis Kritik eher eine Reaktion ist und deshalb etwas weniger anfällig für Wertminderung ist.

Eine wichtige Funktion, der technischen Kultur ist, dass detailierte leidenschaftslose Kritik oft als eine Art Lob verstanden wird (wie in „Unhöflichkeiten erkennen“ im Kapitel Kapitel 6, *Kommunikation* beschrieben), aufgrund der Implikation, dass die Arbeit des Empfängers die Zeit, welche für seine Analyse benötigt wird, es wert ist. Beide dieser Bedingungen – *detalliert* und *leidenschaftslos* – müssen jedoch erfüllt sein, damit das wahr ist. Wenn jemand zum Beispiel eine schlampige Änderung am Code macht, ist es nutzlos (und sogar schädlich) darauf zu antworten, indem man einfach sagt "Das war

schlampig". Schlampigkeit ist letztlich die Eigenschaft einer *Person*, nicht ihrer Arbeit, und es ist wichtig, Ihre Reaktionen auf die Arbeit zu konzentrieren. Es ist viel effektiver, taktvoll und ohne Böswilligkeit, alles zu beschreiben was an der Arbeit falsch ist. Wenn das die dritte oder vierte sorglose Änderung nacheinander von der selben Person ist, ist es angemessen das am Ende Ihrer Kritik zu sagen – wieder ohne Wut – um klar zu machen, dass man das Muster bemerkt hat.

Wenn jemand sich nicht in reaktion auf die Kritik bessert, ist die Lösung nicht mehr oder stärkere Kritik. Die Läsung ist für die Gruppe diese Person aus seiner Position von Inkompetenz zu entfernen, so das es die verletzten Gefühle so gering wie möglich hält; siehe „Übergänge“ später in diesem Kapitel für Beispiele. Was jedoch selten vorkommt. Die meisten Leute reagieren ziemlich gut auf Kritik die spezifisch und detalliert ist, sowie eine klare (wenn auch nicht ausgesprochene) Erwartung einer Verbesserung beinhaltet.

Lob wird natürlich keine Gefühle verletzen, was aber nicht bedeutet, dass sie es weniger vorsichtig nutzen sollten als Kritik. Lob ist ein Werkzeug: Bevor Sie es nutzen, fragen Sie sich, *warum* Sie es nutzen wollen. In der Regel ist es keine gute Idee, Leute für das zu loben, was sie normalerweise machen, oder für Aktivitäten, welche ein normaler und zu erwartender Bestandteil der Teilnahme an der Gruppe sind. Würden Sie damit beginnen, wäre es schwierig, hierin ein Ende zu finden: Sollten Sie *jeden* für die übliche Arbeit loben? Wenn Sie dann manche auslassen, werden diese sich fragen warum. Es ist viel besser, Lob und Dankbarkeit sparsamm zum Ausdruck zu bringen, als Reaktion auf ungewöhnliche oder unerwartete Anstrengungen, mit der Absicht mehr solcher Arbeit zu ermutigen. Wenn es scheint, dass ein Beteiligter sich in einem Zustand dauerhaft erhöhter Produktivität bewegt hat, sollten Sie das Maß an Lob bei dieser Person entsprechend anpassen. Wiederholtes Lob für normales Verhalten wird mit der Zeit sowieso bedeutungslos. Stattdessen sollte diese Person spüren, dass ihr hoher Grad an Produktivität als normal und natürlich erachtet wird, und nur Arbeit, die darüber hinaus geht, sollte besonders zur Kenntnis genommen werden.

Das soll natürlich nicht heißen, dass die Beiträge der Person nicht gewürdigt werden sollten. Denken Sie aber daran, dass wenn das Projekt richtig eingerichtet wurde, alles was diese Person macht, ohnehin sichtbar sein wird, und die Gruppe wird wissen (und die Person wird wissen, dass die Gruppe weiß), was sie alles macht. Neben direktem Lob gibt es auch andere Möglichkeiten die Arbeit von jemand anzuerkennen. Sie könnten beiläufig erwähnen, wärend Sie ein verwandtes Thema besprechen, dass die Person eine Menge Arbeit in dem entsprechenden Bereich geleistet hat und auf dem Gebiet Experte ist; Sie könnten sie öffentlich zu einer Frage über den Code konsultieren; oder vielleicht am effektivsten, könnten Sie die Arbeit, die sie geleistet hat anderweitig nutzen, damit sie sieht, dass andere sich jetzt mit ruhigem Gewissen auf die Ergebnisse ihrer Arbeit verlassen. Es ist wahrscheinlich nicht nötig das alles auf eine berechnete Art zu machen. Jemand der regelmäßig große Beiträge in einem Projekt leistet, wird es wissen, und wird standardmäßig eine einflussreiche Position einnehmen. Es gibt gewöhnlich kein Bedarf für explizite Schritte, um das sicherzustellen, es sei denn Sie spüren, dass ein Beteiligter aus irgendwelchen Gründen nicht entsprechend geschätzt wird.

Verhindern Sie Revierabsteckung

Geben Sie auf Beteiligte acht, die versuchen exklusiven Besitz für bestimmte Bereiche im Projekt abzustecken, und bei denen es scheint, als wollten Sie in den Bereichen, die ganze Arbeit machen, sogar bis zu dem Grad, dass sie Arbeit, die andere anfangen, aggresiv übernehmen. Solches Verhalten mag am Anfang sogar gesund erscheinen. Oberflächlich, sieht es aus, als würde die Person mehr Verantwortung übernehmen, und in dem gegebenen Bereich mehr Aktivität zeigen. Auf lange Sicht ist es allerdings schädlich. Wenn Leute ein "Nicht betreten!"-Schild sehen, bleiben sie weg. Das hat eine reduzierte Überprüfung innerhalb dieses Bereichs zur Folge, sowie größere Zerbrechlichkeit, da der einzelne Entwickler zu einem kritischen Ausfallpunkt wird. Schlimmer noch, es zerbricht den gemeinschaftlichen, egalitären Geist des Projekts. Die Theorie sollte immer sein, dass jeder Entwickler gerne bei jeder Aufgabe und zu jeder Zeit helfen kann. In der Praxis, funktionieren die Sachen ein wenig anders: Leute haben doch ihre Bereiche, in denen sie mehr Einfluss haben, und nicht Experten verweisen oft auf die

Experten auf bestimmten Gebiten des Projekts. Die Hauptsache ist, dass das alles freiwillig ist: Informelle Autorität wird aufgrund von Kompetenz und bewiesenem Urteilsvermögen erteilt, sollte aber nie aktiv *genommen* werden. Selbst wenn die Person wirklich Kompetent ist, ist es trotzdem noch kritisch, dass sie diese Autorität informell hält, durch das Bewusstsein des Projekts, und dass die Autorität sie niemals dazu bringt andere davon auszuschließen, in dem Bereich zu arbeiten.

Die Arbeit von jemand aus technischen Gründen abzuweisen oder zu bearbeiten ist natürlich etwas föllig anderes. Dort ist die Arbeit das entscheidende, nicht wer zufällig den Torwächter gespielt hat. Es kann sein, dass die selbe Person für einen bestimmten Bereich die meisten commits überprüft, so lange er aber nie versucht jemand anders daran zu hinder auch diese Arbeit zu machen, ist wahrscheinlich alles in Ordnung.

Um gegen das einsetzende Entstehen von Revieren anzukämpfen, oder gar sein Auftauchen, sind viele Projekte dazu übergegangen, Namen von Autoren oder Zuständigen in Quellcode-Dateien zu verbieten. Ich stimme diesem Verfahren voll und ganz zu: Wir verfahren so im Subversion-Projekt, und es ist mehr oder weniger eine offizielle Richtlinie bei der Apache Software Foundation. ASF-Mitglied Sander Striker drückt es so aus:

> *Bei der Apache Software Foundation raten wir von der Nutzung von Autor-Markierungen in dem Quellcode ab. Es gibt dafür verscheidene Gründe, mal abgesehen von den rechtlichen Konsequenzen. Bei der gemeinschaftlichen Entwicklung geht es darum, als Gruppe an Projekten zu arbeiten, und sich als Gruppe um das Projekt zu kümmern. Anerkennung ist gut und sollte gewährt werden, allerdings auf eine Art welche keine falsche Anerkennung erlaubt, selbt implizite. Es gibt keine klare Grenze, wann man eine Autor-Markierung machen oder entfernen soll. Fügen Sie Ihren Namen hinzu, wenn sie ein Kommentar bearbeiten? Wenn Sie einen einzeiligen Fix hinzufügen? Entfernen Sie die alten Markierungen wenn sie den Code neu strukturieren, und es zu 95% anders aussieht? Was machen Sie mit Leuten die herumgehen und jede Datei anfassen, und gerade genug ändern, um eine virtuelle Quote zu erreichen, damit ihr Name überall ist?*
>
> *Es gibt bessere möglichkeiten, Anerkennung zu erweisen, und wir bevorzugen es, diese zu nutzen. Aus technischer Sicht sind Autor-Markierungen unnötig; wenn Sie herrausfinden wollen, wer einen bestimmten Abschnitt im Code geschrieben hat, können Sie das Versionsverwaltungssystem dazu konsultieren. Diese Markierungen neigen auch dazu, zu veralten. Wollen Sie wirklich privat zu einem Stück Code kontaktiert werden, welches Sie vor fünf Jahren geschrieben haben und froh sind, es vergessen zu haben?*

Die Quellcode-Dateien eines Software-Projekts sind der Kern seiner Identität; sie sollten die Tatsache wiederspiegeln, dass die Entwicklergemeinschaft im Ganzen dafür verantwortlich ist, und nicht in einzelne Machtbereiche aufgeteilt werden.

Manchmal sprechen sich Leute für die Markierung von Autor oder Zuständigen in Quelldateien aus, mit der Begründung, dass es eine sichtbare Anerkennung derjenigen sei, die am meisten Arbeit geleistet haben. Es gibt mit der Argumentation zwei Probleme. Erstens ergibt sich dadurch die Frage, wieviel Arbeit man leisten muss, um seinen Namen dort auch aufgelistet zu bekommen. Zweitens verschmilzt dadurch die Anerkennung und die Zuständigkeit: Arbeit in der Vergangenheit gemacht zu haben, impliziert keinen Besitz des Bereichs, in dem die Arbeit gemacht wurde, es ist aber schwierig wenn nicht sogar unmöglich, solche Andeutungen zu vermeiden, wenn die Namen von Individuen oben in den Quelldateien aufgelistet sind. So oder so kann die Information über Anerkennung schon aus dem Protokoll der Versionsverwaltung sowie aus anderen Kanälen wie den Archiven der Mailinglisten entnommen werden. Es geht also keine Information verloren, wenn man es in den Quelldateien verbietet. [2]

[2] Ein gutes Gegenargument liefert der Mailinglisten-Thread mit dem Titel *"having authors names in .py files"* unter http://groups.google.com/group/sage-devel/browse_thread/thread/e207ce2206f0beee, insbesondere das Posting von William Stein. Der Kerngedanke ist hier, den-

Wenn Ihr Projekt sich entscheidet einzelne Namen nicht in den Quelldateien zu nennen, sollten Sie es nicht übertreiben. Viele Projekte haben zum Beispiel einen `contrib/` Bereich, in dem kleine Werkzeuge und hilfsscripte getan werden, die oft von Personen geschrieben werden, die an sonsten nicht mit dem Projekt im Zusammenhang stehen. Es ist völlig in Ordnung wenn diese Dateien die Namen der Autoren beinhalten, da sie nicht wirklich von dem Projekt als ganzes gepflegt werden. Wenn andererseits, an einem Hilfsmittel von anderen Personen aus dem Projekt angefangen wird gehackt zu werden, könnte es irgendwann angebracht sein, es in einen weniger isolierten Bereich zu bewegen und, angenommen der ursprüngliche Autor hat keine Einwände, den Namen des Autors entfernen, damit der Code wie jede andere von der Gemeinschaft gepflegte Ressource aussieht. Wenn der Autor im Bezug darauf sensibel ist, sind Kompromisslösungen annehmbar, wie zum Beispiel:

```
# indexclean.py: Entferne die alten Daten aus einem Scanley Index.
#
# Ursprünglicher Autor: K. Maru <kobayashi@nocheinweitereremaildienst.com>
# Derzeit gepflegt von: Dem Scanley Projekt <http://www.scanley.org/>
#                       und K. Maru.
#
# ...
```

Es ist aber besser wenn möglich solche Kompromisse zu vermeiden, und die meisten Autoren lassen sich überreden, da sie froh sind, dass ihr Beitrag zu einem engeren Bestandteil des Projekts gemacht wird.

Wichtig ist es, sich daran zu erinnern, dass es bei jedem Projekt eine fließende Grenze zwischen dem Kern und der Peripherie gibt. Die Hauptdateien des Quellcodes der Software sind ganz klar ein Teil des Kerns, und sollten als von der Gemeinschaft gepflegt, angesehen werden. Begleitende Werkzeuge oder Teile einer Dokumentation können andererseits die Arbeit eines Einzelnen sein, der sie eim wesentlichen alleine pflegt, auch wenn die Werke mit dem Projekt zu tun haben, und sogar verteilt werden. Es muss keine für alle Gültige Regel auf jede Datei angewandt werden, so lange das Prinzip aufrecht erhalten wird, nicht zu erlauben, dass von der Gemeinschaft gepflegte Ressourcen zu den Revieren von Individuen werden.

Der Automatisierungsgrad

Versuchen Sie nicht, Menschen Arbeiten verrichten zu lassen, die an ihrer Stelle von Maschinen gemacht werden könnten. Als Grundregel ist die Automatisierung einer wiederkehrenden Aufgabe mindestens zehn Mal so viel Einsatz wert, wie das, was ein Entwickler aufbringen würde, wenn er sie einmal macht. Bei sehr häufigen oder sehr komplexen Aufgaben, kann dieses Verhältnis sogar leicht auf das zwanzigfache oder mehr ansteigen.

Sich als ein "Projektleiter" anzusehen, im gegensatz zu noch ein weiteren Entwickler, mag an dieser Stelle eine nützliche Einstellung sein. Manchmal sind einzelne Entwickler zu sehr in kleinarbeit verwickelt, um das Gesamtbild zu sehen, und zu erkennen, dass jeder eine menge Mühe daran vergeudet, automatisierbare Aufgaben händisch zu erledigen. Selbst diejenigen, die es erkennen, werden sich nicht unbedingt die Zeit nehmen das Problem zu lösen: Weil jede einzelne Durchführung der Aufgabe sich nicht wie eine riesige Last anfühlt, keiner wird je derart davon genervt, um etwas dagegen zu machen. Was die Automatisierung so wünschenswert macht ist, dass die kleine Last mit der Anzahl der wiederholungen von jedem Entwickler multipliziert wird, und *diese* Zahl wird wiederum mit der Anzahl der Entwickler multipliziert.

ke ich, dass viele der Autoren einer Kultur (dem Akademisch-mathematischen Kreis) entstammen, in der die direkt in den Quellen angesiedelte Anrechnung der Autorschaft als normgerecht und wichtig empfunden wird. Unter solchen Umständen kann es besser sein, Autorennamen in die Quelldateien aufzunehmen und dazu genau auszuführen, was welcher Autor beigetragen hat, weil die Mehrheit der Beteiligten diesen Stil der Anerkennung erwarten wird.

Ich benutze hier den Begriff "Automatisiertung" im weiten Sinne, nicht nur um wiederholte vorgänge zu bezeichnen, bei denen ein oder zwei Variablen sich jedes mal ändern, sondern für jede Art von technischer Infrastruktur die Menschen unterstützt. Das mindeste an standardmäßiger Automatisierung die heutzutage erforderlich ist um ein Projekt zu betreiben, wird in Kapitel 3, *Technische Infrastruktur* beschrieben, aber jedes Projekt kann auch seine eigenen besonderen Probleme haben. Eine Gruppe die an der Dokumentation arbeitet, könnte vielleicht eine Webseite haben wollen, welche zu jeder Zeit die aktuellsten Versionen der Dokumente anzeigt. Da die Dokumentation oftmals in einer Auszeichnungssprache wie XML geschrieben, kann es einen Kompilierungsschritt, meistens ziemlich umständlich, geben um die Dokumente für die Darstellung oder zum herunterladen zu erstellen. Eine Webseite einzurichten, bei dem diese Kompilierung automatisch nach jedem Commit passiert, kann sehr kompliziert und zeitaufwendig sein – es ist es aber wert, selbst wenn Sie mehr als einen Tag brauchen, um es einzurichten. Die übergreifenden Vorteile, Seiten zu haben, die immer aktuell sind, ist enorm, auch wenn die Kosten sie *nicht* zu haben zu einem Zeitpunkt vielleicht nur nach einer kleinen Unbequemlichkeit, für einen einzelnen Entwickler aussieht.

Solche Schritte vorzunehmen, eliminiert nicht nur die verschwendete Zeit, sondern auch die Fesseln und Frustration die entsteht, wenn Menschen Fehler machen (wie es zwangsläufig der Fall sein wird), wenn sie versuchen komplizierte Vorgänge händisch durchzuführen. Dederministische Tätigkeiten, mit mehreren Schritten sind genau das, wofür Computer erfunden wurden; sparen Sie sich ihre Menschen für interesanntere Sachen auf.

Automatisiertes Testen

Automatisierte Testläufe sind für jedes Softwareprojekt hilfreich, insbesondere aber für Open-Source-Projekte, denn automatische Tests (insbesondere Regressionstests) erlauben es Entwicklern sich bei der Änderung von ihnen unbekanntem Code sicher zu fühlen, und ermutigt daruch experimentelle Entwicklung. Da es so schwierig ist, eine Bruchstelle händich zu finden – man muss im wesentlichen raten wo man etwas kapput gemacht haben könnte, und verschiedene Versuche machen, um das Gegenteil zu beweisen – erspart es dem Projekt eine *Menge* Zeit, möglichkeiten zu haben, um solche Bruchstellen automatisch zu erkennen. Leute sind dadurch auch viel entspannter, wenn sie große Schwaden von Code neu gestalten, und trägt auf lange Sicht entsprechend zu der Wartbarkeit der Software bei.

> **Regressionstests**
>
> *Regressionstest* (en. regression test) bedeutet, nach dem wiederauftauchen bereits behobener Fehler zu suchen. Der Sinn von Regressionstests ist, die Wahrscheinlichkeit zu verinngern, dass Änderungen am Code die Software auf unerwartete weise kaputt machen wird. Wenn ein Softwareprojekt anfängt größer und komplizierter zu werden, nimmt die Wahrscheinlichkeit für solche unerwarteten Nebenwirkungen allmählich. Ein guter Aufbau kann die Geschwindigkeit mit der sie zunimmt veringern, kann das Problem aber nicht komplett beseitigen.
>
> Viele Projekte haben aufgrund dessen, eine *Testsreihe* (en. test suite), eine separate Anwendung welche die Software des Projekts auf Arten aufruft, von denen man weiß, dass Sie in der Vergangenheit bestimmte Fehler aufgezeigt haben. Wenn die Testreihe es schafft einer dieser Fehler wieter auftauchen zu lassen, wird das als *Regression* bezeichnet, was soviel heißt wie, dass die Änderung von jemand einen vorher behobenen Bug wieder unbehoben gemacht hat.
>
> Siehe auch http://en.wikipedia.org/wiki/Regression_testing.

Regressionstests sind kein Allheilmittel. Zum einen, funktioniert es am besten, bei Anwendungen mit Schnittstellen für die Kommandozeile. Software die vor allem durch graphische Benutzeroberflächen gesteuert wird, ist viel schwieriger programatisch zu betreiben. Ein weiteres Problem ist, dass das Framework der Testreihe oft ziemlich komplex sein kann, mit einer Lernkurve und einer Wartungslast,

für sich ganz alleine. Diese Komplexität zu reduzieren, ist eines der nützlichsten Sachen, die Sie machen können, auch wenn es vielleicht eine wesentliche Menge an Zeit in Anspruch nehmen kann. Je einfacher es ist neue Tests zu der Testreihe hinzuzufügen, desto mehr Entwickler werden es machen, und desto weniger Fehler werden es in die fertige Version schaffen. Jede Mühe die aufgebracht wird, um das Schreiben von Tests einfacher zu machen, wird im Verlaufe des Projekts um ein Vielfaches zurückgezahlt werden.

Viele Projekte haben eine *"Mache den Build nicht Kaputt!"* Regel, mit folgender Bedeutung: Machen Sie keinen Commit, welches verhindert, dass die Software kompiliert oder läuft. Die Person zu sein, welche den Build kaputt gemacht hat, ist meistens eine Quelle leichter Peinlichkeit und Verippung. Projekte mit Reihen von Regressionstests haben oft eine begleitende Regel: Mache keinen Commit, der die Tests nicht besteht. Es ist am einfachsten zu erkennen, wenn ein Test fehlschlägt, wenn die komplette Testreihe automatisch, jede Nacht durchgeführt wird, und die Ergebnisse an die Entwickler-Liste oder einen gesonderten Verteiler für die Testergebnisse geschickt werden; was ein weiteres Beispiel für Automatisierung ist, die sich lohnt.

Die meisten freiwilligen Entwickler sind bereit, die zusätzliche Zeit zu investieren, um Regressionstests zu schreiben, so lange das Testsystem verständlich und einfach im Umgang ist. Änderungen zusammen mit Tests zu committen, wird als das verantwort Verhalten angesehen, und es ist auch eine einfache Gelegenheit für die Zusammenarbeit: Oft teilen sich zwei Entwickler die Arbeit bei der Behebung eines Fehlers auf, indem der eine den Fix schreibt, und der andere den Test. Der letztere Entwickler kann oft sogar mehr Arbeit haben, und da es schon weniger befriedigend ist einen Test zu schreiben als den Fehler tatsächlich zu beheben, ist es zwingend notwendig, dass es nicht schwieriger ist die Testreihe zu bearbeiten als es sein muss.

Manche Projekte gehen sogar noch weiter, und haben die Anforderung, dass *alle* Bugfixes oder neue Funktionen von Tests begleitet werden. Ob das eine gute Idee ist oder nicht, hängt von vielen Faktoren ab: Die Natur der Software, die Zusammenstellung des Entwicklerteams, und wie schwierig es ist, die Tests zu schreiben. Das CVS (http://www.cvshome.org/) Projekt, hat schon seit langem eine solche Regel. Es ist theoretisch eine gute Richtlinie, da CVS eine Software zur Versionsverwaltung und deshalb sehr risikoscheu ist, in anbetracht der Möglichkeit die Daten der Nutzer zu verunstalten oder falsch handzuhaben. In der Praxis ist das Problem, dass die Regressionstest-Reihe von CVS eine einzige riesige Shellscript-Datei ist (witzigerweise mit dem Namen `sanity.sh`), welche schwierig zu bearbeiten oder erweitern ist. Die schwierigkeit neue Tests hinzu zu fügen, zusammen mit der Forderung, dass Patches von neuen Tests begleitet werden bedeutet, dass CVS im wesentlichen vor Patches abeschreckt. Als ich an CVS gearbeitet habe, sahe ich manche Leute die anfingen an dem Code von CVS zu arbeiten und sogar Patches fertig zu stellen, aber aufgaben als ihnen gesagt wurde, dass sie einen neuen Test zu der `sanity.sh` mussten.

Es ist normal etwas mehr Zeit damit zu verbringen, einen neunen Regressionstest zu schreiben, als an der Behebung des ursprünglichen Fehlers. CVS trieb dieses Phänomen aber ins extreme: man konnte Stunden bei dem Versuch verbringen, sein Test richtig zu gestalten, und es trotzdem falsch machen, da es einfach zu viele unvorhersehbare Komplexitäten gab, bei der Änderung einer 35.000 Zeilen lange Bourne-Shellscript-Datei. Selbst langjährige Entwickler murrten, als sie einen neuen Test hinzufügen mussten.

Diese Situation entstand, weil wir alle nicht den Grad der Automatisierung in Betracht gezogen hatten. Es stimmt zwar, dass der wechsel auf ein echtes Test-Framework – ob eine Eigenanfertigung oder eine fertige – eine riesige Anstrengung gewesen wäre [3]. Es zu unterlassen, hat dem Projekt aber über die Jahre, viel mehr gekostet. Wieviele Bugfixes und Funktionen sind heute *nicht* in CVS, aufgrund der Behinderung durch eine umständliche Testreihe? Woe werden die genaue Zahl nie erfahren, es sind aber sicherlich eine mehr, wie die Anzahl der Bugfixes und Änderungen, auf denen die Entwickler hätten

[3]Man sollte bedenken, dass es nicht notwendig gewesen wäre alle Tests auf das neue Framework zu portieren; beide könnten friedlich neben einander leben, und nur die geänderten Tests würden bei Änderungen übernommen werden.

verzichten müssen, um ein neues Testsystem zu entwickeln (oder ein vorher existierendes zu integrieren). Die Aufgabe hätte eine endliche Zeit beansprucht, wärend der Nachteil, das derzeitige Testsystem weiter zu benutzen, in alle Ewigkeit weiterbestanden hätte, wenn nichts gemacht würde.

Der Punkt ist nicht, dass die strikte Anforderungen, Tests schreiben zu müssen, schlecht ist, oder das es schlecht ist, wenn all Ihre Tests in eine Bourne-Shellscript-Datei geschrieben werden. Es kann hervorragend funktionieren, je nachdem wie Sie es entwerfen und was es testen muss. Das Wesentliche ist einfach, dass, wenn das Testsystem zu einer wesentlichen Behinderung für die Entwicklung wird, etwas getan werden muss. Das gleiche gilt für jede wiederkehrende Aufgabe, welche sich zu einem Hindernis oder Flaschenhals entwickelt.

Behandeln Sie jeden Nutzer wie einen möglichen Freiwilligen

Jede Interaktion mit einem Benutzer ist eine Gelegenheit, einen neuen Freiwilligen zu bekommen. Wenn ein Nutzer sich die Zeit nimmt, auf einer der Mailinglisten des Projekts, eine Nachricht zu schreiben, oder einen Fehler zu melden, hat er sich bereits als jemand gekennzeichnet, der ein größeres Potential zur Beteiligung hat, als die meisten Nutzer (von denen das Projekt niemals hören wird). Greifen Sie dieses Potential auf: Wenn er einen Fehler beschreibt, danken Sie ihn für die Meldung und fragen Sie ihm, ob er es versuchen möchte es zu beheben. Wenn geschrieben hat, um zu sagen, dass eine wichtige Frage in der FAQ gefehlt hat, oder dass die Dokumentation der Anwendung auf irgend eine Art unzureichend war, dann geben Sie das Problem zu (angenommen, es existiert wirklich) und fragen Sie ihr ob er interesiert wäre das fehlenden Material selber zu schreiben. Der Nutzer wird natürlich die meiste Zeit ablehnen. Aber es kostet nicht viel zu fragen, und jedes mal, wenn sie das tun, erinnert es die anderen Zuhörer in dem Forum, dass eine Beteiligung an dem Projekt etwas ist, das jeder leisten kann.

Schränken Sie Ihre Ziele nicht darauf ein, neue Entwickler und Autoren für die Dokumentation zu bekommen. Es zahlt sich zum Beispiel auf lange sicht sogar aus, Leute zu trainieren, die gute Bug-Meldungen schreiben, so lange Sie nicht *zu* viel Zeit mit jeder einzelnen Person verbringen, und sie in Zukunft weiterhin Bug-Meldungen schreiben – was eher wahrscheinlich ist, wenn sie bei der ersten Meldung eine konstruktive Reaktion bekommen haben. Eine konstruktive Reaktion muss nicht die Behebung eines Fehlers sein, auch wenn dass immer ideal ist; es kann auch eine Anfrage nach weiteren Informationen sein, oder gar nur eine Bestätigung, dass das Verhalten *tatsächlich* ein Fehler ist. Menschen wollen zugehört werden. Zweitrangig, wollen sie das ihre Fehler behoben werden. Sie werden möglicherweise nicht immer letzteres zeitnahe geben können, aber Sie (oder vielmehr das Projekt, als ganzes) kann ihnen ersteres geben.

Eine Konsequenz hiervon ist, dass Entwickler keine Wut an Personen auslassen sollten, die gut gemeinte aber vage Bug-Meldungen gemacht haben. Sowas ärgert mich persönlich, ganz besonders; ich sehe Entwickler die es immer wieder auf verschiedenen Open-Source-Mailinglisten machen und der Schaden, den das anrichtet, ist greifbar. Irgend ein unglückseliger Neuling schreibt ein nutzloses Ticket:

> Hallo, ich krieg Scanley nicht zum Laufen. Jedes mal wenn ich es starte, gibt es einfach einen Fehler. Hat jemand schon dieses Problem gehabt?

Ein Entwickler – der diese Art von Ticket schon tausendmal gesehen hat, und kurz bedenkt, dass das beim Neuling nicht der Fall ist – wird folgendermaßen antworten:

> Was sollen wir eigentlich mit so wenig Informationen machen, meine Fresse! Gib uns wenigstens ein paar Details, die Version von Scanley, dein Betriebssystem, und die Fehlermeldung!

Dieser Entwickler hat es nicht geschafft, die Sache aus der Sicht des Nutzers zu sehen, und auch zu bedenken, welche Auswirkung eine solche Reaktion auf alle *anderen* Personen die dem Wechsel

zuschauen, haben könnte. Ein Nutzer der keine Programmiererfahrung hat, und keine vorhergehende Erfahrung mit Fehlermeldungen, wird nicht wissen, wie man ein Ticket schreibt. Wie behandelt man so eine Person richtig? Schulen Sie sie! Und tun Sie es in einer Weise, dass sie wiederkommt:

> Tut mir leid, dass du Probleme hast. Wir werden mehr Informationen brauchen, um herauszufinden, was da los ist. Teile uns bitte deine Version von Scanley, dein Betriebssystem, und den genauen Text der Fehlermeldung mit. Das größte Hilfe wäre eine Auflistung der Befehle, die du ausgeführt hast, und die daraus resultierende Ausgabe. Siehe http://www.scanley.org/wie_man_einen_bug_meldet.html für weiteres.

Diese Art zu reagieren ist sehr viel effektiver, die erforderliche Information aus dem Benutzer herauszubekommen, da der Text aus Benutzersicht geschrieben ist. Erstens drückt das Sympathie aus: *Du hattest ein Problem; wir leiden mit dir* . (Das ist nicht in jeder Reaktion auf eine Bug-Meldung nötig; es hängt davon ab, wie schwerwiegend das Problem ist, und wie verärgert er zu sein scheint.) Zweitens, teilt es ihm mit wie er ein neues Ticket schreiben soll, damit es genügend Details enthält, um nützlich zu sein, anstatt ihn dafür herabzusetzen, dass er es nicht weiß – zum Beispiel erkennen viele Nutzer nicht das "Zeige uns den Fehler" in Wirklichkeit "Zeige uns den genauen Text des Fehlers, ohne etwas auszulassen oder zu kürzen" bedeutet. Wenn Sie zum ersten Mal mit einem solchen Nutzer arbeiten, müssen Sie im Bezug darauf explizit sein. Am Ende geben Sie einen Hinweis auf eine viel detalliertere und vollständige Anleitung, wie man Fehler melden sollte. Wenn Sie erfolgreich mit dem Nutzer Kontakt aufgenommen haben, wird er sich die Zeit nehmen, dieses Dokument zu lesen und sich danach zu richten. Das bedeutet natürlich, dass Sie das Dockument im voraus bereithalten müssen. Es sollte klare Anweisungen geben, welche Art von Information Ihr Entwicklerteam in jedem Ticket sehen möchte. Im Idealfall sollte es sich mit der Zeit auch entwickeln, entsprechend der bestimmten Art von fehlenden Informationen und Falschmeldungen; die Nutzer Ihres Projekts typischerweise an Sie richten.

Die für das Subversion-Projekt erarbeitete Anleitung zum Melden von Fehlern ist so ziemlich ein Standardbeispiel dieser Form (siehe Anhang D, *Beispiel-Anleitung für das Melden von Fehlern*). Beachten Sie, wie diese Anleitung in einer Einladung zum Einsenden eines Patches für den Bug endet. Das geschieht nicht, weil solch eine Einladung zu einem besseren Verhältnis von Patches zu Bugs führen würde – die meisten Nutzer die in der Lage sind, Fehler zu beheben, wissen bereits, dass ein Patch begrüßt wird, und müssen das nicht gesagt bekommen. Der eigentliche Sinn dieser Einladung ist, für alle Leser (insbesondere solche, die nicht vertraut mit dem Projekt oder freier Software im allgemeinen sind) zu betonen, dass das Projekt von freiwilligen Beiträgen lebt. In gewissem Sinne sind die derzeitigen Entwickler des Projekts nicht stärker dafür verantwortlich, den Bug zu beheben, als die Person, die ihn gemeldet hat. Das ist ein wichtiger Punkt, mit dem viele neue Nutzer nicht vertraut sein werden. Wenn sie es einmal begreifen, werden sie bereitwilliger dabei helfen, den Fix zu erstellen, wenn auch nicht, indem sie Code beitragen, sondern indem sie eine Anleitung zur Reproduktion des Fehlers geben, oder indem sie anbieten, die von anderen eingereichten Patches zu testen. Das Ziel ist, jeden Nutzer klar zu machen, dass es keinen *immanenten* Unterschied zwischen ihm und den Personen gibt, die an dem Projekt arbeiten – es ist eine Frage der Zeit und der Mühe, die man hinein steckt, nicht wer man ist.

Die Mahnung sich vor wütenden Antworten zu hüten, gilt nicht gegenüber unhöflichen Nutzern. Ab und zu werden Fehler gemeldet oder Beschwerden eingereicht, die unabhängig von ihrem Informationsgehalt eine spöttische Verachtung für das Projekt beinhalten, aufgrund irgend eines Mangels. Solche Personen sind abwechselnd beleidigend und schmeichelnd, wie beispielsweise diese Person, die an die Subversion-Liste schrieb:

> Wie kommt es, dass nach fast sechs Tagen immer noch keine Binärdateien für Windows hochgeladen wurden?!? Es ist immer wieder das gleiche und es ist ziemlich frustrierend. Warum werden diese Dinge nicht automatisiert, damit sie sofort verfügbar sind?? Wenn ihr einen "RC" Build macht, denke ich eure Idee ist, dass ihr wollt, dass Nutzer den Build testen, trotzdem gebt ihr aber keine Möglichkeit das zu machen. Wazu überhaupt eine Vorlaufphase haben, wenn ihr keine Möglichkeit zum Testen gebt??

Die anfängliche Reaktion zu dieser relativ flamehaften Nachticht war überraschend zurückhaltend: Man wies darauf hin, dass das Projekt eine Richtlinie hatte, keine offiziellen Binärdateien zu veröffentlichen, und man sagte ihm (mit unterschiedlichem Grad an Genervtheit), dass er sich freiwillig melden sollte, sie zu erstellen, wenn sie ihm so wichtig wären. Ob Sie es glauben oder nicht, seine nächste Nachricht fing mit diesen Zeilen an:

> Als allererstes will ich sagen, dass ich Subversion Klasse finde und die Anstrenungen, die damit verbunden sind wirklich zu schätzen weiß. [...]

...und ging dann *wieder* daran, das Projekt dafür zu beschimpfen, dass es keine Binärdateien bereitstellte, wärend er sich immer noch nicht freiwillig meldete, irgend etwas dagegen zu machen. Danach wurde er von ca. 50 leuten auseinandergenommen und ich kann nicht sagen, dass mir das wirklich etwas ausmachte. Die "Null-Toleranz" Richtlinie in Bezug auf Beleidigungen, die in „Unhöflichkeit im Keim ersticken“ im Kapitel Kapitel 2, *Der Einstieg* verfochten wurde, gilt für Personen, die mit dem das Projekt eine fortbestehende Interaktion haben (oder haben möchten). Wenn jemand aber von Anfang an klar macht, dass er eine Quelle beständiger Ärgernisses sein wird, hat es keinen Sinn, ihn willkommen zu heißen.

Solche Situationen sind zum Glück relativ selten, und sie sind wesentlich seltener in Projekten, die sich bereits beim ersten Kontakt die Mühe machen, sich mit Nutzern auf eine konstruktive und zuvorkommende Art zu befassen.

Teilen sie sowohl Verwaltungsaufgaben als auch technische Aufgaben

Teilen Sie sowohl die Bürde der Verwaltung als auch die technische Bürde das Projekt in Gang zu halten. Mit zunehmender Komplexität eines Projekts ist mehr und mehr Arbeit in Management und Informationsfluss erforderlich. Es gibt keinen Grund diese Bürde nicht zu teilen; wobei, sie zu teilen, nicht notwendigerweise eine Hierarchie erfordert – was sich in der Praxis entwickelt, entspricht eher einer Peer-to-Peer-Topologie als einer militärichen Befehlsstruktur.

Manchmal sind die Rollen der Verwaltung formalisiert, und manchmal bilden sie sich spontan. Im Subversion-Projekt haben wir einen Patchesverwalter, einen Übersetzungsverwalter, einen Dokumentationsverwalter, einen Ticketverwalter (wenn auch inoffiziell), und einen Versionsverwalter. Manche dieser Rollen wurden bewusst von uns eingeführt, andere entstanden von allein; da das Projekt wächst, erwarte ich, dass weitere Rollen hinzukommen. Unten werden wir diese und ein paar andere Rollen im Detail untersuchen (außer der des Versionsverwalters, die bereits in „Release-Verwalter“ und „Diktatur durch den Versionsherrn“ im vorhergehenden Kapitel besprochen wurde).

Achten Sie einmal beim Lesen der Beschreibungen darauf, wie keine der Rollen die exklusive Kontrolle über den entsprechenden Bereich erfordert. So hindert der Ticketverwalter andere nicht daran, Änderungen an der Ticket-Datenbank vorzunehmen, der FAQ-Verwalter besteht nicht darauf, der einzige zu sein, der die FAQ bearbeitet, und so weiter. Bei diesen Rollen geht es durchweg um Verantwortung, nicht um eine Monopolstellung. Ein wichtiger Teil der Arbeit des Verwalters eines Bereichs ist es, zu bemerken wenn andere in diesem Bereich arbeiten, und sie anzuleiten, die Dinge ganauso zu erledigen wie der Verwalter, damit die nebeneinander laufenden Anstrengungen einander stärken, anstatt miteinander in Konflikt zu geraten. Der Verwalter eines Bereichs sollte die Abläufe dokumentieren, nach denen er die Arbeit erledigt, damit, sollte einmal gehen, jemand anders die Rolle übernehmen kann.

Manchmal gibt es einen Konflikt: Zwei oder mehr Personen erheben Anspruch auf dieselbe Rolle. Hier gibt es kein Patentrezept. Sie könnten anregen, dass jeder Freiwillige einen Vorschlag (eine "Bewerbung") einreicht, und alle Commit-Berechtigten darüber abstimmen lassen, welche die beste ist. Das ist

aber umständlich und unter Umständen ungeschickt. Ich halte es für besser, die Kandidaten einfach darum zu bitten, das unter sich auszumachen. Meist schaffen sie das auch, und sie werden mit dem Ergebnis zufriedener sein als mit einer von außen aufgezwungenen Entscheidung.

Patchverwalter

In einem freien Software-Projekt, dem eine Menge Patches zugehen, kann es ein Albtraum sein, den Überblick darüber zu behalten, welche Patches angekommen sind und wie über sie entschieden wurde, ganz besonders wenn es auf eine dezentralisierte Art geschieht. Die meisten Patches erreichen das Projekt über die Entwickler-Mailingliste (obwohl manche zuerst in dem Ticket System auftauchen, oder auf externen Webseiten), und es gibt unterschiedliche Routen, die ein Patch nach seiner Ankunft einschlagen kann.

Manchmal schaut sich jemand einen Patch an, findet Fehler darin und schickt ihn zur Berichtigung an seinen Autor zurück. Das führt normalerweise zu einem sich wiederholenden Vorgang – für alle auf der Mailingliste sichtbar –, bei dem der ursprüngliche Autor die überarbeiteten Versionen des Patches wieder zurück schickt, bis der Überprüfende nichts mehr zu beanstanden hat. Es ist nicht immer leicht zu erkennen, wann dieser Vorgang abgeschlossen ist: Wenn der Überprüfende den Patch committet, dann ist er eindeutig vorbei. Tut er es aber nicht, kann es auch daran liegen, dass er einfach nicht genügend Zeit hatte, oder selbst über keine Commit-Berechtigung verfügt, und keinen anderen Entwickler dazu überreden konnte, es zu tun.

Eine weitere häufige Antwort auf einen Patch ist eine freilaufende Diskussion, nicht unbedingt über den Patch selbst, sondern darüber, ob das ihm zu Grunde liegende Konzept gut ist. Zum Beispiel könnte Der Patch einen Fehler beheben, aber das Projekt bevorzugt es, den Fehler auf eine andere Art zu beheben: als Teil der Lösung einer allgemeineren Klasse von Problemen. Oft ist das nicht im Voraus bekannt, und es ist der Patch, der diese Entdeckung anregt.

Gelegentlich geht ein Patch unter völligem Stillschweigen ein. Normalerweise liegt das daran, dass *im Moment* kein Entwickler die Zeit aufbringt, den Patch unter die Lupe zu nemen, jader hofft, dass jemand anders das übernimmt. Da es keine bestimmte Grenze gibt, wie lang jede Person wartet, bis jemand anders den Ball ins Rollen bringt und in der Zwischenzeit immer andere Prioritäten auftauchen, passiert es sehr leicht, dass ein Patch durchs Netz fällt, ohne dass irgend jemand das gewollt hätte. So kann das Projekt einen nützlichen Patch verpassen, und das hat weitere schädliche Nebeneffekte: Es ist entmutigend für denjenigen, der Arbeit in den Patch investiert hat, und es lässt das Projekt als Ganzes den Anschein der Verwahrlosung erwecken, besonders für andere die im Begriff sind, Patches einzureichen.

Die Aufgabe des Patchverwalters ist sicherzustellen, dass Patches nicht "durchs Netz schlüpfen". Das wird erreicht, indem jeder Patch bis zu irgend einem stabilen Zustand verfolgt wird. Der Patchverwalter beobachtet jeden Thread auf der Mailingliste, der sich aus dem Einreichen eines Patches ergibt. Wenn dieser mit einem Commit des Patches endet, tut er gar nichts. Wenn er in eine Schleife von Überprüfung und Überarbeitung übergeht, welche in einer finalen Version, aber ohne Commit endet, schreibt er ein Ticket, das auf die finale Version verweist, sowie auf den entsprechenden Thread, damit es für Entwickler, die daran später anschließen wollen, eine permanente Aufzeichnung gibt. Wenn der Patch ein bereits bestehendes Ticket betrifft, fügt er dem Ticket einen Kommentar mit der relevanten Information hinzu, anstatt ein neues zu eröffnen.

Wenn ein Patch überhaupt keine Reaktion erhält, wartet der Patchverwalter ein paar Tage, uns greift es dann auf, indem er fragt, ob irgend jemand es sich anschauen wird. Darauf gibt es für gewöhnlich eine Reaktion: Ein Entwickler kann erklären, dass er nicht denkt, dass der Patch angewendet werden sollte, und Gründe dafür angeben, oder er kann es sich anschauen, worauf einer der vorher beschriebenen Wege eingeschlagen wird. Wenn es immer noch keine Antwort gibt, wird der Patchverwalter unter Umständen ein Ticket für den Patch eröffnen oder nicht, je nachdem, wie er es für richtig hält, zumindest hat der ursprüngliche Autor *irgendeine* Antwort erhalten.

Einen Patchverwalter zu haben, hat für das Entwicklerteam von Subversion eine Menge Zeit und geistige Energie gespart. Ohne eine designierte Person, die die Verantwortung auf sich nimmt, müsste sich jeder Entwickler die ganze Zeit ständig darüber Sorgen machen, "Wenn ich nicht die Zeit dazu habe, jetzt auf diesen Patch zu antworten, kann ich mich darauf verlassen, dass jemand anders es macht? Sollte ich versuchen ihn im Auge zu behalten? Aber wenn andere das nun auch tun, dann würden wir unnötig unsere Anstrengungen verdoppeln." Der Patchverwalter lässt die zweite Mutmaßung in dieser Situation verschwinden. Jeder Entwickler kann die Entscheidung treffen, die für ihn in dem Moment richtig ist, da er den Patch zum ersten mal sieht. Wenn eine Überprüfung anschließen möchte, kann er das tun – der Patchverwalter wird sein Verhalten entsprechend anpassen. Wenn er den Patch komplett ignorieren will, ist das auch in Ordnung; der Patchverwalter wird dafür sorgen, dass er nicht in Vergessenheit gerät.

Da dieses System nur funktioniert, wenn man sich darauf verlassen kann, dass der Patchverwalter ausnahmslos da ist, sollte die Rolle formal vergeben werden. In Subversion schrieben wir diese Stelle auf der Entwickler-Mailingliste und der für Benutzer aus, bekamen mehrere Freiwillige, und nahmen den ersten der geantwortet hatte. Als die Person abtreten musste (siehe „Übergänge“ später in diesem Kapitel), machten wird das gleiche nochmal. Wir haben nie versucht, mehrere Personen die Rolle teilen zu lassen, da ein Übermaß an Kommunikation zwischen ihnen nötig gewesen wäre; aber bei einer sehr hohen Anzahl von Patches könnte ein mehrköpfiger Patchverwalter vielleicht sinnvoll sein.

Übersetzungsverwalter

Bei Software-Projekten, kann sich "Übersetzung" auf zwei sehr unterschiedliche Dinge beziehen. Es kann die Übersetzung der Dokumentation in andere Sprachen bedeuten, oder es kann die Übersetzung der Software selber bedeuten – womit gemeint ist, dass die Anwendung seine Fehlermeldungen und Texte in der bevorzugten Sprache des Anwenders darstellt. Beides sind komplexe Aufgaben, wenn aber die richtige Infrastruktur erst einmal eingerichtet ist, kann mal sie größtenteils von der anderen Entwicklung trennen. Weil die Aufgaben sind sich in mancherlei Hinsicht ähnlich, kann es Sinn machen (abhängig von Ihrem Projekt) einen einzigen Verwalter für Übersetzungen zu haben um beides zu handhaben, oder es kann besser sein zwei verschiedene Verwalter zu haben.

Im Subversion-Projekt, haben wir einen Verwalter, der sich um beides kümmert. Er schreibt natürlich nicht selber die Übersetzungen – er wird vielleicht bei einer oder zweien aushelfen, aber zum Zeitpunkt dieser Niederschrift, hätte er zehn Sprachen sprechen müssen (zwölf wenn man Dialekte mitzählt) um bei allen mitzuarbeiten! Stattdessen verwaltet er die Teams der freiwilligen Übersetzer: Er hilft ihnen dabei, sich zu koordinieren, und koordiniert zwischen ihnen und dem Rest des Projekts.

Dass ein Verwalter für Übersetzungen nötig ist, liegt zum Teil daran, dass Übersetzer eine sehr andere Menschenart sind, als Entwickler. Sie haben manchmal wenig oder gar keine Erfahrung bei der Arbeit mit einer Versionsverwaltung, oder gar überhapt als Teil eines verteilten Teams von Freiwilligen. In anderer Hinsicht, sind sie aber oft die beste Art von Freiwilligen: Menschen mit Kenntnissen auf einem bestimmten Gebiet, die einen Bedarf sahen, und entschieden sich zu beteiligen. Sie sind meistens bereit zu lernen, und enthusiastisch mit der Arbeit anzufangen. Alles was sie brauchen, ist jemand der ihnen sagt wie. Der Übersetzungsverwalter stellt sicher, dass die Übersetzungen auf eine Art ablaufen, welches die gewöhnlichen Entwicklung nicht behindert. Er dient auch als eine Art representativer der Übersetzer, als geeinigten Körper, wann immer die Entwickler über technische Änderungen informiert werden müssen, die nötig sind, um die Übersetzungsarbeit zu unterstützen.

Die wichtigsten Fähigkeiten dieser Position sind desshalb diplomatische und nicht technische. In Subversion haben wir zum Beispiel eine Richtlinie, nach dem alle Übersetzungen, mindestens zwei Personen haben sollte, die an ihnen arbeiten, weil es ansonsten keine Möglichkeit gibt, dass die Texte überprüft werden. Wenn ein neuer Freiwilliger auftaucht, und anbietet Subversion z.B. in Malagasy zu übersetzen, muss der Übersetzungsverwalter ihn entweder mit jemand zusammenbringen der sechs Monate zuvor geschrieben hat, und auch Interesse bekundigt hat eine Übersetzung in Malagasy zu machen, oder ihn höflich darum bitten, einen *weiteren* Übersetzer für Malagasy zu finden, der mit ihm als Partner

arbeitet. Wenn genügend Leute verfügbar sind, gibt der Übersetzungsverwalter ihnen die nötigen Commit-Rechte, informiert ihnen über die Konventionen des Projekts (wie man etwa Commit-Kommentare schreibt), und hält dann danach ausschau, dass sie sich an diese Konventionen halten.

Unterhaltungen zwischen dem Übersetzungsverwalter und den Entwicklern, oder zwischen dem Übersetzungsverwalter und die Übersetzerteam, werden meistens in der ursprünglichen Sprache des Projekts gehalten – also die Sprache von dem alle Übersetzungen gemacht werden. Für die meisten freien Software-Projekte ist das Englisch, es macht aber keinen Unterschied, so lange das Projek sich darauf einigt. (Obwohl Englisch wahrscheinlich am besten ist, für Projekte die eine breite internationale Gemmeinschaft anlocken wollen.)

Unterhaltungen *innerhalb* eines bestimmten Übersetzungsteams sind jedoch meistens in der gemeinsamen Sprache, und eine der anderen Aufgaben des Übersetzungsverwalter ist es, eine gesonderte Mailingliste für jedes Team einzurichten. So können sich die Übersetzer frei über ihre Arbeit unterhalten, ohne andere auf den Hauptverteilern des Projekts abzulenken, von denen die meisten eh nicht in der Lage wären, die übersetzte Sprache zu verstehen.

Internationalisierung kontra Lokalisierung

Internationalisierung (*I18N*) und *lokalisierung* (*L10N*) beziehen sich beide auf den Arbeitsablauf, eine Anwendung anzupassen, damit sie in anderen linguistischen und kulturellen Umgebungen funktioniert, als der es ursprünglich geschrieben wurde. Die Begriffe werden oft als unter einander austauschbar behandelt, sind aber in wirklichkeit nicht ganz das selbe. Wie http://en.wikipedia.org/wiki/G11n schreibt:

> Die Unterscheidung zwischen beiden, ist subtil aber wichtig: Internationnalisierung ist die Anpassung von Produkten damit sie *potentiell* überall genutzt werden können, wärend die lokalisierung das Hinzufügen spezifischer Funktionen für die Nutzung in einem *bestimmten* Gebiet bezeichnent.

Ihre Software zum umzustellen, damit es verlustfrei Unicode (http://en.wikipedia.org/wiki/Unicode) Textkodierung handhaben kann, fällt zum Beispiel under die Internationnalisierung, da es nicht um eine bestimmte Sprache geht, sondern darum Text von einer Vielzahl an Sprachen annehmen zu können. Ihre Software dazu zu bringen, alle Fehlermeldungen auf Slovenische auszugeben, wenn es erkennt, dass es unter einer slovenischen Umgebung läuft, fällte andererseits under die Lokalisierung.

Die Aufgabe des Übersetzungsverwalters ist desshalb prinzipiell die der Lokalisierung, nicht der Internationalisierung.

Dokumentationsverwalter

Die Dokumentation der Software auf den neusten Stand zu halten, ist eine niemals endende Aufgabe. Jede neue Funktion oder verbesserung die in den Code geht, hat das Potential, eine Änderung an der Dokumentation zu verursachen. Wenn die Dokumentation des Projekts erst einmal einen gewissen Grad der Fertigstellung erreicht hat, werden Sie auch sehen, dass die Patches die von Leuten eingereicht werden, für die Dokumentation sind, nicht für den Code. Das liegt daran, dass es viel mehr Leute gibt, die in der Lage sind Fehler in Prosa zu beheben, als in Code: Alle Nutzer sind Leser, aber nur wenige sind Programmierer.

Patches für die Dokumentation sind meistens viel einfacher zu überprüfen und anzuwenden, als Patches für Code. Es gibt wenig oder nichts zum Testen, und die Qualität der Änderung kann schnell evaluiert werden, alleine indem man ihn durchliest. Da die Menge groß ist, aber die Bürde sie zu überprüfen relativ gering ist, ist das Verhältnis des überschüssigen administrativen Aufwands zu produktiver Arbeit größer als für Patches an dem Code. Desweiteren, werden die meisten Patches wahrscheinlich irgend

eine Art Anpassung erfordern, um einen beständigen Ton in der Stimme des Autors in der Dokumentation zu bewahren. In vielen Fällen, werden sich Patches überschneiden, oder andere Patches beeinflussen, und auf einander abgestimmt werden müssen vor sie committed werden.

Angesichts der Anforderungen an die Handhabung der Dokumentation, und der Tatsache, dass der Codebestand ständig überwacht werden muss damit die Dokumentation aktuell gehalten werden kann, macht es Sinn, eine Person oder eine kleine Teams zu haben, die sich der Aufgabe widmet. Sie können ein Protokoll darüber führen, wo genau und wie die Dokumentation hinter der Software herhikt, und sie können geübte Abläufe haben um große Mengen an Patches auf eine gegliederte Art zu verarbeiten.

Das hindert natürlich andere in dem Projekt nicht daran, Patches mal nebenbei auf die Dokumentation anzuwenden, ganz besonders kleine, wie es die Zeit erlaubt. Und der gleiche Patchverwalter (siehe „Patchverwalter“ früher in diesem Kapitel) kann sowohl Änderungen an dem Code, als auch an der Dokumentation verfolgen, und sie überall ablegen, wo die Entwickler- und Dokumentationsteams sie jeweils haben möchten. (Wenn die gesamte Menge der Patches das übersteigt, was von einem Menschen bewältigt werden kann, ist es wahrscheinlich ein erster Schritt, wenn man auf separate Patchverwalter für Code und Dokumentation wechselt.) Der Sinn eines Dokumentationsteams ist es nicht, Leute zu haben, die sich dafür verantwortlich fühlen, die Dokumentation organisiert, auf dem aktuellsten Stand, und in sich konsistent zu halten. In der Praxis, bedeutet das, die Dokumentation intim zu kennen, den Codebestand zu beobachten, die Änerungen die *andere* an die Dokumentation machen zu beobachten, nach eintreffenden Patches für die Dokumentation ausschau zu halten, und all diese Informationensquellen zu nutzen, um das zu tun, was nötig ist, um die Dokumentation in einem gesungen Zustand zu halten.

Ticketverwalter

Die Anzahl der Tickets im Bugtracker des Projekts nimmt mit der Anzahl der Personen zu, welche die Software benutzen. Deshalb sollten Sie immer noch erwarten, dass die Anzahl offener Tickets, im wesentlichen ohne Grenzen anwächst, selbst während sie Fehler beheben und eine immer robustere Anwendung produzieren. Die Menge der doppelt eingetragenen Tickets wird genau so wie die der unvollständigen und dürftig beschriebenen Tickets zunehmen.

Ticketverwalter helfen dabei, diese Probleme zu mildern, indem sie in die Datenbank gehen, und periodisch alles durchkämmen, auf der Suche nach bestimmten Problemen. Ihre häufigste Aufgabe ist es wahrscheinlich, eintreffende Tickets aufzubessern, entweder weil der Meldende einige der Formularfelder nicht richtig gesetzt hat, oder weil das Ticket ein Duplikat eines bereits in der Dadenbank vorhandenen ist. Offensichtlich wird der Ticketverwalter um so effektiver in der Lage sein, Duplikate zu erkennen, je besser er mit der Bug-Datenbank des Projekts vertraut ist – das ist einer der hauptsächlichen Vorteile ein paar Personen zu haben, die sich auf die Bug-Datenbank spezialisiern, anstatt das alle es versuchen *ad hoc* zu übernehmen. Wenn die Gruppe versucht, das auf eine dezentralisierte Art zu machen, eignent sich nicht ein einzelner ein tieferes Wissen über den Inhalt der Datenbank an.

Ticketverwalter können auch helfen, zwischen den Tickets und den einzelnen Entwicklern zu vermitteln. Wenn eine Menge Bug-Meldungen eintreffen, wird nicht jeder Entwickler die Nachrichten von Verteiler für die Tickets mit der gleichen Aufmerksamkeit lesen. Wenn jemand jedoch weiß, dass das Entwicklerteam ein Auge auf alle eintreffenden Tickets hält, kann sie wenn es angemessen ist, diskret die Aufmerksamkeit bestimmter Entwickler auf spezifische Fehler richten. Das muss natürlich mit einem Gesprür für alles andere was in der Entwicklung abläuft geschehen, und entsprechend den Wünschen und dem Temprament des Empfängers. Es ist desshalb oft besser, wenn Ticketverwalter selber Entwickler sind.

Abhängig davon wie Ihr Projekt den Bugtracker benutzt, können Ticketverwalter auch die Datenbank anpassen, um die Prioritäten des Projekts wider zu spiegeln. Wir in Subversion zum Beispiel, planen wir einen Ticket für eine bestimmte zukünftige Version ein, damit wenn jemand fragt "Wann wird Bug X behoben sein?" wir sagen können "In zwei Versionen" selbst wenn wir kein genaues Datum nennen

können. Die neuen Versionen kann man als Meilensteine in dem Bugtracker eintragen, ein Feld welches in IssueZilla[4] angeboten wird. Grundsätzlich hat jede neue Version von Subversion eine bedeutende neue Funktion, und eine liste spezifischer Fehler die behoben wurden. Wir weisen die entsprechenden Meilensteine allen Tickets zu, die für die Version geplant sind (inklusive der neuen Funktion – es erhält auch einen Ticket), damit man die Bug-Datenbank mit der geplanten Version im Blick anschauen kann. Diese Ziele bleiben jedoch relativ statisch. Wärend neue Fehler gemeldet werden, verschieben sich manchmal die prioritäten, und ein Ticket muss von einem Meilenstein zum anderen verschoben werden, damit jede Version noch im Griff zu behalten bleibt. Das wiederum, wird am besten von Personen gemacht, die insgesamt, einen Sinn dafür haben, was in der Datenbank ist, und wie verschiedene Tickets mit einander verwandt sind.

Eine weitere Sache, die Ticketverwalter machen, ist zu erkennen, wann Tickets veraltet sind. Manchmal wird ein Bug aus Versehen behoben, als Teil einer Änderung an der Software die nicht im Zusammenhang steht. Manchmal ändert das Projekt auch seine Meinung darüber, ob ein bestimmtes Verhalten als Fehler einzustufen ist. Veraltete Tickets zu finden, ist nicht leicht: Der einzige Weg es systematisch zu machen, ist indem man über alle Tickets in der Datenbank geht. Mit der Zeit, wärend die Anzahl der Tickets anwächst, wird sowas allerdings immer winiger praktikabel. Ab einer bestimmten Grenze, ist die einzige Möglichkeit die Datenbank in einem vernünftigen Zustand zu behalten, die Teile-und-herrsche-Methode: Kategorisieren Sie die Tickets, sofort nachdem sie eingetragen wurden und lenken Sie die Aufmerksamkeit der zuständigen Entwickler oder des Teams darauf. Der Empfänger kümmert sich ab dann um das Ticket, bis es seine Lebenszeit beendet hat, und hütet es soweit nötig, bis es erledigt ist oder in Vergessenheit gerät. Wenn die Datenbank derart groß ist, wird der Ticketverwalter eher zu einem globalen Koordinator, der immer weniger Zeit damit verbringt, sich jedes einzelne Ticket anzuschauen, und mehr damit, es in die Hände der richtigen Person zu legen.

FAQ-Verwalter

Die Verwaltung der FAQ ist ein überraschend schwieriges Problem. Anders als bei den meisten anderen Dokumenten in einem Projekt, deren Inhalt im vorraus von den Autoren geplant wird, ist eine FAQ von einer gänzlich reaktiven Natur (siehe Die Pflege einer FAQ-Liste). Egal wie groß es wird, Sie werden nie wissen, was der nächste Eintrag sein wird. Und weil es immer Stückweise erweitert wird, passiert es sehr leich, dass das Dokument als ganzes ohne Zusammenhang oder Organization ist, und sogar doppelte oder mit einander ähnliche Einträge enthält. Selbst wenn es keine solche offensichtlichen Probleme hat, gibt es oft unbemerkte Abhängigkeiten zwischen den Einträgen – Verweise die gemacht werden sollten es aber nicht sind – weil die verwandten Einträge ein Jahr auseinander gemacht wurden.

Die Rolle des Verwalters der FAQ ist zweifältig. Erstens pflegt er die allgemeine Qualität der FAQ indem er damit vertraut bleibt, oder zumindest mit den Themen aller Fragen die darin enthalten sint, damit wenn Personen neue Einträge hinzufügen, die einfach nur Duplikate, oder ähnlich zu bereits bestehenden Einträgen sind, die nötigen Anpassungen gemacht werden können. Zweitens, überwacht er die Mailinglisten und anderen Foren, nach immer wiederkehrenden Fragen und schreibt neue Einträge, auf dieser Grundlage. Letztere Aufgabe kann ziemlich komplex sein: Man muss in der lage sein einen Thread verfolgen zu können, die Kernfragen erkennen zu können, einen Eintrag für die FAQ vorzuschlagen, die Kommentare anderer einzubeziehen (da es unmöglich ist, dass der FAQ-Verwalter ein Experte auf jedem Gebiet ist, welches in der FAQ behandelt wird), und merken, wann dieser Vorgang beendet ist, damit der Eintrag endlich hinzugefügt werden kann.

Der FAQ-Verwalter wird auch normalerweise zum Experten für die Formatierung der FAQ. Es gibt eine menge kleiner Details die es bei der Pflege einer FAQ zu beachten gibt (siehe „Behandeln Sie alle Ressourcen wie Archive“ im Kapitel Kapitel 6, *Kommunikation*); wenn beliebige Leute die FAQ bearbeiten, werden sie oft einige dieser Details vergessen. Das ist in Ordnugn, so lange der FAQ-Verwalter da ist, um nach ihnen aufzuräumen.

[4]IssueZilla ist der Bugtracker, dern wir benutzen; es ist ein Abkömling von BugZilla

Es gibt einige freie Software um die Pflege einer FAQ zu unterstützen. Es ist ok sie zu nutzen, so lange es die Qualität der FAQ nicht gefährdet, seien Sie aber vor all zu viel Automatisierung gewarnt. Manche Projekte versuchen die Pflege der FAQ vollständig zu automatisieren, und erlauben es jedem, FAQ Einträge beizutragen und zu bearbeiten, ähnlich wie bei einer Wiki (siehe „Wikis“ im Kapitel Kapitel 3, *Technische Infrastruktur*). Ich habe das insbesondere bei Faq-O-Matic (http://faqomatic.sourceforge.net/) gesehen, obwohl das daran liegen kann, dass ich Fälle gesehen habe, die lediglich Missbräuche von dem wofür Faq-O-Matic ursprünglich gedacht war. Die komplette dezentralisierung der FAQ-Pflege, reduziert zwar den Aufwand für das Projekt, produziert aber in jedem Fall auch eine dürftigere FAQ. Es gibt keine eine Person mit einem groben Überblick über die ganze FAQ, keiner der merkt, wenn bestimmte Einträge aktualisiert werden müssen, oder komplett überflüssig werden, und keiner der nach den Abhängigkeiten zwischen den Einträgen ausschau hält. Das Ergebnis ist eine FAQ welches es oft nicht schafft, den Nutzern das zu geben, wonach sie gesucht haben und im schlimmsten Fall, sie sogar in die Irre führt. Nutzen Sie welche Werkzeuge Sie auch brauchen um die FAQ ihres Projekts zu pflegen, lassen Sie aber niemals den Komfort dieser Hilfsmittel Sie dazu verleiten die Qualität der FAQ beeinträchtigen.

Siehe den Artikel von Sean Michael Kerner, *The FAQs on FAQs*, bei http://osdir.com/Article1722.phtml, für Beschreibungen und Bewertungen von Open-Source-Hilfsmitteln für die Pflege von FAQs.

Übergänge

Ab und zu kommt es vor, dass ein Freiwilliger in einer dauerhaften verantwortungsvollen Position (z.B. ein Patchverwalter, Übersetzungsverwalter, usw.) nicht mehr in der Lage sein wird, die Aufgaben seiner Position auszuführen. Es kann daran liegen, dass die Aufgabe sich als mehr Arbeit heraus gestellt hat, als er es vermutet hatte, oder aufgrund von gänzlich äußeren Ursachen: Heirat, ein neues Kint, ein neuer Arbeitgeber, oder sonst etwas.

Wenn ein Freiwilliger so überschwemmt wird, bemerkt er es meistens nicht sofort. Es passiert allmählich und es gibt keinen Zeitpunkt, an dem ihm bewussst auffällt, dass er nicht mehr die Pflichten seiner Rolle erfüllen kann. Statt dessen, hört das Projekt eine weile lang nichts von ihm. Dann gibt es eine plötzliche Hektik, wenn er sich dafür schuldig fühlt, dass Projekt so lange Vernachlässigt zu haben, und er arbeitet die Nacht durch, um alles nachzuholen. Dann werden Sie wieder eine Zeit lang nichts von ihm hören, und es kann wieder eine hektische Aktivität geben oder auch nicht. Es gibt aber selten einen unaufgeforderten formalen Abtritt. Der freiwillige macht die Arbeit in seiner Freizeit, also wäre ein Abtritt gleichzeitig ein Eingeständnis an sich selber, dass seine Freizeit permanent veringert ist. Menschen geben so etwas oft nur wiederwillig zu.

Es liegt desshalb an Ihnen und die anderen im Projekt zu erkennen was passiert – oder eher, was nicht passiert – und den Freiwilligen zu fragen was los ist. Die Frage sollte freundlich und komplett frei von Schuldzuweisungen sein. Sie wollen eine Information herausfinden, nicht die Person demütigen. Die Anfrage sollte im allgemeinen für das übrige Projekt sichtbar sein, wenn Sie aber einen besonderen Grund haben, warum es privat gehalten werden sollte, dann ist das auch in Ordnung. Der Hauptgrund es öffentlich zu machen, ist damit, wenn der Empfänger antwortet indem er sagt, dass er nicht mehr in der Lage sein wird die Aufgabe zu übernehmen, es eine Kontext gibt, indem Sie Ihre *nächste* öffentliche Nachricht schreiben können: Eine Anfrage nach einem neuen Freiwilligen, der die Rolle übernimmt.

Manchmal ist ein Freiwilliger nicht in der Lage, die Rolle die er übernommen hat, zu erfüllen, weiß es aber entweder nicht oder will es nicht zugeben. Natürlich kann jeder am Anfang Probleme haben, ganz besonders, wenn die Pflichten komplex sind. Wenn jemand jedoch einfach nicht für die Aufgabe geeignet ist, die er übernommen hat, nachdem jeder ihm alle möglichen Hilfen und Vorschläge gegeben haben, dann ist die einzige Lösung für ihn beiseite zu treten, und jemand anderen es versuchen zu lassen. Und wenn die Person das selber nicht sieht, muss man es ihm sagen. Es gibt denke ich, im Prinzip

nur eine Möglichkeit, dass zu handhaben, es ist aber ein Vorgang mit mehreren Schritten, von denen alle wichtig sind.

Erstens, vergewissern Sie sich, dass Sie nicht irre sind. Sprechen Sie privat mit anderen im Projekt, und schauen Sie ob sie auch das Problem für so ernst halten, wie Sie es tun. Selbst wenn Sie sich schon sicher sind, dienen diese Anfragen, andere wissen zu lassen, dass sie es in betracht ziehen, die Person darum zu bitten, beiseite zu treten. Normalerweise wird keiner dagegen einsprechen – sie werden nur froh sein, dass Sie die heikele Aufgabe übernehmen, weil sie es dann selber nicht müssen!

Kontaktieren Sie als nächstes den in Frage stehenden Freiwilligen, im *privaten* und sagen Sie ihm, freundlich aber direkt, über die Probleme, die Sie sehen. Seien Sie spezifisch, mit sovielen Beispielen wie möglich. Achten Sie darauf, ihm klar zu machen, dass andere versucht haben zu helfen, dass die Probleme aber blieben ohne sich zu verbessern. Sie sollten erwarten, dass es eine weile brauchen wird, diese E-Mail zu schreiben, wenn Sie aber bei dieser Art von E-Mail Ihre Behauptungen nicht untermauern, brauchen Sie sie garnicht erst zu schreiben. Sagen Sie, dass Sie einen neuen Freiwilligen für die Rolle suchen wollen, weisen Sie aber auch darauf hin, dass es viele andere Möglichkeiten gibt, zu dem Projekt etwas beizutragen. Zu diesem Zeitpunkt, sollten Sie nicht sagen, dass Sie mit anderen darüber gesprochen haben; keiner mag es, wenn hinter seinem Rücken Komplotte geschmiedet werden.

Es gibt ein paar Wege, wie die Sache danach verlaufen kann. Die wahrscheinlichste Reaktion ist, dass er Ihnen zustimmen wird, oder zumindest nicht widersprechen will, und bereit sein wird abzutreten. In dem Fall, sollten Sie vorschlagen, dass er selber die Ankündigung macht, und Sie können dann mit einer Nachricht anschließen, indem Sie nach einem Ersatz suchen.

Oder er wird zustimmen, dass es Probleme gegeben hat, aber um ein wenig mehr Zeit bitten (oder noch eine Chance, bei diskreten Aufgaben wie beim Versionsverwalter). Wie Sie darauf reagieren, liegt bei Ihnen, aber was immer Sie auch machen, Sie sollten nicht nur desshalb zustimmen, weil Sie solch eine Vernünftige Bitte nicht abschlagen können. Das würde den Leid nur ausdehnen, nicht verringern. Es gibt oft einen ziemlich guten Grund, die Bitte abzulehnen, nämlich, dass es bereits genügend Gelegenheiten gegeben hat, und das Projekt desshalb bei dem derzeitigen Stand ist. Ich habe es wie folgt in einer E-Mail formuliert, an eine Person welche die Rolle des Versionsverwalters übernommen hatte, aber nicht wirklich dafür geeignet war:

```
> Wenn du mich mit jemand anderem ersetzen willst, bin ich gerne
> bereit die Rolle an ihm weiter zu geben. Ich habe eine Bitte, die
> hoffentlich nicht unvernünftig ist. Ich würde gerne noch eine neue
> Version versuchen, um mich beisen zu können.

Ich verstehe deinen Wunsch vollkommen (habe ich selber durchgemacht!),
ich denke aber, dass wir in diesem Fall nicht "noch einen Versuch"
machen sollten.

Das ist nicht die erste oder zweite neue Version, es ist die sechste
oder siebte... Und bei all denen, ich weiß, dass Sie mit dem
Ergebnis unzufrieden sind (weil wir schon mal darüber
geredet haben). Wir haben also tatsächlich die Noch-einen-Versuch-Sache
durchgezogen. Letztlich muss einer dieser Versuche der letzte sein...
Ich denke diese [vergangene] Version ist es.
```

Im schlimmsten Fall, wird der Freiwillige dem komplett widersprechen. Dann müssen Sie sich damit abfinden, dass die Sache unangenehm wird, und sich trotzdem durchkämpfen. Jetzt, ist die Zeit, zu sagen, dass Sie mit anderen darüber geredet haben (sagen Sie aber immer noch nicht mit wem, bis Sie deren Erlaubnis haben, schließlich waren diese Unterhaltungen privat), und dass Sie nicht denken, dass es gut für das Projekt ist, weiterzumachen, wie bissher. Seien Sie nachdrücklich, aber niemals drohent.

Denken Sie daran, dass bei den meisten Rollen der Übergang in dem Moment statfindet, wenn jemand neues die Arbeit anfängt, *nicht* wenn die alte Person damit aufhört. Wenn es bei der Auseinandersetzung zum Beispiel um die Rolle, von sagen wir dem Ticketverwalter geht, können Sie oder andere einflussreiche Entwickler in dem Projekt, jederzeit eine Anfrage nach einem neuen Ticketverwalter machen. Es ist nicht wirklich notwendig, dass die Person die es vorher gemacht hat damit aufhört, so lange er nicht die Arbeit des neuen Freiwilligen sabotiert (absichtlich oder sonstwie).

Was einen verlockenden Gedanken aufbringt: Statt die Person um seine Abdankung zu bitten, warum sollten Sie es nicht einfach so auslegen als wollten Sie ihm etwas Hilfe beschaffen? Warum sollten Sie nicht zwei Ticketverwalter haben, oder Patchverwalter oder was auch immer die Rolle ist?

Obwohl das sich theoretisch nett anhören mag, ist es im allgemeinen keine gute Idee. Die Rolle eines Verwalters beruht gerade – das genau ist ihr Nutzen – auf der Zentralisierung. Dinge, die auf eine dezentralisierte Weise geregelt werden können, werden im Allgemeinen schon so gehandhabt. Zwei Personen zu haben, die eine Verwaltungsrolle einnehmen, würde nur einen größeren Kommunikationsaufwand bedeuten, sowie das Potential die Verantwortlichkeiten unklar zu machen ("Ich dachte du hast den Verbandskasten mitgenommen!" "Ich? Nee, ich dachte *du* hast den Verbandskasten mitgenommen!"). Es gibt natürlich Ausnahmen. Manchmal arbeiten zwei Personen sehr gut zusammen, oder die Natur der Rolle ist derart, dass sie leicht auf mehrere Personen aufgeteilt werden kann. Diese sind aber wahrscheinlich von keinem großen Nutzen wenn Sie jemanden in einer Rolle versagen sehen, für die er nicht geeignet ist. Wenn er das Problem von vorn herein zu würdigen gewusst hätte, hätte er diese Hilfe vorher schon gesucht. In jedem Fall, wäre es respektlos jemand zeit verschwenden zu lassen, bei der Fortführung Aufgabe die keiner achten wird.

Das wichtigste wenn Sie jemand darum bitten abzutreten, ist die Privatsphäre: Ihm den Raum zu geben, um eine Entscheidung zu treffen, ohne sich dabei so zu fühlen, als würden andere ihn beobachten und auf ihn warten. Ich habe einmal den Fehler gemacht – im Nachhinein ein offensichtlicher Fehler – alle drei Parteien auf einmal anzuschreiben, um den Versionsverwalter von Subversion darum zu bitten, für zwei andere Freiwillige abzutreten. Ich hatte mit den beiden neuen bereits im Privaten geredet und wusste, dass sie gewillt waren die Verantwortung zu übernehmen. Naiver Weise und etwas taktlos dachte ich also, dass ich etwas Zeit und Mühe sparen würde, wenn ich an alle eine E-Mail schreiben würde, um den Übergang anzustoßen. Ich nahm an, dass der derzeitige Versionsverwalter bereits vollkommen über die Probleme im klaren war und sofort sehen würde, dass meine Anfrage vernüngtig war.

Ich irrte mich. Der derzeitige Versionsverwalter war sehr beleidigt, und das mit Recht. Es ist eine Sache, darum gebeten zu werden abzutreten; Es ist eine andere *direkt vor* den Personen, an die man abgeben soll, darum gebeten zu werden. Als ich verstanden hatte, warum er beleidigt war, entschuldigte ich mich. Er trat schließlich ohne Umstände zurück und ist heute weiterhin an dem Projekt beteiligt. Seine Gefühle wurden aber verletzt und es ist natürlich auch nicht der angenehmste Anfang für die Neuen.

Committer

Als die einzig wesentlich ausgeprägte Gruppe von Personen die in allen Open-Source-Projekten gefunden werden kann, verdienen Committer hier besondere Aufmerksamkeit. Committer sind ein unvermeidliches Zugeständnis an die Diskriminierung, bei einem System, welches ansonsten so diskriminierungsfrei wie möglich ist. "Diskriminierung" ist hier aber nicht herabsetzend gemeint. Die Funktion die Committer ausüben ist ganz und gar unerlässlich, und ich denke nicht, dass ein Projekt ohne es erfolgreich wäre. Qualitätskontrolle erfordert, nun ja, Kontrolle. Es gibt immer viele Leute die meinen sie wären qualifiziert Änderungen an einer Anwendung zu machen, und irgend eine kleinere Anzahl, die es auch wirklich sind. Das Projekt, kann sich nicht auf das eigene Urteilsvermögen der Leute verlassen; es muss Normen auferlegen, und nur denen Commit-Berechtigung geben, die diese erfüllen[5]. Ande-

[5]Beachten Sie dass Commit-Zugriff bei einem dezentralisierten Versionsverwaltungssystem etwas leicht Abweichendes bedeutet, da jeder sein eigenes Projektarchiv aufsetzen kann, welches mit dem des Projekts verbunden ist, und sich selber zu diesem Projektarchiv Zugriff geben.

rerseits, entsteht ein Machtverhältnis zwischen Leuten die direkt Änderungen committen können, die direkt beben Leuten die es nicht können. Dieses Verhältnis muss so verwaltet werden, dass es dem Projekt nicht schadet.

In „Wahlberechtigung“ im Kapitel Kapitel 4, *Soziale und politische Infrastruktur*, haben wir beteits die Mechanismen besprochen, wie neue Committer in Betracht kommen. Hier werden wir uns die Normen anschauen, nach denen neue Committer beurteilt werden sollten, und wie dieser Vorgang der weiteren Gemeinschaft präsentiert werden sollte.

Auswahl von Committern

In dem Subversion-Projekt, wählen wir Committer hauptsächlich nach dem Hippokratischen Prinzip: *Erstens, richte keinen Schaden an* . Unsere Hauptkriterien sind nicht technische Fähigkeiten oder auch nur Kentnis des Codes, sondern lediglich, dass der Committer ein gutes Urteilsvermögen zeigt. Urteilsvermögen kann auch einfach bedeuten, zu wissen, was man nicht angehen sollte. Eine Person könnte nur kleine Patches einreichen, relativ kleine Probleme im Code beheben; aber wenn die Patches sich sauber anwenden lassen, keine Bugs enthalten, und weitestgehend mit den Richtlinien des Projekts (über die Formatierung von Komentaren und Code) übereinstimmen, und es genügend Patches gibt, dass sich ein klares Muster zeigt, dann wird ein bestehender Committer ihn für gewöhnlich als Kandidaten für den Commit-Zugriff vorschlagen. Wenn mindestens drei Personen zustimmen und es keine Gegenstimmen gibt, wird ein Angebot unterbreitet. Zugegeben, wir haben dann keinen Beweise dafür, dass die Person in der Lage ist, komplexe Probleme in allen Bereichen des Quellcodes zu lösen, aber das macht keinen Unterschied: Die Person hat klar gemacht, dass sie zumindest in der Lage ist, ihre eigenen Fähigkeiten einzuschätzen. Technische Fähigkeiten können gelernt (und gelehrt) werden, für Urteilsvermögen gilt das im Wesentlichen nicht. Deshalb ist es die eine Sache, bei der Sie sicher gehen wollen, dass die Person sie besitzt, vor Sie ihm Commit-Zugriff erteilen.

Wenn ein neuer Committer vorgeschlagen wird, geht es für gewöhnlich nicht um technischen Fähigkeiten, sondern um das Verhalten der Person auf den Mailinglisten oder im IRC. Manchmal zeigt jemand technisches Können und die Fähigkeit innerhalb der formalen Richtlinien des Projekts zu arbeiten, ist jedoch in den öffentlichen Foren immer streitlustig oder nicht kooperativ. Das ist ein ernstes Bedenken; wenn die Person sich mit der Zeit nicht bessert, selbst als reaktion auf Andeutungen, werden wir ihn keinen Commit-Zugriff geben, egal wie Talentiert er ist. In einer Gruppe von Freiwilligen, sind soziale Kompetenzen, oder die Fähigkeit "gut im Sandkasten zu spielen", genau so wichtig wie die rohen technischen Fähigkeiten. Da alles unter Versionsverwaltung steht, sind die Nachteile einen Committer hinzuzufügen den Sie lieber nicht hätten, nicht so sehr die Probleme die es im Code bereiten könnte (Code Überprüfungen sollten das sowieso schnell aufdecken), sondern dass es irgendwann das Projekt dazu zwingen könnte die commit Berechtigung wegzunehmen – etwas, was niemals angenehm ist und manchmal Auseinandersetzungen provoziert.

Viele Projekte bestehen darauf, dass der potentielle Committer einen gewissen Grad an technischen Kentnissen und Ausdauer vorweist, indem er eine gewisse Anzahl nicht trivialer Patches einreicht – diese Projekte wollen also nicht nur wissen, dass die Person keinen Schaden anrichten wird, sondern auch das er sich wahrscheinlich im gesamten Quellcode bewähren wird. Das ist völlig in Ordnung, seien Sie aber vorsichtig, dass die Commit-Berechtigung nicht anfängt, sich in die Mitgliedschft in einem exklusiven Club, zu verwandeln. Die Frage die in allen Köpfen sein sollte ist "Was wird die besten Ergebnisse für den Code liefern?" nicht "Werden wir den sozialen Status veringern, der mit Commit-Berechtigung in Zusammenhang gebracht wird, wenn wir diese Person aufnehmen"? Der Sinn des Commit-Zugriffs ist nicht das Selbstwertgefühl der Leute zu heben, es geht darum, mit möglichst wenigen Umständen Änderungen am Code zu erlauben. Wenn Sie 100 Committer haben, von denen 10 regelmäßig große

Nichtsdestoweniger gilt das *Konzept* vom Commit-Zugriff, trotzdem noch: "Commit-Zugriff" ist steht für "das Recht, Änderungen am Code vorzunehmen, der in der nächsten Version der Software ausgeliefert wird". In zentralisierten Systemen bedeutet es, direkten Commit-Zugriff zu haben; bei dezentralisierten Systemen bedeutet es, dass die Änderungen standardmäßig ins Projektarchiv des Projekts geladen werden. So oder so, ist es das gleiche; die Mechanik, mit dem es realisiert wird, ist nicht sonderlich wichtig.

Änderungen machen, und die anderen 90 beheben nur Tippfehler und kleine Fehler ein paar mal im Jahr, ist das immer noch besser als nur die 10 zu haben.

Widerruf von Commit-Zugriff

Das erste was zu der Rücknahme von Commit-Zugriff gesagt werden muss ist: Versuchen Sie möglichst gar nicht erst in die Lage zu geraten. Abhängig davon, wessen Zugriff zurückgezogen wird, und warum, können die Diskussionen um solche Maßnahmen sehr geteilt sein. Selbst wenn sie nicht geteilt sind, können sie eine sehr zeitaufwendige Ablenkung von der produktiven Arbeit sein.

Wenn Sie es jedoch tun müssen, sollte die Diskussion im privaten mit den selben Personen geführt werden, die ein Stimmrecht für die *Gewährung* des Zugriffs haben, welche diese Person gerade hat. Die Person selber sollte nicht mit einbezogen werden. Das wirderspricht die gewöhnliche Vorschrift gegen die Geheimhaltung, ist aber in diesem Fall notwendig. Erstens, könnte sonst keiner frei reden. Zweitens, wenn der Antrag fehlschlägt, wollen Sie nicht unbedingt, dass die Person weiß, dass es überhaupt zur Debatte stand, da es Fragen aufwerfen könnte ("Wer war auf meiner Seite? Wer war gegen mich?") die zu der schlimmsten Art von Parteibildung führen. Unter bestimmten seltenen Umständen, kann die Gruppe jemandem sagen wollen, dass der Widerruf in Betracht gezogen wird oder wurde, als Warnung, diese Offenheit sollte aber eine Entscheidung der Gruppe sein. Keiner sollte jemals aus Eigeninitiative jemandem Informationen aus einer Diskussion und einer Wahl preisgeben, von denen andere angenommen haben, dass sie Geheim waren.

Nachdem der Zugriff widerrufen wurde, ist diese Tatsache zwangsläufig öffentlich (siehe „Vermeiden Sie Geheimnisse“ später in diesem Kapitel), also versuchen Sie so Taktvoll wie möglich zu sein, in der Art wie Sie es der Öffentlichkeit präsentieren.

Eingeschränkter Commit-Zugriff

Manche Projekte bieten eine Abstufung bei der Commit-Berechtigung an. Es kann zum Beispiel Freiwillige geben, die freien Zugriff auf die Dokumentation habem, die aber nicht an den Code selber committen können. Häufige Bereiche für den Teilzugriff sind unter anderem die Dokumentation, Übersetzungen, Code für die Anbindung anderer Programiersprachen, bestimmte Dateien um Packete zu erstellen (z.B. Dateien spezifisch zum RPM von RedHat usw.), sowie andere Orte, an denen ein Fehler nicht zu einem großen Problem für das Projekt werden wird.

Da es beim Commit-Zugriff nicht nur darum geht Änderungen zu machen, sonderan auch einen Teil der Wahlberechtigten zu sein (siehe „Wahlberechtigung“ im Kapitel Kapitel 4, *Soziale und politische Infrastruktur*), stellt sich natürlich die Frage: Worüber können teilberechtigte Committer abstimmen? Es gibt keine eine richtige Antwort; es hängt davon ab, welche Arten von Bereichen Ihr Projekt mit Teilzugriff hat. In Subversion haben wir alles relativ einfach gehalten: Ein Committer mit Teilzugriff, kann über Angelegenheiten abstimmen, die exclusiv mit seinem Bereich zu tun haben und auf nichts anderes. Wichtig dabei, ist dass wir eine Möglichkeit haben, um beratende Stimmen abgeben können (im Wesentlichen, schreibt der Committer "+0" oder "+1 (nicht-bindend)" statt einer einfachen "+1" Stimme). Es gibt keinen Grund, Leute komplett zum Schweigen zu bringen, nur weil ihre Stimme nicht formal bindend ist.

Vollständig Commit-Berechtigte können über alles abstimmen, genau so wie sie überall committen können, und nur Vollberechtigte können über die Aufnahmen von neuen Committern jedweder Art abstimmen. In der Praxis, wird die Fähigkeit neue Teilberechtigte aufzunehmen, jedoch weitergegeben: Jeder Vollberechtigte kann für einen neuen Teilberechtigten "bürgen", und teilberechtigte Committer können im wesentlichen neue Committer für den gleichen Bereich wählen (das ist besonders hilfreich, damit die Übersetzungsarbeit glatt verläuft).

Ihr Projekt kann eine etwas andere Anordnung erfordern, abhängig von der Natur der Arbeit, die selben allgemeinen Prinzipien gelten aber für alle Projekte. Jeder Committer sollte über Angelegenheiten

abstimmen können, die in dem Bereich fallen auf den er Zugriff hat, und nicht auf solche die außerhalb davon liegen, und Wahlen über Generelle Abläufe, sollten automatisch auf die vollen Committer gehen, es sei denn es gibt einen Grund (welcher von den vollen Committern entschieden wurde) um den Wahlkreis auszuweiten.

Was die Druchsetzung des Teilzugriffs angeht: Es ist of am besten, wenn das Versionsverwaltungssystem *nicht* den Teilzugriff erzwingt, selbst wenn es dazu in der Lage ist. Siehe „Autorisierung“ im Kapitel Kapitel 3, *Technische Infrastruktur* für weiteres über die Gründe dafür.

Untätige Committer

Manche Projekte entfernen den Zugriff auf das Projektarchiv nach einer gewissen Zeit (sagen wir, einem Jahr) ohne Commit-Aktivität. Ich denke dass das meist nicht hilfreich und sogar aus zwei Gründen kontraproduktiv ist.

Erstens kann es Leute dazu verleiten, annehmbare aber unnötige Änderungen zu committen, nur um zu verhindern, dass ihr Commit-Zugriff abläuft. Zweitens dient es keinen wirklichen Zweck. Wenn die Hauptkriterien, Leuten Commit-Zugriff zu gewähren, ein gutes Urteilsvermögen ist, warum sollte man annehmen, dass das Urteilsvermögen von jemand verfällt, nur weil er dem Projekt eine Weile fernblieb? Selbst wenn er komplett über Jahre verschwindet, sich nicht den Code anschaut, noch die Diskussionen über die Entwicklung mitverfolgt, wird er wenn er wieder auftaucht *wissen* wie fremd er der Sache ist und sich entsprechend verhalten. Sie haben seinem Urteilsvermögen vorher vertraut, warum sollten Sie es nicht immer vertrauen? Wenn ein Diplom nicht abläuft, dann sollte es der commit Zugriff auch nicht.

Manchmal wird ein commiter darum bitten entfernt zu werden, oder in der Auflistung der Committer explizit als abwesend markiert zu werden (siehe „Vermeiden Sie Geheimnisse“ weiter unten für weiteres über diese Liste). In solchen Fällen, sollten das Projekt natürlich den Wünschen der Person nachkommen.

Vermeiden Sie Geheimnisse

Auch wenn die Diskussionen um die Aufnahme eines bestimmten neuen Committers geheim gehalten werden sollte, müssen die Regeln und Abläufe selber nicht geheim seit. Es ist sogar am besten, wenn sie veröffentlicht werden, damit Leute erkennen, dass die Committer nicht irgend einer mysteriösen, prominenten Kaste angehören, die für Normalsterbliche verschlossen ist, sondern dass jeder einfach eintreten kann, indem er gute Patches einreicht, und sich in der Gemeinschaft zu verhalten weiß. Im Subversion-Projekt präsentieren wir diese Information direkt im Dokument der Entwicklerrichtlinien, da diejenigen die am ehesten daran interesiert sind Commit-Zugriff zu bekommen, solche sind die darüber nachdenken, Code zum Projekt beizutragen.

Zusätzlich zu der Veröffentlichung der Richtlinien, sollten Sie die *Liste* der Committer veröffentlichen. Der traditionelle Ort hierfür ist eine Datei namens `MAINTAINERS` oder `COMMITTERS` im obersten Verzeichnis des Quellcodes vom Projekt. Es sollte zuerst alle vollen commit Berechtigten auflisten, gefolgt von verschiedenen Bereichen und die zugehörigen teilberechtigten Committer. Jede Person sollte mit Namen und E-Mail-Adresse aufgeführt sein, wobei die Adresse kodiert sein darf, um Spam zu verhindern (siehe „Verschleierung von Adressen im Archiv“ im Kapitel Kapitel 3, *Technische Infrastruktur*) wenn die Person das bevorzugt.

Da die Unterscheidung zwischen Voll- und Teilzugriff auf das Projektarchiv offensichtlich und gut definiert ist, ist es auch angemessen, wenn diese Liste diese Unterscheidung auch macht. Darüber hinaus, sollte die Liste nicht versuchen die informellen unterschiede anzudeuten, die sich zwangsläufig in einem Projekt ergeben, wie wer besonders einflussreich ist und wie. Es ist ein öffentliches Protokoll, keine Datei für Anerkennungen. Listen Sie die Commiter in alphabetischer Reihenfolge auf, oder in so wie sie angekommen sind.

Anerkennung

Anerkennung ist die hauptsächliche Währung in der Welt der freien Software. Was immer Leute über ihre Motive für die Beteiligung an einem Projekt sagen, ich kenne keine Entwickler die glücklich damit wären, ihre ganze Arbeit anonym zu verrichten, oder unter dem Namen von jemand anderem. Es gibt hierfür konkrete Gründe: Der Ruf innerhalb eines Projekts diktiert ungefähr wieviele Einfluss man hat, und die Beteiligung an einem Open-Source-Projekt kann auch einen indirekten finanziellen Wert haben, weil manche Arbeitgeber bei einer Bewerbung mittlerweile danach Ausschau halten. Es gibt auch vielleicht sogar noch stärkere immaterielle Gründe: Menschen wollen einfach geschätzt werden, und suchen instinktiv nach Zeichen, dass ihre Arbeit von anderen anerkannt wird. Das Versprechen von Anerkennung ist desshalb eines der besten Motivationen, die ein Projekt hat. Wenn kleine Beiträge anerkennung finden, kommen Leute zurück um mehr zu machen.

Einer der wichtigsten Funktionen von Software für die gemeinschaftliche Entwicklung (siehe Kapitel 3, *Technische Infrastruktur*) ist dass es genaue Protokolle darüber führt, wer was wann gemacht hat. Überall wo es möglich ist, sollten Sie bereits vorhandene Mechanismen benutzen um sicher zu stellen, dass diese Anerkennung genau verteilt wird, und seien Sie spezifisch, über die Natur des Beiträgs. Schreiben Sie nicht einfach "Danke an H. Mustermann <hmustermann@beispiel.de>" wenn Sie statt dessen "Danke an H. Mustermann<hmustermann@beispiel.de> für die Bug-Meldung und die Anleitung zur Reproduktion" in einem Kommentar schreiben könnten.

Bei Subversion haben wir eine informelle aber stetige Richtlinie, denjenigen der den Bug gemeldet hat entweder in dem zugehörigen Ticket zu würdigen oder wenn nicht, in dem Commit-Kommentar der Änderung die den Bug behebt. Eine kurze Betrachtung der Kommentare im Subversion-Projektarchiv ergibt, dass mit 14525 ungefähr 10% der Änderungen jemanden bei Namen und E-Mail-Adresse würdigt, üblicherweise die Person, welche den Bug gemeltet oder untersucht hat, der durch die Änderung behoben wurde. Achten Sie darauf, dass diese Person eine andere ist als der Entwickler, der den Commit gemacht hat, dessen Name bereits automatisch vom Versionsverwaltungssystem erfasst wird. Von den ca. 80 Voll- und Teil-Commit-Berechtigten, die Subversion heute hat, wurden 55 in den Commit-Kommentaren (meistens mehrere Male) gewürdigt, bevor sie selber Committer wurden. Das beweist natürlich nicht, dass die Anerkennung ein Grund für die weitere Beteiligung war, baut aber zumindest eine Atmosphäre auf, in der Leute darauf zählen können, dass ihre Beiträge gewürdigt werden.

Es ist wichtig zu unterscheiden, zwischen gewöhnliche Anerkennung und besondere Danksagungen. Wenn ein bestimmter Codeteil, oder irgend ein anderer Beitrag den jemand geleistet hat, diskutiert wird, ist es in Ordnung, ihre Arbeit zu würdigen. Wenn Sie zum Beispiel sagen "Die kürzlichen Änderungen von Daniel an dem Delta-Code bedeuten, dass wir jetzt Funktion X implementieren können" hilft den Leuten zu erkennen, um welche Änderungen es sich handelt, und es würdigt gleichzeitig die Arbeit von Daniel. Es dient andererseits keinen direkten praktischen Zweck, Daniel nur für seine Änderungen an dem Delta-Code zu danken. Es fügt keine Informationen hinzu, da das Versionsverwaltungssystem und andere Mechanismen die Tatsache aufgezeichnet hat, wer die Änderungen gemacht hat. Jeden für alles zu danken, wäre ablenkend und würde letztendlich frei von Informationen, da Danksagungen in so weit effektiv sind, wie sehr sie aus den üblichen positiven Kommentaren herausragen, welche die ganze Zeit ablaufen. Das bedeutet natürlich nicht, dass Sie niemals Leuten danken sollten. Sorgen Sie einfach dafür, dass Sie es auf Arten tun, die nicht zu einer Inflation der Anerkennung führen. Diese Richtlinien zu folgen, wird helfen:

- Je flüchtiger das Forum, desto freier sollten Sie Ihren Dank ausdrücken. Jemandem zum Beispiel im Vorbeigehen, während einer Unterhaltung im IRC, für seinen Bugfix zu danken, ist in Ordnung, genau so wie eine beiläufige Erwähnung in einer E-Mail die hauptsächlich anderen Themen gewidmet ist. Schreiben Sie aber keine E-Mail alleine um jemanden zu danken, es sei denn es ist für eine wirklich ausergewöhnliche Leistung. Sie sollten gleichermaßen nicht die Webseiten des Projekts mit Ausdrücken von Dankbarkeit verunstalten. Wenn Sie erst einmal damit anfangen, wird nie klar sein, wann oder wo man aufhören soll. Und schreiben Sie *niemals* Danksagungen in die Kommentare des Codes;

das wäre nichts als eine Ablenkung vom Hauptzweck der Kommntare, der darin besteht dem Leser beim Verständnis des Codes zu helfen.

- Je weniger jemand mit dem Projekt zu tun hat, desto angemessener ist es sich für etwas zu danken, was er geleistet hat. Das mag sich nicht eingängig anhören, passt aber mit der Einstellung zusammen, dass Sie Lob ausdrücken sollten, wenn jemand mehr beiträgt als Sie erwartet hätten. Es würde deshalb eine geringere Erwartung an Personen ausdrücken, als sie selber an sich haben, wenn Sie sich ständig für regelmäßige Beiträge bedanken, die sie normerweise machen. Wenn überhaupt, wollen Sie das engegengesetzte Ergebnis erzielen!

  Es gibt ab und zu Ausnahmen zu dieser Regel. Man kann jemanden dafür danken seine Rolle zu erfüllen, wenn diese Rolle von Zeit zu Zeit temporäre, intensive Anstrengungnen erfordert. Das Paradebeispiel hierfür ist der Versionsverwalter, der in Zeiten der Veröffentlichung einer neuen Version in einen Gang hochschaltet, ansonsen aber untätig ist (zumindest in seiner Rolle als Versionsverwalter – er kann auch als Entwickler aktiv sein, aber das ist eine andere Angelegenheit).

- Genau so wie Kritik und Anerkennung, sollte Dankbarkeit so spezifisch wie möglich sein. Danken Sie Leuten nicht einfach dafür, dass sie toll sind, selbst wenn das der Fall ist. Danken Sie sie für etwas, dass außergewöhnlich war und es gibt zusätzliche Punkte, wenn Sie auch noch genau sagen, warum das was sie gemacht haben, so toll war.

Im allgemeinen, gibt es immer eine Spannung dazwischen zu gewärleiten, dass die einzelnen Beiträge anerkannt werden, und sicher zu stellen, dass das Projekt eher eine gemeinsame Anstrenung ist als eine Ansammlung einzelner Prachten ist. Bleiben Sie einfach im Klaren über diese Spannung und versuchen Sie sich auf die Seite der Gruppe zu halten, und Sachen werden nicht ausarten.

Abspaltungen

In „Aufspaltbarkeit“ im Kapitel Kapitel 4, *Soziale und politische Infrastruktur*, haben wir gesehen, wie das *Potential* für eine Abspaltung (en. Fork) wichtige Auswirkungen darauf hat, wie ein Projekt geführt wird. Was passiert aber, wenn eine Abspaltung wirklich stattfindet? Wie sollten Sie damit umgehen, und welche Auswirkungen können Sie davon erwarten? Wann sollten Sie umgekehrt, eine Abspaltung *anstoßen*?

Die Antworten hängen davon ab, um welche Art von Abspaltung es sich handelt. Manche Abspaltungen sind aufgrund von freundlichen aber unüberbrückbare Meinungsverschiedenheiten, über die Richtung des Projekts; mehr sind vielleicht sowohl auf technische Argumente als auch auf zwischenmenschliche Konflikte zurückzuführen. Natürlich ist es nicht immer möglich, den Unterschied zwischen den beiden zu erkennen, da technische Argumente auch persönliche Anteile beinhalten können. Alle Abspaltungen haben gemeinsam, dass eine Gruppe von Entwicklern (oder manchmal auch nur ein Entwickler) sich entschieden haben, dass die Kosten mit manchen oder alle anderen Entwicklern zu arbeiten, jetzt schwerer wiegen als der Nutzen.

Wenn ein Projekt sich spaltet, gibt es keine definitive Antwort darüber, welche Abspaltung das "echte" oder "ursprüngliche" Projekt war. Lete werden umgangssprachlich davon sprechen, dass die Abspaltung A aus dem Projekt P kommt, als ob P weiter seinen natürlichen Lauf nimmt wärend A in neue Gebiete divergiert, was aber im wesentlichen eine Aussage darüber ist, wie dieser bestimmte Beobachter das Thema sieht. Es ist im Grunde genommen eine Frage der Sichtweise: Wenn eine ausreichend große Gruppe damit übereinstimmt, dann fängt die Behauptung an wahr zu werden. Es ist nicht so, dass es eine objektive Wahrheit von vornherein gibt, eine welche wir zunächst zweifelsfrei beobachten können. Sondern vielmehr, die Auffassungen *sind* die objektive Wahrheit, da ein Projekt – oder eine Abspaltung – letztendlich eine Entität ist, die sowieso nur in den Köpfen der Menschen existiert.

Wenn diejenigen, welche die Abspaltung anstoßen das Gefühl haben, dass sie aus dem Hauptprojekt einen neuen Ast entsprießen, ist die Frage der Wahrnehmung sofort und einfach beantwortet. Jeder,

sowohl die Entwickler als auch die Nutzer werden die Abspaltung wie ein neues Projekt behandeln, mit einem neuen Namen (vielleicht auf dem alten Namen basierend, aber leicht davon zu unterscheiden), einer eigenen Webseite, und einer anderen Philosopie oder Ziel. Die Sache wird jedoch verwirrender, wenn beide Seiten der Meinung sind, die legitimen Wächter des ursprünglichen Projekts zu sein und desshalb ein Recht haben, ein Recht darauf haben, den ursprünglichen Namen zu benutzen. Wenn es eine Organization gibt, mit Markenrechte auf den Namen, oder rechtliche Kontrolle über die Domaine oder Webseiteb, ist die Angelegenheit über das Recht erledigt: Diese Organisation wird entscheiden wer das Projekt ist und wer die Abspaltung, denn es hält alle Karten in einem Krieg um die öffentliche Wahrnehmung. Natürlich kommen die Sachen meistens nicht so weit: Da jeder bereits weiß, wie die Machtverhältnisse sind, werden sie es vermeiden eine Schlacht zu kämpfen, dessen Ausgang sie bereits kennen, und einfach gleich zum Ende springen.

Zum Glück, gibt es selten Zweifel darüber, welches das Projekt ist, und welches die Abspaltung, denn eine Abspaltung ist im wesentlichen eine Vertrauensfrage. Wenn mehr als die hälfte der Entwickler für welche Richtung die Abspaltung auch immer vorschlägt sind, gibt es normalerweise keinen Grund eine Abspaltung zu machen – das Projekt kann einfach selber diese Richtung einschlagen, es sei denn es wird als Diktatur geführt mit einem besonders sturen Diktator. Wenn andererseits, weniger als die Hälfte dafür sind, ist die Abspaltung eindeutig eine Rebellion einer Minderheit, und is ist sowohl entgegenkommend, als auch vernünftig, dass es sich selber als einen Zweig sieht, und nicht als den Stamm.

Umgang mit Abspaltungen

Wenn jemand droht von ihrem Projekt eine Abspaltung zu machen, bleiben Sie ruhig und denken Sie an Ihre längerfristigen Ziele. Die bloße *Existenz* einer Abspaltung ist es nicht, was einem Projekt schadet; vielmehr ist es der Verlust von Entwickler und Nutzer. Ihr echtes Ziel ist es desshalb nicht, die Abspaltung zu unterdrücken, sondern diese schädlichen Auswirkungen zu minimieren. Sie können darüber sauer sein, oder der Meinung sein, dass die Abspaltung nicht gerecht und unprovoziert war, das aber öffentlich äußern kann einzig und alleine unentschlossene Entwickler entfremden. Statt dessen, sollten Sie Leute nicht dazu zwingen, exklusive Entscheidungen zu treffen, und so kooperativ sein, wie es bei einer Abspaltung machbar ist. Als erstes, sollten Sie nicht die commit Berechtigung zu ihrem Projekt von jemandem zurücknehmen, der sich entschieden hat an der Abspaltung zu arbeiten. Arbeit an der Abspaltung bedeutet nicht, dass die Person plötzlich seine Kompetenz an dem ursprünglichen Projekt zu arbeiten verloren hat; vorherige Committer sollten auch nacher Committer sein. Darüber hinaus, sollten Sie Ihr Wunsch ausdrücken, so kompatibel wie möglich mit der Abspaltung zu bleiben ausdrücken und sagen, dass Sie hoffen, dass die Entwickler die Änderungen zwischen beiden übernehmen wenn es angemessen ist. Wenn Sie administrativen Zugriff auf die Server des Projekts haben, sollten Sie der Abspaltung am Anfang öffentlich Hilfe bei der Infrastruktur anbieten. Bieten Sie ihnen zum Beispiel eine Kopie des Projektarchiv an, mit der kompletten Historie, wenn es keine andere Möglichkeit für sie gibt, daran zu kommen, ohne auf die die historischen Daten verzichten zu müssen (das muss je nach Versionsverwaltungssystem nicht unbedingt notwendig sein). Fragen Sie danach, ob es irgend etwas anderes gibt, was sie brauchen und geben Sie es ihene wenn möglich. Reißen Sie sich ein Bein aus, um zu zeigen, dass Sie ihnen nicht im Weg stehen, und dass Sie wollen, dass die Abspaltung nach seinen eigenen Verdiensten Erfolg hat oder fehlschlägt und sonst nichts.

Der Grund all das zu tun – und es öffentlich zu machen – ist nicht wirklich um der Abspaltung zu helfen, sondern um die Enwickler zu überreden, dass Ihre Seite eine sichere Sache ist, indem Sie möglichst nicht als rachsüchtig erscheinen. In einem Krieg machte es manchmal Sinn (aus strategischer Sicht, wenn auch nicht aus menschlicher Sicht) Leute dazu zu zwingen eine Seite zu wählen, bei freier Software macht es jedoch fast niemals Sinn. Nach einer Abspaltung arbeiten manche Entwickler sogar offen an beiden Projekten, und versuchen ihr möglichstes um beide kompatibel zu halten. Diese Entwickler helfen die Kommunikationspfade nach der Abzweigungung offen zu halten. Sie erlauben es ihrem Projekt von den Interessanten neuen Funktionen in der Abzweigung zu profitieren (ja, die Abzweigung kann Sachen haben, welche Sie haben wollen), und später die Wahrscheinlichkeit einer Zusammenführung vergrößern.

Manchmal wird eine Abzweigung derart erfolgreich, dass auch wenn es selbst von seinen Anstiftern zum Beginn als solches angesehen wurde, es zu der Version wird, die jeder bevorzugt, und letztendlich das Original aufgrund der großen Nachfrage ersetzt. Ein berühmtes Beispiel hierfür war die GCC/EGCS-Abzweigung. Die *GNU Compiler Collection* (*GCC*, vorher der *GNU C Compiler*) ist der beliebteste Open-Source-Compiler für nativen Code und auch einer der portabelsten Compiler der Welt. Aufgrund von Meinungsverschiedenheiten zwischen den Personen die es offiziell pflegten, und Cygnus Software,[6] einer der aktivsten Entwickler von GCC, machte Cygnus eine Abzweigung von GCC namens *EGCS*. Die Abzweigung war absichtlich nicht feindlich gesinnt: Die EGCS-Entwickler versuchten nicht, zu irgend einem Zeitpunkt, ihre Version von GCC als die neue ofizielle Version darzustellen. Statt dessen, konzentrierten sie sich darauf, EGCS so gut wie möglich zu machen, und Patches schneller einzubinden, als die offiziellen Entwickler von GCC. EGCS wurde beliebter, und irgendwann entschieden sich einige größere Vertreiber von Betriebssystemen EGCS anstatt von GCC als ihren standard Compiler auszuliefern. Zu diesem Zeitpunkt, wurde es für alle bei GCC klar, dass an dem Namen "GCC" festzuhalte, wärend jeder zu der EGCS-Abzweigung wechselte, jedem eine unnötigen Namensänderung auferlegen würde, aber nichts machen würde, um den Wechsel zu verhindern. GCC übernahm also den Code von EGCS und es gab wieder eine einzige GCC, nun aber durch die Abzweigung wesentlich verbessert.

Dieses Beispiel zeigt, warum Sie eine Abzweigung nicht immer als etwas ganz und gar schlechtes betrachten können. Eine Abzweigung mag zu der Zeit etwas schmerzhaft und unwillkommen sein, Sie können aber nicht unbedingt absehen ob es Erfolg haben wird. Sie und das übrige Projekt sollten deshalb ein Auge darauf halten, und bereit sein nicht nur alle Funktionen und Code aufzunehmen, wo es möglich ist, sonder im extreemsten Fall sogar der Abzweigung beizutreten wenn es den größten Teil der Aufmerksamkeit des Projekts einnimmt. Sie werden natürlich oft in der Lage sein den wahrscheinlichen Erfolg einer Abzweigung abzusehen, jenachdem wer ihm beitritt. Wenn die Abzweigung von dem größten Kläger im Projekt angefangen wird und von einer Handvoll verärgerten Entwickler die sich sowieso nicht konstruktiv verhalten haben, haben Sie im wesentlichen für Sie das Problem, durch die Abzweigung, erledigt, und Sie müssen sich wahrscheinlich keine Sorgen machen, dass es vom ursprünglichen Projekt irgend welche Schwung wegnimmt. Wenn Sie jedoch sehen, wie einflussreiche und geachtete Entwickler die Abzweigung unterstützen, sollten Sie sich fragen warum. Vielleicht ist das Projekt übermäßig restriktiv gewesen, und die beste Lösung ist es, einige oder alle Maßnahmen, die von der Abzweigung erwägt werden, in das Hauptprojekt einzubinden – im Wesentlichen vermeiden Sie die Abspaltung indem Sie zu ihr werden.

Eine Abspaltung anstoßen

Alle Ratschläge hier gehen davon aus, dass Sie als letztes Mittel eine Abspaltung versuchen. Nutzen Sie alle anderen Möglichkeiten, bevor Sie diesen Schritt erwägen. Es bedeutet fast immer, Entwickler zu verlieren, lediglich mit einem ungewissen Versprechen, später neue zu bekommen. Es bedeutet auch, einem Wettbewerb um die Aufmerksamkeit der Benutzer zu beginnen: Jeder, der dabei ist die Software herunterzuladen, wird sich fragen müssen: "Hmm, will ich diesen oder den anderen?" In welcher Position Sie sich auch immer befinden, die Situation ist chaotisch, da eine Frage entsteht, die vorher nicht da war. Manche Leute behaupten nach dem üblichen Argument der natürlichen Auslese, dass Abspaltungen für das Ökosystem der Software in der Gesamtheit gesund ist: Die Tüchtigsten überleben, was letztlich bedeutet, dass jeder bessere Software bekommt. Das mag aus Sicht des Ökosystems wahr sein, trifft aber nicht die Sicht des einzelnen Projekts. Die meisten Abspaltungen sind nicht erfolgreich und die meisten Projekte sind mit einer Abspaltung nicht glücklich.

Damit einher geht, dass Sie die Drohung einer Abspaltung in einer Debatte nicht als extreme Technik benutzen sollten – "Macht die Sache auf meine Art, oder ich werde das Projekt spalten!" – da jeder sich darüber im Klaren ist, dass eine Abspaltung, die es nicht schafft Entwickler des ursprünglichen Projekts anzulocken, wahrscheinlich nicht lange überleben wird. Alle Beobachter – nicht nur die Entwickler, son-

[6] Jetzt ein Teil von RedHat (http://www.redhat.com/).

dern auch die Benutzer und Packetverwalter der Betriebssysteme – werden ihr eigenes Urteil darüber fällen, welche Seite sie wählen. Sie sollten deshalb gegenüber einer Abspaltung sehr abgeneigt erscheinen, damit Sie, sollte es letztlich doch geschehen, glaubhaft machen können, es sei der einzig mögliche Ausweg gewesen.

Vergessen Sie nicht, *alle* Faktoren bei der evaluierung des möglichen Erfolgs von Ihrer Abspaltung in Erwägung zu ziehen. Wenn zum Beispiel viele der Entwickler in einem Projekt den gleiche Arbeitgeber haben, dann werden sie, selbst wenn Sie verärgert und im privaten für eine Abspaltung sind, es wahrscheinlich nicht laut sagen, wenn Sie wissen, dass ihr Arbeitgeber dagegen ist. Viele Programmierer freier Software denken gerne, dass eine freie Lizenzierung des Codes bedeutet, keine einzelne Firma könne die Entwicklung dominieren. Es ist wahr, dass die Lizenz im letztendlichen Sinne die Freiheit garantiert – wenn andere den dringenden Wunaxh hegen, das Projekt zu spalten, und die Ressourcen dazu haben, können sie das. In der Praxis sind die Entwicklerteams einiger Projekte zum Großteil durch ein Unternehmen finanziert, und es gibt keinen Grund so zu tun, als würde die Unterstützung dieses Unternehmens keinen Unterschied machen. Wenn es die Abspaltung ablehnt, werden seine Entwickler wahrscheinlich nicht daran teilnehmen, selbst wenn sie es insgeheim wünschten.

Wenn Sie trotzdem zu dem Schluss kommen, das Sie sich abspalten müssen, sollten Sie zunächst im privaten dafür Unterstützung suchen, und es dann ohne Feindseligkeit bekannt geben. Selbst wenn Sie wütend oder entäuscht von den derzeitigen Verwaltern sind, lassen Sie es nicht durchblicken. Halten Sie einfach leidenschaftslos fest, was Sie zu der Entscheidung geführt hat, und dass Sie gegenüber dem Projekt von dem Sie abspalten, keine Böswilligkeit hegen. Angenommen, Sie betrachten es als eine Abspaltung (im Gegensatz zu einer notgedrungenen Erhaltung des ursprünglichen Projekts), sollten Sie hervorheben, dass Sie den Code und nicht den Namen abspalten, und wählen Sie einen Namen, der nicht mit dem des ursprünglichen Projekts im Konflikt gerät. Sie können einen Namen wählen, welcher den ursprünglichen beinhaltet oder darauf verweist, so lange es nicht ein Tor für Verwirrung zwischen beiden aufmacht. Es ist natürlich in Ordnung markant auf der Webseite der Abspaltung zu erklären, dass es von dem ursprünglichen Projekt abstammt, und dass es hofft es zu ersetzen. Machen Sie nur nicht das Leben der Benutzer schwieriger indem Sie ihnen aufzwingen, ein Gerangel um die Identität auseinanderzufuseln.

Schließlich können Sie der Sache einen guten Start geben, indem Sie automatisch allen Entwicklern des ursprünglichen Projekts Commit-Rechte zu der Abspaltung geben, inklusive denen die nicht der Notwendigkeit einer Abspaltung zustimmten. Selbst wenn sie niemals den Zugang benutzen, ist Ihre Nachricht klar: Es gibt hier Meinungsverschiedenheiten, aber keine Feinde, und Sie heißen Beiträge aus jeder kompetenten Quelle willkommen.

Kapitel 9. Lizenzen, Urheberrecht[1] und Patente

Die Lizenz die man auswählt hat vermutlich keinen großen Einfluss auf die Einführung des Projektes, solange es eine Open-Source-Lizenz ist. Benutzer wählen Software aufgrund von Qualität und Funktionalität, und nicht wegen Lizenzdetails aus. Trotzdem sollte man ein Grundverständnis über Open-Source-Lizenzen haben um einerseits sicherzustellen, dass die Lizenz zu den Zielen des Projektes passt und andererseits um in der Lage zu sein mit anderen über die Lizenz zu reden. Bitte beachtet, dass ich kein Anwalt und dass dieses Kapitel nicht als juristischer Rat zu sehen ist. Dafür braucht man einen Anwalt oder sollte selbst einer sein.

Terminologie

In jeder Diskussion über Open-Source-Lizenzen stellt man zunächst fest, dass es viele Begriffe für die gleiche Sache gibt: *Freie Software*, *Open Source*, *FOSS*, *F/OSS*, und *FLOSS*. Wir beginnen damit, diese und einige andere Begriffe zu klären.

Freie Software
: Software, die auch im Quelltext frei verteilt und modifiziert werden kann. Der Begriff wurde zuerst von Richard Stallman geprägt, der das Prinzip in der GNU General Public License (GPL) festschrieb, und der die Free Software Foundation (http://www.fsf.org/) gründete um das Konzept bekannt zu machen.

  Obwohl "Freie Software" ungefähr genau soviel Software umfasst wie "Open Source", bevorzugen die FSF und viele andere den Begriff "Freie Software", da er die Idee von Freiheit betont und das Konzept frei verteilbarer Software vor allem als gesellschaftliche Bewegung sehen. Die FSF sieht dass der Begriff zweideutig ist – es könnte auch "umsonst" bedeuten, anstatt "frei" wie in "Freiheit" – findet aber dennoch, dass es alles in allem am besten passt, da andere Varianten im Englischen eigene Zweideutigkeiten haben. (In diesem Buch wird "frei" immer im Sinne von "Freiheit" verwendet, nicht im Sinn von "umsonst".)

Open Source Software
: Freie Software unter einem anderen Namen. Doch der Name spiegelt einen wichtigen philosophischen Unterschied wieder: "Open Source" wurde geprägt durch die Open Source Initiative (http://www.opensource.org/) als eine durchdachte Alternative zu "Freier Software" um diese attraktiver für Unternehmen zu machen; als Entwicklungsmethode und nicht als politische Bewegung. Vielleicht wollte man auch ein anderes Stigma verschwinden lassen, nämlich dass alles was nichts kostet auch von schlechter Qualität ist.

  Obwohl jede freie Lizenz auch "Open Source" ist, und bis auf wenige Ausnahmen auch andersherum, bleiben die meisten Leute bei einem Begriff. Meistens haben diejenigen die "Freie Software" verwenden einen eher politischen oder moralischen Standpunkt, während die die "Open Source" bevorzugen es entweder nicht als eine Frage der Freiheit sehen oder kein Interesse haben, es nach außen zu zeigen. Siehe auch „"Frei" kontra "Open Source"“ in Kapitel 1, *Einleitung* für eine genauere Geschichte dieses Schismas.

  Die Free Software Foundation hat eine vorzügliche – fürchterlich unobjektive, aber nuancierte und recht faire – Herkunft der beiden Begriffe unter http://www.fsf.org/licensing/essays/free-software-for-freedom.html veröffentlicht. Die Open Source Initiative verteilt ihre Sicht auf zwei Seiten:

[1] Im Wesentlichen ist hiermit das im Angloamerikanischen Raum verbreitete "Copyright" gemeint, auch wenn diese nicht ganz mit dem Deutschen Urheberrechtsgesetz vergleichbar ist, werden hier im weiteren nicht zwischen beiden Unterschieden.

http://www.opensource.org/advocacy/case_for_hackers.php#marketing und http://www.opensource.org/advocacy/free-notfree.php.

FOSS, F/OSS, FLOSS
: Wo zwei sind, darf ein drittes nicht fehlen. Genau dies passierte mit Begriffen für freie Software. Akademiker, die vielleicht präzise und umfassende Begriffe gegenüber eleganten bevorzugen, scheinen sich auf FOSS oder F/OSS für "Freie / Open Source Software" zu einigen. Eine andere Variante ist FLOSS für "Freie / Libre Open Source Software" (*libre* steht in vielen Sprachen für "frei" jedoch ohne die Zweideutigkeiten; siehe auch http://de.wikipedia.org/wiki/FLOSS.

  All diese Begriffe bedeuten eigentlich das gleiche: Software die von jedem verändert und verteilt werden kann, manchmal – aber nicht immer – mit der Einschränkung, dass abgeleitete Arbeiten wieder unter den gleichen Bedingungen verteilt werden.

DFSG-verträglich
: Verträglich mit den Debian-Richtlinien für Freie Software (Debian Free Software Guidelines) (http://www.debian.org/social_contract.de.html#guidelines). Dies ist ein weit verbreiteter Test um zu prüfen ob eine Lizenz wirklich frei (open source, libre, etc.) ist. Das Ziel des Debian-Projekts ist ein vollständig freies Betriebssystem, so dass niemand der es installiert daran zweifeln muss ob er das Recht hat einen Teil oder das ganze System zu verändern oder zu verteilen. Die Debian-Richtlinien für Freie Software bestimmen die lizenzrechtlichen Anforderungen die eine Software erfüllen muss um in Debian aufgenommen zu werden. Da das Debian-Projekt gründlich darüber nachgedacht hat, wie man so einen Test erstellt, kamen dabei sehr robuste Richtlinien (siehe http://de.wikipedia.org/wiki/DFSG) heraus, und soweit ich weiss, gibt es weder von der Free Software Foundation noch von der Open Source Initiative ernsthafte Bedenken dagegen. Wenn man weiss, dass eine Lizenz DFSG-verträglich ist, kann man sicher sein, dass sie alle wichtigen Freiheiten einräumt (z.B. die Möglichkeit ein neues Projekt abzuspalten, auch gegen den Willen des Originalautors. Alle hier diskutierten Lizenzen sind verträglich mit der DFSG.

OSI-approved
: Anerkannt durch die Open-Source-Initiative. Dies ist ein anderer oft verwendeter Test ob eine Lizenz alle nötigen Freiheiten erlaubt. Die OSI-Definition von Open-Source-Software basiert auf den DFSG, und beinahe jede Lizenz die die eine Definition erfüllt, erfüllt auch die andere. Über die Jahre gab es einige Ausnahmen, die aber nur seltene Lizenzen betrafen und keine von diesen ist hier relevant. Im Gegensatz zum Debian Projekt hat die OSI eine Liste aller jemals anerkannten Lizenzen unter http://www.opensource.org/licenses/. Damit ist "OSI-anerkannt" eindeutig: Entweder eine Lizenz ist auf der Liste oder eben nicht.

  Die Free Software Foundation stellt auch eine Liste mit Lizenzen unter http://www.fsf.org/licensing/licenses/license-list.html zur Verfügung. Die FSF ordnet die Lizenzen aber nicht nur danach ein, ob sie frei sind, sondern auch danach ob sie kompatibel mit der GNU General Public License ist. Kompatibilität mit der GPL ist ein wichtiges Thema, das in „Die GPL und Lizenz-Kompatibilität“ später in diesem Kapitel besprochen wird.

Proprietär, Closed Source
: Das Gegenteil von "Frei" oder "Open Source". Es steht für Software die unter traditionellen, kostenpflichtigen Lizenzbedingungen (Der Nutzer zahlt pro Kopie der Software) ausgeliefert werden oder zu Bedingungen die restriktiv genug sind, die Dynamik von Open Source zu unterbinden. Auch Software die "umsonst" zur Verfügung gestellt wird, kann proprietär sein, wenn die Lizenz die freie Verteilung und Veränderung der Software verbietet.

  Im allgemeinen sind "proprietär" und "Closed Source" synonym. "Closed Source" impliziert jedoch , dass der Quellcode nicht einmal eingesehen werden kann. Da dies bei der meisten proprietären Software so ist, werden die beiden Varianten meistens nicht unterschieden. Manchmal wird jedoch proprietäre Software veröffentlicht, deren Lizenz es erlaubt, den Quellcode einzusehen. Ver-

wirrender Weise wird dies dann auch "Open Source" oder "Fast Open Source" genannt, doch das ist irreführend. Die *Sichtbarkeit* des Quellcodes ist nicht entscheidend; wichtig ist was man damit tun darf. Die Unterschiede zwischen "proprietär" und "Closed-Source" sind also irrelevant, und man kann die Begriffe synonym verwenden.

Manchmal wird *kommerziell* als Synonym für proprietär verwendet, doch genaugenommen ist das nicht dasselbe. Denn Freie Software kann verkauft werden, solange die Käufer ihre Kopien weitergeben dürfen. Sie kann auch auf anderen Wegen kommerzialisiert werden, zum Beispiel durch Support-Verträge, Dienstleistungen und Zertifizierungen. Es gibt millionenschwere Unternehmen die mit freier Software Geld verdienen, sie richtet sich also weder gegen Kommerzialisierung, noch gegen Unternehmen. Andererseits *ist* sie von Natur aus gegen proprietäre Software. Dies ist der Punkt warum sie sich von althergebrachten "pay per copy" Lizenzmodellen unterscheidet.

public domain
: Niemand hat das Recht, das Kopieren der Software einzuschränken. Dies bedeutet aber nicht, dass die Software keinen Urheber hat. Der Urheber macht aber von seinen Verwertungsrechten keinen Gebrauch. Die Tatsache, dass er die Rechte der Öffentlichkeit einräumt, ändert nichts an seiner Urheberschaft.

  Wenn eine Arbeit "public domain" ist, können Teile davon in anderen lizenzgeschützten Werken benutzt werden. Dann steht *diese Kopie* der Arbeit unter derselben Lizenz wie das Gesamtwerk. Das betrifft aber nicht die Verfügbarkeit der Originalarbeit die immer noch "public domain" ist. Etwas der Öffentlichkeit zu übergeben ist also ein Weg eine Software "frei" zu machen, gemäß den Richtlinien der meisten Organisationen die freie Software zertifizieren. Trotzdem gibt es gute Gründe, eine Lizenz zu verwenden, statt die Software einfach mit allen Rechten und ohne Pflichten herauszugeben: Selbst bei freier Software können Einschränkungen sinnvoll sein, nicht nur für den Urheber, sondern auch für den Lizenznehmer, wie der nächste Abschnitt zeigt.

copyleft
: Eine Lizenz, die das Urheberrecht verwendet um den entgegengesetzten Effekt zu erzielen. Je nach dem wen man fragt, sind Lizenzen gemeint, die die Rechte die wir hier diskutieren einräumen, oder genauer: Lizenzen die diese Rechte nicht nur einräumen sondern sie *erzwingen*, indem sie verlangen, dass diese Rechte mit der Arbeit wandern. Die FSF verwendet ausschließlich die zweite Form; ansonsten steht es Unentschieden: Viele verwenden den Begriff wie die FSF, andere – auch Autoren der Massenmedien – verwenden die erste Variante. Der Unterschied zwischen den Varianten ist vielen aber nicht klar.

  Das bekannteste Beispiel für die genauere Definition ist die GNU General Public License, die verlangt, dass jede abgeleitete Arbeit wieder unter der GPL stehen muss; siehe „Die GPL und Lizenz-Kompatibilität“ weiter unten .

Lizenzaspekte

Obwohl es viele freie Lizenzen gibt, sagen sie in den wichtigen Punkten doch alle dasselbe: jeder den Quellcode bearbeiten kann, jeder die Software im Original oder modifiziert verbreiten darf und dass die Rechteinhaber keine Garantie oder Gewährleistung übernehmen (Haftungsausschluss ist vor allem wichtig, wenn Leute veränderte Software einsetzen ohne es zu wissen.) Die Unterschiede zwischen den Lizenzen reduzieren sich auf einige wenige Punkte:

Kompatibilität mit proprietären Lizenzen
: Einige freie Lizenzen gestatten es, den Code in proprietären Programmen zu verwenden. Das betrifft aber nicht die Lizenz des proprietären Programms: es ist genauso proprietär wie vorher, nur enthält es Code von einer nicht-proprietären Quelle. Beispiele für Lizenzen die das gestatten sind: Die Apache-Lizenz, die Lizenz des X-Konsortiums, Lizenzen im BSD- oder MIT-Stil.

Kompatibilität mit anderen freien Lizenzen
: Die meisten freien Lizenzen sind kompatibel zueinander: Das bedeutet, dass Code der unter Lizenz A entstanden ist und mit Code der unter Lizenz B entstanden kombiniert wird unter jede der beiden Lizenzen gestellt werden kann, ohne die einzelnen Lizenzbedingungen zu verletzen. Die große Ausnahme ist die GNU General Public License, die fordert, dass jede Arbeit die GPL-lizenzierten Code verwendet auch unter der GPL stehen muss, und keine weiteren Einschränkungen hinzufügen darf. Die GPL ist daher nur mit einigen freien Lizenzen kompatible. Das wird in „Die GPL und Lizenz-Kompatibilität“ genauer diskutiert.

Namensnennung erzwingen
: Einige freie Lizenzen verlangen, dass bei Verwendung des Codes eine Notiz erscheinen muss (die Art und Weise ist typischerweise vorgeschrieben), die den Autor oder Rechteinhaber nennt. Diese Lizenzen sind oft kompatibel zu proprietären, da sie nicht dazu zwingen, die abgeleitete Arbeit wieder unter eine freie Lizenz zu stellen, sondern nur dazu das freie Original zu nennen.

Schutz der Marke
: Eine Variante die Namensnennung zu erzwingen. Lizenzen mit einer Bestimmung zum Schutz der Marke, verlangen, dass der Name der Originalsoftware (oder der Rechteinhaber) ohne Genehmigung *nicht* in abgeleiteten Arbeiten verwendet werden darf. Auch wenn man beim Erzwingen der Namensnennung darauf besteht, einen bestimmten Namen zu verwenden, während man beim Schutz der Marke darauf besteht einen bestimmten Namen nicht zu verwenden, drücken doch beide Ansätze dasselbe aus: Der gute Ruf des Originals soll erhalten und verbreitet, aber nicht verwässert werden.

patent snapback
: Both the GNU General Public License version 3 and the Apache License version 2 contain language designed to prevent people from using patent law to take away the rights granted (under copyright law) by the licenses. They require contributors to grant patent licenses along with their contribution, covering any patents licenseable by the contributor that would be infringed by their contribution (or by the incorporation of their contribution into the work as a whole). Then they go further: if someone using software under the license initiates patent litigation against another party, claiming that the covered work infringes, the initiator automatically *loses* all the patent grants otherwise provided for that work by the license, and in the case of the GPLv3 loses their right to distribute under the license altogether.

Schutz der "künstlerischen Integrität"
: Einige Lizenzen (z. B. die Perl Artistic License, oder Donald Knuths TeX License) erfordern, dass bei der Veränderung und Verbreitung klar zwischen der reinen Originalversion und Veränderungen getrennt werden soll. Es werden im Grunde die gleichen Freiheiten wie bei anderen freien Lizenzen eingeräumt, aber es wird verlangt, dass die Integrität des Originalcodes leicht zu überprüfen ist. Diese Lizenzen haben sich nicht über die Sprachen für die sie geschaffen wurden hinaus nicht durchgesetzt und werden in diesem Kapitel nicht weiter erwähnt; sie wurden hier nur der Vollständigkeit halber erwähnt.

Die meisten dieser Bestimmungen schließen sich nicht gegenseitig aus. Die Gemeinsamkeit ist, dass der Lizenznehmer bestimmte Pflichten erfüllen muss für das Recht den Code zu verwenden oder weiter zu verbreiten. Zum Beispiel wollen einige Projekte ihren Namen und ihre Reputation mit dem Code verbreiten, und das ist es ihnen Wert eine zusätzliche Bestimmung zum Markenschutz zu erlassen. Je nachdem wie streng die Bestimmung ist, werden so manche Benutzer ein Programm nehmen das weniger strenge Bestimmungen hat.

Die GPL und Lizenz-Kompatibilität

Die schärfste Trennlinie verläuft zwischen Lizenzen die kompatibel zu proprietärer Software sind solchen die es nicht sind; also zwischen der GNU General Public License und dem Rest. Da das vorrangi-

ge Ziel der GPL-Autoren die Verbreitung freier Software ist, haben sie die Lizenz so gestaltet, dass es unmöglich ist, GPL-lizenzierten Code in proprietären Programmen zu verwenden. Siehe vor allem diese beiden Absätze der GPL (siehe auch http://www.fsf.org/licensing/licenses/gpl.html):

1. Jedes abgeleitete Werk, also alles, was eine gewisse Menge GPL-lizenzierten Code enthält, muss wieder unter die GPL gestellt werden.

2. Es dürfen keine zusätzlichen Einschränkungen hinzugefügt werden, weder auf das Originalwerk, noch auf abgeleitete Arbeiten. Der genaue Wortlaut ist: "Sie dürfen keine zusätzlichen Einschränkungen bzgl. der Ausübung der unter dieser Lizenz gewährten oder zugesicherten Rechte vornehmen."[2]

Mit diesen Bedingungen schafft es die GPL-Freiheit ansteckend zu machen. Steht ein Programm erst einmal unter der GPL, sind die Bedingungen zur Weitergabe *viral* – sie verbreiten sich auf jede Software die diesen Code verwendet. Das macht es unmöglich GPL-lizenzierten Code in proprietären Programmen zu verwenden. Aber genau diese Sätze machen die GPL inkompatible mit einigen anderen freien Lizenzen. Das passiert üblicherweise, wenn die andere Lizenz z.B. verlangt, die Originalautoren zu nennen, denn das widerspricht der Bedingung keine weiteren Einschränkungen einzuführen. Aus Sicht der Free Software Foundation sind diese Folgen wünschenswert, oder zumindest nicht zu bedauern. Die GPL hält die Software nicht nur frei, sondern macht sie zu einem Agenten der versucht auch *andere* Programme dazu zu bringen Freiheit zu fordern.

Die Frage ob dies ein guter Weg ist, freie Software zu verbreiten ist einer der beständigsten heiligen Kriege im Internet (siehe „Vermeiden Sie Heilige Kriege“ in Kapitel 6, *Kommunikation*) und wir werden hier nicht näher darauf eingehen. Für uns ist wichtig, dass Kompatibilität mit der GPL wichtig ist, wenn man eine Lizenz auswählt. Die GPL ist bei weitem die populärste Open-Source-Lizenz: nach http://freshmeat.net/stats/#license liegt sie bei 68%, während die nächst-populäre nur 6% erreicht. Wenn man möchte, dass der Code frei mit GPL-lizenziertem Code gemischt werden kann – und davon gibt es eine Menge – sollte man eine GPL-kompatible Lizenz nehmen. Die meisten Lizenzen die mit der GPL kompatibel sind, sind auch mit proprietären Lizenzen kompatibel, das heisst Code der unter so einer Lizenz steht kann in sowohl in GPL-lizenzierten als auch in proprietären Programmen verwendet werden. Die Ergebnis solcher Kombinationen wären natürlich nicht kompatibel zueinander, da das eine unter der GPL stünde, das andere aber unter einer closed-source Lizenz. Das bezieht sich aber nur auf abgeleitete Werke, nicht auf die ursprüngliche Software.

Glücklicherweise hat die Free Software Foundation eine Liste die aufführt, welche Lizenzen kompatibel zur GPL sind und welche nicht (siehe http://www.gnu.org/licenses/license-list.html). Alle Lizenzen die wir hier diskutiert haben sind auf dieser Liste, auf der einen oder anderen Seite.

Die Wahl einer Lizenz

Wenn man eine Lizenz wählt, die man für das Projekt benutzen will, sollte man wenn irgendwie möglich, eine bereits existierende wählen. Exisitierende Lizenzen sind aus zwei Gründen besser:

- Bekanntheit. Wenn Sie eines der drei oder vier verbreitetsten Lizenzen benutzen, wird man nicht das Gefühl haben, sich um den rechtlichen Rahmen kümmern zu müssen, wenn sie Ihren Code benutzen wollen. Für diese Lizenz, haben sie das nämlich schon vor langem gemacht.

- Qualität. Wenn Sie nicht gerade ein paar Rechtsanwälte parat haben, ist es unwahrscheinlich, dass Sie sich eine rechtlich wasserdichte Lizenz ausdenken. Die erwähnten Lizenzen, sind mit viel Mühe und Erfahrung entstanden; Wenn Ihr Projekt keine äußerst außergewöhnlichen Bedürfnisse hat, werden sie es wahrscheinlich nicht besser machen.

[2] Anm. der Übersetzer: Die Übersetzung der GPL stammt von http://www.gnu.de/documents/gpl.de.html, das englische Original findet man im §10 unter http://www.fsf.org/licensing/licenses/gpl.html

Siehe „Eine Lizenz für Ihre Software“ in Kapitel 2, *Der Einstieg*, um eine dieser Lizenzen auf Ihr Projekt anzuwenden.

Die MIT- / X-Window-System-Lizenz

Wenn Sie wollen, dass Ihr Code die größtmögliche Anzahl an Entwickler und Derivate erreicht, und es sie nicht stört, wenn Ihr Code in proprietären Anwendungen benutzt wird, sollten Sie die MIT- / X-Window-System-Lizenz wählen (der Name kommt durch seine Nutzung für den Code vom ursprünglichen X-Window-System veröffentlicht von dem Massachusetts Institute of Technology). Die grundsätzliche Botschaft dieser Lizenz ist, "Du kannst mit diesem Code machen was du willst." Sie ist mit der GNU GPL kompatibel, sie ist kurz, einfach und leicht zu verstehen:

```
Copyright (c) <Jahr> <copyright Inhaber>

Hiermit wird unentgeltlich, jeder Person, die eine Kopie der Software
und der zugehörigen Dokumentationen (die "Software") erhält, die
Erlaubnis erteilt, uneingeschränkt zu benutzen, inklusive und ohne
Ausnahme, dem Recht, sie zu verwenden, kopieren, ändern, fusionieren,
verlegen, verbreiten, unter-lizenzieren und/oder zu verkaufen, und
Personen, die diese Software erhalten, diese Rechte zu geben, unter
den folgenden Bedingungen:

Der obige Urheberrechtsvermerk und dieser Erlaubnisvermerk sind in
alle Kopien oder Teilkopien der Software beizulegen.

DIE SOFTWARE WIRD OHNE JEDE AUSDRÜCKLICHE ODER IMPLIZIERTE GARANTIE
BEREITGESTELLT, EINSCHLIESSLICH DER GARANTIE ZUR BENUTZUNG FÜR DEN
VORGESEHENEN ODER EINEM BESTIMMTEN ZWECK SOWIE JEGLICHER
RECHTSVERLETZUNG, JEDOCH NICHT DARAUF BESCHRÄNKT. IN KEINEM FALL SIND
DIE AUTOREN ODER COPYRIGHTINHABER FÜR JEGLICHEN SCHADEN ODER SONSTIGE
ANSPRUCH HAFTBAR ZU MACHEN, OB INFOLGE DER ERFÜLLUNG VON EINEM
VERTRAG, EINEM DELIKT ODER ANDERS IM ZUSAMMENHANG MIT DER BENUTZUNG
ODER SONSTIGE VERWENDUNG DER SOFTWARE ENTSTANDEN.[3]
```

Die GNU GPL

Wenn Sie es vorziehen, dass Ihr Code nicht in Proprietären Anwendungen verwendet wird, oder es Ihnen zumindest egal ist, ob es in proprietären Anwendungen verwendet werden kann, sollten Sie die GNU General Public License (de. Allgemeine Öffentliche Lizenz) verwenden (http://www.fsf.org/licensing/licenses/gpl.html). Die GPL ist wahrscheinlich die heute am weitesten verbreitete Lizenz für freie Software; diese breite Bekanntheit, ist an und für sich schon eine der großen Vorteile der GPL.

Wenn man eine Bibliothek schreibt, die hauptsächlich dazu gedacht ist, als Teil anderer Anwendungen verwendet zu werden, sollten sie genau überlegen, ob die Einschränkungen die von der GPL vorgegeben werden, mit den Zielen von Ihrem Projekt vereinbar sind. Manchmal – zum Beispiel, wenn Sie versuchen eine konkurrierende, propritäre Bibliothek, mit der gleichen Funktion, zu ersetzen – kann es aus strategischer Sicht sinnvoller sein, Ihre Lizenz derart zu wählen, dass es in proprietären Anwendungen verwendet werden kann, auch wenn Sie das sonst nicht so gerne sehen würden. Die Free Software Foundation hat sogar eine alternative zur GPL geschrieben, für genau diese Umstände: die *GNU Library GPL*, später zur *GNU Lesser GPL* umbenannt (meistens wird das Kürzel *LGPL* verwendet). Die LGPL hat weniger Einschränkungen als die GPL und kann leichter min nicht freier Software zusammen

[3]Diese freie Übersetzung, von http://de.wikipedia.org/wiki/MIT-Lizenz wird zum besseren Verständnis benutzt, es ist keine offizielle oder im rechtlichen Sinne Anerkannte Übersetzung. In Ihrer Anwendung sollten die Originalfassung verwenden, die Sie hier finden: http://www.opensource.org/licenses/mit-license.php

benutzt werden. Es ist jedoch auch etwas komplex und erfordert etwas Zeit zum Verständnis, wenn Sie also nicht die GPL verwenden, empfehle ich einfach ein Lizenz der Art von MIT/X zu verwenden.

The GNU Affero GPL: A Version of the GNU GPL for Server-Side Code

In 2007, the Free Software Foundation released a variant of the GPL called the *GNU Affero GPL* (http://www.fsf.org/licensing/licenses/agpl.html)[4]. Its purpose is to enforce GPL-like sharing provisions on the growing number of companies that offered hosted services—software that runs on their servers, that users interact with only over the network, and that therefore is never directly distributed to users as executable or source code. Many such services had been using GPL'd software, often with modifications, yet didn't have to share their changes with the world because they weren't distributing any code.

The GNU AGPLv3's solution to this was to take the regular GPL and add a "Remote Network Interaction" clause, stating "*...if you modify the Program, your modified version must prominently offer all users interacting with it remotely through a computer network ... an opportunity to receive the Corresponding Source of your version ... at no charge, through some standard or customary means of facilitating copying of software.*" This expanded the GPL's enforcement powers into the new world of application service providers. The Free Software Foundation recommends that the GNU AGPLv3 be used for any software that will commonly be run over a network.

Note that the AGPLv3 is not directly compatible with GPLv2 (though it is compatible with GPLv3, of course). However, most software licensed under the GPLv2 includes the "or any later version" clause anyway, so you can just shift it to GPLv3 if and when you need to mix it with AGPLv3 code. However, if you need to mix with programs licensed strictly under the GPLv2 (that is, without the "or any later version" clause), then the AGPLv3 won't work.

Although the history of the AGPLv3 is a bit complicated, the license itself is simple: it's just the GPLv3 with one extra clause about network interaction. The Wikipedia article on the AGPLv3 is excellent: http://en.wikipedia.org/wiki/Affero_General_Public_License

Ist die GPL frei oder nicht frei?

Eine der Konsequenzen der GPL, ist die Möglichkeit – zwar klein aber dennoch vorhanden – sich oder oder Ihr Projekt in einer zu leiten, ob die GPL wirklich "frei" ist oder nicht, in Anbetracht der Einschränkungen auf die Verwendung vom Code – nämlich die Einschränkung, dass Code nicht unter irgend einer anderen Lizenz verbreitet werden darf. Für manche, bedeutet diese Einschränkung, dass die GPL weniger "frei ist" als die toleranteren Lizenz, wie die MIT/X Lizenz. Dieses Argument führt natürlich meistens darauf, dass "mehr Freiheit" zwangsläufig besser ist als "weniger Freiheit" (wer ist schließlich nicht für Freiheit?), woraus folgt, dass diese Lizenzen besser sind, als die GPL.

Diese Debatte ist ein weiteres der beliebten heiligen Kriege (siehe „Vermeiden Sie Heilige Kriege“ im Kapitel Kapitel 6, *Kommunikation*). Vermeiden Sie die Beteiligung daran, zumindest in den Foren des Projekts. Versuchen Sie nicht zu beweisen, dass die GPL weniger, gleich viel oder mehr Freiheit bietet als andere Lizenzen. Betonen Sie statt dessen, die spezifischen Gründe, die Ihr Projekt zur Wahl der GPL geführt hat. Wenn ihre weite Verbreitung eines der Gründe war, sollten Sie das sagen. Wenn die Übertragung einer freien Lizenz auf Abgeleitete Werke auch eines der Gründe war, sollten Sie das ebenfalls erwähnen, lassen Sie sich aber nicht in eine Diskussion darüber verleiten, ob der Code dadurch mehr oder weniger "Frei" ist. Freiheit ist ein komplexes Thema und es macht wenig Sinn darüber zu reden, wenn Begriffe als Vorwand benutzt werden vor dem Wesentlichen.

[4] The history of the license and its name is a bit complicated. The first version of the license was originally released by Affero, Inc, who based it on the GNU GPL version 2. This was commonly referred to as the AGPL. Later, the Free Software Foundation decided to adopt the idea, but by then they had released version 3 of their GNU GPL, so they based their new Affero-ized license on that and called it the "GNU AGPL". The old Affero license is more or less deprecated now. If you want Affero-like provisions, you should use the GNU version. To avoid ambiguity, call it the "AGPLv3", the "GNU AGPL", or for maximum precision, "GNU AGPLv3".

Da ich hier jedoch ein Buch schreibe und nicht eine E-Mail an einem Verteiler, gebe ich zu, dass nie das Argument "die GPL ist nicht frei" nie ganz verstanden habe. Die einzige Einschränkung die von der GPL auferlegt wird, ist dass andere daran gehindert werden, *weitere* Einschränkungen zu machen. Zu behaupten, dass sie deswegen weniger frei wäre, erschien mir immer als ob man sagen würde, dass ein Gesetz gegen Sklaverei die Freiheit reduziert, da es manche daran hindert, Sklaven zu besitzen.

(Im übrigen, wenn Sie sich tatsächlich auf eine Debatte darüber einlassen wollen, machen Sie die Sache durch aufrührerischen Gleichnisse nicht schlimmer als nötig.)

Wie sieht es mit der BSD-Lizenz aus?

Eine nicht unwesentliche Menge an freie Software wird unter einer *BSD Lizense* veröffentlicht. Die ursprüngliche BSD Lizenz wurde für die Berkeley Software Distribution verwenden, indem die Universität von Kalifornien wichtige Teile einer Unix Implementierung veröffentlichte. Diese Lizenz (der genaue Text findet man im Abschnitt 2.2.2 auf http://www.xfree86.org/3.3.6/COPYRIGHT2.html#6) wurde im Geiste der MIT/X Lizenz geschrieben, bis auf eine Klausel:

> *Jegliche Werbematerialien, die Funktionen oder die Nutzung dieser Software erwähnen, müssen die folgende Anerkennung einschließen: Dieses Produkt beinhaltet Software die von der Lawrence Berkeley Laboratory der Universität von Kalifornien entwickelt wurde.*

Diese Klausel machte die Ursprüngliche BSD-Lizenz nicht nur inkompatibel zur GPL, sonder setzte auch noch ein gefährliches Beispiel: während andere Organisationen ähnlichen Klausel in *ihre* freie Software schrieben – wobei sie den Namen Ihrer Organisation anstatt von "Lawrence Berkeley Laboratory der Universität von Kalifornien" einsetzten – wurde jedem der die Software verbreiten wollte, eine immer größere Bürde auferlegt, jede einzelne Firma zu erwähnen. Zum Glück, wurden viele Projekte, sich dieser Problematik bewusst und ließen diese Klausel einfach fallen. 1999 schließlich sogar die Universität von Kalifornien.

Das Ergebnis ist die überarbeitete BSD-Lizenz, die einfach die ursprüngliche BSD-Lizenz ist, ohne die Werbeklausel. Diese Geschichte macht den Begriff BSD-Lizense jedoch etwas mehrdeutig: ist damit das Original oder die überarbeitete Fassung gemeint? Ich ziehe deshalb die MIT/X Lizense vor, die im wesentlichen gleichwertig ist, und nicht unter dieser Mehrdeutigkeit leidet. Es gibt jedoch vielleicht ein Grund die BSD Lizenz der MIT/X Lizenz vorzuziehen, die BSD-Lizenz beinhaltet nämlich diese Klausel:

> *Weder der Name der <ORGANIZATION> noch die Namen seiner Teilhaber dürfen zur Werbung oder Förderung von Produkten verwendet werden, die von dieser Software abgeleitet werden, ohne explizite vorherige schriftliche Zustimmung.*

Es ist ohne ein solche Klausel nicht klar, ob jemand der die Software erhält, das Recht hat, den Namen des Lizenzgebers zu verwenden, die Klausel lässt daran aber keinen Zweifel. Organizationen, die sich um Markenrecht sorgen machen, mögen deshalb die überarbeitete BSD-Lizenz vorziehen. Im allgemeinen, impliziert eine liberale Urheberrechts-Lizenz jedoch nicht, dass Empfänger das Recht haben ihre Marken zu benutzen oder zu verwässern – Urheberrecht und Markenrechte sind zwei paar Schuhe.

Wenn Sie die überarbeitete BSD-Lizenz verwenden wollen, finden sieh hier eine Vorlage http://www.opensource.org/licenses/bsd-license.php.

Zuweisung von Urheberrechten

Bei freier Software, mit Codebeiträge von vielen Beteiligten, gibt es drei Möglichkeiten mit der Zuweisung von Urheberrechten umzugehen. Die erste ist das Thema ganz und gar zu ignorieren (Kann ich

nicht empfehlen). Die zweite ist eine *contributor license agreement (de. Lizenzvereinbarung für Beitragende)* (*CLA*) von jedem Beteiligten an dem Projekt einzufordern, die dem Projekt explizit das Recht erteilt, den Code von dieser Person nutzen zu dürfen. Das reicht für die meisten Projekte aus und das schöne daran, ist dass mache Gerichtsbezirke erlauben, dass eine CLA per E-Mail eingereicht wird. Die dritte Möglichkeit ist, die eigentlichen urheberrechtlichen Zuweisungen von den Beitragenden zu bekommen, sodass das Projekt (also irgend eine rechtliche Person, üblicherweise ein gemeinnützige Organisation) der Inhaber aller Urheberrechte ist. Rechtlich gesehen, ist das die am ehesten wasserdichte Lösung, aber auch die am Aufwändigsten für die Beteiligten, weshalb nur wenige Projekte darauf bestehen.[5]

Beachten Sie, dass selbst bei einem zentralen Inhaber des Urheberrechts, der Code immernoch frei ist, denn Open-Source-Lizenzen geben dem Urheber nicht das Recht im nachhinein alle Kopien proprietär zu machen. Selbst wenn das Projekt als rechtliche Person, eine kehrtwende machen würde, und anfangen würde jeglichen Code unter einer restriktive Lizenz zu veröffentliche, würde das kein Problem für die öffentliche Gemeinte bereiten. Die anderen Entwickler würden einfach einen Fork anfangen, ausgehend von der letzten freien Version des Quellcodes, und von da an weitermache, als wäre nichts geschehen. Da sie wissen, dass ihnen diese Möglichkeit offensteht, sind die meisten Freiwilligen kooperative wenn man sie darum bittet eine CLA oder die Übertragung des Urheberrechts zu unterschreiben.

Keine Zuweisung von Urheberrecht

Die meisten Projekte bitten Freiwillige nicht um eine CLA oder der Übertragung des Urheberrechts. Stattdessen, nehmen sie Code an, solange es klar scheint, dass der Beitragende vorsah es in das Projekt zu integrieren.

Üblicherweise ist das auch in Ordnung. Ab und zu, kann sich jemand dazu entschließen auf Urheberrechtsverletzung zu klagen, mit der Behauptung, dass sie die eigentlichen Inhaber von dem in Frage stehenden Code sind und sie sich nie damit einverstanden erklärt haben, dass es mit dem Projekt unter einer Open-Source-Lizenz veröffentlicht wird. Die SCO-Gruppe hat sowas beispielsweise mit dem Linux-Projekt gemacht, siehe http://de.wikipedia.org/wiki/SCO_gegen_Linux für mehr zu diesem Thema. Wenn das passiert, wird das Projekt keine Dokumentation haben, die zeigt, dass der Urheber formal das Recht erteilt hat den Code zu benutzen, was eine rechtliche Verteidigung schwieriger gestalten könnte.

Lizenzvereinbarung der Beitragenden (CLA)

CLAs bieten wahrscheinlich den besten Kompromiss zwischen Sicherheit und Bequemlichkeit. Eine CLA ist üblicherweise ein elektronisches Formular, dass vom Entwickler ausgefüllt wird und an das Projekt gesandt wird. In viele Gerichtsbezirken, reicht es sie per E-Mail einzureichen. Eine sicher digitale Signatur unter Umständen erforderlich sein; fragen Sie einen Anwalt welche Methode für Ihr Projekt angemessen ist.

Die meisten Projekte benutzen meistens zwei verschiedene CLAs, eines für Einzelpersonen und eines für Organisationen. In beiden, steht im wesentlichen jedoch das gleiche: der Beitragende gibt dem Projekt *"...unbefristete, weltweite, uneingeschränkte, unentgeltliche, gebührenfreie, unkündbare Urheberrecht um die Beiträge und davon abgeleitete Werke zu kopieren, verbreiten und unter neuer Lizenz herauszugeben."* Sie sollten natürlich wieder einen Anwalt darum bitten, jede CLA zu überprüfen, wenn Sie es jedoch schaffen all diese Begriffe hineinzubekommen, sollte es kein Problem sein.

Wenn sie von den Beitragenden CLAs anfordern, sollten Sie betonen, dass Sie *nicht* um eine tatsächliche Übertragung des Urheberrechts bitten. Viele CLAs fangen sogar damit an, die Leser daran zu erinnern:

[5]Die Übertragung von bestimmten Rechten ist, im Gegensatz zum amerikanischen copyright, nach deutschem Urheberrecht nicht möglich. Aus diesem Grund sollte bei Open-Source-Lizenzen genau geprüft werden, wie diese nach deutschem Recht einzuordnen sind.

> *Es handelt sich hier ausschließlich um eine Lizenzvereinbarung; es überträgt kein rechtlichen Zuspruch des Urheberrechts und ändert nichts an Ihrem Recht Ihre Beiträge für irgend welche anderen Zwecke zu verwenden.*

Hier einige Beispiele:

- CLAs für Einzelpersonen:
  - http://apache.org/licenses/icla.txt
  - http://code.google.com/legal/individual-cla-v1.0.html
- CLAs für Organisationen:
  - http://apache.org/licenses/cla-corporate.txt
  - http://code.google.com/legal/corporate-cla-v1.0.html

Übertragung vom Urheberrecht

Die Übertragung vom Urheberrecht bedeutet, dass die Beitragende dem Projekt den Besitz der Urheberrechte auf ihre Beiträge zuspricht. Im Idealfall wird das in Papierform gemacht und entweder per Fax oder per Schneckenpost an dem Projekt geschickt.

Manche Projekte bestehen auf die vollständige Übertragung des Urheberrechts, da es nützlich sein kann wenn jeglicher Code einer rechtlichen Person gehört, um die Konditionen der Open-Source-Lizenz vor Gericht durchzusetzen. Wenn keine einzelne Person das Recht hat das zu machen, müssen vielleicht alle Beteiligte kooperieren, von denen vielleicht manche keine Zeit haben oder schlicht nicht erreichbar sind, wenn das Thema aufkommt.

Verschiedene Organisationen sind unterschiedlich streng, wenn es darum geht, diese Rechte von den Beitragenden einzufordern. Manche verlangen einfach eine informelle Stellungnahme von dem Freiwilligen auf einem öffentlichen E-Mail-Verteiler – so etwas wie "Hiermit übertrage ich das Urheberrecht auf diesem Quellcode auf das Projekt, sodass es unter der selben Lizenz veröffentlicht wird, wie der übrige Quellcode." Zumindest, hat ein Anwalt mit dem ich gesprochen habe, mir gesagt, das wäre ausreichend, vermutlich weil es sich um einen Kontext handelt, indem die Übertragung vom Urheberrecht üblich ist und sowieso erwartet wird, und weil es sich auf Seiten des Projekts um ein Akt im guten Glauben handelt, um die wahren Absichten des Freiwilligen zu ermitteln. Die Free Software Foundation andererseits, ist ein Beispiel für das andere Extrem: Sie verlangen von Freiwilligen ein Dokument Physikalisch zu unterschreiben und zu versenden, indem das Urheberrecht formal übertragen wird, manchmal schon für einen einzigen Beitrag, manchmal für aktuelle und zukünftige Beiträge. Wenn der Entwickler ein Angestellter ist, verlangt die FSF, dass der Betrieb es ebenfalls unterschreibt.

Die Paranoia der FSF ist verständlich. Wenn jemand die Bedingungen der GPL verletzt, indem sie eine Teil ihrer Software in eine Proprietäre Anwendung verwenden, wird die FSF dagegen vor Gericht ankämpfen müssen, und in dem Fall, wollen sie, dass ihre Uhrheberrechte, so wasserdicht wie möglich sind. Da die FSF der Inhaber vom Uhrheberrecht für viele beliebte Anwendungen ist, sehen sie das als eine ernstzunehmende Möglichkeit an. Ob ihre Organisation im gleichen Maße peinlich genau arbeiten muss, müssen Sie mit Beratung durch Anwälte entscheiden. Im Allgemeinen, sollten Sie CLAs bevorzugen, wenn es keinen bestimmte Grund das Urheberrecht vollständig zu Übertragen; dass macht die Angelegenheit einfacher für alle beteiligten.

Doppelte Lizenzierung

Manche Projekte versuchen sich durch ein duales Lizenzmodell zu finanzieren, bei dem proprietäre Derivate dem Inhaber des Urheberrechts bezahlen können um den Code in ihre Projekte verwenden zu

dürfen, der Code für andere Open-Source-Projekte aber unter einer frei bleibt. Das funktioniert natürlich meistens besser für Code-Bibliotheken als für einzelnen Anwendungen. Die genauen Bedingungen unterscheiden sich von Fall zu Fall. Oftmals ist die freie Lizenz die GNU GPL, da es bereits andere daran hindert, den davon abgedeckten Code in ihre Proprietären Anwendungen, ohne Erlaubnis des Urhebers zu benutzen, manchmal ist es jedoch eine eigens geschriebene Lizenz mit dem gleichen Effekt. Ein Beispiel für ersteres ist die hier beschriebene MySQL-Lizenz http://www.mysql.com/company/legal/licensing/; ein Beispiel für Letzteres ist die hier beschriebene Lizenzierung der Sleepycat Software http://www.oracle.com/technology/software/products/berkeley-db/htdocs/licensing.html.

Sie mögen sich vielleicht fragen: Wie kann ein Urheber eine proprietäre Lizenz für eine Gebühr anbieten, wenn die Bedingungen der GNU GPL vorgibt, dass der Code unter keiner restriktiveren Lizenz verbreitet werden darf? Die Antwort darauf, ist dass die Bedingungen der GPL vom Urheber auf alle anderen auferlegt; es steht dem Urheber deshalb offen, sich selber diese Bedingungen *nicht* aufzuerlegen. Man kann sich das so vorstellen, dass der Urheber unendlich viele Kopien der Software in einem Eimer hat, es kann sich entscheiden welche Lizenz dafür gilt: GPL, proprietär, oder etwas anderes. Das Recht das zu machen ist nicht an die GPL oder irgend einer anderen Open-Source-Lizenz gebunden; es ist einfach ein durch das Urheberrecht gegebene Möglichkeit.

Duale Lizenzierung ist attraktiv, da es freie Software eine Möglichkeit bietet, eine verlässliche Einkommensquelle zu sichern. Leider, kann es auch die üblichen Abläufen von Open-Source-Projekte behindern. Das Problem, ist das jede Freiwillige die einen Beitrag macht, jetzt an zwei verschiedene Einheiten einen Beitrag macht: Der freien Version vom Code und der proprietären Version. Auch wenn die Freiwillige gerne zur freien Version etwas beiträgt, da es bei Open-Source-Projekten üblich ist, mag es ihr nicht ganz genehm sein, zum teils-proprietären Einkommen von jemand anderem etwas beizutragen. Dieses Unannehmlichkeit wird noch verschlimmert, da die duale Lizenzierung von dem Urheber erfordert, formale, unterschriebene Übertragungen der Urheberrechts einzusammeln, um sich vor einem aufgebrachten Freiwilligen zu schützen, der im Nachhinein einen Teil der Gebühren einfordert. Durch diesen Vorgang, werden freiwillige offenkundig, mit der Tatsache konfrontiert, dass sie Arbeit machen, die jemand anderem Geld einbringt.

Das wird nicht allen Freiwilligen stören; schließlich werden ihre Beiträge auch an die Open-Source-Version gehen, und dass mag ihr Hauptinteresse sein. Duale Lizenzierung ist trotzdem ein Fall bei dem der Urheber sich selber ein Recht zuspricht, die andere im Projekt nicht haben, und wird deshalb irgendwann garantiert Spannungen verursachen, zumindest bei manchen Freiwilligen.

In der Praxis scheinen Firmen die eine duales Lizenzmodell wählen keine wirklich gleichberechtigte Entwicklergemeinschaft aufzubauen. Sie bekommen von außen kleine Bugfixes und Patches gegen Unschönheiten, müssen jedoch die die meiste Arbeit intern erledigen. Zack Urlocker, der Vizepräsident für Marketing bei MySQL, sagte mir beispielsweise, dass die Firma aktive Entwickler im Allgemeinen sowieso einstellt. Die Entwicklung unterliegt deshalb, wenn auch unter der GPL lizenziert, im wesentlichen der Kontrolle der Firma, wenn auch mit der (äußerst unwahrscheinlichen) Möglichkeit, dass jemand der wirklich unzufrieden damit ist, wie die Firma mit dem Projekt umgeht, einen Fork von dem Projekt machen könnte. In welchem Maße diese Bedrohung, die Politik der Firma präventiv gestaltet, weiß ich nicht, in jedem Fall scheint MySQL jedoch keine Akzeptanzprobleme, in der Open-Source-Welt oder sonstwo zu haben.

Patente[6]

Software-Patente sind derzeit ein brisantes Thema für freie Software, da sie eine echte Bedrohung bereitstellen, wogegen sich die freie Software-Gemeinschaft nicht alleine wehren kann. Mit Uhrheberrecht und Markenrechte kommt man immer irgendwie klar. Wenn ein Teil vom Code aussieht als ob es

[6]Das Patentrecht in den USA erlaubt Patente auf Software, bzw. Ideen. Das europäische Patentrecht ist wesentlich restriktiver. Die Patentierbarkeit "computerimplementierter Erfindungen" wurde im Juli 2005 vom EU Parlament mit einer Mehrheit von 95% abgewiesen. Das Thema ist dennoch auch für Europäer relevant, wenn sie wollen, dass ihre Software in den USA verwendet wird.

die Urheberrechte von jemand verletzt, kann man den Teil einfach neu schreiben. Wenn es sich herausstellt, dass jemand das Markenrecht auf den Namen von Ihrem Projekt hat, müssen Sie im schlimmsten Fall den Namen ändern. Ach wenn das vorübergehend etwas Unangenehm wäre, macht es auf lange Sicht keinen Unterschied, denn der eigentliche Code erfüllt immer noch seine Aufgabe.

Ein Patent bietet jedoch eine pauschale Verfügung über die Implementierung einer Idee. Es macht weder einen Unterschied wer den Code schreibt, noch welche Sprache dafür verwendet wird. Sobald jemand ein freies Software-Projekt beschuldigt, ein Patent zu verletzen, muss das Projekt entweder aufhören diese Funktion zu implementieren, oder sich auf eine teure und zeitaufwendigen Gerichtsverhandlung gefasst machen. Da die Anstifter solcher Klagen meistens Firmen mit tiefen Taschen – diejenigen mit den Ressourcen und der Neigung sich Patente überhaupt anzueignen – sind, können es sich die meisten freien Software-Projekte jedoch nicht letztere Möglichkeit leisten, und müssen sofort aufgeben, auch wenn sie der Meinung sind, das Patent wäre vor Gericht wahrscheinlich nicht durchsetzbar. Um solch eine Situation zu vermeiden, fangen freie Software-Projekte an, von vorn herein defensiv zu programmieren. Sie vermeiden patentierte Algorithmen, auch wenn sie die beste oder einzig verfügbare Lösung zu einem Problem wären. [7]

Umfragen und Einzelberichte zeigen jedoch, dass nicht nur die Mehrheit der Open-Source-Programmierer, sondern die Mehrheit *aller* Programmierer der Meinung sind, dass Software Patente abgeschafft werden sollten.[8] Open-Source-Programmierer sind meistens besonders Empfindlich wenn es um dieses Thema geht, und weigern sich an Projekte teilzunehmen, die sich zu sehr mit der Sammlung und Durchsetzung von Software-Patente assoziieren. Wenn Ihre Organisation Software-Patente sammelt, sollten Sie klar, öffentlich und unwiderruflich erklären, dass die Patente niemals gegen Open-Source-Projekte durchgesetzt werden, und dass sie nur als Verteidigung benutzt werden, sollte irgend eine andere Organisation gegen Ihre eine Klage einreichen. Das ist nicht nur das einzig richtig Vorgehen, es ist gute Beziehungspflege mit der Open-Source-Gemeinschaft.[9]

Leider ist die Sammlung von Patenten als Schutz ein vernünftiges Vorgehen. Das derzeitige Patentrecht, zumindest in den USA, ist von Natur aus ein Wettrüsten: wenn ihre Konkurrenten viele Patente habe, ist ihre beste Defensive, sich selber möglichst viele Patente anzueignen, sodass wenn Sie von einer Patentklage betroffen sind, Sie mit einer ähnlichen Drohung reagieren können – danach setzen sich beide Parteien meistens hin und arbeiten ein Übereinkommen aus, sodass keiner von beiden etwas zahlen muss, außer natürlich ihren Anwälten.

Der Schaden für freie Software der durch Patente entsteht ist jedoch heimtückischer als Bedrohungen bei der Entwicklung von Code. Software-Patente fördern eine Umgebung der Geheimhaltung unter den Entwicklern von Firmware, die sich berechtigte Sorgen machen, dass die Veröffentlichung von Details über die Schnittstellen ihren Konkurrenten eine technische Hilfe gibt, die darauf aus sind gegen ihnen Patentklagen zu erheben. Das ist nicht nur eine theoretische Gefahr; es geschieht beispielsweise, offenbar schon seit einiger Zeit in dem Markt für Grafikkarten. Viele Hersteller von Grafikkarten geben ungern die Spezifikationen für ihre Karten heraus, die gebraucht werden um performate Open Source Treiber für ihre Karten zu schreiben. Dadurch wird es für freie Betriebssysteme unmöglich diese Karten im vollen Umfang zu unterstützen. Was bringt die Hersteller dazu, sowas zu machen? Es macht schießlich keinen Sinn für Sie *gegen* Software Unterstützung zu arbeiten; mehr Betriebssysteme mit Unterstützung für ihre Produkte, bedeutet schließlich mehr verkaufte Karten. Es stellt sich jedoch heraus, dass hinter den Türen der Entwickler all diese Hersteller, gegenseitig gegen die Patente der anderen Verstoßen, manchmal bewusst, manchmal aus versehen. Patente sind etwas derart unvorhersehbares und derart weitreichend, dass kein Hersteller für Grafikkarten sich ganz sicher sein kann, selbst nachdem

[7] Sun Microsystems und IBM haben auch zumindest eine Geste gemacht, von der anderen Seite des Problems, indem sie eine Vielzahl – jeweils 1600 und 500 – ihrer Software Patente, für die Nutzung durch die freie Software-Gemeinschaft freigegeben haben. Ich bin kein Anwalt und kann deshalb nicht beurteilen, inwiefern diese Bewilligungen etwas Nützen, aber selbst wenn das alle wichtigen Patente sind, und ihre Bedingungen sie wirklich für die Nutzung in irgend einem Open-Source-Projekt freigeben, ist es dennoch nur ein Tropfen auf dem heißen Stein.
[8] Siehe http://lpf.ai.mit.edu/Whatsnew/survey.html für eines dieser Umfragen.
[9] RedHat sich hat beispielsweise Zugesichert, dass Open-Source-Projekte sicher vor ihre Patente sind, siehe http://www.redhat.com/legal/patent_policy.html.

sie nach Patenten gesucht haben. Hersteller wagen es deshalb nicht, die vollständigen Spezifikationen ihrer Schnittstellen offenzulegen, da es dadurch viel zu einfach für Konkurrenten wäre, sich die Patente herauszusuchen, die von ihnen verletzt werden. (Die Natur dieser Situation ist natürlich derart, dass man keine schriftliche Bestätigung über die Ereignisse von den Parteien erhalten würde; Ich habe davon durch persönliche Unterhaltungen erfahren.)

Manche freie Software-Lizenzen, beinhalten Klausel, um gegen Software Patente anzukämpfen, oder zumindest um gegen sie zu warnen. Die GNU GPL hat beispielsweise folgende Passagen:

> 7. Sollten Ihnen infolge eines Gerichtsurteils, des Vorwurfs einer Patentverletzung oder aus einem anderen Grunde (nicht auf Patentfragen begrenzt) Bedingungen (durch Gerichtsbeschluß, Vergleich oder anderweitig) auferlegt werden, die den Bedingungen dieser Lizenz widersprechen, so befreien Sie diese Umstände nicht von den Bestimmungen dieser Lizenz. Wenn es Ihnen nicht möglich ist, das Programm unter gleichzeitiger Beachtung der Bedingungen in dieser Lizenz und Ihrer anderweitigen Verpflichtungen zu verbreiten, dann dürfen Sie als Folge das Programm überhaupt nicht verbreiten. Wenn zum Beispiel ein Patent nicht die gebührenfreie Weiterverbreitung des Programms durch diejenigen erlaubt, die das Programm direkt oder indirekt von Ihnen erhalten haben, dann besteht der einzige Weg, sowohl das Patentrecht als auch diese Lizenz zu befolgen, darin, ganz auf die Verbreitung des Programms zu verzichten.
>
> [...]
>
> Zweck diese Paragraphen ist nicht, Sie dazu zu bringen, irgendwelche Patente oder andere Eigentumsansprüche zu verletzen oder die Gültigkeit solcher Ansprüche zu bestreiten; dieser Paragraph hat einzig den Zweck, die Integrität des Verbreitungssystems der freien Software zu schützen, das durch die Praxis öffentlicher Lizenzen verwirklicht wird. Viele Leute haben großzügige Beiträge zu dem großen Angebot der mit diesem System verbreiteten Software im Vertrauen auf die konsistente Anwendung dieses Systems geleistet; es liegt am Autor/Geber, zu entscheiden, ob er die Software mittels irgendeines anderen Systems verbreiten will; ein Lizenznehmer hat auf diese Entscheidung keinen Einfluß.[10]

Die Apache Lizenz, Version 2.0 (http://www.apache.org/licenses/LICENSE-2.0) beinhaltet ebenfalls Bedingungen gegen Patente. Es legt zuerst fest, dass jeder der Quellcode unter der Lizenz veröffentlicht, eine Gebührenfreie Lizenz impliziert, für jegliche Patente die sie halten mögen, der sich auf den Code anwenden ließe. Zweitens und am geistreichsten, bestraft es jeden der auf Patent-Verletzung in dem betreffenden Werk klagt, indem es automatisch im Moment einer solchen Klage, ihnen diese implizite Lizenz entzieht:

> 3. Lizenzierung von Patanten. Gemäß den Bedingungen dieser Lizenz, bewilligt jeder Beitragende Ihnen hiermit eine unbefristete, weltweite,

[10]Diese Übersetzung von Katja Lachmann im Auftrag der S.u.S.E. GmbH – http://www.suse.de und überarbeitet von Peter Gerwinski [http://www.peter.gerwinski.de/], G-N-U GmbH – http://www.g-n-u.de, wird hier zum besseren Verständnis verwendet und ist keine offizielle oder im rechtlichen Sinne anerkannte Übersetzung. Die Englische Originalfassung finden Sie hier http://www.gnu.org/licenses/gpl.html. Die vollständige deutsche Übersetzung der GPL finden Sie hier http://www.gnu.de/documents/gpl.de.html

uneingeschränkte, gebührenfreie, unwiderrufliche (außer wie in diesem Abschnitt beschrieben) Lizenz auf Patente um das Werk zu erstellen, benutzen, anzubieten, zu verkaufen, importieren oder anderweitig zu übertragen, wobei besagte Lizenz sich lediglich auf von dem Beitragenden Lizenzierbare Patentansprüche bezieht, zwangsläufig verletzt durch ihre Beiträge oder der Kombination ihrer Beiträge mit dem Werk an dem diese Beiträge gemacht wurden. Sollten Sie gegen irgend jegliche juristische Person auf Verletzung von Patenten Klagen (einschließlich einer Gegenforderung in einem Gerichtsverfahren) unter der Behauptung, dass das Werk oder ein in dem Werk einbezogener Beitrag, direkt oder teilweise Patente Verletzt, verfallen alle Ihnen durch diese Lizenz gewährte Patente für das betreffende Werk nach dem Zeitpunkt an dem besagte Klage eingereicht wird.[11]

Auch wenn es nützlich ist, sowohl rechtlich als auch politisch, freie Software auf diese Art gegen Patente zu schützen, reichen diese Schritte letztendlich nicht aus, um der gruseligen Wirkung die Patente auf freie Software haben, Einhalt zu gebieten. Lediglich Änderungen an dem Inhalt oder der Interpretation vom internationalen Patentrecht können das erreichen. Um mehr über das Problem zu erfahren, und wie dagegen angekämpft wird, schauen sie auf http://www.nosoftwarepatents.com/. Der Artikel auf Wikipedia http://en.wikipedia.org/wiki/Software_patent bietet ebenfalls viele nützliche Informationen zum Thema Software-Patente. Ich habe auch eine Blog-Eintrag geschrieben, der die Argumente gegen Software-Patente zusammenfasst http://www.rants.org/2007/05/01/how-to-tell-that-software-patents-are-a-bad-idea/.

Weitere Quellen

Dieses Kapitel war lediglich eine Einführung in das Thema der Lizenzierung von freier Software. Ich hoffe zwar, dass es genügend Informationen enthält, damit Sie mit ihrem Projekt anfangen können, würde jede ernstgemeinte Untersuchung der Lizenzprobleme schnell den Rahmen dieses Buchs sprengen. Hier sind einige weitere Quellen zum Thema der Open-Source-Lizenzierung:

- *Understanding Open Source and Free Software Licensing* von Andrew M. St. Laurent. Veröffentlicht durch O'Reilly Media, erste Edition August 2004, ISBN: 0-596-00581-4.

  Dieses Buch dreht sich um die ganze Komplexität, bei der Lizenzierung von Open-Source-Software, inklusive vieler Themen die hier ausgelassen wurden. Siehe http://www.oreilly.com/catalog/osfreesoft/ für weiteres.

- *Make Your Open Source Software GPL-Compatible. Or Else.* von David A. Wheeler, auf http://www.dwheeler.com/essays/gpl-compatible.html.

  Dieser gut geschriebene Artikel erzählt, warum es wichtig ist, Lizenzen zu benutzen, die zur GPL kompatibel sind, selbst wenn sie nicht die GPL verwenden. Es streift auch viele weitere Lizenzfragen und hat viele ausgezeichnete Links.

- http://creativecommons.org/

  Creative Commons ist eine Organisation die eine Reihe weiterer flexibler und liberale Urheberrechte vorantreiben, entgegen der üblichen Praxis beim Urheberrecht. Sie bieten diese Lizenzen nicht nur für Software, sondern auch für Schriften, Kunst, und Musik an, die sich alle bequem online auswählen lassen; manche sind Copyleft-Lizenzen, manche sind nicht Copyleft aber dennoch frei, weitere sind wie das traditionelle Urheberrecht aber etwas aufgelockert. Die Webseite der Creative Commons bie-

[11] Diese freie Übersetzung wird hier zum besseren Verständnis verwendet und ist keine offizielle oder im rechtlichen Sinne Anerkannt Übersetzung. Die Englische Originalfassung finden Sie hier http://www.opensource.org/licenses/apache2.0.php.

tet eine sehr klare Darstellung worum es geht, wenn ich eine Seite wählen müsste um die philosophische Bedeutung der freien Software-Bewegung zu zeigen, wäre es diese.

Anhang A. Systeme zur Versionsverwaltung

Dies sind alle Open-Source-Versionsverwaltungssysteme, die mir Mitte 2007 bekannt waren sowie einige, die ich Ende 2011 hinzugefügt habe. Diejenigen, die ich regelmäßig nutze, sind Subversion und Git; aber auch Bazaar und CVS nutzte ich in großem Maße. Mit den anderen habe ich kaum oder keine Erfahrung, und die hier aufgeführten Informationen habe ich von ihren Websites übernommen. Siehe auch http://en.wikipedia.org/wiki/List_of_revision_control_software.

Subversion – http://subversion.tigris.org/

Subversion wurde vor allem als Ersatz für CVS geschrieben – d.h. seine Herangehensweise an die Versionsverwaltung ist ungefähr die gleiche wie bei CVS, aber ohne die Probleme und fehlende Funktionen, welche die meisten Nutzer von CVS verärgern. Eines der Ziele von Subversion ist, dass der Übergang nach Subversion von Leuten, die bereits mit CVS vertraut sind, als relativ glatt empfunden wird. Es ist hier nicht genug Platz, um detalliert die Funktionen von Subversion zu behandeln; besuchen Sie die Webseite des Projekts für weitere Informationen. *[Disclaimer: Ich bin an der Entwicklung von Subversion beteiligt, und es ist das einzige System welches ich regelmäßig benutze.]*

GIT – http://git.or.cz/

GIT ist ein Projekt, welches von Linus Torvalds gestartet wurde, um den Quellcode des Linux-Kernels zu verwalten. GIT war zunächst relativ auf die Bedürfnisse der Kernel-Entwicklung konzentriert, ist aber darüber hinaus gewachsen und wird mittlerweile von anderen Projekten als dem Linux-Kernel benutzt. Auf der Webseite steht "...entwickelt um sehr große Projekte schnell und effizient zu verwalten; es wird hauptsächlich von verschiedenen Open-Source-Projekten benutzt, insbesondere vom Linux-Kernel. Git gehört zu der Kategorie der verteilten Verwaltungssysteme für Quellcode, ähnlich wie z.B. GNU Arch oder Monotone (oder BitKeeper aus der proprietären Welt). Jedes Arbeitsverzeichnis von GIT ist ein vollwertiges Repository mit der Möglichkeit Revisionen zu verfolgen, unabhängig von der Verfügbarkeit des Netzwerks oder einem Server."

Mercurial – http://www.selenic.com/mercurial/

Mercurial ist ein verteiltes Versionsverwaltungssystem, welches unter anderem "vollständige Kreuzindexierung von Dateien sowie Änderungen an mehreren Dateien; Bandbreite und Prozessor sparende HTTP- und SSH-Protokolle für die Synchronisierung; beliebige Merges zwischen den Zweigen von Entwicklern; integrierte autonome webbasierte Benutzeroberfläche; [unterstützung für] UNIX, MacOS X, und Windows" und mehr (die vorangehende Liste von Funktionen stammt von der Webseite von Mercurial).

Bazaar – http://bazaar.canonical.com/

Bazaar (oder bzr) ist ein verteiltes Versionsverwaltungssystem mit Schwerpunkt auf einfacher Benutzbarkeit und flexiblem Datenmodel. Es ist ein offizielles GNU-Projekt und das ursprüngliche Versionsverwaltungssystem der freien Software-Projekt-Hosting-Site Launchpad.net. Bazaar unterstützt verteilte Versionsverwaltung vollständig: sämtliche Arbeit findet in Zweigen statt, und jedem Entwickler steht üblicherweise die vollständige Änderungsgeschichte des Zweiges zur Verfügung. Zweige können auf dezentralisierte Weise miteinander gemergt werden, aber Bazaar kann auch so konfiguriert werden, dass es zentralisiert arbeitet. Bazaar nahm seinen Anfang aus einem Fork von GNU Arch, wurde dann jedoch von Grund auf neu geschrieben und hat nun zu GNU Arch keine direkte Beziehung mehr.

SVK – http://svk.elixus.org/

Auch wenn es auf Subversion aufbaut, ähnelt SVK wahrscheinlich eher einigen der dezentralisierten Systeme weiter unten als Subversion. SVK unterstützt verteilte Entwicklung, lokale Commits, ausgeklügelte Zusammenführung von Änderungen, und die Fähigkeit, Bäume aus Versionsverwaltungssystemen außer SVK zu spiegeln. Siehe seine Webseite für weitere Details.

Veracity – http://veracity-scm.com/

Veracity ist ein verteiltes Versionsverwaltungssystem; Genau wie bei Git, Mercurial, Bazaar usw. arbeitet jeder Entwickler auf einem vollständigen lokalen Projektarchiv und Änderungen werden auf Bedarf zwischen den Projektarchiven per Push und Pull übertragen. Von den Befehlen her ist es diesen System sehr ähnlich, aber Veracity beinhaltet zusätzlich zum Versionieren von Dateien eine verteilte Fehlerdatenbank, die gemeinsam mit den Dateien versioniert wird. In anderen Worten: Veracity versucht, alle Artefakte aufzunehmen, die tatsächlich zur Entwicklung benötigt werden — nicht allein den Quelltext-Baum, sondern auch die Fehlerberichte — und sie durch das Versionsverwaltungssystem verfügbar zu machen. Das ist ein ehrgeiziges Ziel, und solange ich noch nicht die Gelegeneit hatte, es zu benutzen, bin ich an Erfahrungsberichten sehr interessiert. Es wird hauptsächlich von SourceGear, Inc. entwickelt, einem Unternehmen mit einer langen Geschichte im Hinblick auf Versionsverwaltung und Softwarekonfigurationsmanagement.

Siehe auch Fossil für ein ähnliches System.

Fossil – http://www.fossil-scm.org/

Fossil ähnelt Veracity dahin gehend, dass es ein verteiltes Versionsverwaltungssystem ist, das mehr als nur Quelltextdateien versioniert: es versioniert die Fehlerdatenbank, aber auch ein verteiltes Wiki, und ein verteiltes Blog. Die weiteren Features umfassen einen standardmäßigen "autosync"-Modus, um nicht-konfligierende Änderungen automatisch zusammenzuführen (d.h. Fossil kann sowohl auf zentralisierte als auch auf dezentralisierte Weise arbeiten, was theoretisch auch für andere dezentralisierte Systeme gilt, aber es scheint so, als ob Fossil den zentralisierten Arbeitsablauf besser unterstützt). Auch wird es mit einem Web-Interface ausgeliefert, mit dem die Benutzer durch das Projektarchiv navigieren können.

Fossil wird hauptsächlich durch Dr. Richard Hipp entwickelt, der möglicherweise besser bekannt ist als Autor der SQLite Datenbank-Engine. Genau wie Veracity, habe ich Fossil bislang nicht benutzt; sollten Sie es benutzen, dann lassen Sie mich bitte wissen, wie es läuft.

CVS – http://www.nongnu.org/cvs/

CVS gibt es schon seit langem, und viele Entwickler sind bereits damit vertraut. Zu seiner Zeit war es revolutionär: Es war das erste Open-Source-Versionsverwaltungssystem, das Entwicklern Zugang über weite Netze anbot (so weit ich weiß), und es erlaubte erstmals anonyme, rein lesende Checkouts, was neuen Entwicklern einen einfachen Weg bot, sich an Projekten zu beteiligen. CVS verwaltet lediglich Dateien, keine Verzeichnisse; es bietet Verzweigung an, Tags und gute Geschwindigkeit auf Client-Seite, kann aber nicht so gut mit großen oder binären Dateien umgehen. Es unterstützt auch keine atomaren Commits.*[Disclaimer: Ich war ca. fünf Jahre lang aktiver Entwickler bei CVS, bevor ich dabei half, das Subversion-Projekt zu starten, um es zu ersetzen.]*

Darcs – http://abridgegame.org/darcs/

"David's Advanced Revision Control System ist noch ein weiterer Ersatz für CVS. Es wurde in Haskell geschrieben, und wurde schon auf Linux, MacOS X, FreeBSD, OpenBSD und Microsoft Windows

benutzt. Darcs beinhaltet auch ein cgi script, mit dem man sich den Ihnhalt des Repositorys anschauen kann."

Arch – http://www.gnu.org/software/gnu-arch/

GNU Arch unterstützt sowohl verteilte als auch die zentralisierte Entwicklung. Entwickler committen ihre Änderungen zu einem "Archiv", welches lokal sein kann, und die Änderungen können zu anderen Archiven geschoben oder gezogen werden, wie es die Verwalter der Archive für richtig halten. Wie solch eine Methodik impliziert, hat Arch feinere Unterstützung für Merges als CVS. Arch erlaubt es auch einfach Zweige von Archiven zu erstellen, auf denen man keinen Commit-Zugriff hat; siehe die Webseiten von Arch für weitere Details.

monotone – http://www.venge.net/monotone/

"monotone ist ein freies verteiltes Versionsverwaltungssystem. Es bietet einen einfachen, aus einer Datei bestehenden auf Transaktionen basierenden Speicher, mit vollständig entkoppeltem Betrieb und einem effizienten Peer-to-Peer-Synchronisationsprotokoll. Es versteht historisch abhängige Merges, leichtgewichtige Zweige, integrierte Überprüfung und Tests durch Dritte. Es benutzt kryptographische Benennung von Versionen sowie Client-seitige RSA-Zertifikate. Es bietet gute Unterstützung für Internationalisierung, hat keine externen Abhängigkeiten, läuft unter Linux, Solaris, OSX, und Windows, und ist unter der GNU GPL lizenziert."

Codeville – http://codeville.org/

"Warum noch ein weiteres Versionsverwaltungssystem? Alle anderen Versionsverwaltungssysteme erfordern, dass Sie genau auf die Beziehungen zwischen Zweigen achthaben, um nicht wiederholt die selben Konflikte zusammenführen zu müssen. Codeville ist sehr viel freizügiger. Es erlaubt Ihnen ein Commit von oder zu einem Repository zu machen, jederzeit und ohne unnötige wiederholte Merges."

"Codeville arbeitet, indem es für jede Änderung, die gemacht wird, eine Kennung erstellt, und sich eine Liste aller Änderungen die je auf einer Datei angewandt wurden behält, sowie die letzte Änderung die jede Zeile in jeder Datei geändert hat. Wenn es einen Konflikt gibt, überprüft es, ob eine der beiden Seiten bereits auf die andere angewandt wurde, und wenn das der Fall ist, gewinnt die andere Seite automatisch. Wenn es einen Konflikt gibt, der wirklich nicht automatisch zusammengeführt werden kann, verhält sich fast genau wie CVS."

Vesta – http://www.vestasys.org/

"Vesta ist ein portables SCM [Software Configuration Management] System dessen Ziel es ist, die Entwicklung fast jeder Größe zu unterstützen, von ziemlich klein (unter 10.000 Codezeilen) bis sehr groß (10.000.000 Codezeilen)."

"Vesta ist ein ausgereiftes System. Es ist das Ergebnis von über 10 Jahren Forschungs- und Entwicklungsarbeit an dem Compaq/Digital Systems Forschungszentrum, und wurde von der Compaq-Gruppe für den Alpha Processor über zweieinhalb Jahre lang produktiv genutzt. Die Alpha-Gruppe hatte über 150 aktive Entwickler an zwei Standorten die tausende Meilen auseinaner lagen, an den Ost- und Westküsten der USA. Die Gruppe hat Vesta genutzt, um Quellcode-Daten von einer Größe bis zu 130MB zu kompilieren, die je 1.5 abgeleitete Daten produzierten. Die Kompilierungen die an der Ostküste gemacht wurden, haben an einem durchschnittlichen Tag 10-15 GB an Daten produziert, die alle von Vesta verwaltet wurden. Obwohl Vesta mit Softwareentwicklung im Sinn entworfen wurde, hat die Alpha-Gruppe die Flexibilität des System demonstriert, indem sie es für die Hardwareentwicklung benutzte, Commits mit den Dateien der Beschreibungssprache für die Hardware machten, sowie ihre Simulationen und andere abgeleitete Objekte Builds mit dem System von Vesta gemacht haben. Die Mitglieder der ehema-

ligen Alpha-Gruppe, jetzt ein Teil von Intel, benutzen heute weiterhin Vesta bei einem neuen Mikroprozessor-Projekt."

Aegis – http://aegis.sourceforge.net/

"Aegis ist ein Software-Configuration-Management-System, basierend auf Transaktionen. Es bietet einen Rahmen, in dem ein Team von Entwicklern an vielen Änderungen unabhängig voneinander arbeiten können, und Aegis koordiniert die Integration dieser Änderungen zurück in die zentralen Quellcode-Dateien der Anwendung, mit so wenig Störungen wie möglich."

CVSNT – http://cvsnt.org/

"CVSNT ist ein fortgeschrittenes Versionsverwaltungssystem welches auf mehreren Platforment läuft. Es ist mit der Industrienorm des CVS-Protokolls kompatibel und unterstützt viele weitere Funktionen. ... CVSNT ist Open Source, freie Software und unter der GNU GPL lizenziert." Seine Funktionen sind unter anderem die Authentifikation mit allen üblichen CVS-Protokollen, sowie das Windows eigene SSPI und Active Directory; Unterstützung für secure transport, mittels sserver oder verschlüsseltem SSPI; es ist plattformübergreifend (läuft in Windows- oder Unix-Umgebungen); NT version ist vollständig mit dem Win32 System integriert; MergePoint-Verarbeitung bedeutet, dass Sie keine Tags mehr brauchen, um einen Merge zu machen; es wird aktiv entwickelt.

META-CVS – http://users.footprints.net/~kaz/mcvs.html

"Meta-CVS ist ein Versionsverwaltungssystem, welches um CVS gebaut wurde. Obwohl es die meisten Funktionen von CVS behält, inklusive aller Netzwerk-Unterstützung, ist es mächtiger als CVS, und einfacher zu benutzen". META-CVS listet auf seiner Webseite unter anderem folgende Funktionen: Versionierung von Verzeichnisstrukturen, verbesserte Handhabung verschiedener Dateitypen, einfachere und benutzerfreundlichere Erzeugung von und Zusammenführung von Zweigen, Unterstützung für symbolische Verweise, Listen von Attributen für versionierte Daten, verbesserter Import von Daten dritter Parteien, sowei einfache Aufrüstung bereits bestehender CVS-Archive.

OpenCM – http://www.opencm.org/

"OpenCM wurde als ein sicherer, hoch integrierter Ersatz für CVS entworfen. Eine Liste der wesentlichen Funktionen kann auf seiner Webseite gefunden werden. Auch wenn es nicht so viele Funktionen wie CVS hat, unterstützt es einige nützliche Dinge, die CVS fehlen. In Kürze bietet OpenCM erstklassige Unterstützung für die Umbenennung und Konfiguration, kryptographische Authentifizierung, Kontrolle der Zugriffsberechtigung, und erstklassige Verzweigung."

PRCS – http://prcs.sourceforge.net/

"PRCS, das Project Revision Control System, ist das Frontend für einen Satz von Programmen, die es (wie CVS) ermöglichen mit Gruppen von Dateien und Verzeichnessen als Entitäten umzugehen, und dabei kohärente Versionen der gesamten Gruppe zu bewahren. ... Sein Sinn ist ähnlich dem von SCCS, RCS, und CVS, ist aber (zumindest laut seinen Entwicklern) sehr viel einfacher als irgend eines dieser Systeme."

ArX – http://www.nongnu.org/arx/

ArX ist ein verteiltes Versionsverwaltungssystem, welches Funktionen für Verzweigung und Zusammenführung anbietet, kryptographische Prüfung der Integrität von Daten, sowie die Fähigkeit, Archive einfach auf jedem HTTP-Server zu veröffentlichen.

SourceJammer – http://sourcejammer.org/

"SourceJammer ist ein Verwaltungssystem für Versionen und Quellcode welches in Java geschrieben ist. Es besteht aus einer Server-seitigen Komponente, welche die Dateien und die Historie der Versionen, Commits und Checkouts usw. und andere Befehle verarbeitet sowie einer Client-seitigen Komponente die Anfragen an den Server sendet und die Dateien auf dem Dateisystem des Clients verwaltet."

FastCST – http://www.zedshaw.com/projects/fastcst/index.html

"Ein 'modernes' System welches Gruppen von Änderungen über Revisionen von Dateiein benutzt und verteilt arbeitet anstatt zentralisiert. Sobald Sie eine E-Mail-Adresse haben, können Sie FastCST benutzen. Für breitere Verteilung brauchen Sie lediglich einen FTP- und/oder HTTP-Server oder Sie können den 'server' Befehl benutzen um direkt Ihre Inhalte zu verteilen. Alle Änderungen sind universell eindeutig, und haben eine Menge Metadaten, also können Sie alles ablehnen, was Sie nicht wollen, bevor Sie es ausprobieren. Merges werden realisiert, indem die Zusammenführung eines Satzes von Änderungen mit dem derzeitigen Inhalt des entsprechenden Verzeichnisses verglichen wird, nicht, indem der Versuch unternommen wird, es mit einem anderen Satz von Änderungen zusammen zu führen."

Superversion – http://www.superversion.org/

"Superversion ist ein verteiltes Versionsverwaltungssystem für mehrere Benutzer welches auf Changesets basiert. Sein Ziel ist eine professionelle Open-Source-Alternative zu kommerziellen Lösungen an die Seite zu stellen, die gleich einfach bedienbar ist (oder sogar einfacher) und gleich leistungsfähig. Tatsächlich ist intuitive und effiziente Bedienbarkeit bereits seit dem Anfang seiner Entwicklung eines der vorrangigen Ziele von Superversion."

Anhang B. Freie Bugtracker

Unabhängig davon welchen Bugtracker ein Projekt benutzt: es wird immer Entwickler geben, die oft und gerne etwas daran auszusetzen haben. Das scheint in Bezug auf Bugtracker häufiger zuzutreffen, als bei irgend einem anderen der üblichen Entwicklungswerkzeuge. Ich denke das liegt daran, dass Bugtracker so visuell und interaktiv sind, dass es leicht ist, sich die Verbesserungen vorzustellen die man vornehmen würde (wenn man nur die Zeit hätte), und diese Verbesserungen auch laut zu benennen. Nehmen Sie die Verbesserungen mit einem gewissen Vorbehalt – viele der hier aufgeführten Bugtracker sind ziemlich gut.

In dieser Auflistung wird das Wort "Ticket" benutzt, um Einträge in dem Tracker zu bezeichnen. Denken Sie aber daran, dass jedes System seine eigene Begriffen verwenden kann, wobei der Begriff etwas wie "Issue", "Artifact", "Bug" oder etwas anderes sein könnte.

Hinweis: Dieser Überblick stammt größtenteils noch aus dem Jahre 2005, seitdem wurden einige neue Open-Source-Bugtracker geschrieben. So könnten wir die Liste sicherlich einmal aktualisieren, aber bis dahin erhalten Sie aktuelle Informationen unter http://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems, http://www.dmoz.org/Computers/Software/Configuration_Management/Bug_Tracking/Free/, http://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems und http://www.webresourcesdepot.com/9-free-and-open-source-bug-tracking-softwares/

Redmine – http://www.redmine.org/

Redmine ist ein relativ junges (Stand 2011) und recht ausgefeiltes Projekt-Management-System. Es geht etwas über einen Bugtracker hinaus, denn es bietet auch Wikis, Diskussionsforen und andere Funktionen, dennoch scheint die Fehlerverfolgung sein Kernstück darzustellen. Es hat eine ziemlich intuitive webbasierte Benutzerschnittstelle, flexible Konfiguration (mehr als ein Projekt, rollenbasierte Zugriffskontrolle, benutzerdefinierte Felder usw.), Gantt-Diagramme, Kalender, bidirektionale E-Mail-Interktion und mehr. Wenn Sie ein neues Projekt beginnen würden und den Bugtracker frei wählen könnten, dann wäre Redmine eine gute Wahl.

Bugzilla – http://www.bugzilla.org/

Bugzilla ist sehr verbreitet, wird aktiv entwickelt und scheint seine Nutzer ziemlich glücklich zu machen. Ich benutze eine angepasste Variante bei meiner Arbeit seit mittlerweile vier Jahren, und ich mag es. Man kann es nicht sonderlich individualisieren, aber auf eine gewisse Art kann das als Vorteil betrachtet werden: Bugzilla-Installationen neigen so zu einem recht ähnlichem Aussehen, und da viele Entwickler bereits mit der Bedienoberfläche vertraut sind, fühlen sich auf gewohntem Terrain.

GNATS – http://www.gnu.org/software/gnats/

GNU GNATS ist einer der ältesten Open-Source-Butracker und weit verbreitet. Seine größten Vorteile sind die Vielfalt seiner Bedienung (er kann nicht nur durch den Browser sondern auch per E-Mail oder über Kommandozeile bedient werden) und die Speicherung der Tickets im Klartext. Die Tatsache, dass alle Daten der Tickets als Textdateien gespeichert werden, erleichtert es, eigene Programme zu schreiben, um die Daten analysieren und zu verarbeiten (zum Beispiel um Statistiken zu erstellen). GNATS kann E-Mails auch auf verschiedene Arten automatisch annehmen und sie den entsprechenden Tickets hinzufügen, was auf Grundlage von Mustern in den E-Mail-Headern geschieht, das erleichtert die Protokollierung des Austauschs zwischen Nutzern und Entwicklern.

RequestTracker (RT) – http://www.bestpractical.com/rt/

Auf der Webseite von RT steht "RT ist ein für Unternehmen geeignetes Ticket-System, welches einer Gruppe von Personen erlaubt, Aufgaben, Probleme und Anfragen die von einer Gemeinschaft von Nutzern eingereicht werden, intelligent und effizient zu verwalten", womit das Wesentliche gesagt ist. RT hat eine relativ ausgefeiltes Webinterface, und scheint ziemlich weit verbreitet zu sein. Die Optik der Oberfläche wirkt zunächst etwas komplex, bedarf aber lediglich einer Eingewöhnungsphase. RT ist unter der GNU GPL lizenziert (aus irgend einem Grund wird das auf der Webseite nicht recht deutlich).

Trac – http://trac.edgewall.com/

Trac ist etwas mehr als ein Butracker: es ist in Wirklichkeit ein integriertes System von Wiki und Bugtracker. Es nutzt Wikiverweise um Tickets, Dateien, Changesets der Versionsverwaltung, und einfache Wikiseiten miteinander zu verbinden. Es ist recht einfach einzurichten, und lässt sich mit Subversion integrieren (siehe Anhang A, *Systeme zur Versionsverwaltung*).

Roundup – http://roundup.sourceforge.net/

Roundup ist relativ einfach zu installieren (es wird lediglich Python 2.1 oder höher benötig) und einfach zu benutzen. Es kann per Browser, E-Mail oder Kommandozeile bedient werden. Die Vorlagen für Ticket-Daten und die Webinterface können genauso angepasst werden, wie Teile seiner Logik für die Übergänge zwischen verschiedenen Zuständen.

Mantis – http://www.mantisbt.org/

Mantis ist ein Web-basierter Bugtracker, geschrieben in PHP, der MySQL als Datenbanksystem nutzt. Es hat die Funktionen, die man erwartet. Ich perönlich finde die Oberfläche sauber, intuitiv, und angenehm für die Augen.

Flyspray – http://www.flyspray.org/

Flyspray ist ein in PHP geschriebener Bugtracker. Seine Webseite beschreibt es als "unkompliziert", und die Liste seiner Funktionen beinhaltet: Unterstützung für mehrere Datenbanken (derzeit MySQL und PGSQL); mehrere Projekte; 'Beobachtung' von Aufgaben, mit Benachrichtigung bei Änderungen (mittels E-Mail oder Jabber); umfassende Historie von Aufgaben; CSS-Themes; Datei-Anhänge; erweiterte Suchfunktionen (aber auch einfach zu bedienende); RSS-/Atom-Kanäle; Wiki und Klartext-Eingabe; Abhängigkeitsdiagramme.

Scarab – http://scarab.tigris.org/

Scarab ist als hochgradig anpassbarer, vollständiger Bugtracker gedacht, mehr oder weniger die Vereinigung aller Funktionen die von anderen Bugtrackern angeboten werden: Daten-Eingabe, Abfragen, Berichte, Benachrichtigungen an interesierte Parteien, gemeinschaftliches Sammeln von Kommentaren und Verfolgung von Abhängigkeiten.

Es ist über administrative Webseiten anpassbar. Sie können in einer einzigen Scarab-Installation mehrere aktive "Module" (Projekte) haben. Innerhalb eines Moduls können Sie neue Arten von Tickets anlegen (Mängel, Verbesserungen, Aufgaben, Support-Anfragen, usw.) und beliebige Attribute hinzufügen, um den Tracker an die spezifischen Erfordernisse ihres Projekts anzupassen.

Ende 2004 näherte sich Scarab seiner 1.0-Version.

Debian Bug Tracking System (DBTS) – http://www.chiark.greenend.org.uk/~ian/debbugs/

Das Debian-Bugtracking-System ist insofern ungewöhnlich, als dass alle Eingaben und Änderungen per E-Mail gemacht werden: Jedes Ticket bekommt seine eigene Mailadresse. Das DBTS skaliert ziemlich gut: http://bugs.debian.org/ hat zum Beispiel 277,741 Tickets.

Da die Bedienung über gewöhnliche E-Mail-Anwendungen erfolgt, eine Umgebung mit der die meisten Personen vertraut sind und zu der sie leicht Zugang haben, ist das DBTS geeignet für die Handhabung großer Mengen eingehender Meldungen, die eine schnelle Klassifizierung und Reaktion benötigen. Es gibt natürlich auch Nachteile. Entwickler müssen Zeit investieren, um das Befehlssystem zu lernen, und Nutzer müssen ihre Bug-Meldungen ohne ein Web-Formular eingeben, das sie beim Schreiben in der Auswahl der nötigen Informationen unterstützen würde. Es gibt Programme, die den Nutzern zu helfen, bessere Bug-Meldungen zu schreiben, wie das Kommandozeilenprogramm **reportbug** oder das `debbugs-el`-Paket für Emacs. Die meisten Leute werden diese Werkzeuge aber nicht benutzen; sie werden die E-Mail einfach per Hand schreiben, und so werden sie die Richtlinien, die für die Meldung von Bugs von Ihrem Projekt erarbeitet wurden, befolgen oder eben auch nicht.

DBTS hat ein Webinterface mit reinem Lesezugriff, um Tickes anzuschauen und abzufragen.

Trouble-Ticket Tracker

Solche Systeme sind eher darauf ausgelegt, Tickets für einen Informationsschalter zu überwachen, als Bugs in Software. Sie werden wahrscheinlich einen gewöhnlichen Bugtracker hilfreicher finden. Solche Systeme habe ich hier der Vollständigkeit halber aufgeführt, da auch auch ungewöhnliche Projekte vorstellbar sind, bei dem ein Trouble-Ticket-System besser passt als ein herkömmlicher Bugtracker.

- **WebCall** – http://myrapid.com/webcall/

Bluetail Ticket Tracker (BTT) – http://btt.sourceforge.net/

BTT liegt irgendwo zwischen Trouble-Ticket-Tracker und Bugtracker. Es bietet Funktionen für den Datenschutz, was unter Open-Source-Bugtrackern etwas ungewöhnlich ist: Nutzer des Systems werden in Mitarbeiter, Freunde, Kunden, oder Anonyme eingeteilt, und es stehen, je nachdem zu welcher Kategorie man gehört, mehr oder weniger Daten zur Verfügung. Es bietet etwas E-Mail-Integration, die Bedienung mittels Kommandozeile und Mechanismen um E-Mails in Tickets umzuwandeln. Es hat auch Funktionen zur Pflege von Informationen, die nicht mit einem spezifischen Ticket zu tun haben, wie z.B. interne Dokumentation oder FAQs.

Anhang C. Warum sollte es mich kümmern, welche Farbe der Fahrradschuppen hat?

Es sollte Sie nicht kümmern; es macht nicht wirklich einen Unterschied, und Sie haben besseres mit Ihrer Zeit zu tun.

Die berühmte E-Mail von Poul-Henning Kamp (aus der ein Ausschnitt in Kapitel 6, *Kommunikation* erscheint) ist eine wortgewandte Abhandlung darüber, was bei Gruppendiskussionen dazu neigt, schief zu laufen. Es ist hier mit seiner Genehmigung kopiert und übersetzt. Die URL des Originals ist http://www.freebsd.org/cgi/getmsg.cgi?fetch=506636+517178+/usr/local/www/db/text/1999/freebsd-hackers/19991003.freebsd-hackers.

```
Betreff: Ein Fahrradschuppen (egal welche Farbe) auf grünerem Gras...
Von: Poul-Henning Kamp <phk@freebsd.org>
Datum: Sat, 02 Oct 1999 16:14:10 +0200
Message-ID: <18238.938873650@critter.freebsd.dk>
Absender: phk@critter.freebsd.dk
Bcc: Blind Distribution List: ;
MIME-Version: 1.0



[bcc an committers und hackers]


Mein letztes Schreiben wurde aureichend gut aufgenommen, sodass ich mich
nicht davor scheue ein weiteres zu senden, und heute habe ich die Zeit
und Muße es zu tun.


Ich hatte ein paar Probleme zu entscheiden, wie ich so etwas
herausgeben soll, dieses mal als bcc an committers und
hackers, vermutlich das Beste, was ich machen kann.
Ich bin selber nicht bei hacker angemeldet, aber dazu später mehr.


Was mich dieses Mal angestoßen hat ist der "sleep(1) sollte mit
Sekunden-Bruchteile funktionieren" Thread, welcher unser Leben zu viele
Tage lang geplagt hat – wahrscheinlich sind es schon ein paar Wochen,
ich will mir nicht mal die Mühe machen, nachzuschauen.


An diejenigen die diesen speziellen Thread verpasst haben: Herzlichen
Glückwunsch.


Es war ein Vorschlag, dass sleep(1) sich richtig verhalten sollte, wenn
```

man ihr ein Argument gibt, das kein Integer ist, und dieses bestimmte Buschfeuer in Gang setzte. Ich werde nicht mehr darüber sagen, weil es eine viel kleinere Angelegenheit ist, als man von der Länge des Threads erwarten würde, und sie hat schon viel mehr Aufmerksamkeit geerntet, als einige der *Probleme* die wir hier gehabt haben.

Die sleep(1) Saga ist das eklatanteste Beispiel einer Fahrradschuppen-Diskussion, die wir je bei FreeBSD gehabt haben. Der Vorschlag war gut ausgearbeitet, wir wären mit OpenBSD und NetBSD kompatibler geworden, und trotzdem mit jedem Code vollständig kompatibel geblieben, den irgend jemand je geschrieben hat.

Es wurden jedoch so viele Einwände, Vorschläge und Änderungen aufgeworfen das man meinen könnte, die Änderungen hätten alle Löcher in einem schweizer Käse gestopft oder den Geschmack von Coca Cola verändert, oder etwas ähnlich gravierendes.

"Worum geht es bei diesem Fahrradschuppen?", haben mich einige gefragt.

Es ist eine lange Geschichte, bzw. eher eine alte Geschichte, aber sie ist in Wirklichkeit ziemlich kurz. C. Northcote Parkinson schrieb Anfang der 1960er ein Buch namens "Parkinsons Gesetz", welches eine Menge Einblicke in die Dynamik des Management beinhaltet.

Sie können es bei Amazon finden, und vielleicht auch im Bücherregal ihres Vaters, es ist sowohl seinen Preis als auch die Zeit es zu lesen wert. Wer Dilbert gut findet, wird auch Parkinson mögen.

Jemand sagte mir neulich er hätte es gelesen und fand, dass lediglich 50% sich heutzutage anwenden ließe. Das ist verdammt gut, würde ich sagen, viele der modernen Bücher über Management liegen deutlich darunter, und dieses ist über 35 Jahre alt.

Bei dem spezifischen Beispiel, bei dem es um einen Fahrradschuppen geht, ist die andere entscheidende Komponente ein Atomkraftwerk, was denke ich die Zeit, in der das Buch geschrieben wurde, anschaulich macht.

Parkinson zeigt, wie Sie zum Vorstand gehen und Zustimmung für den Bau ein multi-millionen oder gar Milliarden Dollar teures Atomkraftwerk bekommen können, wenn Sie aber einen Fahrradschuppen bauen möchten sich in endlosen Diskussionen verzetteln werden.

Parkinson erklärt, dass das daran liegt, dass ein Atomkraftwerk so gewaltig, so kostspielig und so kompliziert ist, dass Menschen es nicht begreifen können, und anstatt es zu versuchen eher auf die Annahme zurückfallen, jemand anderes hätte bereits alle Details überprüft bevor es so weit gekommen ist. Richard P. Feynmann gibt in seinen Büchern einige interessante und treffende Beispiele, die mit Los Alamos zu tun haben.

Einen Fahrradschuppen andererseits kann jeder übers Wochenende bauen, und noch genug Zeit übrig haben, um das Spiel im Fernsehen zu schauen. Egal wie gut Sie sich also vorbereiten, egal wie vernünftig Sie bei Ihrem Vorschlag sind, irgend jemand wird die Chance ergreifen um Ihnen zu zeigen, dass er seine Arbeit macht, dass er aufpasst, dass er *da* ist.

In Dänemark nennen wir das "seinen Fingerabdruck hinterlassen". Es geht um den persönlichen Stolz und Ansehen, es geht darum irgendwo drauf zeigen zu können und zu sagen "Da! Das habe *ich* gemacht". Es ist ein starker Wesenszug bei Politikern, aber auch in den meisten Menschen vorhanden, wenn sie dazu die Gelegenheit bekommen. Denken Sie einfach an Fußabdrücke im nassen Zement.

Ich verneige mich mit Respekt vor dem ursprünglichen Absender des Vorschlags, da er wärend dieser Bombardierung durch den Pöbel bei seinen Waffen blieb, und die Änderung mittlerweile eingepflegt wurde. Ich hätte der Sache schon nach weniger als einer Handvoll Nachrichten den Rücken gekehrt.

Was mich wie zuvor erwähnt zurück auf den Grund bringt, warum ich nicht mehr bei -hackers angemeldet bin:

Ich habe mich von -hackers vor mehreren Jahren abgemeldet, weil ich nicht mehr mit der E-Mail-Flut mithalten konnte. Seitdem habe ich aus demselben Grund mehrere andere Mailinglisten verlassen.

Und ich bekomme immer noch jede Menge E-Mails. Vieles wird nach /dev/null über Filter geleitet: Leute namens [weggelassen] werden es genau so wenig auf meinem Bildschirm schaffen wie Commits an Dokumente in Sprachen, die ich nicht verstehe, genauso wie ähnliche Portierungen. All diese Sachen und mehr gehen über den Jordan, ohne dass ich je was von ihnen erfahre.

Aber trotz dieser scharfen Zähne vor meinem Posteingang bekomme ich immer noch zu viele E-Mails.

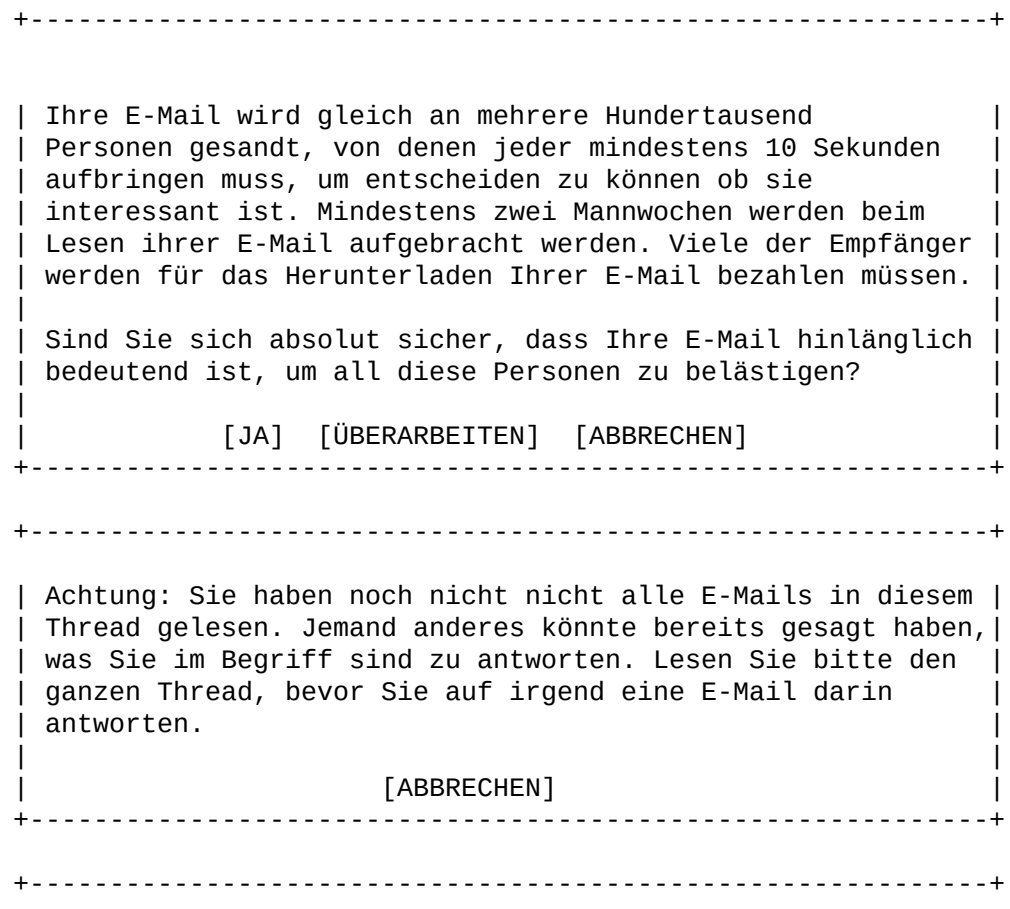
Hier kommt nun das grünere Gras ins Bild:

Ich wünschte, wir könnten das Maß an Rauschen auf unserern Listen verringern, und ich wünschte, wir ließen die Leute gelegentlich einen Fahrradschuppen bauen, und es kümmert mich nicht wirklich, in welcher Farbe sie ihn streichen.

Der erste dieser Wünsche zielt darauf ab, E-Mails zivilisiert, sensibel und intelligent zu benutzen.

Wenn ich griffige, präzise Kriterien definieren könnte, wann eine E-Mail beantwortet werden sollte und wann nicht, denen jeder zustimmen und folgen würde, wäre ich ein glücklicher Mensch, aber ich bin zu weise, das auch nur zu versuchen.

Lasst mich aber ein paar Pop-Up-Meldungen vorschlagen, die ich gerne von E-Mail-Programmen implementiert sähe, die auftauchen, sobald Leute versuchen, E-Mails an Listen zu schreiben oder zu beantworten, von denen sie möchten dass ich sie abonniere:



```
| Achtung: Ihr E-Mail-Programm hat Ihnen nicht einmal die    |
| ganze Nachricht gezeigt. Es folgt logich daraus, dass Sie  |
| sie unmöglich komplett gelesen und verstanden haben können.|
|                                                            |
| Es ist unhöflich, auf eine E-Mail zu antworten, bevor Sie  |
| sie ganz gelesen und darüber nachgedacht haben.            |
|                                                            |
| Eine Zeitbeschränkung wird Sie zur Beruhigung eine Stunde  |
| lang daran hindern, auf irgend welche E-Mails in diesem    |
| Thread zu zu antworten.                                    |
|                                                            |
|                      [ABBRECHEN]                           |
+------------------------------------------------------------+

+------------------------------------------------------------+

+------------------------------------------------------------+
| Sie haben diese E-Mail mit einer Geschwindigkeit von mehr  |
| als N.NN zps verfasst. Es ist i.A. möglich, schneller als  |
| mit einer Geschwindigkeit von A.AA zps zu denken und zu    |
| schreiben, und Ihre Antwort ist deshalb wahrscheinlich     |
| unpassend, undurchdacht und/oder emotional.                |
|                                                            |
| Eine Zeitbeschränkung wird Sie zur Beruhigung eine Stunde  |
| am Versenden vom E-Mails hindern.                          |
|                                                            |
|                      [ABBRECHEN]                           |
+------------------------------------------------------------+
```

Der zweite Teil meines Wunsches ist eher emotional. Offensichtlich haben die Kapazitäten welche wir für die Besatzung dieses unfreundlichen Feuers in dem sleep(1) Thread hatten, trotz ihrer vielen Jahre bei dem Projekt, es nie als wichtig erachtet, diese kleine Handlung durchzuführen, warum sind sie also jetzt derart aufgebracht über jemand anderen, sehr viel unerfahreneren der es erledigt ?

Ich wünschte ich wüsste es.

Was ich weiß ist, dass Argumente es nicht schaffen werden, diese "Reaktionären Konservativen" aufzuhalten. Es mag sein, dass diese Leute über ihren eigenen Mangel an greifbaren Beiträgen in der letzten Zeit frustriert sind, oder es ist ein schlimmer Fall von "Wir sind alt und knauzig, WIR wissen, wie sich die Jugend verhalten sollte".

So oder so, ist es für das Projekt sehr unproduktiv, ich habe aber keine Vorschläge es aufzuhalten. Das beste was ich vorschlagen kann, ist sich davor zurückzuhalten, die Ungeheuer zu füttern, die auf den Mailinglisten lauern: Ignoriere sie, antworte nicht auf sie, vergiss dass sie da sind.

Ich hoffe wir können eine stärkere und breitere Basis von Mitwirkenden bei FreeBSD bekommen, und ich hoffe dass wir zusammen die alten Knauser und die [Ausgelassen]en dieser Welt daran hindern können, sie durchzukauen, auszuspukken und abzuschrecken bevor sie überhaupt ein Bein auf den Boden kriegen.

An die Leute die da draußen gelauert haben, vor diesen Untieren zurück geschreckt haben: Ich kann mich nur entschuldigen und euch dazu ermutigen, es trotzdem zu versuchen, das ist nicht die Umgebung die ich in dem Projekt haben möchte.

Poul-Henning

Anhang D. Beispiel-Anleitung für das Melden von Fehlern

Dies ist eine leicht abgewandelte Kopie der online verfügbaren Anleitung des Subversion-Prokjekts für neue Benutzer, wie Fehler gemeldet werden sollten. Siehe „Behandeln Sie jeden Nutzer wie einen möglichen Freiwilligen“ im Kapitel Kapitel 8, *Leitung von Freiwilligen*, wo dargestellt wird, warum es wichtig ist, dass ein Projekt derartige Anleitungen hat. Das ursprüngliche Dokument befindet sich bei http://svn.collab.net/repos/svn/trunk/www/bugs.html.

```
                    So melden Sie Fehler in Subversion


Dieses Dokument erläutert, wie und wo Fehler Sie Fehler melden sollten.
(Es ist keine Liste aller noch bestehender Fehler — diese können Sie
stattdessen hier bekommen.)


Wo Sie einen Fehler melden sollten
----------------------------------

    * Liegt der Fehler in Subversion selbst, senden Sie eine E-Mail
      an users@subversion.tigris.org. Sobald der Fehler bestätigt wird,
      kann jemand, vielleicht Sie, ihn in das Ticketsystem eingeben.
      (Oder wenn Sie sich ziemlich sicher sind, können Sie auch
      gleich an unsere Entwickler-Liste schreiben,
      dev@subversion.tigris.org. Sind Sie sich aber nicht sicher, ist
      es besser zuerst an users@ zu schreiben; dort kann Ihnen
      jemand sagen, ob das Verhalten, das Sie beobachtet haben,
      beabsichtigt ist oder nicht.)


    * Liegt der Fehler in der APR-Bibliothek, melden Sie dies
      bitte auf einer dieser Mailinglisten: dev@apr.apache.org,
      dev@subversion.tigris.org.


    * Liegt der Fehler in der Neon-HTTP-Bibliothek, melden Sie
      dies bitte bei: neon@webdav.org, dev@subversion.tigris.org.


    * Liegt der Fehler in Apache-HTTPD 2.0, melden Sie dies
      bitte an die beiden folgenden Mailinglisten: dev@httpd.apache.org,
      dev@subversion.tigris.org. Auf der Mailingliste der
      Apache-httpd-Entwickler herrscht Hochbetrieb, es ist also möglich,
      dass Ihre Meldung übersehen wird. Sie können auch eine Bug-Meldung
      unter http://httpd.apache.org/bug_report.html einreichen.

    * If the bug is in your rug, please give it a hug and keep it snug.
```

Wie Sie einen Bug melden sollten

Vergewissern Sie sich erst, dass es ein Fehler ist. Wenn Subversion sich nicht so verhält wie Sie es erwarten, schauen Sie in der Dokumentation und den Archiven der Mailverteiler nach und weisen Sie nach, dass es sich so verhalten sollte, wie Sie es erwarten. Wenn es allerdings etwas Offensichtliches ist, wie wenn Subversion mal eben Ihre Daten zerstört und Ihren Bildschirm dazu bringt, Rauch auszuspucken, dann können Sie Ihrem Urteil vertrauen. Wenn Sie sich aber nicht sicher sind, fragen Sie lieber zuerst auf der Benutzer-Mailingliste (users@subversion.tigris.org) oder im IRC bei #svn auf irc.freenode.net nach.

Wenn Sie sich dann sicher sind, dass es ein Fehler ist, ist es das wichtigste, eine einfache Beschreibung und eine Anleitung zu finden, nach der der Fehler reproduziert weden kann. Wenn der Fehler als Sie ihn ursprünglich gefunden haben, fünf Dateien über zehn Commits brauchte, versuchen Sie ihn mit nur einer Datei und einem Commit auszulösen. Je einfacher die Anleitung, desto wahrscheinlicher wird ein Entwickler ihn erfolgreich reproduzieren und beheben können.

Wenn Sie die Anleitung schreiben, erklären Sie nicht einfach in Worten, was Sie gemacht haben, um den Fehler zu bekommen. Schreiben Sie stattdessen ein genaues Protokoll, welche Befehle Sie eingegeben haben und was diese ausgegeben haben. Sie sollten dies über Kopieren und Einfügen tun. Wenn Dateien dafür gebraucht werden, sollten Sie die Namen der Dateien angeben, und sogar ihren Inhalt, wenn Sie meinen, dass diese relevant wären. Am besten ist es, wenn Sie Ihre Anleitung in Form eines Skripts bündeln würden, so etwas hilft ungemein.

*Nur kurz um etwas aus dem Weg zu räumen: Sie *haben* doch die neuste Version, oder? :-) Der Fehler wurde vielleicht schon behoben; Sie sollten deshalb Ihre Anleitung mit der aktuellen Entwickler-Version von Subversion ausprobieren.*

Zusätzlich zu der Anleitung für die Reproduktion, brauchen wir auch eine komplette Beschreibung der Umgebung in der du den Fehler reproduziert hast. Das heißt:

* Ihr Betriebssystem
* Die Version und/oder Revision von Subversion
* Der Compiler und die Einstellungen die Sie benutzt haben um Ihr Build von Subversion zu erstellen
* Etwaige Änderungen, die Sie an Subversion vorgenommen haben
* Die Version von Berkeley DB die Sie mit Subversion benutzen, falls überhaupt
* Alles andere, was möglicherweise relevant sein könnte. Wobei

Sie lieber zuviel Informationen geben sollten, als zu wenig.

Wenn Sie all dies gemacht haben, sind Sie bereit die Meldung zu schreiben. Fangen Sie mit einer klaren Beschreibung des Fehlers an. D.h. sagen Sie, wie Sie erwarten, dass sich Subversion verhalten sollte, und im Gegensatz dazu wie es sich wirklich verhalten hat. Auch wenn der Fehler Ihnen offensichtlich erscheint, muss das nicht unbeningt für jemand anderes gelten, also vermeiden Sie am besten Ratespiele. Danach sollten Sie sich der Beschreibung der Umgebung zuwenden und der Anleitung zur Reproduktion des Fehlers. Wenn Sie auch eine Spekulation über die Ursache anstellen wollen, oder sogar einen Patch anbieten können, um den Fehler zu beheben, wäre das super — siehe http://subversion.apache.org/docs/community-guide/#patches für eine Anleitung, wie Patches eingereicht werden sollten.

Schreiben Sie all diese Informationen an dev@subversion.tigris.org, oder wenn Sie bereits dort gewesen sind und darum gebeten wurden ein Ticket anzulegen, gehen Sie direkt zum Bugtracker und folgen der dortigen Anleitung.

Danke. Wir wissen, dass es eine Menge Arbeit ist, eine effektive Bug-Meldung zu schreiben, aber eine gute Meldung kann einem Entwickler Stunden der Arbeit ersparen, und der Fehler wird so viel eher behoben.

Anhang E. Copyright

License:

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Adaptation" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

   b. "Collection" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.

   c. "Creative Commons Compatible License" means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.

d. "Distribute" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

e. "License Elements" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

f. "Licensor" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

g. "Original Author" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

h. "Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

i. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

j. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or

public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

k. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights.

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant.

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

e. For the avoidance of doubt:

i. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

iii. Voluntary License Schemes. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

b. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a

Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

c. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose

of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability.

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this

License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in

the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of the License.

Creative Commons may be contacted at http://creativecommons.org/.