

Software-Engineering und Software-Qualität in Open-Source-Projekten

Jan Tobias Mühlberg
Department of Computer Science
University of York
muehlber@cs.york.ac.uk

York, den 25. Januar 2006

Copyright

Diese Arbeit unterliegt den Bedingungen einer *creative commons*-Lizenz. Sie darf vervielfältigt, verbreitet, öffentlich aufführt, bearbeitet und kommerziell genutzt werden, sofern die resultierenden Arbeiten unter identischen Lizenzbedingungen weitergegeben und eine Nennung des Autors der Ursprungsarbeit erfolgt.



Abstract

The paper at hand is based on the preliminary results of a meta-study on literature dealing with open source software (OSS). It discusses the impact of development mechanisms towards the quality of the resulting products. The author concludes that OSS is in general at least comparable and performs sometimes better than closed source software. Its main shortcomings had been found while analysing the maintainability of the source code. The paper discusses the reasons for those deficiencies and shows options for avoiding them. It is intended to be a basis for future discussion of OSS software engineering.

Überblick

Ausgehend von vorläufigen Ergebnissen einer Metastudie zu Open-Source-Software (OSS), befasst sich die vorliegende Arbeit mit der Frage, inwieweit die bei der Entwicklung größerer OSS-Projekte eingesetzten Entwicklungsmechanismen Auswirkungen auf die Qualität ihrer Produkte haben. Der Autor kommt zu dem Ergebnis, dass OSS in der Regel als mit Closed-Source-Software vergleichbar oder besser angesehen wird und qualitative Defizite insbesondere den Bereich der Weiterentwickelbarkeit der Software betreffen. Im Folgenden werden das Zustandekommen dieser Defizite analysiert und Möglichkeiten zu deren Vermeidung aufgezeigt. Die Arbeit ist primär als Grundlage für eine weitere Diskussionen der Thematik Open-Source-Software-Engineering gedacht.

Inhaltsverzeichnis

- 1 Einleitung
- 2 Betrachtete Qualitätskriterien
- 3 Software-Qualität bei Open-Source-Software
- 4 Kritik am Open-Source-Ansatz
- 5 Zusammenfassung
- 6 Danksagung

Literatur

1 Einleitung

Die aus der Ökonomie bekannte Pareto-Verteilung¹ gibt es auch in der Software-Entwicklung: In der Regel lassen sich mit 20% des gesamten Entwicklungsaufwandes bereits 80% der geplanten Funktionalität eines Projektes umsetzen, während umgekehrt oftmals 20% eines Systems 80% des Arbeitsaufwandes verursachen. Angewandt auf die Qualität einer Software könnte die Regel auch in „20% der Einzelkomponenten einer Software verursachen 80% der auftretenden Fehler und des mit dem Betrieb und der Weiterentwicklung der Software einhergehenden Wartungsaufwandes“ umformuliert werden.

Kritisch ist das insofern, als dass Software-Produkte viel stärker als materielle Produkte aufeinander aufbauen – eine Eigenschaft, die insbesondere im Bereich von Open-Source-Entwicklungen² beobachtet werden kann. Gerade in der Möglichkeit der Wiederverwendung von bereits geschriebenem Quelltextes in anderen Pro-

¹Nach Vilfredo Frederico Pareto (1848 – 1923), ital. Ingenieur, Ökonom und Soziologe. Pareto fand heraus, dass in den von ihm untersuchten Volkswirtschaften 20% der Bevölkerung 80% des Gesamteinkommens verdienen (vgl. Tarascio, [26, S. 115 ff.]).

²Unter einer *Open-Source-Software* wird im Sinne dieses Textes stets eine *Free/Libre/Open-Source-Software* verstanden. Die philosophisch zweifellos sehr interessante Unterscheidung zwischen diesen Strömungen ist im Rahmen der vorliegenden Arbeit irrelevant.

jekten, liegt eine der Stärken des Open-Source-Ansatzes, die zu der beachtlichen Entwicklungsgeschwindigkeit führt, wie sie verschiedene Open-Source-Projekte an den Tag legen. Berücksichtigt man die in den Projekte angewandte Entwicklungsmethodik – das bunte Durcheinander des Basar-Stils³ und die o.g. häufige Wiederverwendung von Software-Komponenten – so ließe sich im Vergleich zwischen katedralenartig gebauter proprietärer und der Open-Source-Software eine Verschiebung der Pareto-Verteilung zu Ungunsten der Open-Source-Software vermuten. Erstaunlicherweise werden jedoch gerade Laufzeitstabilität und Wartbarkeit häufig als die Stärken von Open-Source-Entwicklungen herausgestellt.

Die vorliegende Arbeit befasst sich mit der Frage, inwieweit die bei der Entwicklung größerer Open-Source-Projekte eingesetzten Entwicklungsmechanismen Auswirkungen auf die Qualität der resultierenden Software haben. Hierfür werden zuerst die betrachteten Qualitätskriterien definiert und anschließend der Einfluss von Entwicklungsstil und -methodik auf diese Faktoren dargestellt. Den Kern der Arbeit stellt eine kritische Auseinandersetzung mit dem Open-Source-Entwicklungsmodell dar.

2 Betrachtete Qualitätskriterien

Qualitätskriterien für Software zu definieren ist in der Regel schwierig. Für den einen Anwender spielen dabei vielleicht die „Ergonomie“ und der schwammige Begriff „Sicherheit“ eine Rolle, für den anderen wird die „Performance“ der Software ausschlaggebend sein. Ganz selbstverständlich versteht jeder Anwender oder Entwickler unter jedem dieser Begriffe etwas völlig anderes. Tatsächlich wäre selbst die Verwendung von Eigenschaften wie „Schönheit“ oder „Eleganz“ aus der Sicht eines Software-Entwicklers ganz und gar nicht abwegig.

Aufgrund der Tatsache, dass letztere Eigen-

³Die Begriffe „Kathedrale“ und „Basar“ als Metaphern für Organisationsstrukturen in der Software-Entwicklung wurden vor allem von Raymond in [21] geprägt.

schaften begrifflich nur sehr schwer fassbar und erstere kaum objektiv bewertbar sind, beschränkt sich diese Arbeit auf die Betrachtung eines weiteren wichtigen Aspektes der Qualität von Software-Produkten: Unter dem Begriff *Funktionssicherheit* fassen wir all jene Eigenschaften zusammen, die Verfügbarkeit einer Software ausmachen. Das sind insbesondere deren *Zuverlässigkeit* und *Weiterentwickelbarkeit*⁴.

Sowohl die Zuverlässigkeit einer Software als auch deren Wartbarkeit hängen ganz entscheidend von den im Rahmen des Entwicklungsprozesses angewendeten Techniken und den technischen Fähigkeiten der Entwickler ab. Ebenfalls nicht unterschätzt werden dürfen hierbei deren persönliche Motivation und Disziplin.

In der „klassischen“ (wir sollten besser „akademischen“ sagen) Softwareentwicklung gibt es verschiedene gut voneinander separierte Prozesse. Diese umfassen beispielsweise, die Planung, die Anforderungsanalyse, einen mehrstufigen Software-Entwurf, die eigentliche Umsetzung und nicht zuletzt Tests und Integration (vgl. Boehm, [7] und [8]). Die Sicherstellung qualitativer Ansprüche an das Ergebnis dieser Prozesse wird in Unternehmen durch die Realisierung eines Qualitätsmanagements – ein weiterer Entwicklungsprozess – erreicht. Dieses geschieht klassischerweise in Anlehnung an die Normenreihe ISO 9000, die wiederum insbesondere Anforderungen an das Management eines Unternehmens spezifiziert. Die Steuerung offener Entwicklergemeinschaften nach diesen Richtlinien wird häufig als inpraktikabel dargestellt (vgl. bspw. Robbins, [22] in [9]) und erscheint auch dem Autor als nicht realistisch.

Als ebenso wichtig wie ein konsequentes Qualitätsmanagement kann der Einsatz moderner Software-Engineering-Techniken angesehen werden. Insbesondere um eine hohe Wartbarkeit und Erweiterbarkeit eines Software-Systems zu erreichen, steht hierbei die Arbeit in der Entwurfsphase der Entwicklung sowie eine konsequente Einhal-

⁴Der Begriff *Wartbarkeit* bezeichnet im Sinne dieser Arbeit die Wartbarkeit des Quelltextes einer Software und wird somit als Synonym zum Begriff *Weiterentwickelbarkeit* verwendet.

tung der entworfenen Systemspezifikation an erster Stelle. Beispielsweise ermöglicht erst eine gut durchdachte Modularisierung des Systems die Verteilung kleiner Teilaufgaben auf eine große Gruppe von Mitarbeitern. Sie erlaubt es dem einzelnen Entwickler das Gesamtsystem auf einer hohen Abstraktionsebene zu betrachten und reduziert damit dessen Komplexität ganz erheblich. Der Entwickler kann sich auf den für ihn interessantesten Bereich konzentrieren (vgl. Arief et al., [1] in [12]). Auch dies scheint dem Ansatz der verteilten Software-Entwicklung, wie sie in Open-Source-Projekten üblich ist, zu widersprechen – der Fokus der Entwickler liegt hier nämlich vorrangig bei der Produktion von Quelltext, also bei der eigentlichen Umsetzungsphase, die im akademischen Software-Engineering-Ansatz nur einen vergleichsweise kleinen Anteil an der gesamten Entwicklungszeit ausmacht. Die Konzeptionsphase scheint dagegen vernachlässigt zu werden (vgl. Wilson, [30]).

Wie also erreichen Open-Source-Projekte dennoch eine hohe Zuverlässigkeit und Wartbarkeit?

3 Software-Qualität bei Open-Source-Software

Betrachten wir zunächst den Ablauf und die Organisation von Open-Source-Projekten: Die meisten Projekte beginnen damit, dass ein Einzelner oder eine Gruppe von Entwicklern eine erste Version der Software schreibt und sie im Netz frei zur Verfügung stellt. Damit werden sowohl Benutzer als auch weitere Entwickler „eingeladen“, die Software zu verwenden oder Beiträge zu ihrer Entwicklung zu leisten. Wie auch Raymond in [21] darstellt, sind insbesondere die Anwender von großer Wichtigkeit für die weitere Entwicklung des Projektes. Sie werden in Open-Source-Projekten vielmehr als Mitentwickler denn als bloße Verbraucher angesehen. Auch die Entwickler werden nicht einfach in Strömen auf das Projekt aufspringen (vgl. O’Reilly, [19]). Ausgehend von Raymonds und O’Reillys Aussagen ist eher von einem zum Wachstum des Benutzerstammes proportionalen Wachstum der Entwicklergemeinde aus-

zugehen. Die weitere Entwicklung des Projektes wird normalerweise von dessen Gründern oder den aktivsten Entwicklern gesteuert. Die Aufgaben dieser Projektleitung bestehen vorrangig darin, offene Aufgabe und deren Prioritäten festzulegen, zu entscheiden welche Quelltextbeiträge in das Projekt aufgenommen werden und die Planung für die Freigabe neuer Versionen vorzunehmen (vgl. Samoladas et al., [23] und Fielding et al., [13] in [11]). Tatsächlich fällt der Führungsgruppe damit eine Machtposition zu, die dem Entwicklungsprozess nach außen hin ein fast traditionelles Aussehen verleiht. Sehr charakteristisch für Open-Source-Projekte ist jedoch, dass offene Aufgaben in der Regel nicht direkt vergeben werden sondern vielmehr die Entwickler ihren Fähigkeiten und Interessen entsprechende Aufgaben auswählen. Im Gegensatz zu der Arbeit von Bauer und Pizka (vgl. [3] in [2]) geht der Autor in dem vorliegenden Text also sehr wohl von einem „geordnet-chaotischen“ Entwicklungsprozess in Open-Source-Projekten aus. Wie später gezeigt wird, sind nämlich insbesondere die „Chaos-Prozesse“ auf eine hohe Wartbarkeit des Produktes und seines Quellcodes angewiesen – unabhängig von der Mitwirkung und dem finanziellen Engagement von Unternehmen aus der Wirtschaft.

Die offensichtlichen Mechanismen zur Sicherstellung hoher qualitativer Ansprüche an ein Open-Source-Projekt stellt die bereits weiter oben erwähnte Auswahl der Beiträge der Entwickler durch die Projektleitung und die Priorisierung der offenen Aufgaben des Projektes dar. Wie Garcia und Steinmueller in [14] darlegen, laufen die Projektleiter dabei jedoch Gefahr, durch den „Fork“ eines neuen Tochterprojektes⁵ „bestraft“ zu werden. Sie weisen jedoch darauf hin, dass dies nur sehr selten vorkommt und geben als Gründe hierfür vor allem die Art der hierarchischen Strukturen innerhalb eines Projektes an: Open-Source-Projekte sind wissensbasierte Gesellschaften, in denen vor allem technisch überlegenen Entwicklern die Projektlei-

⁵Da der Quelltext einer Open-Source-Software frei ist, haben Entwickler jederzeit die Möglichkeit, diesen als Ausgangspunkt für ein neues Projekt zu verwenden und keine weiteren Beiträge zur Entwicklung des Mutterprojektes mehr zu leisten. Der Begriff „forking“ (engl., Gabelung) bezeichnet den Vorgang der Abspaltung eines Tochterprojektes.

tung obliegt. Deren Schiedssprüche werden von andere Entwickler möglicherweise eher als rein managementorientierte Entscheidungen akzeptiert. Das Wohlwollen der Masse der Entwickler ist aus der Sicht des Autors von ganz entscheidender Bedeutung für die Durchsetzung qualitativer Ansprüche und Richtlinien in einer Entwicklergemeinde, deren Mitglieder nicht mit finanziellen Mitteln motiviert werden können⁶.

Wie Warsta und Abrahamsson (vgl. [29] in [10]) feststellen, entsprechen die in Open-Source-Projekten anzutreffenden Entwicklungsprozesse weitestgehend dem aus der Agilen Software-Entwicklung⁷ bekannten Vorgehensweisen. Verschiedene Autoren, beispielsweise Voightmann und Coleman (vgl. [28]), erklären, dass die hierbei entscheidenden Entwicklungsparadigmen – (1) Prüfung des Quelltextes durch viele Entwickler; (2) häufige Veröffentlichung des Quelltextes, auch in frühen Stadien des Entwicklungsprozesses; (3) gute, vorrangig autodidaktische Weiterbildung der Entwickler – insbesondere bei der Herstellung von Software für Anwendungsbereiche, in denen eine besonders hohe Funktionssicherheit gefragt ist, entscheidende Vorteile aufweisen. „Given enough eyeballs, all bugs are shallow“ (Raymond, [21]), die viel und kontrovers zitierte goldene Regel der Open-Source-Entwicklung; gewiss ist sie richtig, jedoch hat wohl bislang noch niemand wirklich ausreichend Augäpfel auf ein einzelnes Stück Quelltext angesetzt. Trotz aller Kritik an dieser sehr pragmatischen Behauptung (vgl. bspw. Bezroukov, [6]) ist es wichtig, darauf hinzuweisen, dass die Offenlegung von Quelltexten überhaupt erst die unabhängige Analyse und das schnelle Beheben von Fehlern einer Software ermöglicht. Thomas weist beispielsweise darauf hin (vgl. Thomas, [27]), dass viele Open-Source-Projekte auf-

⁶Wie verschiedene aktuelle Beispiele zeigen – allen voran Firmen wie IBM oder RedHat – gibt es immer mehr Software-Entwickler, die aus ihrer Tätigkeit in Open-Source-Projekten monetären Gewinn ziehen. Ghosh et al. belegen jedoch in [15, Part 4: Survey of Developers], dass nur etwa 16% der in einer repräsentativen Umfrage befragten Entwickler direkt mit der Entwicklung von Open-Source-Software Geld verdienen und die Open-Source-Bewegung damit primär auf freiwillige Zuarbeiten angewiesen ist.

⁷Der Einsatz agiler Prozesse in der Software-Entwicklung wird beispielsweise von Beck in [4] beschrieben.

grund geringer Anwender- und Entwicklerzahlen ohne einen auch nur ansatzweise hinreichenden Einsatz von Test- und Verifikationstechniken auskommen müssen. Eine Orientierung an Methoden der Agilen Software-Entwicklung impliziert also keinesfalls eine konsequente Anwendung damit verbundener Testmethoden⁸. Resultierend daraus besteht bei vielen Projekte ein hoher Bedarf an zusätzlichem Einsatz von Entwicklungsmethodik und der konkreten Durchführung von Test- und Verifikationstätigkeiten ([ebd.]).

Um die Qualität, insbesondere die Weiterentwickelbarkeit von Open-Source-Software messbar zu machen, bedienen sich verschiedene Arbeiten der direkten Analyse des Quelltextes.

In [24] analysieren Schach et al. 365 Versionen des Linux-Kernels hinsichtlich der gegenseitiger Abhängigkeiten zwischen 17 ausgewählten Komponenten und dem gesamten Kernel. Ein hohes Maß an gegenseitigen Abhängigkeiten sollte in Software-Systemen insbesondere deshalb vermieden werden, weil sie die Wahrscheinlichkeit negativer Auswirkungen eines Fehlers in einer Software-Komponente auf andere Komponenten erhöhen. Auch machen sie die Software dadurch, dass Änderungen in einem Modul häufig auch Änderungen in anderen Modulen erzwingen, schlechter weiterentwickelbar. Des Weiteren verlangen derartige Abhängigkeiten einem Entwickler sehr viel Verständnis für das Gesamtsystem der Software ab. Sie erlauben es ihm nicht, sich auf eine Komponente mit klar abgegrenzter Funktionalität zu konzentrieren und verstärken damit die genannten Probleme (vgl. Page-Johnes, [20]). Schach et al. kommen zu dem Schluss, dass der Linux-Kernel ein gut definiertes modulares Design aufweist – der Indikator hierfür ist das lediglich lineare Anwachsen des Quelltextes über den betrachteten Versionen. Jedoch schließen sie aus dem exponentiellen Anwachsen gegenseitiger Abhängigkeiten, dass der Betriebssystemkern ohne ein grundlegendes Redesign in zukünftigen Versionen nur noch sehr schwer erweiterbar sein wird. Leider

⁸Die von Beck propagierte Testmethode, das „Test-Driven Development“, erfordert beispielsweise, dass Tests vor den zu implementierenden Programmkomponenten spezifiziert und umgesetzt werden (vgl. Beck, [5]).

gibt es keine neueren Arbeiten, in denen die von Schach et al. untersuchten Merkmale für gegenwärtige Kernel-Versionen ausgewertet werden.

Zu ähnlichen Ergebnissen kommen Samoladas et al. in [23]. In der Arbeit vergleichen sie verschiedene Open-Source-Projekte und partiell sogar Open-Source- mit Closed-Source-Software anhand eines „Wartbarkeitsindex“. Dieser ergibt sich aus der Betrachtung verschiedener aus dem Quelltext bestimmbarer Werte wie dessen Größe, Komplexität und Selbsterklärungskraft durch Kommentare. Bei der Untersuchung von insgesamt neun Projekten in jeweils verschiedenen Entwicklungsstufen stellen sie fest, dass Open-Source-Projekte mit ähnlichen Problemen bezüglich der Weiterentwickelbarkeit zu kämpfen haben wie Closed-Source-Produkte, Open-Source-Entwicklungen der nicht-quelloffenen Software in dieser Hinsicht jedoch mindestens ebenbürtig ist und teilweise bessere Bewertungen erhält.

Insbesondere die Arbeit von Samoladas et al. weist konkret darauf hin, dass die anfangs erwähnte Pareto-Verteilung auf die jeweils untersuchten Software-Projekte zutrifft und es tatsächlich nur etwa 20% des Quelltextes sind, die zu schwerwiegenden Problemen bei der Funktionssicherheit der Produkte führen. Offensichtlich wird das große Potenzial, das quelloffene Software einzig durch ihre Quelloffenheit besitzt, oftmals nicht hinreichend genutzt.

4 Kritik am Open-Source-Ansatz

Der vorangestellte Abschnitt zeigt, dass Open-Source-Software in der Regel mit ähnlichen Problemen wie Closed-Source-Software zu kämpfen hat. Letztlich stellen weder Closed-Source noch Open-Source das Patentrezept für sichere, verlässliche und wartbare Software dar. Wie beispielsweise die Arbeit von Madanmohan und De ([18]) zeigt, wird Open-Source-Software dennoch in beachtlichem Umfang in kommerziellen Produkten eingesetzt; beispielsweise in Anwendungen aus dem Bereich der Netzwerksicherheit, einem Anwendungsgebiet in dem die qualitativen Eigen-

schaften der verwendeten Einzelkomponenten eine besonders wichtige Rolle spielen. Tatsächlich besteht aus der Sicht eines Angreifers kein großer Unterschied zwischen einer Closed-Source- und einer Open-Source-Anwendung. In beiden Fällen hat er zumindest die Möglichkeit den Objekt-Code⁹ einer Software zu untersuchen und erhält damit in jedem Fall Einblick in alle implementierungsspezifischen Details eines Software-Systems – auch wenn das im Fall der Closed-Source-Software etwas umständlicher ist. Der Closed-Source-Ansatz ist grundsätzlich ein Mechanismus aus der Geschäftswelt, der auf die Durchsetzung von Rechtsvorschriften angewiesen ist. Mit Sicherheit und Qualität hat er nichts zu tun (vgl. Witten et al., [31]). In gleicher Weise kann auch die Open-Source-Bewegung als eine eher philosophisch orientierte Strömung ohne jeden kommerziellen Hintergedanken verstanden werden (vgl. bspw. Stallman, [25]). Entscheidend ist jedoch, dass das bereits weiter oben erwähnte viele-Augen-Prinzip der Open-Source-Software das Potenzial gibt, ein weitaus höheres Niveau an Funktionssicherheit zu erreichen, als das bei Closed-Source-Software der Fall wäre. Witten et al. ([31]) stellen in ihrer Arbeit fest, dass die Offenheit des Quelltextes dazu beiträgt, die Sicherheit eines Software-Systems zu erhöhen und dass Open-Source-Software hinsichtlich bestimmter Klassen von Fehlern tatsächlich weniger anfällig ist. Auch bieten sie über die Verwendung öffentlich zugänglicher Datenbanken für Fehler dem Anwender eine höhere Transparenz bezüglich der Existenz und dem Bearbeitungsstatus bekannter Schwachstellen in dem jeweiligen Software-System (vgl. Koru und Tian, [17]). Vor allem in Einsatzgebieten, in denen der Anwender ein sehr hohes Maß an Sicherheit benötigt, wird ihm gar nichts anderes übrig bleiben als vorrangig Open-Source-Software zu verwenden: In einem großen und komplexen Szenario würde erst die Offenlegung des Quelltextes eine „rekursive“ Prüfung und Zertifizierung aller Einzelkomponenten und des Gesamtsystems ermöglichen. Dagegen

⁹Unter dem Begriff Objekt-Code wird der in Maschinsprache übersetzte Quelltext eines Programmes verstanden. Dieser stellt die letzten Endes lauffähige Version des Programmes dar.

erhöht die Geheimhaltung des Quelltextes jedoch in keiner Weise die Sicherheit einer Anwendung.

Um so tragischer stellt sich in diesem Zusammenhang das schlechte Abschneiden von Open-Source-Produkten bei der Analyse des Quelltextes hinsichtlich seiner Weiterentwickelbarkeit dar. In einem Entwicklungsprojekt, dessen expliziter Fokus darauf liegt, Dritten den freien Zugang zu den internen Strukturen der Software zu ermöglichen, sollte die Wartbarkeit des Quelltextes immer oberste Priorität haben. Die Integration einer großen Anzahl von Entwicklern in ein Software-Projekt ist anders nicht durchführbar. Insbesondere weil Neulinge andernfalls beim Einstieg in ein Projekt sehr viel Zeit in die Einarbeitung in dessen komplexe Struktur investieren müssen, die sie besser in das *Design* und die Umsetzung der anvisierten Erweiterungen des Projektes investieren möchten und sollen. Tatsächlich werden von vielen Autoren insbesondere Versäumnisse in der Designphase der Software-Entwicklung bei Open-Source-Projekten kritisiert. Beispielsweise weist Jørgensen in [16] darauf hin, dass eine der entscheidenden Hemmschwellen in dem von ihm untersuchten Open-Source-Projekt darin besteht, dass es für Entwickler die sich mit Designfragen befassen, sehr schwer ist, konstruktive Kritik bezüglich ihrer Vorschläge zu erhalten. In [30] formuliert Wilson das Problem folgendermaßen:

„Great programmers can work effectively without explicit design or coordination, but when average programmers try to emulate that improvisation, the results are rarely pretty.“¹⁰

Er erläutert ferner, dass Problemstellungen aus dem Bereich des Software-Designs und der Software-Architektur in Open-Source-Gemeinschaften oft nicht oder nur widerwillig diskutiert werden.

Die weiter oben aufgeführten Analysen von Quelltexten aus Open-Source-Projekten haben gezeigt, dass beispielsweise die dem Linux-Kernel

¹⁰Sinngemäß: Wirklich gute Entwickler können tatsächlich ohne expliziten Entwurf und ohne Organisation entwickeln. Wenn durchschnittliche Programmierer jedoch versuchen, derart zu improvisieren, sind die Ergebnisse weniger gut.

zugrundeliegende Modularisierung durchaus tragfähig ist. Um so wichtiger ist es daher, Designfragen auch für weitaus speziellere Probleme im Software-Entwurf zu stellen und zu diskutieren. Ein Open-Source-Projekt, das sich Herausforderungen der Gegenwart stellt und seinen Entwicklungsfokus auf eine strikte Trennung der Einzelkomponenten und deren Wartbarkeit richtet – selbst wenn das unter Umständen den Verzicht auf die schnelle Umsetzung neuer Funktionalität bedeutet – wird es wesentlich leichter haben, mit den Herausforderungen der Zukunft umzugehen. Beispielsweise wird der Einsatz formaler Methoden zur Verifizierung einer komplexen Anwendungssoftware oder gar eines ganzen Betriebssystems hohe Anforderungen an die Qualität des Quelltextes des betreffenden Projektes stellen. Open-Source-Software, die für den Einsatz in sicherheitskritischen Anwendungsbereichen geeignet sein soll, wird früher oder später den Bedürfnissen der Anwender nach Verifizierung und Zertifizierung nachkommen müssen. Kostenfaktoren sind hierbei erst einmal zweitrangig; viel wichtiger ist es, dass Open-Source-Entwicklungen ihre diesbezüglichen Vorzüge durch ein gezieltes *design for maintainability*, ausbauen. Das bedeutet insbesondere einen erhöhten Einsatz softwarearchitektonischer Maßnahmen, die explizit auf eine Erhöhung der Wartbarkeit des Quelltextes abzielen und in allen Phasen des Entwicklungsprozesses Anwendung finden.

5 Zusammenfassung

Die vorliegende Arbeit basiert auf umfangreichen Literaturanalysen zum Thema Open-Source-Software und Software-Engineering in Open-Source-Projekten. Es wurden verschiedene Vorzüge der in Open-Source-Entwicklungen eingesetzten Software-Engineering-Praktiken diskutiert und aufgezeigt, dass Open-Source-Software, im Falle von Projekten mit größeren Nutzer- und Entwicklerzahlen, der Meinung der gängigen Literatur zufolge, in qualitativer Hinsicht durchaus mit kommerzieller Closed-Source-Software konkurrieren kann. Ferner wird herausgearbeitet, dass die größten Defizite von Open-Source-Produkten

im Bereich der Wartbarkeit des Quellcodes liegen. Diese Probleme stellen in verschiedener Hinsicht gleichzeitig Chancen und Herausforderungen für die Open-Source-Gemeinschaft dar:

1. Einige größere Open-Source-Projekte werden in absehbarer Zeit nur noch sehr schwer erweiterbar sein, ein grundlegendes Redesign wird unter Umständen nötig.
2. Die Möglichkeit, Redesign und Refactoring sehr effizient realisieren zu können, gilt als einer der entscheidenden Vorzüge der Agilen Softwareentwicklung. Zu beweisen, dass dies auch bei umfangreichen Projekten realisierbar ist, stellt eine große Herausforderung für die Zukunft dar.
3. Die Chancen, die sich daraus ergeben, betreffen insbesondere die Verlässlichkeit der betroffenen Projekte und bieten ihnen die Möglichkeit, sich in Hinblick auf Verifizierung und Zertifizierung deutlich von Closed-Source-Entwicklungen abzuheben.

Der Autor kommt damit zu dem Schluss, dass eine stärkere Fokussierung auf Designfragen nötig ist, um die Stärken des Open-Source-Ansatzes gezielter nutzen zu können.

6 Danksagung

Die vorliegende Arbeit basiert zu einem sehr großen Teil auf einer im Rahmen des Projektes INNODES¹¹ entstandenen und unter <http://innodes.fh-brandenburg.de/bibdb> frei zugänglichen Bibliographie zum Themenkomplex „Open-Source-Software und Innovation“. Ich danke allen Mitarbeitern für ihre Beiträge zu diesem Projekt und hoffe, dass wir den zugrundeliegenden Datenbestand auch für die nächste Zeit weiter pflegen und damit einen Beitrag zur wissenschaftlichen Auseinandersetzung mit der Thematik „Open-Source-Software“ leisten können.

¹¹siehe <http://innodes.fh-brandenburg.de/>

Literatur

- [1] Budi Arief, Cristina Gacek, and Tony Lawrie. Software Architectures and Open Source Software – Where can Research Leverage the Most? In Feller et al. [12], pages 3 – 5. Available online at <http://opensource.ucc.ie/icse2001/>; visited June 19th 2005.
- [2] Matthias Bärwolff, Robert A. Gehring, and Bernd Lutterbeck, editors. *Open Source Jahrbuch 2005: Zwischen Softwareentwicklung und Gesellschaftsmodell*, Berlin, March 2005. Lehmanns Media – LOB.de. Available online at <http://www.opensourcejahrbuch.de/2005/>; visited on April 21th 2005.
- [3] Andreas Bauer and Markus Pizka. Der Beitrag freier Software zur Software-Evolution. In Bärwolff et al. [2], pages 95 – 112. Available online at <http://www.opensourcejahrbuch.de/2005/>; visited on April 21th 2005.
- [4] Kent Beck. *Extreme Programming*. Addison-Wesley, München, 2001.
- [5] Kent Beck. *Test-Driven Development*. Addison-Wesley, München, 2003.
- [6] N. Bezroukov. A Second Look at the Cathedral and the Bazaar. *First Monday*, 4(12), December 1999. Available online at http://firstmonday.org/issues/issue4_12/bezroukov/; visited January 14th 2005.
- [7] Barry W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226 – 1241, December 1976.
- [8] Barry W. Boehm. A spiral model of software development and maintenance. *IEEE Computer*, 21(5):61 – 72, May 1988.
- [9] Joseph Feller, Brian Fitzgerald, Frank Hecker, Scott Hissam, Karim Lakhani, and André van der Hoek, editors. *Meeting Challenges and Surviving Success: The 2nd Workshop on Open Source Software Engineering*. ACM, 2002. Available online at <http://opensource.ucc.ie/icse2002/>; visited January 14th 2005.
- [10] Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors. *Taking Stock of the Bazaar: Proceedings of the 3rd Workshop on Open Source Software Engineering*. ACM, 2003. Available online at <http://opensource.ucc.ie/icse2003/>; visited June 28th 2005.
- [11] Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani, editors. *Perspectives on Free and Open Source Software*, Cambridge, July 2005. The MIT Press Ltd.
- [12] Joseph Feller, Brian Fitzgerald, and André van der Hoek, editors. *Making Sense of the Bazaar: 1st Workshop on Open Source Software Engineering*. ACM, 2001. Available online at <http://opensource.ucc.ie/icse2001/>; visited June 19th 2005.
- [13] Roy T. Fielding, James D. Herbsleb, and Audris Mockus. Two Case Studies of Open Source Software Development: Apache and Mozilla. In Feller et al. [11], pages 163 – 209.
- [14] Juan Mateos Garcia and W. Edward Steinmueller. The Open Source Way of Working: A New Paradigm for the Division of Labour in Software Development?, January 2003. Available online at http://siepr.stanford.edu/programs/OpenSoftware_David/oswp1.pdf; visited on October 24th 2005.
- [15] Rishab Aiyer Ghosh, Bernhard Krieger, Ruediger Glott, Gregorio Robles, and Thorsten Wichmann. Free/Libre and Open Source Software: Survey and Study – FLOSS. Final Report, International Institute of Infonomics University of Maastricht, The Netherlands; Berlecon Research GmbH Berlin, Germany, June 2002. Available online at <http://www.infonomics.nl/FLOSS/report/index.htm>; visited January 18th 2005.
- [16] Niels Jørgensen. Putting it all in the Trunk: Incremental Software Development in the FreeBSD Open Source Project. *Information Systems Journal*, 11(4):321 – 336, 2001.
- [17] A. Günes Koru and Jeff Tian. Defect Handling in Medium and Large Open Source Projects. *IEEE Software*, 21(4):54 – 61, 2004.

- [18] T. R. Madanmohan and Rahul De. Open Source Reuse in Commercial Firms. *IEEE Software*, 21(6):62 – 69, 2004.
- [19] Tim O’Reilly. Ten Myths about Open Source Software. Website, 2000. Available online at http://opensource.oreilly.com/news/myths_1199.html; visited on January 8th 2005.
- [20] Meilir Page-Jones. *The practical guide to structured systems design*. Yourdon Press Computing Series, New York, NY, USA, 2nd edition, 1988.
- [21] Eric Steven Raymond. The Cathedral and the Bazaar. Website, 2000. Available online at <http://www.catb.org/~esr/writings/cathedral-bazaar/>; visited on January 9th 2005.
- [22] Jason E. Robbins. Adopting OSS Methods by Adopting OSS Tools. In Feller et al. [9], pages 42 – 44. Available online at <http://opensource.ucc.ie/icse2002/>; visited January 14th 2005.
- [23] Ioannis Samoladas, Ioannis Stamelos, Lefteris Angelis, and Apostolos Oikonomou. Open Source Software Development Should Strive for Even Greater Code Maintainability. *Communications of the ACM*, 47(10):83 – 87, October 2004.
- [24] Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the Linux Kernel. *IEE Proceedings - Software*, 149(1):18 – 23, 2002.
- [25] Richard M. Stallman. Why Software Should Be Free. Website, July 2005. Available online at <http://www.gnu.org/philosophy/shouldbefree.html>; visited on July 22nd 2005.
- [26] Vincent J. Tarascio. *Pareto’s Methodological Approach to Economics*. The university of North Carolina Press, Chapel Hill, North Carolina, USA, 1968.
- [27] Craig Thomas. Improving Verification, Validation, and Test of the Linux Kernel: the Linux Stabilization Project. In Feller et al. [10], pages 133 – 136. Available online at <http://opensource.ucc.ie/icse2003/>; visited June 28th 2005.
- [28] Michael P. Voightmann and Charles P. Coleman. Open Source Methodologies and Mission Critical Software Development. In Feller et al. [10], pages 137 – 141. Available online at <http://opensource.ucc.ie/icse2003/>; visited June 28th 2005.
- [29] Juhani Warsta and Pekka Abrahamsson. Is Open Source Software Development Essentially an Agile Method? In Feller et al. [10], pages 143 – 147. Available online at <http://opensource.ucc.ie/icse2003/>; visited June 28th 2005.
- [30] Greg Wilson. Is the Open-Source Community Setting a Bad Example? *IEEE Software*, 16(1):23 – 25, 1999.
- [31] Brian Witten, Carl Landwehr, and Michael Caloyannides. Does Open Source Improve System Security? *IEEE Software*, 18(4):57 – 61, 2001.