Koordination und Kommunikation in Open-Source-Projekten

MATTHIAS ETTRICH

Die Entwicklung von Software ist ein komplizierter und schwer vorhersehbarer Prozess. Komplexe Softwaresysteme neigen nicht nur dazu, fehlerbehaftet zu sein, sondern in der Herstellung auch teurer zu werden, als erwartet, und immer hinter ihrem Zeitplan zurückzuliegen. Die Antworten der Softwaretechnik heißen Prinzipien, Methoden, komplexe mehrphasige Systementwicklungsmodelle und organisierendes Projektmanagement. Umso erstaunlicher sind die technischen Erfolge in der freien Softwareentwicklung. Die offensichtlichen Vorteile sind der fehlende Zeitdruck und die fehlenden Kosten. Doch auch das ist kaum hinreichend, um zu erklären, wie ein loses, weltweites Netzwerk von Hackern ohne erkennbare Planung und kaum erkennbarer Organisation es schafft, Softwaresysteme im Millionen-Zeilen-Bereich zu erstellen, zu pflegen und weiterzuentwickeln – und das unbezahlt und in ihrer Freizeit. Wie schaffen es freie Projekte, ohne autorisierte Projektleitung und Management, notwendige Entscheidungen zu treffen und die anfallenden Aufgaben sinnvoll zu verteilen?

1. Projektleitung: Entscheidungsfindung und die Rolle der Maintainer

Das Grundprinzip der Entscheidungsfindung in freien Softwareprojekten ist einfach: Diejenigen, die die Arbeit tun, entscheiden. Im Umkehrschluss bedeutet das, dass jeder Teilnehmer selbst entscheidet, woran er oder sie arbeiten will, und das dann auch tut. Wenn jeder genau das macht, was er oder sie will, warum kommt dann etwas zustande, das von außen betrachtet klare Züge von Strategie und richtigen Entscheidungen trägt und normalen Endanwendern zugute kommt? Oder umgekehrt gefragt: Wenn alle nur machen, was sie wollen, und keine kontrollierende und leitende Instanz existiert, warum gibt es dann nicht endlose Diskussionen und ein heilloses Durcheinander? Letzte Antworten auf diese Fragen lassen sich ohne gründliche und systematische Forschung nicht geben. Derzeit wissen wir, dass es funktioniert, ahnen aber lediglich, warum. Hier sind einige lose Gedanken, die zum besseren Verständnis des Entscheidungsprozesses in freien Softwareprojekten beitragen können:

- In jedem Betrieb und in jeder Arbeitsgruppe gibt es Mitarbeiter, die so gut sind, dass man sie weder beaufsichtigen noch herumkommandieren muss. Sie wissen, was zu tun ist. Sie sind kompetent, sie sind motiviert und sie sind der Traum eines jeden Chefs. In jedem erfolgreichen Open-Source-Projekt gibt es solche Mitarbeiter: Überdurchschnittlich intelligente, kreative und nicht zuletzt gebildete Menschen mit einem breiten Interesse an Informationstech-

nologie und ihren Anwendungen. Zwischen ihnen entsteht ein Gleichgewicht, das dazu führt, dass jede oder jeder genau an den Dingen arbeitet, die er oder sie am besten kann. Projekte sind nur dann erfolgreich, wenn sie von solchen Mitarbeitern getragen werden, und sie ziehen solche Mitarbeiter an, wenn sie erfolgreich sind. Das ist vergleichbar mit anderen Teams. Eine Fußballmannschaft ist nur dann wirklich gut, wenn sie gute Spieler hat. Umgekehrt wechseln gute Spieler am ehesten zu einer guten Mannschaft. Gute Teams werden damit besser, mittelmäßige oder schlechte Teams schlechter. Es wäre falsch zu behaupten, alle Entwickler im Open-Source-Umfeld seien technisch überragend. Nur ein Bruchteil aller gestarteten Open-Source-Projekte ist wirklich erfolgreich und schafft es, hinreichend neue Entwickler und Anwender zu begeistern. Die sich durchsetzenden Projekte aber schaffen es unter anderem deshalb, weil sie von wirklich guten Entwicklern getragen werden.

- In einem technischen Softwareumfeld trifft man selten Entscheidungen; man findet Lösungen. In einem solchen Lösungsfindungsprozess bewertet man Alternativen und wählt die beste, oder zumindest die jeweils am besten realisierbare aus. „Richtig" und „falsch" werden innerhalb eines solchen logischen Bezugssystems nahezu objektive Begriffe. In einer Welt, in der sich im allgemeinen leicht beweisen lässt, wer recht hat, und alle Parteien offen für logisch korrekte Beweisführungen sind, gibt es naturgemäß wenig Konflikte.
- In Fragen in denen keine Einigung erzielt werden kann, weil unterschiedliche Werte, unbeweisbare Annahmen oder verschieden geartete Zielsetzungen dies verhindern, wird ein evolutionärer Ansatz verfolgt: Jede Gruppe geht ihren eigene Weg, bis entweder objektive Kriterien das erfolgreichere Konzept ausweisen oder sich ein Weg als Einbahnstraße herausstellt. Schließen sich zwei Konzepte gegenseitig aus, wird das erfolgreichere das unterlegene Konzept verdrängen, weil es mehr Anwender und letztendlich mehr Entwickler anzieht. Ungesunde Konflikte werden damit durch gesundes Konkurrieren gelöst.

Ähnliche Konzepte lassen sich in traditionellen Softwareunternehmen nur begrenzt umsetzen. Entscheidungen und Strategie sind hier nicht auf rein technische Kriterien beschränkt, sondern müssen eine Vielzahl weiterer Faktoren berücksichtigen. Technologie muss nicht nur nützlich und damit prinzipiell verkaufbar sein, sondern sie muss von dem jeweiligen Unternehmen in dessen spezifischem Marktsegment gewinnbringend absetzbar sein. Softwareentwicklung ist teuer, also kann ohne strategische Planung nicht einfach ins Blaue hinein programmiert werden. Wo es aber keine bezahlten Arbeitsplätze gibt, kann auch keiner auf Grund einer strategischen Fehlentscheidung seinen Arbeitsplatz verlieren. Und wo Softwareentwicklung nichts kostest, wird durch langwierige und später verworfenene Fehlentwicklungen kein Geld verschwendet. Planung und Strategie können damit risikofrei durch Versuchen und Ausprobieren ersetzt werden, Konstruktion wird zu Evolution.

Hierzu kommen die hohe technische Qualifikation, die Lernbereitschaft und die Motivation der Mitarbeiter, die zu einer Produktivität führen, die über dem in-

dustriellen Durchschnitt zu liegen scheint. Zwei Dinge mögen hier entscheidend sein: Erstens leisten Menschen mehr, wenn sie zu 100% hinter den ihnen aufgetragenen Aufgaben stehen und selbst Entscheidungen treffen dürfen. Zweitens findet ein natürlicher Auswahl- und Anpassungsprozess statt. Ein unqualifizierter Kollege in einem Open-Source-Projekt kann nach einem missglückten Anlernversuch ohne Konsequenzen einfach freundlich ignoriert werden, ausgebrannte Kollegen verlassen das Projekt schon alleine dadurch, dass sie nichts mehr beitragen. Wenn ein Teilprojekt keinen Spaß mehr macht, wird es liegen gelassen, bis es ein anderer übernimmt, dem es wirklich Spaß macht. Solche Möglichkeiten haben Unternehmen nicht, sie müssen sich der sozialen Verantwortung stellen, die mit der vertraglichen Bindung zwischen Arbeitnehmer und Arbeitgeber einhergeht, und sie haben eine Verantwortung ihren bezahlenden Kunden gegenüber.

Ganz ohne Struktur kommt jedoch auch ein freies Projekt nicht aus. Diese Struktur besteht aber nicht aus vordefinierten Rollen mit Arbeitsbeschreibungen, die in einem geordneten Verfahren mit den am besten geeigneten Kandidaten besetzt werden. Sie entsteht vielmehr dynamisch aus der Gruppe heraus, der natürlichen, sozialen Veranlagung des Menschen folgend. Genauso wie eine kleine Gruppe Menschen in der Lage ist, gemeinsam zu spielen, zu jagen, zu kochen oder einen Frühjahrsputz in der Wohnanlage durchzuführen, so ist es auch möglich, gemeinsam Software zu entwickeln. Jeder sucht sich die Rolle, die am besten zu ihm oder ihr passt, und einige übernehmen automatisch eine Führungsrolle in einem gewissen Bereich, getrieben vom Charakter und gestützt auf durch Arbeit und Wissen erworbenen Respekt der Mitmenschen. Eine solche auf persönliche Relationen gestütze Selbstorganisation skaliert natürlich nicht. Dass es bei größeren Softwareprojekten dennoch funktioniert, verdanken wir einer besonderen Eigenschaft guter Software: Sie ist modular aufgebaut, aus relativ unabhängigen Teilen. Große freie Softwareprojekte folgen der Struktur der Software. Für jedes unabhängige Teil gibt es in der Regel auch ein unabhängiges Teilprojekt, das sich nach den oben genannten Prinzipien selbst organisiert.

Eine Rolle, die sich in allen Teilprojekten herauskristallisiert, ist die Rolle des Verwalters oder *Maintainers*. Diese Entwickler führen alle Fäden zusammen und tragen letztlich die Verantwortung. Je nach Persönlichkeit des Betreffenden kann sich die Tätigkeit im Einzelfall zwischen aktiver Projektleitung oder reaktiver Serviceleistung bewegen. Die ersten Maintainer sind immer die ursprünglichen Autoren des Codes. Später, wenn sich diese anderen Aufgaben zuwenden, werden es in der Regel diejenigen Entwickler, die am aktivsten am Code arbeiten. Zu den Aufgaben der Maintainer gehört das Überprüfen von Code-Beiträgen, das Sicherstellen der Funktionsfähigkeit, das Ausarbeiten und Einhalten eines Releaseplanes und das Kommunizieren mit Mitentwicklern. Wenn Entwickler Codeänderungen vorschlagen bzw. Änderungen in Form von Patches einreichen, hat ein Maintainer die Möglichkeit, diese anzunehmen, für eine spätere Version zurückzustellen, oder sie zurückzuweisen. Dies ist eine sehr wichtige Aufgabe und mit einer großen Verantwortung verbunden. Der falsche Mann oder die falsche Frau auf einer solchen Position kann einem Projekt viel Schaden zufügen. Umgekehrt kann der richtige Maintainer viel Gutes für ein Projekt tun. Einen Patch zurückzuweisen kann auf eine Art gesche-

hen, bei der sich der Entwickler selbst zurückgewiesen fühlt. Oder es geschieht auf eine Art, bei der der Entwickler mit Freuden den Patch überarbeitet. Lässt man die technischen Aufgaben beiseite, gleicht die Rolle des Maintainers der Rolle eines Personalmanagers. Es geht darum, die Mitarbeiter am Erfolg des Projektes teilhaben zu lassen, sie fühlen zu lassen, dass das Projekt allen gehört. Und es geht darum, neue Entwickler für das Projekt zu gewinnen. Das geschieht beispielsweise dadurch, dass „dumme" Fragen nicht belächelt, sondern beantwortet werden, und dass unerfahrene Entwickler nicht schief angeguckt und ausgegrenzt werden, sondern ihnen geholfen wird, Erfahrungen zu sammeln und besser zu werden. Zusätzlich nehmen manche Maintainer auch die Rolle eines Schlichters wahr, falls es einmal zu emotional geladenen Diskussionen im Team kommen sollte.

Bei funktionierenden Projekten sind die Maintainer die am besten für diese Aufgabe geeigneten Entwickler. Sie sind diejenigen, die in den Augen ihrer Mitentwickler am meisten über den Code wissen, am aktivsten daran arbeiten und auch in nicht-technischen Fragen rund um das Projekt den Respekt ihrer Mitentwickler genießen. Der Grund ist basisdemokratisch und einfach: Ein schlechter oder inaktiver Maintainer wird schnell durch einen anderen ersetzt, im schlimmsten Fall über das Spalten des Projektes, auch *forking* genannt. Die Mechanismen sind dieselben wie bei einer Hobbyfußballmannschaft auf einem Bolzplatz: Benimmt sich der Kapitän daneben, spielt er schlecht oder bezieht er andere Spieler nicht genug ein, so übernimmt entweder ein anderer den Job, die Spieler wechseln zu einer anderen Mannschaft oder sie machen ihre eigene Mannschaft auf.

Bei aller Analyse bleibt die einfache Einsicht, dass Softwareentwicklung erstaunlich reibungslos funktioniert, wenn Softwareentwickler das Sagen haben. Das ist an sich nicht verwunderlich: Mathematiker, Informatiker, Physiker und andere natur- und strukturwissenschaftlich ausgebildete Akademiker stellen eine mathematisch-logische Elite, der die Verständigung mit ihresgleichen relativ leicht fällt. Erstaunlicher ist, dass viele Unternehmen dieses wertvolle intellektuelle Kapital ungenutzt zu lassen scheinen und keinen technischen Karriereweg für produzierende Ingenieure anbieten. Die Comic-Serie „Dilbert", von allen mir bekannten Programmierern gerne gelesen, zeugt davon. Programmierer werden als bloße Implementierer eingesetzt, während strategische Entscheidungen auf sogenannten „höheren" Ebenen getroffen werden, dominiert von Leuten mit ein- oder zweijährigen MBA-Ausbildungen oder ehemaligen Programmierern, die den Kontakt zum Produkt und seiner Entwicklung längst verloren haben. Die Freie-Software-Gemeinschaft stellt den radikalsten Gegenentwurf zur dilbertesken Ingenieursausbeutung dar: Eine Gemeinschaft, basierend auf geteiltem Expertenwissen und gemeinsamen Interessen, in der Rang nur durch technisches Wissen und Taten erlangt und erhalten wird.

Mit fröhlichem „Draufloshacken" in vielen kleinen Teams ist es aber nicht getan. Software will auch qualitätsgesichert, dokumentiert, lokalisiert und veröffentlicht sein. Was in einem kleinen Team noch ein relativ überschaubarer Prozess ist, stellt sich in größeren Projekten als fast übermenschliche Herausforderung dar, zumal Softwarentwickler vor geplanten Releases gerne dazu tendieren, „noch eben schnell" ein neues Feature einzubauen, das „ganz bestimmt" nichts kaputtmacht.

Open-Source-Jahrbuch 2004, hrsg. von Robert A. Gehring und Bernd Lutterbeck, Berlin (Lehmanns Media) 2004.

2. Planung und Projektkoordination

Planung in Open-Source-Projekten entsteht nicht zuletzt aus der Notwendigkeit koordinierter Releases. Software durchläuft einen Reifezyklus, der von einer ersten Veröffentlichung über diverse *Alpha-* und *Beta-Releases* zu einer endgültigen Version führt. In diesen Testphasen wird versucht, die Software einer möglichst breiten Anwendergruppe schmackhaft zu machen, um über frühe Rückmeldungen Fehler schnell beheben zu können. In einem großen Open-Source-Projekt mit dutzenden Teilprojekten und hunderten von Mitentwicklern ist es eine Herausforderung, die Test- und Veröffentlichungszyklen aller Teilprojekte zu synchronisieren, ohne ein allzu großes Maß an die Kreativität und Freude hemmender Bürokratie einzuführen. Verschiedene Projekte haben hier verschiedene Lösungen gefunden. Allen Lösungen gemein ist die Definition sogenannter Meilensteine, ab denen die Software „eingefroren" wird. Eingefroren bedeutet, dass nur noch Änderungen erlaubt sind, die dokumentierte Fehler beheben, nicht aber Änderungen, die neue Funktionalität hinzufügen.

Klassische strategische Entwicklungsplanung im Sinne von: „Wer arbeitet wann und woran?" findet kaum statt, kann aber mangels Weisungsbefugnis auch nicht stattfinden. Dies stellt eine interessante Herausforderung an das Entwicklungsmodell: Wie verhindert man auseinanderdriftende, inkompatible Entwicklungen? Was passiert, wenn verschiedene Entwickler oder Gruppen von Entwicklern parallel an verschiedenen Teilen der Software arbeiten, und sich Wochen später herausstellt, dass die Neuentwicklungen überhaupt nicht zusammenpassen? Betriebe leisten sich hierfür Projektleiter, vorausschauende Planung und, falls es nötig sein sollte, ein Meeting aller involvierten Entwickler. Diese Mittel stehen im verteilten Entwicklungsmodell im Open-Source-Umfeld in der Regel nicht zur Verfügung. Ziel ist es deshalb, Parallelität und den dadurch geschaffenen Bedarf an Kommunikation und Planung möglichst zu vermeiden. Dies glückt im Regelfall dank zwei einfacher Mechanismen:

- Alle Mitentwickler arbeiten immer mit der jeweils aktuellen Version des gemeinsamen Codes.
- Code-Änderungen werden möglichst früh und häufig in den gemeinsamen Code-Bestand eingespielt.

„Jeweils aktuelle Version" bedeutet hierbei nicht das letzte öffentliche Release, sondern der aktuelle, häufig erst wenige Stunden oder Minuten alte Code. Ein Arbeitstag beginnt in der Regel mit einem Softwareupdate auf die letzte Version und endet mit dem Einspielen der geschaffenen Änderungen. Dies legt dem einzelnen Softwareentwickler eine interessante Verantwortung auf: Die Software muss für Mitentwickler auch nach dem Einspielen der Änderungen funktionsfähig und damit einsetzbar sein. Ich kann zum Beispiel nicht größere strategische Umbauten vornehmen, wenn dies andere Entwickler über mehrere Wochen stark behindern würde, weil deren Software auf Funktionen meiner Software angewiesen ist. Das Entwicklungsmodell gleicht damit in gewisser Weise dem Straßenbau: Auch hier muss im laufenden Betrieb ausgebessert und erweitert werden. Muss eine Straße einmal für kürzere Zeit vollständig gesperrt werden, muss zunächst eine Umleitung gefunden

und ausgeschildert werden. Was zunächst als Nachteil erscheint, entpuppt sich in der Praxis als qualitätssichernder Vorteil. Softwareentwicklungsprozesse wie das populäre „extreme Programmieren“ (Wells 1999) fordern genau das gleiche: Jegliche Codeänderungen sollten am besten gleich, spätestens aber nach einem Tag in den Hauptquellbaum eingespielt und damit den Mitentwicklern zur Verfügung gestellt werden. Diese ständigen Updates helfen, böse Überraschungen inkompatibler Parallelentwicklungen zu vermeiden, und sie sorgen dafür, dass die Software auch während des Entwicklungszyklus niemals gänzlich auseinanderfällt, sondern jederzeit verwendbar ist.

Auch wenn die Software jederzeit funktionsfähig ist, und das Projekt es schafft, Releases und Testzyklen zu koordinieren, – wie stellt man sicher, dass nicht an den Bedürfnissen der Anwender vorbei entwickelt wird? Im traditionellen Softwareentwicklungsprozess kommen an dieser Stelle strategische Planungen ins Spiel, die sich auf Kundenuntersuchungen, Marktstudien, Rückmeldungen der Verkaufsflotte und nicht zuletzt auf die Erfahrungen der Chefs im Marketing, Verkauf und Produktmanagement stützen. Kein Unternehmen kann es sich leisten, eine neue Version ihres Hauptproduktes herauszubringen, die die Kunden so nicht haben wollen. Verständlicherweise wird alles versucht, dieses schlimmste Szenario im Vorfeld auszuschließen.

Freie Software hat keine bezahlten Spezialisten für Marketing, Verkauf oder Produktmanagement. Trotzdem stürzt sich eine große Benutzergemeinde auf jede neue Version, häufig noch vor der endgültigen Freigabe. Ein Grund hierfür ist sicherlich der günstige Preis von 0 Euro. Viel wichtiger aber ist die grundlegende Anwenderorientierung Freier Software. Aus Prinzip ist Freie Software im höchsten Maße an den aktuellen und dringlichsten Bedürfnissen der Anwender ausgerichtet, weil es die Anwender selbst sind, die die Software weiterentwickeln. An erster Stelle bei der Implementierung einer bestimmten Funktionalität steht fast immer der Wunsch, diese Funktionalität selbst nutzen zu können. Kommerzielle Softwarehersteller argumentieren manchmal, Freie Software sei etwas für Freaks und Technikverliebte, nicht aber für Endanwender. Das ist unaufrichtig argumentiert, denn auch Informatiker und andere naturwissenschaftlich ausgebildete Akademiker sind Endanwender, und es sind diese Endanwender, die für die Weiterentwicklung der Software sorgen. Was die Anwenderfreundlichkeit kommerzieller Software angeht, muss darauf hingewiesen werden, dass in einem kommerziellen Umfeld der Anwender nur indirekt auf die Software einwirken kann, nämlich durch seine Kaufentscheidung. Neue Funktionen werden dann implementiert, wenn sie sich verkaufen lassen oder dem Hersteller andere direkte Vorteile bringen. Im Umkehrschluss bedeutet dies, dass für den Anwender wichtige Funktionen nicht implementiert werden, wenn sich diese für den Hersteller negativ auswirken könnten. Offene Standards, definierte Protokolle, Zusammenarbeit mit anderen Produkten und die generelle Anpassungsfähigkeit der Software sind hier als Konfliktfelder zu nennen. Freie Software steht bei solchen Konflikten immer auf Seiten des Anwenders – eine Software, die von Anwendern für Anwender entwickelt wird. Dahingestellt sei, dass Freie Software manchmal die Hürden für Einsteiger etwas höher legt.

Open-Source-Jahrbuch 2004, hrsg. von Robert A. Gehring und Bernd Lutterbeck, Berlin (Lehmanns Media) 2004.

Doch wie schafft es ein freies Softwareprojekt, Anwender dafür zu begeistern, die Software selbst zu verbessern? Wie kommen Informatiker dazu, umsonst zu programmieren, statt als Berater gutes Geld zu machen? Und was treibt Anwender ohne Programmiererfahrung dazu, lieber programmieren zu lernen, um fehlende Funktionalität selbst implementieren zu können, anstatt einfach ein kommerzielles Produkt käuflich zu erwerben?

3. Die Projektgemeinde

In einem Open-Source-Projekt gibt es nicht nur Programmierer, sondern auch Tester, Grafikdesigner, technische Schreiber für die Dokumentation und Übersetzer, die die Software und die Dokumentation an verschiedene Sprachen anpassen. Die Übergänge sind fließend. So beginnen zum Beispiel viele Programmierer ihre Freie-Software-Laufbahn als Übersetzer. Das Gros der Mitarbeiter kommt immer aus der Anwendergemeinde der Software selbst. Übersetzer beginnen typischerweise als Anwender, die sich daran stören, dass Software nicht oder nur unzureichend lokalisiert ist. Zufriedene Anwender werden schnell zu Testern, weil sie sich für die neuesten Versionen interessieren. Fortgeschrittenere Tester haben Gedanken und Vorschläge, wie die Software zu verbessern sei. Über diese Vorschläge kommen sie in Kontakt mit den Entwicklern und lernen diese besser kennen. Über kurz oder lang entsteht der Drang, selbst Änderungen an der Software vorzunehmen. Aus anfänglich kleinen Patches werden schnell größerere Patches, und ehe man sich versieht, ist man Teil einer internationalen Community.

Daneben spielt aber auch die persönliche Anwerbung eine wichtige Rolle. Viele Entwickler kommen zur Freien Software, weil sie jemanden kennen, der bereits in einem Projekt mitarbeitet. Das können entweder alte Bekanntschaften sein, aus einem Leben vor oder außerhalb der Freien Software, oder aber neue Bekanntschaften, die auf Messen oder Kongressen gemacht wurden.

Diese Erklärungsmodelle sind aber nicht gänzlich hinreichend. Sie erklären, warum Entwickler dazu beitragen, ein Projekt zu vollenden. Sie erklären aber nicht, was Entwickler dazu treibt, ein neues Projekt anzufangen. „Ein neues Projekt" heißt in diesem Fall nicht unbedingt losgelöst von allen anderen Projekten. Alle größeren Softwareprojekte sind modular aufgebaut. Neben einem zentralen Kern bestehen sie aus einer Vielzahl kleinerer, relativ unabhängiger Teilprojekte. Wie eigenständige Projekte auch, bedarf jedes Teilprojekt zumindest eines Initiators, und einer großen Zahl Vollender. Initatoren werden nicht angeworben, sondern von ihrem inneren, kreativen Drang getrieben. Ein kurzer Artikel über ein erfolgreiches Softwareprojekt in einem Computermagazin kann schon ein Auslöser sein, diesem Drang freien Lauf zu lassen.

Neue Mitarbeiter in einem Unternehmen durchlaufen üblicherweise ein Trainingsprogramm. Sie bekommen einen Mentor als Ansprechpartner, der sie mit ihren Kollegen und Kolleginnen bekannt macht, mit der Unternehmensstruktur und den für sie relevanten Prozessen. In einem freien Projekt gibt es weder Einführungskurse noch professionelle Betreuer und schon gar keine Meetings mit physischer Anwesenheit. Trotzdem integrieren freie Softwareprojekte ständig neue Mitarbeiter aus al-

len Teilen der vernetzten Welt, die nach kurzer Anlernungszeit effizient und koordiniert zusammenarbeiten. Wie funktioniert diese Kommunikation?

4. Kommunikation in Open-Source-Projekten

Die Kommunikation unter den Teilnehmern eines Open-Source-Projektes steht auf zwei wichtigen Säulen: Dem Internet als schnelle und billige Möglichkeit, Daten auszutauschen und der englischen Sprache als international akzeptiertes menschliches Sprachprotokoll. Englisch hat sich als der kleinste gemeinsame Nenner klar durchgesetzt, als Sprache der Wahl auf Konferenzen, im Web, im Programmcode, für Dokumentation und für die Kommunikation der Entwickler untereinander. Das hat nichts mit falsch verstandenem Internationalismus à la Telekom-Werbesprüchen zu tun, oder gar einer Missachtung der eigenen Sprache. Die Gründe sind vielmehr rein pragmatischer Natur: Kein freies Projekt kann es sich leisten, Entwickler anderer Sprachgruppen auszugrenzen. Dazu ist das Interesse, an Freier Software zu arbeiten, einfach zu gering. Selbst in verhältnismäßig großen Ländern wie Deutschland oder Frankreich lässt sich national kaum eine kritische Masse von Teilnehmern erreichen. Und selbst wenn ein nationales Softwareprojekt technisch erfolgreich wäre – gegen die Konkurrenz eines internationalen Projektes, das Kompetenz aus allen Teilen der vernetzten Welt rekrutierten kann, hätte es keine Chance. Verständlichkeit steht dabei über sprachlicher Richtigkeit oder gar Sprachästhetik: die Mehrheit der Programmierer haben Englisch als zweite oder dritte Fremdsprache erlernt, zum Teil erst durch das Programmieren im Erwachsenenalter. Anglisten bzw. Linguisten mit Hang zur Pedanterie mag das missfallen. Freunde aktiver Sprachentwicklung und neuentstandener Kreoldialekte können dabei aber viel Spaß haben.

Neben Englisch setzen Open-Source-Entwickler eine Vielzahl von technischen Hilfsmitteln zur Kommunikation und Zusammenarbeit über das Internet ein. Das gemeinsame Grundmerkmal dieser Werkzeuge ist, dass sie wiederum frei sind. Dies ermöglicht allen teilnehmenden Entwicklern, die gleichen Werkzeuge zu benutzen und falls notwendig anzupassen und zu verbessern. Es ist aber auch eine politische Dimension dabei: Das Einsetzen nicht-freier Werkzeuge im Entwicklungsprozess wird allgemein als Makel empfunden und nur in Ausnahmefällen toleriert, bei denen keine langfristigen Abhängigkeiten geschaffen werden.

Die am häufigsten verwendeten technischen Werkzeuge sind:

E-Mail:

Persönliche E-Mail von einem Entwickler zu einem anderen ist ein unschätzbarer Kommunikationskanal. E-Mail ist schnell und kann für verschiedenste Arten von Daten, auch größerer Mengen, benutzt werden. Der Erhalt einer E-Mail stört nicht, und die Empfänger einer Nachricht können selbst entscheiden, wann und ob sie diese beantworten wollen.

Noch viel wichtiger als persönliche Nachrichten sind allerdings sogenannte Mailinglisten. Mailinglisten werden üblicherweise für ein bestimmtes Thema eingerichtet, das sich auch im Namen der Liste wiederspiegelt. Wenn als Beispiel für das KDE-Projekt eine Konferenz im tschechischen Nove Hrady in Südböhmen geplant

wird, geschieht dies auf einer Mailingliste *novehrady@kde.org*. Alle an der Konferenz Interessierten können die Nachrichten, die auf diese Liste geschickt werden, mitlesen und auch selbst Nachrichten an die Liste schicken. Für alle anderen wird üblicherweise ein elektronisches Archiv zur Verfügung gestellt, aus dem man sich über vorangegangene Diskussionen informieren kann.

Mailinglisten als Medium sind nicht ganz unproblematisch. Ihr größter Vorteil – die einfache Benutzung – ist auch ihr größter Nachteil: Jeder kann sie benutzen. Größere Projekte, oder Projekte die sich mit aktuell sehr interessanten Themen beschäftigen, leiden daher oftmals darunter, dass ihre Kommunkationskanäle von Beiträgen aus der Peripherie des Projektes überschwemmt werden, von Anwendern, beiläufig Interessierten oder einfach nur Leuten, die sich langweilen und gerne im Netz ihre Meinungen verkünden. Diese offenen Diskussionen sind wichtig, um das Interesse an einem Projekt zu fördern, können aber für die Arbeit des Projektes äußerst hinderlich sein. Wenn Entwickler jeden Abend nur ein bis zwei Stunden Zeit für Freie Software haben, möchten sie verständlicherweise die Hälfte dieser Zeit nicht mit dem Lesen von E-Mails verbringen, nur um die wenigen relevanten Beiträge herauszufiltern.

Das Verhältnis von relevanten zu irrelevanten Beiträgen wird als „signal/noise ratio" bezeichnet. Überwiegen die irrelevanten oder weniger relevanten Beiträge auf einer Mailingliste (niedrige „signal/noise ratio"), werden überlichweise weitere Mailinglisten eingerichtet, die sich mit spezielleren Themen beschäftigen (weniger Beiträge, dafür aber höhere „signal/noise ratio"). Der Schreibzugriff auf solche Listen kann technisch auf eine bestimmte Gruppe eingeschränkt werden. Oftmals entscheiden die Mitglieder einer solchen eingeschränkten Liste darüber, wer neu aufgenommen wird und welche Kritierien neue Mitglieder erfüllen müssen.

Internet Relay Chat (IRC):

IRC ist ein offenes Chat Protokoll, manchmal auch als *Instant Messaging* bezeichnet. Es erlaubt einer Gruppe von Teilnehmer, sich direkt, Satz für Satz, über Tastatur zu „unterhalten". Auf IRC-Servern gibt es hunderte von verschiedenen Kanälen, die sich jeweils einem Thema widmen. IRC wird oft benutzt, um sich besser kennenzulernen oder einfach nur die Zeit zu vertreiben. Es ist aber auch eine Möglichkeit, „mal eben schnell" eine Antwort auf eine Frage zu bekommen, oder technische Probleme direkt zu diskutieren, ohne stundenlang komplexe E-Mails austauschen zu müssen. Bei Gruppen mit mehr als 4 Personen funktioniert Chat im Allgemeinen besser als eine Telefonkonferenz.

World Wide Web (WWW):

Am World Wide Web, umgangssprachlich nahezu zum Synonym für das Internet geworden, führt auch für die Freie-Software-Gemeinschaft kein Weg vorbei. Jedes Open-Source-Projekt unterhält eine Webseite, über die Informationen rund um das Projekt erhältlich sind, einschließlich der zum Herunterladen des Quellcodes. Häufig werden auch Web-basierte Diskussionsforen zu Themen rund um das Projekt angeboten. Diese öffentlichen Diskussionstafeln wurden in den letzten Jahren so erfolgreich, dass sie das klassische „News"-Protokoll aus Usenet-Tagen fast voll-

ständig verdrängt haben. Neuerdings werden auch dynamische Webseiten verwendet, *Wiki* genannt, die von Betrachtern der Seite direkt im Webbrowser verändert werden können (Leuf und Cunningham 2001).

File Transfer Protocol (FTP):

Vor einigen Jahren noch waren FTP-Server die wichtigsten Adressen, um Freie Software auszutauschen. Zwar hat das Web diese Funktion weitgehend übernommen, aber noch gehört FTP nicht ganz zum alten Eisen. Insbesondere für große Pakete wird es auch heute noch gerne eingesetzt. Das Prinzip ist einfach: Jedermann kann Dateien anonym in das öffentlich schreibbare (aber nicht lesbare) „Incoming-Verzeichnis" heraufladen. Ein Verwalter überprüft diese neuen Dateien regelmäßig und verschiebt sie nach erfolgreicher Prüfung in öffentlich lesbare Bereiche des Servers. Das manuelle Prüfverfahren verhindert, dass Softwarepiraten die freien Server zum Austausch geklauter, kommerzieller Software missbrauchen können.

Source Code Management (SCM):

Ein zentraler Bestandteil eines jeden Softwareprojektes ist ein zentrales Depot, in dem der gesamte Code eines Projektes – Programmcode, Grafiken, Übersetzungen und Dokumentation – verwaltet wird. Ein solches Depot wird als *Source Code Management System (SCM)* bezeichnet. Das zur Zeit am häufigsten eingesetzte System heißt *Concurrent Versions System (CVS)* (Vesperman 2003), der wahrscheinlichste Nachfolger nennt sich Subversion (CollabNet). Vereinfacht gesagt, ermöglicht ein solches System einer großen Gruppe von Entwicklern, zeitgleich an demselben Programm zu arbeiten und die Beiträge der Mitentwickler nachzuvollziehen. Jeder kann jederzeit für jede einzelne Zeile Code feststellen, wer die Zeile eingefügt oder verändert hat, zu welchem Zweck und wann. Im Bedarfsfall ist es auch einfach, einmal gemachte Änderungen wieder rückgängig zu machen oder das gesamte Projekt so zu rekonstruieren, wie es zu einem ganz bestimmten Zeitpunkt ausgesehen hat. Verwalter einzelner Module können auch automatisch informiert werden, falls jemand Änderungen eingespielt hat, oder den Zugang auf eine bestimmte Gruppe von Entwicklern beschränken.

Infrastrukturservices:

Vor einigen Jahren war es für freie Softwareprojekte noch eine Herausforderung, die zur Entwicklung notwendige Infrastruktur bereitzustellen. Meistens wurden Netzwerk-Bandbreite, Plattenplatz und Prozessorleistung für die Server von universitären Einrichtungen „abgezweigt", häufig ohne ausdrückliche Zustimmung der Leitung. Heute ist dies kein Problem mehr. Verschiedene Organisationen und Firmen bieten Rundumpakete an, die freie Projekte kostenlos nutzen können. Beispiele für solche Paketlösungen sind SourceForge,[1] das deutsche BerliOS Projekt,[2] oder der Savannah-Server der Free Software Foundation.[3] Die Zahl der in diesen vier Systemen beherbergten freien Softwareprojekte geht in die Zehntausende. Da-

1 Die Homepage des SourceForge-Projekts befindet sich unter http://sourceforge.net.
2 Näheres zu BerliOS unter http://www.berlios.de.
3 Zu finden unter http://savannah.gnu.org.

runter sind zugegebenermaßen viele gescheiterte oder verwaiste Projekte, aber auch viele gesunde Projekte im Wachstum.

Neben den technischen Werkzeugen darf ein wichtiges Kommunikationsmedium nicht vergessen werden: Der persönliche Kontakt von Mensch zu Mensch. Die meisten erfolgreichen Open-Source-Projekte beginnen früher oder später, sich real auf Konferenzen zu treffen, oder sie nutzen Community Events wie Messen oder allgemeine Veranstaltungen zu Freier Software als Rahmen, um sich zu treffen. Die Erfahrung lehrt, dass solche Treffen zu einem merkbaren Produktivitätsschub im Projekt führen, neben dem positiven Effekt der intensiv geleisteten Design- und Entwicklungsarbeit auf den Treffen selbst. Wir führen das einerseits auf gesteigerte Motivation zurück, andererseits darauf, dass Kommunikation über elektronische Medien deutlich besser funktioniert, wenn man den Gegenüber persönlich kennt. Der einzige Nachteil dieser Treffen, das Ausgrenzen von Entwicklern die aus diversen Gründen nicht teilnehmen können, wird dadurch mehr als ausgeglichen. Es ist trotzdem wichtig für freie Projekte, einer möglichst breiten Gruppe von Mitarbeitern die Teilnahme zu ermöglichen, u.a. durch die Wahl günstig gelegener und billiger Austragungsorte und durch direkte finanzielle Unterstützung finanzschwächerer Entwickler. Für Firmen oder andere Institutionen sind Kongresse und Entwicklertreffen die wahrscheinlich einfachste und effektivste Art, Freie Software finanziell zu unterstützen, ohne in die Projekte direkt einzugreifen.

Auch die beste Kommunikation kann nicht verhindern, dass Uneinigkeiten unter den Teilnehmern entstehen können. Solange das Ziel dasselbe ist, besteht Vermittlungspotential. Wenn aber die Zielsetzungen in einer Gruppe zu sehr auseinanderdriften, sind getrennte Wege oftmals die einzige Alternative zu endlosen und lähmenden Diskussionen. Viele Projekte sehen sich im Laufe ihrer Geschichte vor einer solchen „Gabellung“ – zu Englisch: „fork“.

5. Forking

Mit *forking* bezeichnet man das Abspalten eines neuen Projektes von einem bestehenden. Ein *fork* gilt häufig als ein Schreckensszenario innerhalb der Freie-Software-Gemeinschaft, spaltet er doch auch die Entwicklergruppe. Zusammenarbeit einer größeren Entwicklergruppe, so die Theorie, sei effektiver als zwei kleinere Entwicklergruppen, die untereinander konkurrieren. Wie man dazu auch stehen mag, die Möglichkeit zu einem *forking* ist eines der grundlegendsten Rechte, die ein Anwender Freier Software hat. Software, die diese Möglichkeit nicht bietet, kann nicht als Freie Software bezeichnet werden. Die ständige Gefahr eines potentiellen *forkings* zwingt Entwickler auf gleicher und respektvoller Ebene zusammenzuarbeiten. Einzelne Mitarbeiter einer kommerziellen Firma können damit drohen, die Firma zu verlassen und vielleicht ein paar Kunden mitzunehmen. Unzufriedene Mitarbeiter eines Open-Source-Projektes hingegen nehmen nicht nur ein paar Kunden mit, sondern gleich die ganze „Firma“ und einen guten Teil der Mitarbeiter.

Als Konzept wird *forking* also nicht nur toleriert, sondern als Voraussetzung für Freie Software geradezu eingefordert. In der Praxis durchgeführte *forkings* können

aber durchaus als „feindlich" angesehen werden und die durchführenden Entwickler sich unbeliebt machen. Inwieweit diese Einstellung sich auf die Softwareentwickler selbst erstreckt oder aber nur von Anwendern und Fans Freier Software lautstark vorgetragen wird, kann ohne gründliche Untersuchung nicht geklärt werden. Persönlich sehe ich das Recht zu „forken" als zu wichtig an, um die Inanspruchnahme dieses Rechtes scharf zu verurteilen, auch wenn das *forking* gegen den Willen der ursprünglichen Entwicklerschaft durchgeführt wird. c*ode-forkings* können auftreten, wenn große Teile eines Teams unterschiedlicher Meinung sind und keine Einigung mehr möglich ist. Das hat meistens mit technischen Einschätzungen zu tun, oft aber auch mit den Persönlichkeiten und persönlichen Vorlieben der involvierten Entwickler. Eine technische Alternative zu c*ode-forkings* ist die erhöhte Konfigurierbarkeit der Software, oftmals unterstützt durch eine Komponentenarchitektur. Bei einer aus einzelnen Komponenten aufgebauten Anwendung stellt im Grunde genommen erst der Anwender über verschiedenartige Optionen die eigentliche Anwendung nach persönlichen Vorlieben zusammen. Er oder sie entscheidet individuell, welche Komponenten hineingehören und welche nicht. Die Entscheidungsfindung, welcher Weg denn jetzt der „richtige" sei, wird damit den Anwendern überlassen. Über Rückmeldungen der Anwenderschaft kann das Projekt dann die Voreinstellungen so wählen, dass die meisten Benutzer mit der Software zufrieden sind, ohne dass Anwender und Entwickler mit anderen Vorlieben ausgeschlossen werden.

Forks treten aber auch dann auf, wenn ein Projekt allmählich einschläft, d.h. wenn die Maintainer nicht mehr schnell genug auf Vorschläge und Anfragen von Seiten der Benutzerschaft reagieren wollen oder können. Ein neues Team kann hier einspringen, möglicherweise gegen den Willen der ursprünglichen Entwickler, aber wahrscheinlich im Sinne der gegenwärtigen Anwender. Ein solcher *fork* bedeutet in der Praxis ein allmähliches Auseinanderdriften der Codebasis und damit meistens eine dauerhafte Trennung der Projekte. Es gibt aber auch Fälle, bei denen das „geforkte" und erfolgreichere Projekt später als neuer Hauptzweig in das ursprüngliche Projekt zurückgeführt wird. der *fork* wird damit zur Kur, die frischen Wind in alte Projekte bläst. Ein wichtiges Beispiel hierfür ist das Herzstück aller Freien Software, die *GNU Compiler Collection (GCC).* Als das Projekt vor einigen Jahren ins Stocken geriet, entstand eine zweite Gruppe, die mit dem *Experimentellen GNU Compiler System* (EGCS) einen Fork ins Leben rief (Henkel-Wallace 2002). EGCS lieferte Funktionalität, auf die die Anwendergemeinde lange gewartet hatte. Das neue System konnte seinem Urvater damit schnell an Popularität den Rang ablaufen, und zwang so dessen Maintainer zum handeln. Schließlich wurden Anpassungen an der Projektstruktur vorgenommen und beide Softwarebäume erneut unter dem ursprünglichen Namen GCC zusammengeführt.

Eine andere Art von *forks* sollte aber nicht vergessen werden, nämlich sogenannte konzeptionelle *forks*. Hierbei wird nicht etwa die Codebasis übernommen und verändert, sondern die Projektidee. Dies kann im Großen oder im Kleinen geschehen. In einem Textbearbeitungsprogramm kann ein Entwickler sich beispielsweise dahingehend entscheiden, eine neue Komponente für die Rechtschreibprüfung zu entwickeln, anstatt mit den anderen Entwicklern an der bereits existierenden zu arbeiten. Über einen gewissen Zeitraum wird das Projekt dann zwei verschiedene

Komponenten für die gleiche Aufgabe besitzen, bis eine sich als die bessere herausstellt und für die Kerndistribution ausgewählt wird. Diese Art von Konkurrenz ist ein wesentlicher Bestandteil der freien Softwareentwicklung, auch wenn Außenstehende manchmal den Kopf schütteln, warum es Dutzende von IRC-Clients, Texteditoren oder E-Mail-Programmen geben muss. Die Antwort ist einfach: Weil es Programmierer gibt, die die nötige Neugier besitzen oder ein technisches Interesse daran haben, diese zu schreiben. Kooperation und Zusammenarbeit kommt erst an zweiter Stelle. Sie wird dann gewählt, wenn sie anderen, persönlicheren Zielen förderlich erscheint. Ein Projekt kann sich zum Ziel setzen, das am meisten benutzte, anwenderfreundlichste E-Mail Programm zu schreiben. Ein solches Projekt wird höchstwahrscheinlich mit anden Projekten und vor allem Anwendern kooperieren. Ein anderes Projektes kann es sich zum Ziel machen, ein E-Mail Programm zur eigenen Verwendung gänzlich alleine zu schreiben. Kooperation ist hier eher hinderlich. Die beteiligten Entwickler dafür zu kritisieren, wäre wie Freizeitmusiker dafür zu kritisieren, dass sie ihre Zeit damit verschwenden, Musikstücke einzustudieren und sich selbst vorzuspielen, statt mit einem Verlag und einer Musikhandlung zu kooperieren und einfach CDs zu kaufen. Am Ende des Tages geht es um ein wunderschönes Hobby und den Drang, etwas mit den eigenen Händen zu schaffen, beim Programmieren wie beim Musizieren.

6. Zusammenfassung

Freie Softwareentwicklung ist ein dynamischer, evolutionärer Prozess. Das Open-Source-Konzept garantiert nicht automatisch, dass qualitativ überragende Software in kürzester Zeit entwickelt wird. Auf jedes erfolgreiche Open-Source-Projekt kommen hunderte von Projekten, die scheitern. Damit ein Projekt erfolgreich ist, müssen viele verschiedene Faktoren zusammenkommen, einschließlich des richtigen Timings und einer gewissen Portion Glück. Die wichtigste Überlebensfrage ist dabei immer, inwieweit ein Projekt es schafft, genügend Entwickler anzuziehen und zu halten. Anwender sind für ein Projekt nur an zweiter Stelle entscheidend: Als Motivationsfaktor und als Basis, neue Entwickler zu rekrutieren.

Die Motivation der einzelnen Entwickler ist sehr individuell. Sie reicht vom Eigeninteresse an der fertigen Software über technisch-wissenschaftliche Neugier, z.B. wie eine bestimmte Art von Software funktioniert, bis hin zu ganz persönlichen Zielsetzungen, z.B. ob man selbst in der Lage ist, eine bestimmte Software zu schreiben. Manche wollen auch einfach nur Spaß haben.

Koordination und strategische Planung innerhalb freier Softwareprojekte beschränken sich im wesentlichen darauf, einer möglichst großen Anzahl von Gleichgesinnten eine Arbeitsumgebung zur Verfügung zu stellen, in der sie fair und effektiv kooperieren können. Unterschiedliche Projekte finden unterschiedliche Lösungen, wobei größere Projekte auf Grund des komplexeren Release-Prozesses einen höheren Bedarf an Verwaltung und Regeln haben als kleiner Projekte.

Kommunikation innerhalb der Projekte geschieht über modernste Internet-Technologien. Ohne Internet kann es keine Entwicklung Freier Software geben, da

lokale Gemeinschaften nicht in der Lage sind, die notwendige kritische Masse aufzubringen.

Literatur

Adams, Scott: *The Official Dilbert Website*,
online http://www.dilbert.com (29.11.2003)
CollabNet Inc.: S*ubversion.tigris.org, a compelling replacement for CVS*,
online http://subversion.tigris.org (29.11.2003)
The GCC Home Page,
online http://gcc.gnu.org (29.11.2003)
Henkel-Wallace, David (2002): *A new compiler project to merge the existing GCC forks*,
online http://www.goof.com/pcg/egcs.html (29.11.2003)
Irchelp.org: Internet Relay Chat (IRC) help archive,
online http://www.irchelp.org (29.11.2003)
The K Desktop Environment Homepage,
online http://www.kde.org (29.11.2003)
KDE.news,
online http://dot.kde.org (29.11.2003)
KDE Mailing Lists,
online http://www.kde.org/mailinglists (29.11.2003)
KDE FTP Mirrors,
online http://www.kde.org/mirrors/ftp.php (29.11.2003)
Leuf, Bo und Ward Cunningham (2001): *The Wiki Way: quick collaboration on the Web*,
Boston: Addison-Wesley,
online Information unter http://www.wiki.org (29.11.2003)
Vesperman, Jennifer (2003): *Essentials CVS*, O'Reilly & Associates.
World Wide Web Consortium,
online http://www.w3c.org (29.11.2003)
Wells, Don (1999): *Extreme Programming, a gentle introduction*,
online http://www.extremeprogramming.org (29.11.2003)