Dieser Artikel ist Teil des

Open Source Jahrbuch 2005

erhältlich unter http://www.opensourcejahrbuch.de.

Das Open Source Jahrbuch 2005 enthält neben vielen weiteren interessanten Artikeln ein Glossar und ein Stichwortverzeichnis.

Der Beitrag freier Software zur Software-Evolution

ANDREAS BAUER UND MARKUS PIZKA

Ohne ein gesondertes Interesse an der Thematik Software-Evolution zu haben, hat die stetig wachsende Open-Source-Bewegung über die letzten zwei Jahrzehnte trotzdem auf bemerkenswerte Weise äußerst erfolgreiche Konzepte, Methoden und Techniken für die kontinuierliche und kontrollierte Weiterentwicklung komplexer Software-Systeme etabliert. Diese Evolutionstechniken repräsentieren *Best Practices*, die ebenso zur Lösung aktueller und höchst kritischer Probleme bei der Pflege und Weiterentwicklung kommerzieller Software-Systeme herangezogen werden könnten. Im vorliegenden Artikel stellen die Autoren einige dieser Prinzipien aus der Perspektive erfahrener Open-Source-Entwickler dar. Dabei wird deutlich, dass der extrem dynamische Prozess der Entwicklung freier Software eng mit dem konstanten Wachstum der Code-Basen und stets veränderlichen Projektgrößen verbunden ist. Weiter wird argumentiert, dass ein Großteil des Erfolgs von Open-Source-Software tatsächlich auf den erfolgreichen Umgang mit diesem kontinuierlichen Wandel zurückzuführen ist.

1. Einführung

> „Virtue is more to be feared than vice, because its excesses are not subject to the regulation of conscience." – Adam Smith (1723–1790)

Die initiale Entwicklung und stetige Evolution von Open-Source-Software-Produkten weist erstaunliche Ähnlichkeiten mit den Ideen der freien Marktwirtschaft auf. Die liberale Bereitstellung, Nutzung und Veränderung von Software, wie sie in Open–Source-Umgebungen praktiziert werden, teilen die Prinzipien des Wirtschaftsliberalismus (Smith 1776), indem sie einen Raum für ungehinderten Handel, Veränderung und Wachstum etablieren. Infolgedessen könnte man spekulieren, dass sich die Entwicklung von Produkten und Prozessen in Open-Source-Umgebungen auf lange Sicht gesehen als überlegen gegenüber allen anderen Modellen herausstellen wird, weil das selbstregulierende Wechselspiel zwischen Angebot und Nachfrage für kontinuierliche Selektion und Verbesserung sorgen wird.

Tatsächlich gibt es zunehmend Hinweise für diese Hypothese. Die anfangs eher unbedeutende Open-Source-Gemeinde hat sich zu einem umfangreichen freien Markt

für Software-Artefakte mit tausenden Teilnehmern und unzähligen Produkten weltweit entwickelt. Aufgrund der enormen Energie in diesem Markt haben einige Produkte, wie GNU/Linux oder der Übersetzer GCC eine hinreichend hohe Qualität für breiten kommerziellen Einsatz erreicht. Zusätzlich kann dieser Markt aufgrund seiner Größe und Vielfalt schnell auf verändernde bzw. neue Anforderungen reagieren – z. B. die Entwicklung neuer Gerätetreiber. Neben den Produkten entwickelt sich auch der Entwicklungsprozess selbst innerhalb dieses Marktes ständig weiter. Die Entstehung strukturierter Kommunikationsplattformen, [1] die Einführung von Rollen – *Maintainer* – und die Verbreitung von Elementen agiler Methoden (Beck 1999) stützen diese Behauptung.

Nun soll dieser Artikel nicht den Eindruck erwecken, Open Source sei die *silver bullet* (Brooks Jr. 1995) des Software-Engineerings, denn es ist offensichtlich, dass auch dieses Modell inhärente Defizite besitzt und nicht in jeder Situation zwangsläufig zu besseren Ergebnissen führen muss. Demgegenüber wird im Folgenden argumentiert, dass im Zuge freier Software in der Vergangenheit Konzepte entstanden sind, die potentiell auch großen kommerziellen Softwareprojekten, mit Millionen von Zeilen von Code, dabei helfen können, sich sogar über Jahrzehnte auf „gesunde" Art und Weise weiterzuentwickeln. Unsere eigenen praktischen Erfahrungen mit einigen weithin bekannten Open-Source-Produkten wie der GNU Compiler Collection (GCC) oder GNU/Linux sowie unsere Urheberschaft und Beteiligung an kleineren Open–Source-Produkten, haben uns den Nutzen des Entwicklungsmodells freier Software sowie den daraus resultierenden Architekturen klar vor Augen geführt.

Anhand dieser Erfahrungen können wir guten Gewissens argumentieren, dass einige der nachstehend diskutierten Open-Source-Prinzipien und -Praktiken in kommerzielle bzw. proprietäre Software-Organisationen zur kontinuierlichen Verbesserung bestehender Produkte und Prozesse übernommen werden können und sollten.

Abschnitt 2. klärt den Kontext unserer Arbeit und grenzt die Gültigkeit der Aussagen ein. Dabei wird der Begriff Open Source als „freie Software" präzisiert, es werden verwandte Arbeiten zur Evolution freier Software beschrieben und ein Zusammenhang mit Forschungsarbeiten im Bereich Software-Evolution hergestellt. Danach werden wir in 3. einen detaillierten Blick auf die Evolution der Prozesse in freien Softwareprojekten werfen, bevor wir uns in 4. auf die damit zusammenhängende Evolution der technischen Architektur der Produkte konzentrieren werden. In 5. schließen wir den Artikel mit einer kurzen Zusammenfassung der wesentlichen Resultate unserer Studie ab und diskutieren die Übertragbarkeit der Ergebnisse auf nichtfreie Software-Systeme und -Umgebungen.

2. Open Source und Freie Software

Wenn wir uns in diesem Artikel auf Open-Source-Software beziehen, beschränken wir uns nicht auf die aktuell sehr populäre Open-Source-Bewegung an sich, die größ-

1 Siehe SourceForge, Open Source Technology Group: http://sourceforge.net/.

tenteils durch das GNU/Linux-Projekt[2] inspiriert wurde. Wir sprechen vielmehr von Software, deren Nutzungs- und Änderungsrechte dem (sogar noch älteren) Ideal der freien Software an sich entsprechen. Diese Freiheit beschränkt sich nicht auf eine Erlaubnis zum Lesen des Quellcodes, wie bei einer „Redefreiheit", sondern erlaubt uneingeschränkte und kostenlose Nutzung, Weitergabe, Modifikation, ganz wie beim allseits beliebten „Freibier". Der Hauptunterschied liegt also darin, dass Redefreiheit in der zivilisierten Welt eher als ein menschliches Grundrecht anzusehen ist, mit dem noch garantiert ist, dass damit auch etwas erreicht werden kann, wohingegen Freibier einzig und alleine deshalb positiv auffällt, weil es nichts kostet und die künftige Verwertung gänzlich offen lässt.

Gemäß der GNU GPL[3] der Free Software Foundation (FSF) – die Lizenz des GNU/Linux-Projekts – darf Quellcode modifiziert und von einer dritten Person verwendet werden, solange diese Programmmodifikationen ebenso frei und offen bleiben und die ursprünglichen Copyright-Hinweise nicht verändert werden. Die FSF glaubt, dass dies der beste Weg sei, das Recht der Nutzer zum Verstehen und ggf. Anpassen von Programmen zu erfüllen. Wenngleich diese Freiheiten schon sehr weit gehen, sind BSD-artige Lizenzen[4] noch viel liberaler und erlauben einer dritten Person mehr oder weniger mit den Sourcen zu machen, was sie will – sogar Modifikationen der freien Software als proprietäre *Closed Source* zu vertreiben.

Der Rest dieses Artikels beschäftigt sich also mit den Erfahrungen der Autoren im Umgang und in der Entwicklung von *freier* Software im Allgemeinen, anstatt mit ihren jeweiligen speziellen Ausprägungen, die unter anderem auch Trends obligen. Die Begriffe Open Source und „freie Software" werden synonym für Software verwendet, die ähnlich wenigen Nutzungsbeschränkungen unterliegt wie Freibier.

2.1. Mythen und Klischees

Eine weit verbreitete falsche Vorstellung ist, dass Open Source oder *free software* in einem chaotischen Prozess von Freizeit-Hackern, die nicht die gleichen Ansichten bezüglich Wartbarkeit haben wie ihre „professionellen" oder akademischen Entsprechungen, erstellt wird und in mehr oder weniger brauchbaren Software-Produkten mündet. Bedauerlicherweise kann auch der Titel des bekannten *Papers* von Eric Raymond „The Cathedral and the Bazaar" (Raymond 1998) leicht dahingehend missverstanden werden, wodurch das Chaos-Klischee zusätzlich unterstützt wird.

Doch selbstverständlich ist das weit gefehlt: Freie Software entsteht nicht auf einem chaotischen Basar. Wie wir unten zeigen werden, ist die Entwicklung freier Software oft sehr gut organisiert und bedient sich strukturierter Prozesse mit klar definierten Rollen. Infolgedessen gibt es eine große Anzahl freier Software-Produkte, deren Qualität kommerziellen Varianten mindestens ebenbürtig ist und folglich unsere These unterstützt.

2 GNU/Linux: http://www.linux.org/
3 GNU General Public License: http://www.fsf.org/licenses/gpl.html
4 The BSD License: http://www.opensource.org/licenses/bsd-license.html

Leider muss sich die Freie-Software-Bewegung allerdings sehr wohl vorwerfen lassen, dass sie die Resultate und Trends in der Forschung im Bereich der Software-Evolution ignoriert. Umgekehrt ignoriert die andere, nicht-freie Seite die Umstände, unter denen ihre Lieblingstexteditoren, Compiler oder Betriebssysteme ins Leben gerufen wurden. Dieser Artikel zielt darauf ab, diese Lücke zwischen den Errungenschaften der freien und der „professionellen" Softwareentwickler aus dem akademischen Umfeld und in der Industrie ein Stück weit zu schließen.

2.2. Wirklichkeit

Um nur einige wenige der erfolgreichen freien Software-Systeme zu nennen, weisen wir auf die BSD-basierten Betriebssysteme FreeBSD, NetBSD, OpenBSD und Darwin[5] hin, die ihren Ursprung hauptsächlich in Ideen und Code aus den 80er Jahren haben. Dank der hohen Qualität dieser Produkte, die sich über einen langen Zeitraum entwickelt hat und der „Freibier"-Philosophie der BSD-Lizenz, basieren kommerzielle Betriebssysteme von wichtigen Anbietern wie Apple heute auf einem BSD *Open Source Kernel*, der nach Charles Darwin – dem Urvater der Evolutionstheorie (Darwin 1859) – benannt ist. Dies unterstreicht die Bedeutung der erfolgreichen Evolution von Software-Produkten, da insbesondere bei solchen komplexen Systemen eine langfristige erfolgreiche Reifung notwendig ist, um die notwendige Qualität zu erreichen.

Sogar Free-speech-Projekte wie GCC oder GNU/Linux werden heute zu einem großen Teil von dem finanziellen Engagement großer Konzerne wie IBM oder Red Hat getragen, die versierte Entwickler, Geld und andere Ressourcen für Softwareprojekte zur Verfügung stellen, bei denen jeder den Quellcode lesen und verändern kann (Harris et al. 2002). Das Ergebnis muss natürlich, ganz entgegen der verbreiteten Fehleinschätzung des Chaos-Prozesses, ein wartbares Produkt sein, da Wartbarkeit ein wesentlicher Grund dafür ist, dass so alte „Dinosaurier-Projekte" wie GCC, *BSD, Emacs, GNU/Linux usw. heute in ihren Bereichen nach wie vor erfolgreich sind.

Offensichtlich hat die Open-Source-Bewegung ihre eigenen Konzepte und Techniken entwickelt, die es großen Softwareprojekten erlaubt, sich auch bei ständig wechselnden Anforderungen langfristig erfolgreich und gesund weiterzuentwickeln. Obwohl viele der freien Software-Produkte unter dem Aspekt der technischen Innovation nicht mehr interessant sind, sind sie heutzutage alles andere als irrelevant. Ein Teil des offenkundigen Erfolges eines Betriebssystems wie GNU/Linux muss deshalb zwangsläufig an der Art und Weise liegen, wie sich diese Software an Veränderungen sowohl der technischen Realität als auch der Anzahl der Personen, die etwas zu dem Projekt beitragen (siehe Abschnitte 3.1., 3.2.), anpasst.

2.3. Verwandte Arbeiten zur Software-Evolution

Aufgrund der großen Lücke zwischen proprietärer Softwareentwicklung, akademischer Forschung und den Praktiken der freien Software-Gemeinde ist es nicht ver-

5 (Open)Darwin http://www.opendarwin.org/, http://developer.apple.com/darwin/

wunderlich, dass nach wie vor nur wenige Forschungsarbeiten auf die Weiterentwicklung freier Software-Produkte abzielen. Ausnahmen sind Nakakoji et al. (2002), Lehey (2002), Succi und Eberlein (2001) und Godfrey und Tu (2000). Auf der anderen Seite spielen in der Welt der freien Software eine beträchtliche Anzahl praktischer Konzepte und Techniken für die Evolution vorhandener Software eine wichtige Rolle, wie zum Beispiel Konfigurationsmanagement (z. B. CVS), Regressionstests (Savoye 2002), *Refactoring* (Opdyke 1992), Analyse von Quell-Code, Code-Generatoren und *Separation of Concerns* – um nur einige zu nennen. Wir verzichten bewusst auf die weitere Aufzählung dieser langen Liste, aber es sollte offensichtlich werden, dass die kontinuierliche Weiterentwicklung freier Software, wie sie in dieser Arbeit beschrieben wird, zahlreiche Verbindungen mit verschiedenen konkreten Evolutionstechniken besitzt. Indes konzentrieren wir uns in dieser Arbeit auf die *Prinzipien* der Evolution freier Software, unabhängig von bestimmten Techniken.

Lehmans wohlbekannte acht Gesetze der Software-Evolution (von Lehman 1969 bis Lehman und Ramil 2001) zielen auf die fundamentalen Mechanismen ab, die der Dynamik der Software-Evolution zugrunde liegen. Wie wir sehen werden, entspricht freie Software den Gesetzen Lehmans ganz hervorragend und das, obwohl Lehmans Gesetze und der Entwicklungsprozess freier Software unabhängig voneinander entstanden sind: Freie Software verändert sich ständig (Gesetz I), die Komplexität nimmt zusehends zu (Gesetz II) und die Selbstregulierung des Evolutionsprozesses ist offensichtlich (Gesetz III). Unsere im Folgenden beschriebenen Beobachtungen stützen auch die verbleibenden Gesetze IV bis VIII. Im Gegenzug ist die weitreichende Übereinstimmung der freien Software-Entwicklung mit Lehmans Gesetzen eine Möglichkeit, den Erfolg der Open-Source-Softwareentwicklung zu erklären.

3. Evolution des Entwicklungsprozesses

Anders als bei vielen proprietären Softwareprojekten beginnt die Entwicklung freier Software meist ohne jeden zusätzlichen Verwaltungsaufwand. Typische frühe Projektphasen zu Strukturierung und Koordination der noch folgenden Phasen, wie zum Beispiel eine exakte Anforderungsanalyse, spielen oftmals überhaupt keine Rolle. Ebenso ist es jedoch offensichtlich, dass der administrative Aufwand nicht konstant bleiben kann, wenn das Projekt wächst. Hierfür gibt es viele bedeutende Beispiele (Lehey 2002, Nakakoji et al. 2002, Cubranic und Booth 1999).

In der Tat ist der Entwicklungsprozess bei freier Software höchst dynamisch, skaliert mit der darunterliegenden Architektur und auch mit der Anzahl und den Fähigkeiten der Menschen, die am Projekt beteiligt sind. Den *einen* Entwicklungsprozess für freie Software gibt es allerdings nicht, sondern sich weiter entwickelnde Prozesse, die stark mit der Komplexität der resultierenden Produkte selbst verwoben sind.

3.1. Unausweichliche technische Veränderungen

Einige der Veränderungen im Entwicklungsprozess freier Software beruhen eher auf technischen Veränderungen als auf Entscheidungen der Entwickler. Zum Beispiel

war der weit verbreitete Übersetzer GCC ursprünglich Mitte der 80er Jahre als schneller und praxisnaher C-Compiler auf 32-Bit-Plattformen entwickelt worden, welche 8-Bit-Bytes adressieren und mehrere Allzweckregister[6] besitzen. Heutzutage unterstützt der GCC mehr als 200 verschiedene Plattformen (Pizka 1997) sowie zahlreiche weitere Programmiersprachen. Sein Kern besteht aus über 900 000 Programmzeilen, und während das GCC Projekt aus einem zunächst simplen E-Mail-Verkehr (später auch über Usenet) zwischen Entwicklern des Kern-Teams entstanden ist, funktioniert der derzeitige Entwicklungsprozess heute grundlegend anders. Das Projekt hat offensichtlich nicht nur seine ursprünglichen Ziele geändert, sondern auch die Anzahl und auf gewisse Weise auch die Art der Mitwirkenden in Bezug darauf, wie sie sich an der fortlaufenden Evolution von GCC beteiligen. Das Handbuch der Version 3.2.2 [7] listet die Namen von 302 verschiedenen Mitwirkenden auf, was natürlich ein starker Kontrast zu 1984 ist, als Richard Stallman die erste Version eines damals eher einfachen System-Compilers für potenzielle GNU-Plattformen alleine erstellte.

Diese drastischen Veränderungen wurden hauptsächlich durch die technischen Fortschritte möglich, die sich seit der Geburt von GCC ereignet haben. Hierzu gehören besonders die weltweite Ausbreitung des Internets mit all seinen neuen Transfer–Protokollen, die während des Projekts entstanden sind (siehe das *hypertext transfer protocol*, http). Im Speziellen zieht das GCC-Projekt heute aus den folgenden technischen Fortschritten seine Vorteile:

- Automatisches Management von Mailinglisten mit Zugriff auf durchsuchbare Archive und Web-Oberflächen: so werden die Bemühungen eines weltweit verbreiteten Netzwerks von Entwicklern koordiniert und gesteuert,
- (Öffentliche) CVS-Server mit Web-Oberflächen, die sowohl verschiedene Versionen als auch unabhängige Entwicklungen innerhalb des Projekts verfolgen,
- Eine große Anzahl von (http und ftp) *mirror sites*, welche die Verfügbarkeit der relevanten Daten weltweit erhöhen,
- Die Einführung und das Interesse an neuen Sprachen (z. B. Java, Haskell) und neuen Hardware-Plattformen (z. B. IA-64-Architektur),
- Ein modernes und automatisiertes Fehlerverfolgungssystem, das weltweit via World Wide Web zugänglich ist,
- *compile farms*, die an einem Punkt zentralen Zugang zu mehreren Plattformen bieten und für gewöhnlich durch Firmen aus der Industrie gefördert werden, die ein beträchtliches Interesse an Open-Source-Produkten haben,
- Eine wachsende Anzahl peripherer Projekte, die auf dem GCC aufbauen, aber ihren eigenen Fortschritt unabhängig managen (z. B. Glasgow Haskell Compiler (GHC), Echtzeit-Java-Implementierung, Mercury Compiler).

6 GNU Compiler Collection Internals: http://gcc.gnu.org/onlinedocs/gccint/

7 Using the GNU Compiler Collection: http://gcc.gnu.org/onlinedocs/gcc-3.2.2/gcc/

Projekt	Alter	Code-Zeilen
Linux kernel 2.4.2	1991	2 437 470
Mozilla	1998	2 065 224
XFree86 4.0.3	1987	1 837 608
GCC 2.96-20000731	1984	984 076
GDB / DejaGnu-20010316	Mitte 1980er	967 263
Binutils 2.10.91.0.2	Mitte 1980er	690 983
glibc 2.2.2	Frühe 1990er	646 692
Emacs 20.7	1984	627 626

Tabelle 1: Größe von Open-Source-Projekten (Wheeler 2002)

Das gilt auch für andere große und erfolgreiche Open-Source-Projekte, die sich meistens durch Anwendung derselben Tools (CVS, RCS, BugZilla usw.) unter ähnlichen Umständen oder technischen Bedingungen weiterentwickeln. Weitere Beispiele sind *BSD, Mozilla und der Linux Betriebssystem-Kernel. Wenn wir die wachsende Anzahl der Mitwirkenden, z. B. bei FreeBSD und GCC vergleichen, sehen wir Ähnlichkeiten zwischen beiden Projekten: Im Jahre 1995, als FreeBSD 2.0 freigegeben wurde, hatte es 55 Mitwirkende (d. h. Anwender mit Schreibzugriff auf das *code repository*); bis Ende 2002 war es insgesamt 319 Leuten gestattet, Änderungen selbst direkt an der Code-Basis einzupflegen (Lehey 2002).

3.2. Änderung der Organisation

Die wahre Herausforderung beim Aufbau und Unterhalt eines erfolgreichen Entwicklungsprozesses für ein freies Software-Produkt besteht darin, technische Veränderungen entsprechend den begleitenden sozialen Veränderungen einzuführen und zwar dann, wenn das Projekt eine „kritische Größe" überschreitet. Die kritische Größe hängt nicht allein von der Anzahl der Zeilen im Quell-Code ab, sondern auch von den Programmmodulen, der Anzahl der Mitwirkenden und im Grunde genommen auch von beliebigen Metriken, die es uns erlauben, Schlussfolgerungen über die Komplexität eines Software-Produktes zu ziehen.

Die heutzutage florierenden freien Softwareprojekte haben alle ihre kritische Größe mindestens einmal überschritten, gewöhnlich in der Anzahl der Zeilen und oft auch im Hinblick auf die Zahl der Mitwirkenden. Wie aus Tabelle 1 ersehen werden kann, besteht die Linux-Version 2.4 aus mehr als 2 400 000 Code-Zeilen und außerdem deuten ihre derzeitigen *changelogs* eine mindestens gleich große Zahl von aktiven Entwicklern an, wie sie etwa an FreeBSD beteiligt sind.

Diese Zahlen weisen auf die Tatsache hin, dass organisatorische Aufgaben bei erfolgreicher freier Software nicht durch eine einzelne Person gelöst werden können (z. B. durch den ursprünglichen Initiator eines Projekts). Wenn eine kritische Größe einmal erreicht ist, entsteht so oder so eine „Geschäftsführung", und das Gremium

Projekt	Name des Gremiums	Mitglieder
FreeBSD	Core	15
GCC	Steering Committee	12
Debian	Leader & Technical Committee	8
Mozilla	Project Managers („Drivers“)	13
KDE	Core Group	20

Tabelle 2: Führungsgremien in freien Softwareprojekten

koordiniert und kontrolliert die weitere Evolution des Produktes. In Konsequenz daraus wird ein reifes Projekt wie GCC nicht mehr von Richard Stallmann vorangetrieben, sondern von einem „Lenkungsausschuss“, der heute aus 12 professionellen Software-Entwicklern besteht, die teilweise von Software-Anbietern bezahlt werden (z. B. Red Hat, IBM, Apple), um sich auf die Entwicklung einer freien *compiler suite* zu konzentrieren (siehe Tabelle 2). Während die Mitglieder alle Experten auf ihren Gebieten sind, wird von keinem einzelnen erwartet, ein vollständiges Verständnis für den gesamten Code von 900 000 Zeilen zu haben.

Wie aus Tabelle 2 ersehen werden kann, scheinen freie Software-Produkte gut mit einem Lenkungsausschuss bestehend aus 10–20 Mitgliedern auszukommen. Offensichtlich ist diese Mitgliederzahl groß genug, um ein komplexes Projekt zu managen, aber klein genug, um sicherzustellen, dass die Entwicklung nicht aufgrund interner Debatten und unglücklicher Politik gehemmt wird. Interessant ist auch, dass sich die aufgelisteten Projekte bezüglich Größe und Alter sehr ähnlich, aber historisch kaum miteinander verbunden sind und dennoch gemeinsam 10–20 Kernmitglieder als eine angenehme Größe empfinden, um die „richtigen“ Entscheidungen für das jeweilige Projekt zu treffen.

Es sollte an dieser Stelle herausgestellt werden, dass die Strategie auch perfekt zu den Ergebnissen umfangreicher Studien über den Erfolg und das Scheitern von kommerziellen Softwareprojekten passt, wie z. B. die wohl bekannten CHAOS-Berichte der Standish Group (The Standish Group International 1999, 1995). Diese Studien zeigen eine Tendenz zum Scheitern eines Projektes, je größer es wird. Daher wird stark empfohlen, ein Projekt überschaubar klein zu halten, da es ansonsten sehr wahrscheinlich scheitern wird. Das ist genau der Grund, warum agile Methoden wie Extreme Programming (XP) sich absichtlich weigern, „riesige“ Projekte in einem einzelnen Schritt auszuführen. Unglücklicherweise scheint die Verbreitung dieser grundlegenden Lektion in kommerziellen Umgebungen sehr langsam zu sein. Im Gegensatz dazu gewährleistet die Selbstregulierung durch Wettbewerb und Auslese innerhalb freier Softwareprojekte eine passende Projektgröße. Das Projekt wird je nach Bedarf fortlaufend in kleinere Teilbereiche aufgespalten, wobei den Mitgliedern des Entwicklerteams neue Wartungsaufgaben zugeordnet werden, die ihrem Können, ihrer Erfahrung und ihren aktuellen Kompetenzen entsprechen. Letztendlich skaliert die gesamte Organisation mit der Gesamtgröße des Produktes.

Der Weg, wie der Führungsausschuss nominiert wird unterscheidet sich bei den verschiedenen freien Softwareprojekten. Während Debian zum Beispiel demokratische Wahlen einsetzt, beruht die Auswahl im KDE-Team ausschließlich auf den Verdiensten der Mitglieder um das Projekt. Jeder kann Mitglied der KDE-Kerngruppe werden, aber der spezielle Bewerber muss sich selbst durch herausragende Beiträge und Hingabe über einen beachtlichen Zeitraum hinweg ausgezeichnet haben. Dessen ungeachtet stellen sowohl KDE als auch Debian sicher, dass vorrangig die passendsten und begabtesten Leute die Verantwortung übernehmen und nicht die, die am lautesten schreien, am ältesten oder jüngsten sind usw. – wie es oft in kommerziellen Umgebungen der Fall ist. Dies ist für sich ein Evolutionsprozess, der Personen mit Wissen und praktischen Fähigkeiten fördert anstatt reine Autoritätspersonen.

3.3. Konfigurations- und Änderungsmanagement

Trotz der Ähnlichkeiten zwischen den verschiedenen „großen" freien Softwareprojekten, gibt es viele verschiedene Ansätze für das Änderungsmanagement. Diese konvergieren jedoch mehr und mehr aufgrund des Einsatzes ähnlicher Werkzeuge (CVS, RCS, BitKeeper, patch/diff usw.), welche die teilweise sehr komplexen Aufgaben automatisieren.

Wie Nakakoji et al. (2002) beobachtet haben, scheinen die verschiedenen Ansätze im Open-Source-Änderungsmanagement in etwa gleich erfolgreich zu sein, da es bedeutende Beispiele für den Erfolg der einzelnen Techniken gibt. In ihrem Dokument werden die verschiedenen Änderungshistorien „Evolutionsmuster" genannt, welche sich selbst hauptsächlich durch die Art abgrenzen, wie Änderungen getestet, geprüft und mit dem Projekt zusammengeführt werden.

Linus Torvalds und Alan Cox, zwei treibende Kräfte der Linux-Kernel-Entwicklung, haben in einigen Interviews Bedenken gegenüber dem Gebrauch von öffentlichen CVS-Servern zur Abwicklung des Konfigurationsmanagements erhoben, und das, obwohl gleichzeitig viele wichtige Linux-Module auf diese Weise unter Versionskontrolle stehen (Southern und Murphy 2002). Auf der anderen Seite scheinen GCC und *BSD mit den Einrichtungen und den Möglichkeiten öffentlicher CVS-Server zufrieden zu sein, trotz gelegentlicher *commit wars* (Lehey 2002). Von einem *commit war* wird dann gesprochen, wenn Entwickler, die an denselben Teilen des Codes arbeiten, laufend die Änderungen des jeweils anderen überschreiben. Bei dem wichtigsten Linux-Kernel (dem von Linus Torvalds) kann das nicht passieren, weil Linus Torvalds als der Projektleiter persönlich entscheidet, welche Änderungen angenommen und welche lieber fallen gelassen oder durch andere *Maintainer* für ihre eigenen Entwicklungsbäume aufgegriffen werden.

Bezüglich des Änderungsmanagements scheinen Open-Source-Projekte dieselben Dinge zu tun, wie sie bei proprietärer Software vorgefunden werden können (siehe auch van der Hoek 2000). Man sollte auch darauf hinweisen, dass sogar CVS selbst seine Wurzeln in der freien Software-Welt (Cederqvist et al. 1993) hat, was einfach zeigt, wie sich ein Open-Source-Programm, das als eine Hand voll Shell-Skripten, die im Usenet verbreitet wurden, begann, sich zu einem De-facto-Standard entwickelt

hat – eben weil es stets fest integriert in die Entwicklung anderer freier Software–Produkte war und deren Unterstützung zum Ziel hatte. Im Klartext heißt das, dass die zahlreichen Produkte, die mit Hilfe von CVS entwickelt wurden, tatsächlich den Entwicklungsprozess dieses *toolkits* selbst ausgelöst haben, bis hin zu dem Punkt, an dem es ein unersetzlicher Eckpfeiler sowohl für freie als auch kommerzielle Projekte wurde.

4. Evolution der Architektur

In einem freien Software-Projekt ist es nicht nur die soziale Struktur, die sich einer sich verändernden Realität anpasst, sondern auch die Architektur selbst, die sehr oft eher einer natürlichen Entwicklung unterliegt, als das Ergebnis einer sorgfältig geplanten Anforderungsanalyse zu sein. Wiederum ein gutes Beispiel für diese Behauptung ist das kürzlich durch das GCC-Projekt angenommene, von Intel vorgeschlagene *cross vendor application binary interface*, das anfangs auf negative Resonanz der Nutzer stieß, weil es Inkompatibilitäten mit früheren C++ Programmen verursachte. Der Schritt war jedoch notwendig, um mit den Binärpaketen kompatibel zu sein, die von anderen (kommerziellen) Compilern erzeugt werden, die auch Intels neue (64-Bit-) Hardware und einen Großteil der von dem neuesten ISO-Standard für die Sprache C++ vorgeschlagenen Features unterstützen (Intel Corporation 2001).

4.1. Modulare Programme und Schichten

Das Verändern der ABI eines existierenden Compilers ist keinesfalls eine triviale Aufgabe. Im Falle des GCC allerdings war diese Änderung durch das modulare Design der Architektur möglich: Der Aufbau erlaubt es, den Großteil der Code-Generierung auf eine plattformunabhängige Art und Weise (siehe Abbildung 1) zu erledigen, denn viele der notwendig gewordenen Veränderungen im GCC Back-End sind für die späten Phasen der Übersetzung, in denen die Repräsentation der Register Transfer Language (RTL) auf plattformabhängige *templates* in Maschinensprache abgebildet wird, völlig transparent.

Andere Projekte wie der Linux-Kernel, Mozilla oder *BSD haben ähnliche logische und physische Programmmodule, die eine optimierte Koordination der Beiträge der Mitwirkenden und einen besseren Ansatz für das Änderungsmanagement ermöglichen. Die Mehrheit dieser Module aber war nicht notwendigerweise ersichtlich und hat folglich auch nicht bestanden, als diese Projekte vor mehr als zehn Jahren initiiert wurden. Deswegen muss diese vernünftige Strukturierung aus dem Evolutionsprozess selbst resultieren, der stattfindet, wenn eine wachsende Anzahl an Mitwirkenden neue Quelltexte, Ideen und Lösungen beisteuert, um den bereits existierenden Code an vielen Stellen zu verbessern.

4.2. Verstrickung von Prozess und Architektur

Wir behaupten also, dass in gut gewarteten, erfolgreichen freien Software-Produkten die technische Struktur der zugrunde liegenden Architektur stets mit der Organisa-

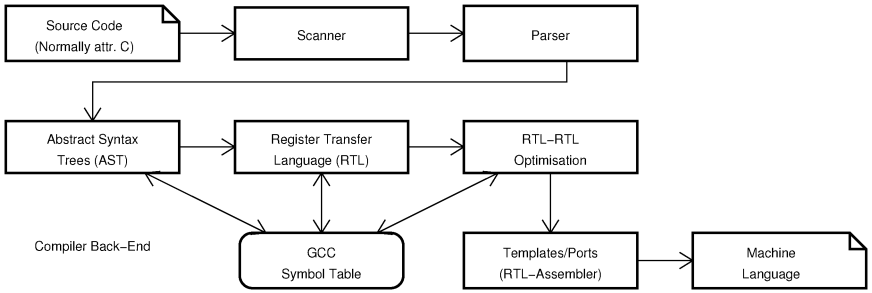

Abbildung 1: Logische und physische Module des GCC Kerns

tion des Projekts verstrickt ist. In anderen Worten: Jede wichtige Änderung in der Organisation des Projekts muss zwangsläufig zu einer Veränderung der technischen Architektur des Produkts führen und umgekehrt.

Abbildung 2 zeigt die starken Wechselbeziehungen zwischen der technischen Struktur des GCC und der Gesamtorganisation des Projekts. Sie zeigt auch, wie Ergänzungen zur Code-Basis zuerst durch eine öffentliche Überprüfung in Mailinglisten koordiniert werden, um offensichtliche Fehler zu eliminieren bevor sie in das *cvs repository* geschrieben werden. Im Falle des GCC durchlaufen sogar die *Maintainer* selbst diesen Prozess, um die anderen Mitwirkenden über die anstehenden Änderungen zu informieren. Tatsächlich ist es nicht ungewöhnlich, dass sich eine Änderung im Form eines Programmfragments (*patch*) noch während dieses Prozesses weiterentwickelt, bevor der *patch* ein fester Bestandteil der GCC-Suite ist. Der Grund dafür ist, dass große oder komplexe *patchs* gut verstanden werden müssen, bevor sie angenommen werden können. Folglich ist ein Prozess der konstanten Verfeinerung von Nöten, mit dem sich der *patch* an die sich kontinuierlich ändernde Code-Basis annähert. Unsere eigenen Erfahrungen in der GCC-Entwicklung haben gezeigt, dass dieser Prozess manchmal Wochen oder sogar Monate in Anspruch nimmt (Bauer 2003).

Folglich unterliegen die Gesamtorganisation und Reorganisation der Arbeiten der Mitwirkenden einem „natürlichen" Prozess, der hauptsächlich durch Notwendigkeiten und Begründungen getrieben wird und nicht durch Autorität. Aufgrund der starken Verstrickung von Entscheidungen bzgl. der Architektur und der Organisation des Projekts muss sich ein großer Teil der technischen Struktur genauso „natürlich" entwickeln: Die Lösung, die die anwendbarste, widerstandsfähigste oder die am leichtesten zu wartende ist, hat letztendlich Erfolg – vielleicht nicht sofort, aber sicherlich asymptotisch gesehen. Also eine „gesunde" Art der Evolution, die auch für viele proprietäre Projekte wünschenswert wäre, aber durch den Einfluss zusätzlicher Interessen nicht stattfindet.

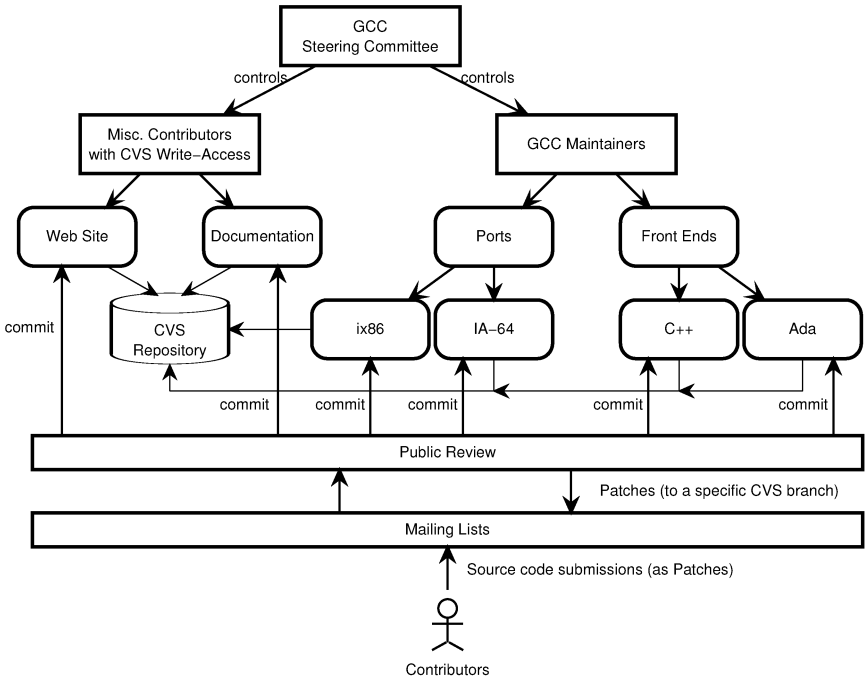

Abbildung 2: Die wesentlichen Elemente der GCC Projektstruktur und Organisation

4.3. Projekt- und Interprojekt-Abhängigkeiten

Obwohl kommerzielle Software-Anbieter natürlich um die Weiterentwicklung ihrer Software besorgt sind, haben sie oft eine statische, produktorientierte Projektorganisation, welche im strengen Kontrast zur Bedeutung des Wortes „Evolution" steht. In der Tat gibt es Fälle, in denen die Projektstruktur in proprietären Projekten mehr die geographische Verteilung der Firma widerspiegelt, als den Zweck und die Ziele des Produkts. Zum Beispiel ist das Team in Stadt *A* mit einer Aufgabe *a* betraut und das Team in Stadt *B* ist verantwortlich für Aufgabe *b*. Kann von Software wirklich erwartet werden, dass sie sich unter solchen statischen Bedingungen vernünftig entfaltet?

Ein anderes beliebtes Open-Source-Projekt mit kommerziellen Wurzeln stützt unsere Thesen. Als Netscape die Communicator-Quellen im Jahr 1998 (Cubranic und Booth 1999) veröffentlichte, wurde das Mozilla-Projekt geboren, das für viele Jahre ein Beispiel für ein unwartbares und deshalb erfolgloses Open-Source-Projekt war. Heutzutage ist Mozilla grundlegend anders als der Communicator und das Projekt hat viele Veränderungen in der Architektur durchlebt, die auch zu erfolgreichen kommerziellen Anwendungen der neu entstandenen Mozilla-Komponenten geführt haben. Abbildung 3 zeigt, wie die Komplexität dieses einst einzigen, riesigen Produktes syste-

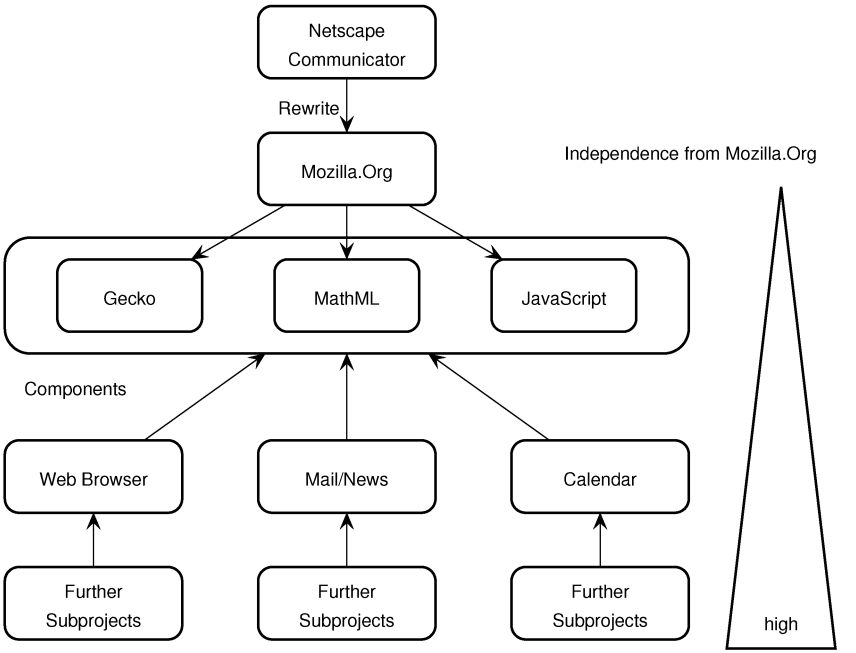

Abbildung 3: Interprojekt-Abhängigkeiten im Mozilla-Projekt

matisch in mehrere handhabbare Projekte aufgebrochen wurde, die mit dem weiteren Aufteilen des Projekts mehr und mehr an Flexibilität gewannen und unabhängiger von der Mozilla-Projektorganisation wurden.

Tatsächlich veröffentlichten im April 2003 die Mozilla-Projektleiter eine Erklärung, in der sie ihre bis dahin größte Aufteilung der Architektur ankündigten. Mozilla als solches wird nur als der Spitzname des Projekts weiterbestehen, während sich seine Kernkomponenten Mail, News und Web-Browser in separate Projekte verwandeln, wie sie die neue Roadmap widerspiegelt. Die Argumentation hinter diesen neuen Roadmap-Elementen lässt sich auf das Bevorzugen von Qualität statt Quantität zurückführen. Man muss sogar weniger tun, dafür aber besser und mit fehlerfreien Erweiterungsmechanismen. (*Mozilla Development Roadmap* 2003)

Momentan besteht Mozilla aus annähernd 50 „Kernprojekten" und über 2 Millionen Code-Zeilen, hat 13 Projektleiter und über 1 000 aktive Mitwirkende. Das Projekt hat sich letztendlich von seinen alten, nicht wartbaren Wurzeln befreit und floriert so sehr, dass Netscape in diesen Tagen seine Browser auf Mozilla basiert – anstatt umgekehrt.

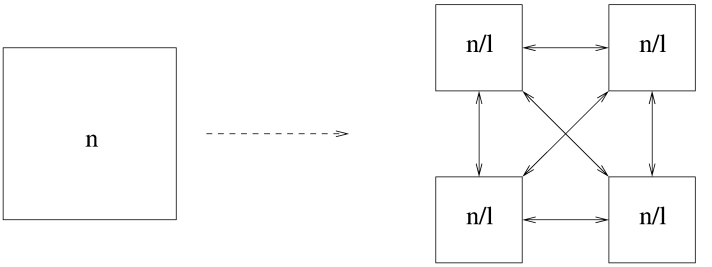

Abbildung 4: Code-Splitting

4.4. Komplexitätsreduktion durch Code-Aufteilung

Eine einfache, exemplarische Rechnung hilft, den Erfolg dieser Aufteilungsstrategie zu verstehen. Aus den Ergebnissen der Forschungsarbeiten zur Kostenschätzung, z. B. *function points* und COCOMO (Boehm 1981), ist die *Diseconomy of Scales* (Diversifikationsnachteil) wohl bekannt, das heißt, die Kosten eines Software-Projekts wachsen nicht linear mit dessen Größe, sondern wesentlich stärker.

Entsprechend dieser Erkenntnisse modelliert die COCOMO-Methode die Kosten eines Projekts mit Hilfe der Formel $A \cdot n^B$, wobei n ein Maß für die Größe des Systems ist (z. B. Quell-Code in KLOC – 1 000 Code-Zeilen) und $A, B > 1$ ist. Nun gehen wir von einem Stück Code aus, das aus n Modulen (ersetzbar durch KLOC) besteht. Weiter nehmen wir vereinfachend an, dass die Programmabhängigkeiten innerhalb dieses Programmfragments quadratisches Wachstum der Komplexität verursachen, d. h. $B = 2$. Dann kann die Gesamtkomplexität ausgedrückt werden durch folgende Formel:

$$C_{old} = O(n^2). \tag{1}$$

Wenn wir das Stück Code in l sorgfältig separierte Teile aufspalten, vermindert sich die Gesamtsystemkomplexität demnach auf

$$C_{new} = O\left(\frac{n^2}{l} + c \cdot l\right). \tag{2}$$

Mit anderen Worten: Die Komplexität C_{new} des gesamten Produkts wird signifikant reduziert. Die absolute Reduktion wächst sogar mit n und l. Die Konstante c ist ein linearer Faktor und repräsentiert die neuen Vermittlungskomponenten, die zwischen den nun getrennten l Teilen eingeführt werden müssen. Wegen der Entkopplung der l Teile können wir auch annehmen, dass die neuen Vermittler selbst keine quadratische Komplexität verursachen. Technisch wird dies durch den Gebrauch von schlanken Schnittstellen erreicht, wie sie von primitiven Operationen zur Interprozesskommunikation von Betriebssystemen oder gekapselten Datenstrukturen bereitgestellt werden.

Vom algorithmischen Standpunkt her bleibt die Komplexität – wenig überraschend – quadratisch in (1) und (2), aber in absoluten Zahlen wird C_{new} immer

kleiner sein als C_{old}, wenn n hinreichend groß gegenüber c ist. Die *Maintainer* von freien Softwareprojekten gewährleisten diese sinnvolle Teilung des Codes, sobald eine durchzuführende Änderung und die Verständlichkeit des Codes dies erfordern.

4.5. Inkrementelle Überarbeitungen

Erfolgreiche freie Softwareprojekte besitzen die Fähigkeit zur Durchführung inkrementeller Überarbeitungen von schwer zu wartenden Programmteilen. Mozilla[8] ist ein sehr gutes Beispiel für diese scheinbar gängige Praxis, denn Mozilla hat heute so gut wie keine Zeile Quell-Code mehr mit seinem Vorgänger Communicator gemeinsam, obwohl es niemals als Ganzes in einem einzigen Schritt neu aufgesetzt wurde.

Als kommerzielles Unternehmen machte die Firma Netscape allerdings einen schweren strategischen Fehler, als sie sich entschied, auf die vollständige Neuimplementierung der Code-Basis von Communicator zu warten, was letztendlich durch das Mozilla-Projekt geschah. Netscape war in dieser Zeit dazu gezwungen, die Version 5.0 des Communicators zu überspringen und hatte dadurch wertvolle Marktanteile hauptsächlich an den Internet Explorer von Microsoft verloren. Hätte Netscape die Prinzipien der erfolgreichen Evolution freier Software besser verstanden, so wäre dieser Schaden sicherlich abwendbar gewesen.

Ungeachtet der Verluste für Netscape hatte das Projekt Mozilla, das mit seiner freien und offenen Organisation keinem nennenswerten Marktdruck unterliegt, überwältigenden Erfolg. Im Rahmen von Mozilla sind eine ganze Reihe qualitativ hochwertiger Komponenten entstanden, die heute in vielen kommerziellen Anwendungen vorgefunden werden können (z. B. Borland Kylix API, AOL Web-Browser, Netscape Communicator, verschiedene eingebettete Web-Browser für Mobiltelefone und PDAs).

Dieser Fall zeigt die große Diskrepanz zwischen kommerzieller und freier Umgebung. In der freien Umgebung war das Vorgehen wegen der flexiblen, selbstgesteuerten Evolution sehr erfolgreich. In der kommerziellen Umgebung scheiterte das Vorhaben aufgrund fehlgeleiteter und unrealistischer Planung.

Wenn wir den Linux-Kernel als eine Neuimplementierung von AT&T UNIX, BSD oder MINIX betrachten, kommen wir zu ähnlichen Schlüssen: Noch vor einem Jahrzehnt wäre es für Firmen ein Desaster gewesen, das unreife Linux zu einem Baustein ihres Geschäftsmodells zu machen, aber für die freie Software-Gemeinde haben sich die letzten zehn Jahre intensiver Entwicklung als extrem erfolgreich herausgestellt. Und sogar im aktuellen Linux-Kern werden immer noch größere Teile schrittweise neu geschrieben. Ein aktuelles Beispiel dafür ist die Diskussion über die Restrukturierung des IDE-Layers.[9]

Die GCC-Suite, die sogar noch älter als Linux und Mozilla ist, unterliegt ebenso großen Veränderungen. Das Ziel des AST-Projekts[10] ist, die Handhabung der *abstract syntax trees* im Backend neu zu schreiben und dabei einen großen Teil der Optimierung von der Ebene der *register transfer language* auf die AST-Ebene anzuheben.

8 The Mozilla Project: http://www.mozilla.org/

9 siehe das Kernel Mailing List Archive http://www.uwsg.iu.edu/hypermail/linux/kernel/

10 Abstract Syntax Tree Optimizations http://www.gnu.org/software/gcc/projects/ast-optimizer.html

Hierzu gibt es Unterprojekte, wie den SSA-Zweig, der hauptsächlich von Red Hat gefördert wird, damit dieser verteilte und stufenweise Erneuerungsprozess zu einem Erfolg führt. Es ist offensichtlich, dass es keine triviale Aufgabe ist, ein 20 Jahre altes und gemächlich gewachsenes Backend eines Compilers zu restrukturieren. Frühere, erfolgreiche Überarbeitungen deuten jedoch auf die Tatsache hin, dass der Erfolg eines Open-Source-Projekts eng mit der Fähigkeit verbunden ist, sich selbst zu restrukturieren, wann und wo es notwendig wird. Wir gehen davon aus, dass dies ein weiteres wichtiges Prinzip der erfolgreichen Evolution langlebiger freier Softwareprojekte ist.

5. Fazit

Das erwiesenermaßen erfolgreiche Entwicklungsmodell freier Software ist eine ausgezeichnete Quelle für Prinzipien und Praktiken erfolgreicher Software-Evolution.

Im Gegensatz zu den meisten proprietären Softwareprojekten, ist die fortlaufende und uneingeschränkte Evolution ein inhärenter Bestandteil freier Software. Üblicherweise ist die einzige Konstante in einem freien Software-Projekt die ständige Veränderung. In Anbetracht Lehmans Gesetze der Software-Evolution (Lehman und Ramil 2001) ist es nicht überraschend, dass diese Strategie langlebige und qualitativ hochwertige Softwareprodukte hervorgebracht hat.

Der Prozess der Entwicklung freier Software ist alles andere als chaotisch sondern der Prozess und die Organisation skalieren mit der Größe des Projekts. Das heißt, Projekte starten fast ohne *overhead* und können rapide wachsen. Wenn jedoch eine bestimmte Größe überschritten wird, werden Regelungen, Lenkungsauschüsse und Werkzeuge je nach Bedarf hinzugefügt, d. h. der Prozess entwickelt sich. Die resultierende Organisation korreliert stark mit der technischen Struktur des Produkts und nicht mit der geographischen Verteilung der Teams oder dem Organigramm des Unternehmens.

Kommerzielle Software-Organisationen könnten aus der dynamischen Evolution von Prozessen, entsprechend dem Wachstum des technischen Produkts, ebenfalls großen Nutzen ziehen. Derzeit verwenden sie oft genau einen Prozess, der entweder starr in allen Projekten befolgt wird oder für die Projekte zuerst zugeschnitten werden muss. In der Regel gibt es keine Evolution des Prozesses wie bei freier Software, die die aktuellen Bedürfnisse des Projektes widerspiegelt. Obendrein schaffen es kommerzielle Softwareprojekte oft nicht, den richtigen Mitarbeitern die richtigen bzw. geeigneten Aufgaben zuzuweisen.

Natürlicher Wettbewerb und Auslese innerhalb von freien Software-Prozessen betonen eher fachliches Können als Autorität und Rang (in einem Unternehmen). Das erhöht die Qualität der Ergebnisse. Es ist sehr wahrscheinlich, dass eine Art von Wettbewerb kombiniert mit dynamischer Zuteilung von Rollen innerhalb von Unternehmen auch die Qualität von nicht-freien Softwareprodukten erhöhen würde. Natürlich würde das einen radikalen kulturellen Wechsel innerhalb der meisten Organisationen erfordern, aber dank agiler Methoden sind einige dieser Veränderungen bereits in kommerzielle Umgebungen diffundiert.

Neben der Evolution des Prozesses stehen einige der interessantesten Prinzipien freier Software in Zusammenhang mit der engen Verstrickung von einem sich ändernden Entwicklungsprozess und der Evolution der technischen Architektur, d. h. des Produkts selbst. Die Architektur des Produkts wird kaum vorausgeplant, sondern entwickelt sich frei mit den wechselnden Erfordernissen und mit der Größe des Produkts. Zu bestimmten Zeitpunkten werden die Architektur und Organisation in mehr oder weniger isolierte Teile aufgespalten, was zu unabhängig wartbaren Modulen oder sogar zu völlig neuen Produkten führt.

Die Entwicklung der Architektur wird von inkrementellen Überarbeitungen begleitet. Der Umfang der Überarbeitung wird allein durch die notwendige Änderung und die verfügbaren Ressourcen bestimmt. Im Gegensatz zu nicht-freien Umgebungen sind Überarbeitungen nicht durch nicht-technische Aspekte, wie mangelnde Rechte oder statische Verantwortlichkeiten, beschränkt. Das wiederum reduziert die Notwendigkeit für teure und ineffiziente *workarounds*, welche die Komplexität erhöhen und die Qualität und Wartbarkeit rapide sinken lassen. Freie Software entwickelt sich auf eine gesunde und natürliche Weise durch behutsames Erweitern und schonungslose inkrementelle Überarbeitung, bei der am Ende der am besten geeignete Code überlebt.

Die hier zusammengefassten Beobachtungen stützen unsere Anfangsthese, dass die freie Software-Bewegung Evolutionsstrategien hervorbringt, die weniger liberalen Umgebungen überlegen sind; ähnlich wie der Wirtschaftsliberalismus für die Ökonomie.

Bekanntmachung

Diese Arbeit wurde vom Deutschen Bundesministerium für Bildung und Forschung (BMBF) als Teil des Projekts ViSEK (Virtuelles Software- Engineering- Kompetenzzentrum) gefördert.

Literatur

Bauer, A. (2003), Compilation of Functional Programming Languages using GCC — Tail Calls, Master's thesis, Institut für Informatik, Technische Universität München. http://home.in.tum.de/~baueran/thesis/.

Beck, K. (1999), 'Embracing Change with Extreme Programming', *Computer* **32**, S. 70–77.

Boehm, B. (1981), *Software Engineering Economics*, Prentice-Hall.

Brooks Jr., F. P. (1995), *The Mythical Man-Month*, Addison Wesley.

Cederqvist, P. et al. (1993), *Version Management with CVS*, Signum Support AB.

Cubranic, D. und Booth, K. S. (1999), Coordinating open-source software development, *in* 'Eighth IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises', IEEE Computer Society Press, Stanford, CA, USA, S. 61–65.

Darwin, C. (1859), *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*, John Murray, London.

Godfrey, M. W. und Tu, Q. (2000), Evolution in Open Source Software: A Case Study, *in* 'Proceedings of the ICSM 2000', San Jose, CA, S. 131–142.

Harris, N. et al. (2002), *Linux Handbook / A guide to IBM Linux solutions and resources*, IBM International Technical Support Organization.

Intel Corporation (2001), *Intel Itanium Software Conventions and Runtime Architecture Guide*, Intel Corporation, Santa Clara, California. Intel document SC–2791, Rev. No. 2.4E.

Lehey, G. (2002), Evolution of a free software project, *in* 'Proceedings of the Australian Unix User's Group Annual Conference', Melbourne, Australia, S. 11–21.

Lehman, M. M. (1969), The Programming Process, techn. Bericht RC2722, IBM Research Centre, Yorktown Heights, NY.

Lehman, M. M. und Ramil, J. F. (2001), 'Rules and Tools for Software Evolution Planning and Management', *Annals of Software Engineering* .

Mozilla Development Roadmap (2003). http://www.mozilla.org/roadmap.html.

Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K. und Ye, Y. (2002), Evolution patterns of open-source software systems and communities, *in* 'Proceedings of the international workshop on Principles of software evolution', S. 76–85.

Opdyke, W. F. (1992), Refactoring Object-Oriented Frameworks, PhD thesis, University of Illinois at Urbana-Champaign.

Pizka, M. (1997), Design and Implementation of the GNU INSEL Compiler gic, Technical Report TUM I-9713, Technische Universität München.

Raymond, E. S. (1998), 'The Cathedral and the Bazaar'. Revision 1.40, http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

Savoye, R. (2002), DejaGnu – The GNU Testing Framework, techn. Bericht, Free Software Foundation.

Smith, A. (1776), *The Wealth of Nations*, William Strahan, London.

Southern, J. und Murphy, C. (2002), 'Eine zielgerichtete Explosion', *Linux Magazin* .

Succi, G. und Eberlein, A. (2001), Preliminary Results from an Empirical Study on the Growth of Open Source and Commercial Software Products, *in* 'Proceedings of the Workshop on Economics-driven Software Engineering Research, EDSER 3', Toronto, Canada, S. 14–15.

The Standish Group International, I. (1995), 'CHAOS'.

The Standish Group International, I. (1999), 'CHAOS: A Recipe for Success'.

Wheeler, D. A. (2002), 'More Than a Gigabuck: Estimating GNU/Linux's Size', http://www.dwheeler.com/sloc/. Version 1.07.

van der Hoek, A. (2000), Configuration Management and Open Source Projects, *in* 'Proceedings of the 3rd International Workshop on Software Engineering over the Internet', Limerick, Ireland.