Seminar „Open Source Software Engineering“

Wintersemester 2004

Qualitätssicherung bei Open-Source-Projekten

István Bartkowiak

bartkowi@inf.fu-berlin.de

Betreuer: Christopher Oezbek

Berlin, den 10. April 2005

Zusammenfassung

Die Entwicklung von Open-Source-Software (OSS) war von Beginn an geprägt durch die Problematik, die Qualität des Produktes nur schwer beurteilen zu können. Durch die massiv-parallele Entwicklung und das Fehlen bewährter Kontrollstrukturen der konventionellen Softwareentwicklung mussten neue Strategien ersonnen werden, um die Qualität sicherzustellen. Diese Arbeit unternimmt eine Bestandsaufnahme der etablierten Prozesse der Open-Source-Szene und beleuchtet die noch vorhandenen Konfliktfelder.

Inhaltsverzeichnis

1 Einleitung

Einer der herausragendsten Vorteile des Open-Source-Prinzips ist die Möglichkeit, jederzeit Einblick in den Quellcode nehmen zu können. Besonders bei sicherheitskritischen Produkten (s. Abschnitt 8.2), die in sensiblen Bereichen eingesetzt werden sollen, ist dies eine unabdingbare Voraussetzung für deren Verwendung. Sowohl der Anwender, der mit der Software arbeiten möchte, als auch Dritte, die mit der Sicherheitsüberprüfung beauftragt wurden, können sich davon überzeugen, ob der Quellcode *sicher* ist [uoD00].

Ein Nachteil, der aus den Freiheiten bei der Entwicklung entsteht, ist das Fehlen jedweder Dokumentation – nur wenige Projekte können derartiges vorweisen. Eine Dokumentation ist allerdings erforderlich, wenn man ein Produkt auf Defekte – sowohl architektonischer als auch algorithmischer Art – hin untersuchen möchte, siehe dazu Abschnitt 2 ff.

Änderung und Anpassung von Programmquellcode ermöglichen die Entwicklung von datenschutzfreundlichen Produkten und Technologien im weiteren Sinne und stellen somit eine wichtige Voraussetzung für einen verantwortungsvollen Umgang mit digitalisierten Daten dar (s. Abschnitt 6). Dass Software stets als potentiell mängelbehaftet angesehen werden muss, zeigte sich in der Vergangenheit häufig. An dieser Tatsache wird auch OSS nicht viel ändern können. Gerade deshalb sind formalisierte Verfahren zur Defektbehebung notwendig, um den Vorteilen des Open-Source-Gedankens genügend Freiraum zu verschaffen.

Hinweis an den Leser Diese Arbeit stützt sich auf möglichst aktuelle Quellen, um den Stand der Entwicklung zu skizzieren. Dabei werden zunächst die Quellenaussagen mit einem entsprechenden Quellenverweis wiedergegeben und anschließend kommentiert. Diese Vorgehensweise soll dazu dienen, die einzelnen Perspektiven unverzerrt darzulegen.

1.1 Was ist unter Qualität zu verstehen?

Um sich über die Qualitätsfragen bei der Softwareentwicklung sinnvoll verständigen zu können, muss genau definiert werden, nach welchen Kriterien geurteilt werden soll. Ein weit verbreitetes Verständnis von Qualität ist der **Reifegrad** eines Produktes [Kad00]. Jedoch wird hier ein Verständnisproblem durch ein anderes ersetzt, da auch der Begriff „Reifegrad“ schwer zu fassen ist. Er lässt sich zwar nach genauen Vorgaben quantitativ messen. Ebenso kann qualitativ beurteilt werden, ob ein Produkt den Erwartungen entspricht – vorausgesetzt es wurde vorher genau festgelegt, welche Produkteigenschaften einer Bewertung unterzogen werden sollten. Sowohl bei der quantitativen, wie auch bei der qualitativen Herangehensweise sind als Ergebnis aber stets nur Momentaufnahmen zu bekommen.

Ein anderer Ansatz ist daher, den **Entstehungsprozess** einer Software in die Qualitätsbeurteilung mit einzubeziehen. Dieser prozessbezogene Ansatz liefert unter der Voraussetzung einer vollständigen und widerspruchsfreien Produktspezifikation sogar ein Mittel der Belegbarkeit von Qualität, s. dazu auch Abschnitt 7).

Darüber hinaus beschrieb die Wirtschaft einen weiteren Ansatz, die Qualität eines Produktes zu beurteilen. Es ist zwar von Vorteil, belegen zu können, dass eine Software spezifikationskonform entstanden ist, doch ist dies nicht hilfreich, wenn niemand etwas mit dem Produkt anfangen kann. Ob ein Produkt zum Einsatz kommt, entscheiden der Preis und die **Integrierbarkeit** in laufende Prozesse. Beides lässt sich empirisch ermitteln und wird gemeinhin als *Kosten-Nutzen-Rechnung* bezeichnet.

1.2 Voraussetzungen für eine Qualitätsbeurteilung

Zu Beginn der Open-Source-Bewegung konnte oft beobachtet werden, dass bei Projekten ein Systementwurf schlichtweg fehlte [Koc01]. Das Vorhandensein eines solchen Entwurfes ist aber unabdingbar. Es muss bei einer Qualitätsprüfung stets ein Bauplan vorliegen, der als Basis für die Entstehung des Produktes dient. Dies ist auch der Grund, weshalb die Übertragung von bewährten Qualitätssicherungsverfahren auf OSS-Projekte in der Vergangenheit scheiterte. Nachträgliche Korrekturen von Architekturfehlern der Software waren besonders aufwendig und schwierig. Da nicht bekannt war, wie weitreichend die Folgen einer Änderung sein würden, konnte eine Korrektur ganz neue Defekte hervorbringen oder das Produkt schlichtweg ruinieren. Mittlerweile hat man aus diesen Fehlern gelernt und erkannt, dass eine Qualitätssicherung auch bei OSS-Projekten bereits von Beginn an erfolgen muss. Sie kann daher als wohlwollender Berater verstanden werden *(Beraterfunktion)*.

Die Fehlervermeidung ist die effektivste Fehlerbehebung. Deshalb sollte man darauf achten, dass zu jedem Zeitpunkt des Entwicklungsprozesses das Auftreten von Fehlern ein bestimmtes Niveau nicht übersteigt [Hen01] (s. dazu auch Abschnitt 8.2).

1.3 Schaffung eines Qualitätsbewusstseins

Bei der Problematik der Qualitätsbeurteilung gibt es natürlich auch andere Ansätze. So verfolgt [Bac95] die These, dass Software nur so gut zu sein braucht, wie vom Benutzer erwartet wird *(discipline of utilitarian approach)*. Dieser Ansatz mag verwundern, da man argumentieren kann, dass sich Erwartungen stets ändern können. Doch sollte nicht vergessen werden, dass man schon seit jeher mit Etappenzielen gearbeitet hat. Es bringt keine Vorteile bereits von Anfang an alles zu wollen. Vielmehr ist es sinnvoll, sich Stück für Stück an festgelegte Ziele *(Meilensteine)* heranzuarbeiten. Wenn man Qualität als Lösungsmenge versteht und die Meilensteine als Problemmenge, so kann man sich sukzessive vorarbeiten, ohne das Risiko einzugehen, bereits gewonnenes Terrain wieder zu verlieren.

Ins Gegenteil verkehrt hat diese Philosophie die Firma *Microsoft*. Da sie ausschließlich Closed-Source-Software (im Folgenden als *CSS* bezeichnet) herstellt und die eigene Monopolstellung um jeden Preis verteidigt, verfolgt sie die Prämisse *„Sell the right product at the right time." [„Verkaufe das richtige Produkt zur richtigen Zeit"]*. So setzt sie sich selbst Fertigstellungstermine *(deadlines)* und verfolgt diese um jeden Preis, um Konkurrenten einen Schritt voraus zu sein. Es zeigt sich jedoch stets, dass sie die eigenen Termine so gut wie nie halten kann. Als Konsequenz werden Funktionalitäten *(features)* kalkuliert weggelassen und unbedingt notwendige derart implementiert, dass sie oberflächlich zu funktionieren scheinen. Notwendige Korrekturen werden dann nach und nach veröffentlicht, so dass sich hier der Begriff *Bananensoftware* etabliert hat – also Software, die erst beim Kunden reift und die letztlich gewollte Qualität erheblich später erreicht.

1.4 Schaffung von Qualitätsstandards

Neue Zertifizierungs- und Prozessstandards tolerieren mittlerweile die Verwendung nahezu jeden Entwicklungsprozesses. Diese Austauschbarkeit macht diese Standards zu sog. *Metaprozessen*. So erlaubt der Standard ISO 9126 mit seinen 21 Eckpunkten aus 6 Schwerpunktbereichen eine Übertragung auf OSS-Projekte. Da der Standard aus dem professionellen Sektor stammt, kann ein standardkonformes OSS-Projekt den gleichen Stellenwert erlangen wie ein konventionell geführtes Projekt (s. Abschnitt 7). Weist ein OSS-Produkt eine hohe Qualität auf, so wird dies selten den Entwicklern direkt mitgeteilt. Vielmehr erwarten diese Fehlerberichte und Verbesserungsvorschläge. Daher kann keine Kritik sehr wohl als gute Kritik verstanden werden.

1.5 Begriffsdefinitionen

Um sich einen Überblick zu verschaffen, an welchen Stellen unerwünschte Produkteigenschaften auftreten – Juristen sprechen an dieser Stelle von Produktmängeln –, ist es notwendig, einige Begriffe zu definieren, s. dazu [Som04].

- **Bug** - Der Begriff *Bug* wird oft für jede Art von Fehler in einer Software verwendet. Für eine Betrachtung bei der Qualitätssicherung ist dies jedoch bei weitem unzureichend. Um einen Mangel an einem Produkt zu kennzeichnen, mag dies jedoch genügen.

- **Error** - Beschreibt alle Probleme, die mit der Bedienung einer Software durch den Benutzer auftreten. So kann beispielsweise eine falsch programmierte Programmoberfläche ungültige oder widersprüchliche Werte entgegennehmen, die von der dahinter liegenden Programmlogik nicht verarbeitet werden können.

- **Defect** - Jede Art von Fehler in der Programmlogik wird hier als Defekt verstanden. Dabei muss beachtet werden, dass semantische Fehler Logikfehler verursa-

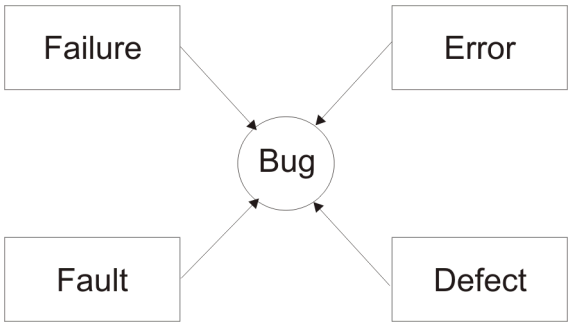

Abbildung 1: Begriffshierarchie

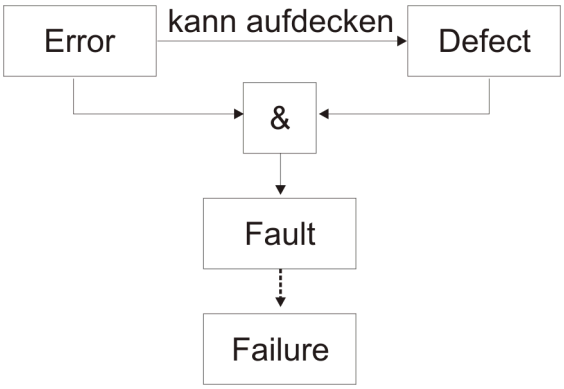

Abbildung 2: Begriffsbeziehungen

chen. Aber auch das Fehlen von Programmlogik kann zu solchen Logikfehlern führen.

- **Fault** - Ein *Fault* bezeichnet einen ungewollten Programmzustand. Er kann durch einen Defekt impliziert sein, jedoch auch durch äußere Einflüsse eintreten, z.B. beim Zugriff auf externe Ressourcen. Ist die Software in der Lage, derartige Zustände zu erkennen und selbstständig abzufangen, wird sie gemeinhin als robust verstanden. Im qualitativen Sinn wird sie dagegen als fehlertolerant bezeichnet.

- **Failure** - Dies ist der schwerwiegenste Mangel, der bei einer Software eintreten kann. Er bezeichnet das Versagen der Software im Sinne eines „Absturzes".

Abb. 1 verdeutlicht die Begriffshierarchie, Abb. 2 setzt die Begriffe zueinander in Beziehung.

1.6 Verwaltung von Softwarefehlern

Wie Softwarefehler verwaltet werden, hängt stark vom jeweiligen Projekt ab. So wurde bei *Debian GNU/Linux* eine Gruppeneinteilung vorgenommen, wobei jede dieser Gruppen einen genau festgelegten Zuständigkeitsbereich besitzt [Mie02]. Um eine unkomplizierte Erreichbarkeit zu gewährleisten und um eine Vorfilterung der eingehenden Berichte durchzuführen, ist jede Gruppe mit einer gesonderten E-Mail-Adresse erreichbar. Jene dezentrale Strukturierung der Verwaltungsaufgaben erlaubt den Gruppen, unabhängig voneinander zu arbeiten. Damit dies auch effektiv funktioniert und möglichst keine Abhängigkeiten auftreten, ist die Softwarearchitektur ebenfalls modular aufgebaut. Somit sind Änderungen an einzelnen Modulen erlaubt, ohne dass eine Gruppe auf die Fertigstellung einer anderen Gruppe warten muss. Debian besteht aus über 8.000 sog. Paketen, die jeweils ein eigenes Projekt darstellen. Jedem dieser Unterprojekte ist ein Betreuer *(maintainer)* zugeordnet, der sein Produkt verantwortet. Die Granularität der Vorfilterung erhöht sich innerhalb des Debian-Projektes weiter, so dass jedes einzelne Unterprojekt eine eigene E-Mail-Adresse besitzt, die sich aus dem Namen des Unterprojektes zusammensetzt. Damit die Arbeiten koordiniert erfolgen, überwacht der Betreuer den Fortschritt und führt eine Prioritätssetzung der eingehenden Berichte durch. Des Weiteren obliegt ihm die Aufgabe, seine untergeordneten Projektbeteiligten zu motivieren und alle von ihnen benötigten Ressourcen bereitzustellen. Bei geographisch nicht getrennten Gruppen sogar die Sicherstellung von Räumlichkeiten und technischer Ausstattung und die Bestellung von „Futtermitteln“ [Koc01].

1.7 Handhabung von Fehlerberichten

Als favorisierte Kommunikationsmethode hat sich bei OSS-Projekten die E-Mail durchgesetzt. Dabei geht man sogar noch einen Schritt weiter und setzt sog. Nachrichtenverteiler *(mailing lists)* auf. Diese nehmen einzelne Fehlerberichte in Form einer E-Mail an und schicken sie als Kopie an alle eingetragenen Interessenten. Auf diese Weise verfügt jeder, der sich bei einem solchen Verteiler registriert hat, über die gleichen Informationen [Mie02]. Positiver Nebeneffekt dieser Kommunikationsart ist die Ungezwungenheit: Ein lockerer Umgangston verführt zu freiem Denken und motiviert, unorthodoxe Lösungsvorschläge zu unterbreiten. Da E-Mails genau betrachtet oft nur reinen Text als Information enthalten, ist deren Archivierung und die Suche nach Stichwörtern denkbar einfach. Zusätzlich kann man mit standardisierten Vorlagen und Markierungen *(tags)* eine automatisierte Verarbeitung realisieren, der sich eine Fehlerbaumanalyse anschließen kann.

Bestrebungen in diese Richtung verfolgt das *Mozilla Project* [Org04], welches mit einer Fehlerdatenbank *(bug tracking system)* arbeitet. Das Projekt nennt diese Datenbank *Bugzilla*, die neben dem reinen Fehlerbericht auch dessen Bearbeitungsstatus vorhält. Die Eingabe von neuen Fehlern *(posting)*, deren Verwaltung und die automatische Suche nach Konflikten und Abhängigkeiten stellt momentan den aktuellen Stand des Machbaren dar und erlaubt die Kanalisierung und Zielorientierung der Kommunikation [Wie04].

Analog zu kommerziellen Softwareunternehmen findet bei den Debian-Projekten eine Aufteilung der Fehlerberichte in Prioritätsklassen statt. Dabei wird unterschieden zwischen *kritisch*, *schwer wiegend*, *wichtig*, *normal*, *behoben* und *Wunschliste*. Letzterer Klasse werden auch Anregungen und Verbesserungsvorschläge zugeordnet, die im Zusammenhang mit einer bestimmten Funktionalität stehen.

2 Software Errors

Eine große Schwäche von OSS war in der Vergangenheit das Fehlen einer leicht zu bedienenden Programmoberfläche und einer guten Produktdokumentation. Diese Aufgabe übernahmen daher sog. Distributoren (z.B. Suse/Novell, RedHat etc.), die neben dem eigentlichen Zusammenstellen von einzelnen OSS-Paketen zunehmend auch dazu passende Oberflächen erstellten oder zu dessen Programmierung anregten. Durch Letzteres findet in der OSS-Szene langsam ein Wandel statt, der neben der reinen Funktion einer Software auch dessen ergonomische Bedienung in den Vordergrund stellt. Dennoch: Die Hauptaufgabe der Distributoren wird auch in Zukunft darin bestehen, einzelne OSS-Pakete in ein Produkt zu integrieren, Installationsroutinen zu entwickeln, gute Dokumentationen zu erstellen und nicht zuletzt Wartung und Produktunterstützung *entgeltlich* anzubieten [uoD00].

2.1 Möglichkeiten der Benutzerbefragung

Neben der Theorie entwickelte die Praxis eigene Methoden, um den Benutzer bei Problemen zu unterstützen. Da mit OSS direkt kein Geld zu verdienen ist, sondern nur über daran geknüpfte Dienstleistungen, mussten neue unkonventionelle Wege gefunden werden, mit den Benutzern zu kommunizieren. Als Beispiel sei hier erneut das Mozilla Project genannt, welches das bekannte Prinzip der Diskussionsforen *(newsgroups)* für diesen Zweck nutzt. Der Benutzer kann sich hier mit anderen austauschen, seine Probleme schildern und nach passenden Lösungen suchen. Häufig findet sich ein erfahrener Benutzer oder sogar ein Entwickler, der sich der Probleme annimmt [Org04].

Um die Benutzerbefragungen für den Anwender angenehm zu gestalten, bietet die Bugzilla benutzerfreundliche Eingabeassistenten *(user-friendly wizards)* an, die bei der Entgegennahme von Fehlerberichten behilflich sind. Bei Debian ist es dagegen gängige Praxis E-Mail-Berichte zu erstellen, die neben der reinen Fehlerbeschreibung auch den Namen des Paketes enthalten. Die zuständigen Betreuer des Paketes erhalten stets eine Kopie der E-Mail ebenso wie alle beim Nachrichtenverteiler `debian-bugs-dist` registrierten Entwickler [Gra02]. Gerade an derart großen Projekten wie Debian zeigt sich immer stärker die Bereitschaft, Fehlermeldungen automatisiert und nach Kategorien sortiert zu erfassen [uSE03]. Eine der großen Herausforderungen in naher Zukunft wird sein, dem Benutzer das Gemeinschaftsgefühl zu vermitteln und ihn in den Entwicklungsprozess mit einzubeziehen.

2.2 Möglichkeiten der Benutzerhilfe

Oft wird argumentiert, dass einer der größten Nachteile von *freier* Software sei, dass kein Anspruch auf Hilfe seitens des Herstellers – also des Autors der Software – bestehe [uoD00]. Dass ein derartiger Anspruch tatsächlich nicht besteht, täuscht jedoch darüber hinweg, dass auch die kommerziellen Anbieter von CSS ihre Dienste meist nur gegen ein Entgelt anbieten. Wenn sich der Benutzer dann selbst helfen muss, ist er auf eine gute Produktdokumentation angewiesen. Nicht jedem ist damit geholfen, einen kurzen Blick in den Programmquellcode zu werfen und daraus das weitere Vorgehen abzuleiten. Der bereits erwähnte Mangel an Dokumentation bei OSS legt den Finger auf die Wunde: Kann nur der Quellcode als Hilfe dienen, macht sich schnell Ratlosigkeit breit. Genau an dieser Stelle setzen erneut die Diskussionsforen an. In der Regel helfen sich hier die Benutzer gegenseitig. So bietet das Mozilla-Projekt zusätzlich eine FAQ *(frequently asked questions [häufig gestellte Fragen])*, die die brennendsten Probleme der Benutzer behandelt [Org04]. Als Analogon zu verstehen ist die Bug-FAQ: Sie listet die am häufigsten berichteten Softwarefehler auf, für die noch keine direkte Lösung existiert. Ebenso wie E-Mails lassen sich auch die Foren nach Schlüsselbegriffen *(key words)* durchsuchen.

Als Ergänzung dienen den OSS-Projekten ihre produktbezogenen Internetseiten. Hier werden das Projekt und das Produkt vorgestellt und vermarktet, seine Funktionen beschrieben und sonstige Neuheiten angekündigt [Kru04]. Eine weitere wichtige Funktion ist die Angabe der Download-Server *(mirrors)*, von denen die Software und Korrekturen *(patches)* bezogen werden können und die Bereitstellung von Benutzerhandbüchern erfolgt, wie dies z.B. bei *FreeBSD* und Debian der Fall ist. Beide Projekte bieten auf diese Weise mehrsprachige Installations- und Nutzungsanleitungen an. Eine Arbeit, die sich der Kategorisierung von OSS-Projekten unter diesem und anderen Gesichtspunkten widmet, findet sich unter [Mat03]. In einer weiteren Studie [uSE03] hat sich herausgestellt, dass mit zunehmender Größe eines Projektes die Ausführlichkeit der Dokumentation steigt. Im Besonderen können Datenbankprojekte eine gute Dokumentation vorweisen.

3 Software Defects

Die Programmierung von Software ist ein kreativer Prozess. Meist ist nur klar, welche Funktionalitäten das spätere Produkt aufweisen soll, nicht jedoch, wie diese zu realisieren sind. Bei OSS-Projekten arbeiten viele Autoren zusammen, um passende Lösungen zu finden. Dabei verfolgt jeder Autor aber seine ganz eigene Philosophie, so dass die Erhaltung einer vorher festgelegten Vorgehensweise nur schwer zu erreichen ist [uoD00]. Rahmenvorgaben, die den Entwicklern als Richtschnur dienen können, sind deshalb unerlässlich.

Gerade bei iterativen Entwicklungsprozessen besteht die Gefahr, dass nur die neuen Systemteile getestet werden, die Verträglichkeit innerhalb des Gesamtsystems jedoch

als gegeben erhofft wird. Dies birgt das Risiko, dass Unverträglichkeiten erst spät erkannt werden und unter Umständen unkorrigierbar sind.

Die *Open-Source-Definition* (OSD) [www05] enthält keine Verpflichtungserklärung zur Dokumentation oder Spezifikation, da dies für viele kleine Projekte ein Hindernis darstellen würde. Allerdings ist das Vorhandensein zumindest einer Spezifikation für die Bewertung eines Produktes unabdingbar.

Damit der Open-Source-Gedanke nicht verletzt wird, enthält die OSD einen Passus, der die absichtliche Konfusion des Codes *(obfuscated source code)* verbietet. Es ist sinnlos, den Programmquellcode zwar einsehen zu können, dieser jedoch durch sinnlose Variablenbezeichner unverständlich gemacht wurde. Der direkte Zugriff auf den Quellcode soll dazu verleiten, Lösungen für Probleme durch Veränderungen an der Programmlogik vorzunehmen. Dies ist keineswegs nachteilig – jedoch muss darauf geachtet werden, dass derartige Korrekturen *(patches)* nicht überhastet erfolgen. Es sollte stets eine Systemverträglichkeitsprüfung durch die Programmierer oder entsprechende Prüfer vorgenommen werden, um sicherzustellen, dass eine Korrektur nicht mehr Schaden anrichtet als sie behebt. Vorteilhaft hierbei ist die Tatsache, dass es keine engen Termine gibt und die Entwickler freiwillig arbeiten. Der fehlende Termindruck und die persönliche Motivation der Entwickler senken das Defektrisiko erheblich.

Das Sichten von Quellcode und dessen Begutachtung *(code review)* ist ein altbewährtes Prinzip der konventionellen Qualitätssicherung und im kommerziellen Umfeld häufig im Einsatz. Bei CSS ist dies nur Mitgliedern der entsprechenden Entwicklungsabteilung möglich. Wird ein Defekt von diesen Spezialisten übersehen, kann er von niemand anderem mehr direkt aufgedeckt werden [Wie04]. Vielleicht war dies auch der Ausgangspunkt und Motivator für die Entstehung der Open-Source-Philosophie. Hierbei zu beachten ist, dass die wahrgenommene Qualität eines Softwareproduktes nicht von der Defektanzahl abhängt, sondern wie schwerwiegend jeder einzelne Defekt im Produktiveinsatz ist [Bac95].

3.1 Defektklassifizierung

Die einfachste und effektivste Methode eine Klassifizierung von Defekten vorzunehmen, ist die funktionsorientierte Klassifizierung. Dabei wird jeder Defekt genau der Funktion zugeordnet, die am meisten unter ihm leidet [Org04]. Das Mozilla-Projekt beispielsweise definiert Teams, die für sechs verschiedene Funktionsfelder verantwortlich sind und sich um die Behebung der Defekte kümmern (s. Abb. 3).

3.2 Defektprävention

Eine goldene Regel bei der Softwareentwicklung ist die Feststellung, dass Software immer Defekte enthält. Dies resultiert aus der Erfahrung, dass Defekte meist sehr viel später bekannt werden. Häufig deckt erst die Praxis Defekte auf, die ein Fehlverhalten der Software verursachen. Setzt man sich als Ziel, nur eine absolut defektfreie Softwa-

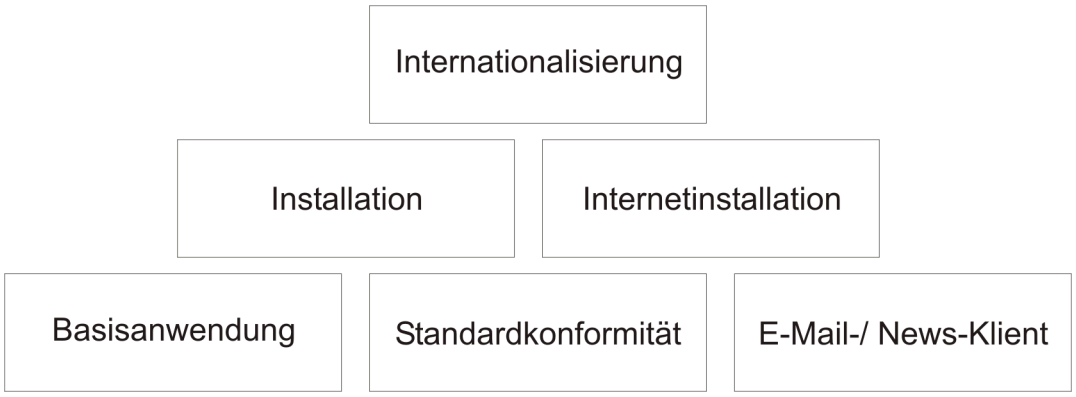

Abbildung 3: Qualitätssicherungsteams des Mozilla-Projektes

re zu veröffentlichen, ist fraglich, ob diese jemals fertiggestellt werden kann [Mie02]. Als pragmatische Lösung hat man sich damit abgefunden, bewusst *unfertige* Produkte freizugeben. Dabei erhofft man sich, Defekte durch den Praxiseinsatz der Software funktionsorientiert zu entdecken. Werden keine Fehler mehr gemeldet, so bedeutet dies lediglich, dass es keine Funktion in der Software mehr gibt, die unter einem Defekt leidet. Dieser Ansatz impliziert, dass die Qualitätssicherung frühzeitig zum Einsatz kommen muss – idealerweise bereits in der Spezifikationsphase. Wie bereits skizziert, birgt die Veränderbarkeit des Quellcodes das Risiko, dass sich Defekte erneut einschleichen können, da jeder Änderungen vornehmen darf [KK00]. Aus diesem Grund sollte man Korrekturen immer von der selben Quelle beziehen, da leider nicht davon ausgegangen werden kann, dass alle Quellen, bei denen die Software verfügbar ist, auf dem gleichen Stand sind. Zusätzlich ist es ratsam, skeptisch zu bleiben und die neue Version vor deren Einsatz ausgiebig zu testen.

Der größte Vorteil von OSS gegenüber CSS ist das *Viele-Augen-Prinzip*. Voneinander unabhängige Programmierer führen Quellcodebegutachtungen durch und können so eine Kompetenz aufbauen, die kein kommerzieller Softwareanbieter vorweisen kann. Es ist verständlich, dass nicht jeder dieser Entwickler von Beginn an das Projekt, in dem er tätig werden möchte, überschaut. Daher ist es bei OSS-Projekten erforderlich, Neulinge in die Projektarbeit einzuführen und sie nicht gleich Änderungen an kritischen Bereichen vornehmen zu lassen [Hen01].

Auch bei der Konzeptionsphase kann man das Entstehen von Defekten effektiv verhindern. Hier ist es wichtig, dass implizite Anforderungen explizit formuliert werden. Nicht jeder Projektbeteiligte hat die gleichen Vorstellungen vom fertigen Produkt. Besonders bei einer heterogenen Entwicklergemeinde, die z.B. mehrere Portierungen einer Software für zueinander inkompatible Hardware-Plattformen entwickeln möchte, müssen die besonderen Bedingungen, die damit einhergehen, berücksichtigt werden. Des weiteren lassen sich mit sehr einfachen Vorgaben an den Programmierstil positive Ergebnisse erzielen. So ist es gängige Praxis, dass Codierungsstandards *(coding conven-*

tions) und regelmäßige Quellcodebegutachtungen bereits zum Beginn eines Projektes festgelegt werden.

Ein weiterer Ansatz, der besonders bei OSS-Projekten effektiv funktioniert, ist die problemorientierte Sicht: Wird ein Defekt entdeckt, ermittelt man zunächst dessen Konsequenzen [Bac95]. Dies ist zwar ein sehr mühsamer Prozess, bringt jedoch die Gewissheit, dass die Defektbeseitigung nachhaltig ist. Auf der einen Seite impliziert die Behebung eines Defektes zwar die Behebung des Produktmangels. Dies sollte aber durch eine entsprechende Validierung bestätigt werden. Andererseits impliziert eine wohlüberlegte Gewichtung der Konsequenzen eine praxisorientierte Gewichtung der Defekte. Auf diese Weise kann man schwerwiegende Defekte leichter identifizieren und scheinbar schwerwiegende, die in der Praxis nicht relevant sind, aussortieren. Abschließend bleibt festzuhalten, dass eine Mängelbeseitigung mitnichten einer Defektbeseitigung gleichkommt. Die Praxisrelevanz spielt hier eindeutig eine übergeordnete Rolle.

3.3 Defektkompensation

Die konventionelle Herangehensweise bei Defekten hatte stets die Behebung des Produktmangels im Auge. Dabei wurde ausschließlich darauf hingearbeitet, dass das Programmverhalten korrigiert wird [Gra02]. Diese Art der Defektbehebung ist vergleichbar mit der Behandlung von Symptomen einer Krankheit. Bei OSS-Projekte wird dagegen stets die Ursache für einen Produktmangel ermittelt, da in der Regel immer ein Spezialist an genau der richtigen Stelle tätig ist. Dies könnte man so verstehen, dass der Hausarzt gleichzeitig Spezialist für jedes mögliche Fachgebiet der Medizin ist. Da CSS in der Regel entgeltlich angeboten wird, kann eine Produkthaftungsverpflichtung seitens des Herstellers bestehen, die aus dem Kaufvertrag abgeleitet wird. Zwar versuchen die meisten Hersteller durch Ausschlussklauseln dieser Verpflichtung zu entgehen, doch gelingt dies nicht immer. So wird beim Hersteller ein Mängelbericht stets als Mängelgutachten verstanden, das nachweist, dass die Mängelfreiheit des Produktes im juristischen Sinne nicht gegeben ist. Steht der Hersteller dann in der Pflicht nachzubessern, bedeutet dies zusätzliche Kosten. Fehlerberichte seitens der Benutzer sind daher oft nicht gerne gesehen und werden verheimlicht.

4 Software Faults

Fehlzustände einer Software sind nicht nur für den Benutzer unangenehm und ärgerlich, sondern stellen auch ein ernst zu nehmendes Risiko dar. Unter Sicherheitsaspekten sind derartige Fehlzustände nicht tolerierbar und sollten um jeden Preis unterbunden werden. Dies ist kein auf OSS beschränktes Problem, sondern vielmehr ein Indikator wie gründlich der Systementwurf erfolgte und ob die jeweilige Softwarearchitektur tragfähig ist. Letzterer Punkt lässt sich mit sog. Lasttests überprüfen und stellt hohe Anforderungen an das Qualitätsmanagement.

4.1 Unit Tests und Test Suites

Zu Beginn eines Testlaufes steht erneut die Quellcodebegutachtung. Auf Grundlage dieser Analyse werden Testfälle hergeleitet und mit der Spezifikation abgeglichen. Dabei ist im Besonderen darauf zu achten, dass der zu testende Systemteil auch Werte toleriert, die laut Spezifikation als ungültig deklariert wurden. Dies begründet sich in der Tatsache, dass nicht davon ausgegangen werden kann, dass der Systemteil hundertprozentig spezifikationskonform implementiert wurde. OSS-Projekte leiden zusätzlich darunter, dass nicht immer eine Spezifikation vorliegt und somit die Quellcodebegutachtung die einzige Grundlage darstellt, um Testfälle zu formulieren [Wie04]. Liegen genügend Testfälle vor und existieren passende Testdaten, so kann ein automatisierter Testlauf durchgeführt werden, der reproduzierbare Ergebnisse liefert. Zusätzlich ist eine Produktivanalyse möglich, die mit Hilfe von Prototypen oder Vorgängerversionen Eingabemuster seitens des Benutzers ermittelt hat und deren Ergebnisse in die Testläufe miteinfließen können.

Je größer ein Projekt und dessen Produkt wird, desto komplexer und aufwendiger wird auch dessen Test. So ist es wenig überraschend, dass die Bereitschaft zu ausführlichen und gewissenhaften Tests mit der Größe des Projektes abnimmt [uSE03]. Es macht daher Sinn, den Systemtest als ein eigenständiges Projekt auszugliedern und unabhängig weiterzuentwickeln. Da die Tests von der eigentlichen Softwareentwicklung nun entkoppelt sind, können auch die Benutzer diese Aufgabe übernehmen. Sie verwenden jeweils aktuelle Versionen des Produktes und ersinnen passende Testfälle, die dann das Test-Projekt ergänzen. Auch bei der Umsetzung von Testverfahren verwendet man möglichst Werkzeuge *(test suites)* der Open-Source-Szene, wobei deren Verfügbarkeit auch deren Einsatzfelder bestimmt. So existieren wesentlich mehr Test Suites für Java-basierte Projekte als für C/C++-Projekte. In diesem Bereich wird sich in naher Zukunft noch einiges tun, nicht zuletzt deshalb, weil C/C++ als Programmiersprache am häufigsten genutzt wird und somit ein enormer Bedarf besteht.

Bei Linux bestand von Anfang an das Bedürfnis, viel und ausführlich zu testen. Dies mag nicht weiter verwundern, handelt es sich doch um ein Betriebssystem von dem jede darauf laufende Software abhängig ist. Doch wird in den seltensten Fällen angegeben, welches die Testfälle waren, unter denen neuen Funktionen und Korrekturen getestet wurden [Tho03]. Dies hat zur Folge, dass die tatsächliche Korrektheit des neuen Programmquellcodes nur schwer überprüft werden kann. Ein anderes Beispiel verfolgt Testverfahren dagegen beispielhaft: Das *GNU C Compiler Project* gibt für jede Quellcodeänderung die durchgeführten Testfälle mit an, so dass die Korrektheit belegbar und nachvollziehbar wird. Das Projekt kann im Gegensatz zum Linux-Kernel-Projekt davon profitieren, dass die Programmiersprache C nach Industriestandards spezifiziert wurde und somit eine verlässliche Referenz vorliegt. Beim Linux-Kernel-Projekt sind neue Funktionen dagegen tendenziell schlecht spezifiziert, so dass eine verlässliche Validierung nahezu unmöglich ist. Ebenso finden neue Funktionen kaum den Weg in schriftlichen Vorgaben oder gar Spezifikationen. Die konsequente Reaktion einiger Entwickler war daher, sich ausschließlich auf das Testen zu konzentrieren und zu spezialisieren.

Dies hat eine neue Methodik von Testverfahren hervorgebracht: Die Entwicklung von Kleinsttestverfahren *(micro benchmarks)*, die permanent laufen und ständig die Funktion der aktuellen Quellcodes testen. Auf diese elegante Weise können Anfragen von Entwicklern binnen Stunden beantwortet werden, die bei kommerziellen Prozessen um eine Größenordnung länger brauchen würden. Die *Open Source Development Lab (OSDL)* gehen sogar noch einen Schritt weiter und entwickelten die *Scalable Test Platform (STP, http://www.osdl.org/stp/)*. Mit deren Hilfe können zukünftige Kernel-Versionen festgelegten Last- und Leistungstests unterzogen werden. Da Linux im kommerziellen Servermarkt mittlerweile eine etablierte Größe darstellt, findet auch eine zunehmende Professionalisierung der Testverfahren statt, z.B. *IBM Linux Technology Center, Linux Stabilization Project*. Um den gesamten Prozess des Testens komplett zu automatisieren, entwickelt das OSDL momentan den sog. *Patch Lifecycle Manager (PLM)* und kombiniert ihn mit der oben erwähnten STP. Damit ist es möglich Vergleichsdaten zu gewinnen, um die Entstehung neuer Fehlzustände im Kernel besser verstehen zu können.

4.2 Massentests

Der Grundgedanke hinter Massentests ist die Überlegung, dass die Benutzer – durch ihre Arbeit mit dem Produkt – eine hervorragende Ergänzung zu den vorher erfolgten Testprozessen darstellen. Dadurch gewinnt der gesamte Bereich der Produktests eine zusätzliche Praxisbezogenheit. Fehlerberichte seitens der Benutzer erlauben Korrekturen am Programmquellcode, Verbesserungsvorschläge dagegen Optimierungen an der Ergonomie – also der Benutzbarkeit des Produktes [uSvE04]. Eine Studie [uSE03] zeigte, dass mehr als die Hälfte der OSS-Projekte begründete Fehlerberichte erhielt und dass eine Rückmeldung durch die Benutzer umso schneller erfolgte, je größer das Projekt war. Offensichtlich korreliert hier die Rückmeldungsbereitschaft mit der Anzahl der Benutzer eines Produktes. Im Besonderen profitieren Internetanwendungen von solchen Massentests, da hier der Feldtest auf vielen unterschiedlichen Plattformen erfolgt. Derartige Test wären für die Entwickler enorm aufwendig und würden die Ressourcen vieler Projekte übersteigen.

Eine interessante Sichtweise erläutert [Gra02]. Dort wird die Entwickler- und Anwendergemeinde von OSS als *„kollektive Intelligenz“* bezeichnet, die nicht nur in der Lage ist, Produktmängel effektiv zu entdecken, sondern auch diese genauso effektiv zu beseitigen. Dabei steigt die Kompetenz dieser Intelligenz mit der Anzahl der Beteiligten.

5 Software Failures

Beim Zusammenbruch von Programmen entstehen prinzipbedingt die größten Schäden, da hier keine rettenden Eingriffe mehr vorgenommen werden können. In den meisten Fällen stellt ein Zusammenbruch für einen Benutzer ein Hindernis bei der Benutzung der Software dar. Oft genug verliert er aber auch wichtige Daten oder erleidet andere

Arten von wirtschaftlichen Schäden. Besonders im professionellen Einsatz können so Schäden in beträchtlicher Höhe entstehen. Um diesen Umstand zu entschärfen müssen Softwarearchitekturen verwendet werden, die diesen Gefahren Rechnung tragen und den Verlust von Daten oder die Entstehung von Schäden auf ein Minimum reduzieren.

5.1 Plug-in-Architektur

Als Beispiel sei hier das OSS-Projekt *Eclipse* genannt, welches die Grundlage für Viele Anwendungen darstellt. Die Architektur von Eclipse geht vom sog. *„Kamälion-Prinzip"* aus, welches sich den Gegebenheiten anpasst [Dau03]. Ohne jedwede Ergänzung erfüllt Eclipse so gut wie keinen Zweck. Das Potential offenbart sich erst durch die Verwendung von Erweiterungen *(plug-ins)*. So wird Eclipse mit Hilfe dieser Module exakt dem Anwendungszweck angepasst und stellt die optimale Lösung dar. Die starke Kapselung der funktionalen Teile macht sie austauschbar und Eclipse nahezu resistent bei Fehlfunktionen in einem der Module. Betrachtet man Eclipse etwas distanzierter so kann man beobachten, dass sich das OSS-Prinzip im offenen Architekturkonzept widerspiegelt. Auch Eclipse ist dezentral konzipiert und organisiert, die funktionalen Teile können unabhängig voneinander entwickelt und beliebig kombiniert werden [Sau04]. Gerade in der Softwareentwicklung ist dies eine gern gesehene Eigenschaft, da dort ein ganzer Anwendungszoo notwendig ist, um alle Stationen des Entwicklungsprozesses umzusetzen. Verwendet man ausschließlich CSS entsteht eine Abhängigkeit von vielen Herstellern, die ein hohes Maß an Verwaltungsaufwand und Investitionskosten generiert. Dagegen ist die optimalste Lösung für OSS-Projekte wiederum OSS: Die Lösung des Problems wird aus vielen Teillösungen zusammengestellt *(top-down)* und bedarfsgerecht zusammengefasst. So stellt Eclipse stets eine ideale Entwicklungsplattform für OSS dar. Die kontinuierliche Entwicklung von neuen Erweiterungen wird diesen Effekt in Zukunft sicherlich noch verstärken.

5.2 Szenarienspezifische Anpassungen

Wie bereits erwähnt bietet OSS eine gute Möglichkeit sich Anwendungssoftware maßzuschneidern. Sogar wenn sich ein OSS-Produkt bereits im Produktiveinsatz befindet und sich die Bedingungen leicht ändern, ist eine Anpassung schnell erfolgt. Entweder indem man selbst Hand anlegt oder indem man die Gemeinschaft um Hilfe bittet [uoD00]. Bereits bei der Auswahl von OSS-Produkten steht eine breite und weiter wachsende Auswahl zur Verfügung. Dabei induziert die Einsatzumgebung die Adaptierung und Selektion von Produkten, so dass die eigentliche Herausforderung darin besteht, zu wissen was man will.

6 Produktverantwortung

Die Verantwortlichkeit für ein Produkt aus dem OSS-Bereich ist nicht zu verwechseln mit der legaldefinierten Produkthaftung der nationalen Rechtsprechung. Die Mitglieder eines OSS-Projektes *fühlen* sich für ihr Produkt verantwortlich. Haftbar gemacht werden, können sie dagegen nicht. Durch die Möglichkeit der tiefgreifenden Produktanalyse durch unabhängige Dritte, stellt der Verlust eines Rechtsanspruches jedoch kein Problem für den Anwender dar [uoD00]. Er kann bei einer entsprechenden Wahl einer Vertrauensperson oder -stelle davon ausgehen, dass der Programmquellcode *sauber* ist. Ein größeres Problem könnte dagegen die nicht garantierte zeitnahe Behebung eines Produktmangels sein. Ferner birgt die Offenlegung des Quellcodes auch sicherheitsrelevante Risiken: So kann die Latenz zwischen der Entdeckung eines Defektes und dessen Behebung geschickt dazu benutzt werden, Schäden anzurichten oder Spionage zu betreiben. Doch wenn man diesen Gedankengang weiterverfolgt, wird schnell ersichtlich, dass sich CSS für den Einsatz in kritischen Bereichen durch ihre erheblich höhere Latenz eher disqualifiziert. Gemäß der OSD ist zwar festgelegt, welchen Charakter ein OSS-Projekt haben muss, durch die geographische Verteilung der Beteiligten ist eine Sanktionierung bei Verstößen jedoch unmöglich. Ähnlich verhält es sich mit der Berücksichtigung von nationaler Gesetzgebung bei OSS-Projekten. Aus diesen Freiheiten erwächst für den Benutzer eine besondere Verantwortung zum Selbstschutz. Der regelmäßige Besuch der Internetseiten von für den Anwender sicherheitsrelevanten Projekten sollte daher zur Selbstverständlichkeit werden. Eigenverantwortlichkeit im Umgang mit digitaler Datentechnik wird in Zukunft ohnehin immer wichtiger werden.

Projektteams der verwendeten Produkte sollten stets bevorzugte Ansprechpartner sein, wenn es um Probleme mit der Software geht. Meist leitet sich die Verantwortlichkeit aus der Projektstruktur selbst ab *(generische Produktverantwortung)*, da sich die jeweiligen Autoren in der Verantwortung für *ihren* Quellcode sehen [uSvE04]. Sie bieten freiwillig und meist zeitnah Lösungsbeschreibungen *(work-around, patch-work)* an, die in den entsprechenden Datenbanken zu einem Projekt abgerufen werden können. Mit dem Anwachsen der Komplexität eines OSS-Produktes steigt die Problematik der Aufgabenverteilung. Meist wird versucht, das Projekt immer weiter zu zergliedern, damit die einzelnen Produktteile unabhängig voneinander entwickelt werden können. Dies ist jedoch riskant, da nicht immer sichergestellt werden kann, dass sich Freiwillige für einen Produktteil finden, die diesen weiterentwickeln und dessen Fehler korrigieren. Des Weiteren steigen die Reibungsverluste durch Kommunikation und die Gefahr der Teamzerplitterung.

Damit OSS professionellen Ansprüchen genügen kann, muss eine konstante Produktunterstützung und kontinuierliche Weiterentwicklung des Produktes gewährleistet sein. Da dies in den seltensten Fällen direkt von den Projekten geleistet werden kann, haben sich Firmen spezialisiert, die erstere Leistung entgeltlich anbieten, sog. *Distributoren*. Die zweite Voraussetzung kann von großen Projekten derart gelöst werden, dass sie altbekannte hierarchische Strukturen bilden [Kru04] wie beispielsweise die *Apache*

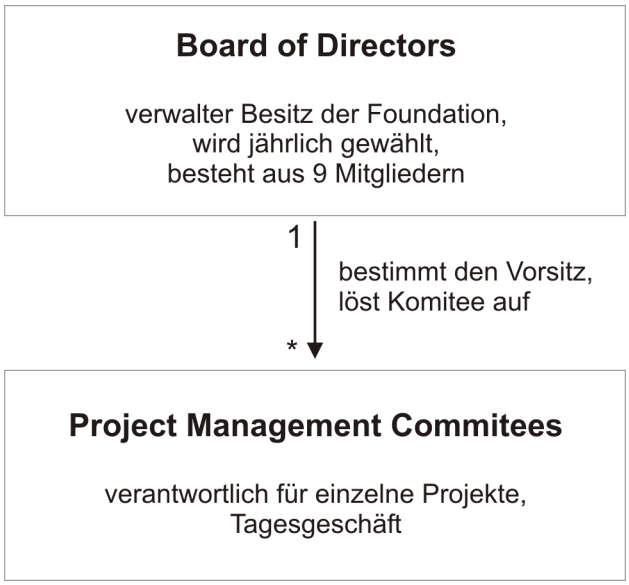

Abbildung 4: Verwaltung der Apache Software Foundation

Software Foundation (ASF), s. Abb. 4.

In [Gan03] argumentiert der Autor, dass Programmierfehler sich bei OSS leichter einschleichen können als bei der kommerziellen Konkurrenz. Dem muss jedoch widersprochen werden. Es ist zwar korrekt, dass jeder im Programmquellcode Änderungen vornehmen darf, doch erfolgen diese unter ständiger Überwachung durch die Gemeinschaft (s. Seite 10). Bei CSS existiert diese Überwachung nicht, da dieses Prinzip die Überprüfung durch unabhängige Dritte unterbindet. Ferner muss berücksichtigt werden, dass die Zahl der eingeteilten Programmierer im Vergleich zu OSS bei CSS verhältnismäßig gering ist. Vorher festgelegte Kontrollen sind dagegen wirkungslos. Sie können bei Personalknappheit leichter unterlaufen werden. Genau wie bei den meisten CSS-Projekten werden auch bei OSS Änderungen protokolliert und können jederzeit einem Autor zugeordnet werden. Dies geht sogar soweit, dass ein Autor von einem Projekt ausgeschlossen werden kann, wenn er dem Projekt nachweislich geschadet hat. Hingegen sind veröffentlichte Produktversionen bei CSS gegen jede Änderung verschlossen worden und können ohne authorisierten Zugriff auf die Quellcodes durch niemanden mehr geprüft werden. Dass dies eine trügerische Sicherheit vorspiegelt, ist nicht zuletzt durch das Knacken von unter Verschluss gehaltenen kryptographischen Algorithmen belegt worden. Gerade im Sicherheitsbereich ist die Verfolgung des CSS-Prinzips grob fahrlässig und wird in der Zukunft noch viele Schäden verursachen.

6.1 Das Team-Modell

Möchte man selbst ein OSS-Projekt angehen, so stellt sich die Frage nach der Aufgabenverteilung. In der Open-Source-Szene haben sich zwei grundweg verschiedene Ansätze etabliert, die sich auch miteinander kombinieren lassen. Das *Team-Modell* zeichnet sich dadurch aus, dass geschlossene Entwicklerteams nach formalen Prozessen an einem Projekt arbeiten. Externe Entwickler unterstützen diese mit sporadischen Beiträgen [uSvE04]. Die Einreichung solcher Beiträge erfolgt nach einem definierten Verfahren, welches diese auf Qualität, Konsistenz und Fehlerfreiheit prüft. Jedes Team arbeitet an einem genau definierten Teil des Produktes, so dass stets ein einzelner oder eine ganze Gruppe als Verantwortlicher herangezogen werden kann. Der Trend geht dabei ebenfalls in Richtung Gemeinschaftsmodell. Bei neueren Projekten ist es üblich, die Namen eines externen Autors mit in die Protokolle aufzunehmen, um jederzeit nachvollziehen zu können, welche Beiträge dieser für das Projekt geleistet hat [Gra02].

6.2 Das Verdienst-Modell

Das Verdienst- oder auch Reputations-Modell setzt auf die freiwillige Teilnahme von Entwicklern an der Weiterentwicklung oder Fehlerbereinigung eines Produktes. Dabei geht es ausschließlich um das Gewinnen von Ansehen in der Entwicklergemeinde *(community)* eines Projektes. Nicht zu letzt deshalb, wird es auch von professionell tätigen Entwicklern als Plattform benutzt, um die eigene Kompetenz zu präsentieren und die eigenen Berufschancen zu verbessern [uSvE04]. Da eine Organisationsstruktur weitgehend fehlt, können auch die Benutzer zum Teil der Entwicklergemeinde werden. Sie erstellen aufgrund ihrer eigenen Erfahrungen im Umgang mit dem Produkt geeignete Produktdokumentationen und nehmen aktiv an Testläufen neuer Varianten des Produktes teil. So gewinnen nicht nur die Autoren der Programmquellcodes das Gefühl, ein Teil des Projektes zu sein, sondern auch die Benutzer.

Auch die ASF arbeitet nach diesem Prinzip einer Leistungsgesellschaft [Kru04]. Besonders ambitionierte Entwickler erarbeiten sich Ansehen durch konstruktive Vorschläge und Korrekturen für das Produkt. Dabei ist hervorzuheben, dass der jeweils erworbene Verdienst mit der Gewährung von Privilegien gekoppelt ist. Je größer der Beitrag eines Entwicklers im Projekt, desto mehr Rechte erhält er. Ab einem bestimmten Punkt kann er selbst Einfluss auf die Entwicklungsrichtung des Produktes nehmen. Entsprechend hat sich mit der Zeit eine Hierarchie auf Grundlage von Privilegien herausgebildet (s. Abb. 5).

Bei strittigen Themen wird nach dem demokratischen Mehrheitsprinzip abgestimmt, wobei höher privilegierte ein Vetorecht besitzen. Durch die relativ flache Hierarchie können derartige Streitfragen schnell geklärt werden. Bei einer Abstimmung hat jeder Stimmberechtigte die Wahl zwischen *Ablehnung* (-1), *keine Meinung* (0) oder *Zustimmung* ($+1$).

Generell kann man an dieser Stelle festhalten, dass Projekte meist von altgedienten oder besonders aktiven Mitgliedern geleitet werden [Gra02]. Da in aller Regel das Pro-

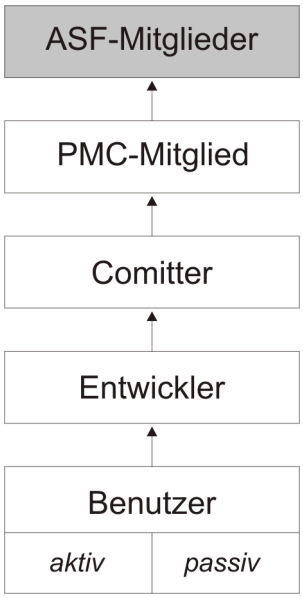

Abbildung 5: Privilegienhierarchie der ASF

dukt mit der Größe des OSS-Projektes wächst, bilden sich ab einem bestimmten Zeitpunkt sog. Kerngruppen *(core teams)* heraus, die für die Quellcodeintegration verantwortlich sind. Allein schon die Menge an Beiträgen der externen Entwickler macht dies erforderlich.

6.3 Das Hybrid-Modell

Die Kombination der beiden Organisationsmodelle wird als Hybrid verstanden. Zu beobachten ist diese Verschmelzung bei extrem großen OSS-Projekten [uSvE04]. Hier liefern ganze Teams sporadische Beiträge an ein Unterprojekt. Dabei spielen die OSS-Distributoren eine besondere Rolle, da sie Erfahrungen aus Vertrieb und Produktunterstützung mit einbringen können. Eine Ausnahme bildet hierbei Linux: Es existiert kein offizielles Entwicklerteam, sondern sechs Vertrauensleute von *Linus Torvalds* [Gra02]. Sie sammeln alle Korrekturen und Vorschläge von freiwilligen Entwicklern und filtern diese. Als Schöpfer von Linux trifft Linus Torvalds dann die letzte Entscheidung über deren Verwendung und Integration.

7 Produktzertifizierung

Soll OSS im professionellen Umfeld eingesetzt werden, muss die Möglichkeit der Zertifizierung und Verifizierung bestehen. Da dies bei OSS nicht immer sichergestellt werden kann, ist eine Suche nach entsprechenden Produkten teilweise recht aufwendig

[uoD00]. Dennoch liegen die Vorteile von OSS auf der Hand. Man kann sich selbst von der Korrektheit des Programmquellcodes überzeugen und virulenten Code oder Hintertüren *(backdoors)* aufspüren. Grundsätzlich bietet die OSD keine Garantie für die Übereinstimmung von Quell- und ausführbaren Binärcode, so dass diese Schwachstelle böswillig ausgenutzt werden kann *(replacement attacks)*. Möchte man dagegen eine *echte* Zertifizierung im Sinne von Sicherheit nach innen *(security)* und Schutz nach außen *(safety)*, müssen alle Elemente des Systems zertifiziert sein. Dies beginnt bei der zu verwendenden Anwendungssoftware und endet bei der gewählten Hardware auf der die Software ausgeführt werden soll. Diese rekursive Zertifizierung konnte bisher nur im militärischen Sektor durchgeführt werden, da dort Auftraggeber und Benutzer dieselbe Einrichtung waren. OSS ermöglicht diese absolute Sicherheit erstmalig auch im zivilen Bereich. Dabei kann die OSS-Philosophie als konsequente Fortsetzung der kryptographischen Sichtweise verstanden werden. Dort gilt die Prämisse, dass jeder nicht offengelegte Verschlüsselungsalgorithmus unsicher ist. Wenn nun ein sicherheitskritisches System auf dem aktuellen Stand gehalten werden soll, dann erzwingt dies, dass jede Korrektur mit einer digitalen Signatur versehen werden muss, die die Herkunft der Korrektur belegt. Folgerichtig lässt sich auch die Entwicklungsgeschichte eines Produktes lückenlos festhalten.

Bisherige Methoden der Zertifizierung waren sehr teuer und aufwendig (bzgl. des Aufwandes s. [Gru02] über die Erteilung des intern. anerkannten *Deutschen IT-Sicherheitszertifikates*). Daher wurden nur stabile Revisionen *(z.B. stable o. released)* eines Produktes zertifiziert, die einen Versionswechsel auf oberster Ebene *major release* vollzogen hatten. Es ist auch heutzutage noch erforderlich, dass die Versionswechsel mit deren Zertifizierung synchronisiert werden müssen. Für Privatanwender ist zertifizierte Software sicherlich wünschenswert, doch war dies in der Vergangenheit schlichtweg unbezahlbar. Um diese Lücke zu schließen, übernehmen die Distributoren von OSS zunehmend diese Aufgabe. Sie authentifizieren die zusammengestellten Programmquellcodes und zertifizieren deren Kompilate. Auf diese Weise kann der Anwender auf die Herkunftsgarantie der verwendeten OSS vertrauen insofern er dem Distributor vertraut.

Auch bei der Zertifizierung hilft eine Erhöhung der Abstraktion der Softwareentwicklungsprozesse. So könnte man mit einem entsprechend geeigneten Prozess OSS zertifizieren, ganz gleich auf welche Weise diese entstanden ist. Derartige Prozesse sind z.B. der *ISO 9000* für Hersteller, der *ITSEC* (europäisch) oder der *CC* (international) (s. Abb. 6).

Die Voraussetzungen einer Zertifizierung sind dabei die Qualitätssicherung beim Hersteller und die Verfügbarmachung der Programmquellcodes. Beides stellt bei OSS prinzipiell kein Problem dar, wenn von vornherein das Projekt auf eine Zertifizierung ausgerichtet wurde. Die Zertifizierungsstelle prüft die spezifizierte Funktionalität, die Qualität der Umsetzung, die Vertrauenswürdigkeit der Projektbeteiligten und die Einhaltung von Sicherheitsstandards. Letzteres kann dabei von der beauftragten Zertifizierungsstelle und nationaler Gesetzgebung anhängen.

Die hundertprozentige Sicherstellung der Vertrauenswürdigkeit von OSS-Produkten

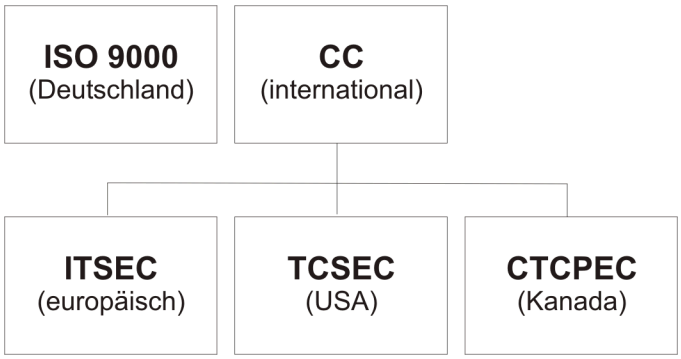

Abbildung 6: Hierarchie der Zertifizierungsstandards

bedeutet auch weiterhin einiges an Aufwand. So muss nicht nur das Produkt selbst geprüft werden, sondern auch alle Werkzeuge, die zu dessen Entstehung beigetragen haben [KK00]. Theoretisch wäre es möglich, dass zertifizierter Programmquellcode mit einem manipulierten Übersetzer *(compiler)* in Maschinencode überführt wurde. In solch einem Fall ist mitnichten davon auszugehen, dass die Software noch vertrauenswürdig ist. Das Zertifikat geht bei einer solchen Überführung verloren. Bei kryptographischen Algorithmen besteht jeher eine Nachweispflicht über deren Korrektheit. Keine Bankgesellschaft würde das Risiko eingehen, unzertifizierte Algorithmen einzusetzen. Daher stellt auch das sehr beliebte Online-Banking im privatem Bereich ein Sicherheitsrisiko dar, da kaum ein Anwender auf Sicherheitszertifikate achtet. Damit die Vertrauenswürdigkeit von OSS am Ende erhalten bleibt, muss die Zertifizierung an jeder Stelle im Zertifizierungsprozess gegengeprüft werden, um Manipulationen auszuschließen. Man verfolgt derzeit sogar den Gedanken der Open Hardware, der die Prinzipien von OSS auf die Entwicklung und Herstellung von technischen Bauteilen überträgt.

Robuste und sichere Produkte müssen auf einer sicheren Basis aufbauen, die CSS prinzipbedingt nicht bieten kann. Ein Versuch, sichere Produkte auf einer unsicheren Grundlage zu entwickeln, ist kompliziert und aufwendig. Ein gewisses Restrisiko wird immer bestehen [Neu00]. Dennoch gibt es derartige Ansätze, die darauf abzielen, entweder den sicheren Teil zu kapseln oder aber den unsicheren. Dies geschieht mit sog. Umhüllungsverfahren *(wrapper technology)* und stellt eher eine Notlösung als ein tragendes Konzept dar. Im Juni 2001 erfolgte die Standardisierung von Linux durch die *Free Software Group* und die Freigabe einer entsprechenden Test Suite *(Linux Base Certification Suite, http://sourceforge.net/projects/lsb)*. Dies ermöglichte es, Linux-Varianten einfach und schnell zu zertifizieren, so dass im April 2003 bereits 18 Zertifikate ausgestellt waren [Ere03].

8 Weiterentwicklung von OSS

Eine der größten Risiken, die man bei CSS eingeht, ist die Abhängigkeit von proprietären Formaten bei der Datenspeicherung. In den seltensten Fällen können sich konkurrierende Hersteller von CSS auf einen Standard einigen [Gra02]. So kann es passieren, dass ein Hersteller vom Markt verschwindet und dessen Produkt nicht mehr weiterentwickelt wird. Bei einem Wechsel der technischen Ausstattung entsteht dann das Problem, dass die Veränderbarkeit und Lesbarkeit von archivierten Daten unmöglich wird. Den einzigen Ausweg bei dieser Problematik stellen offene Standards dar. Eine goldene Regel bei der Archivierung von digitalen Daten ist es, möglichst weit verbreitete und offene Standards zu verwenden. Gerade bei OSS kann man konstatieren, dass mehrheitlich offene Standards verwendet werden (z.B. *XML, PostScript, OpenPGP, MP3* etc.) [uSvE04]. Diese positive Eigenschaft ist besonders dann hilfreich, wenn man Systemteile austauschen möchte. In der Regel ist ein OSS-Produkt durch ein anderes ersetzbar, doch sollte dies bereits vorher bestätigt werden.

Gerade wenn OSS-Produkte in vielen Bereichen zu finden sind, liegt eine besondere Verantwortung bei den Autoren. Sie sollten sicherstellen, dass ihre Produkte kontinuierlich weiterentwickelt werden oder zumindest für Ersatzlösungen sorgen. Die Professionalisierung von OSS-Projekten erfordert in zunehmenden Maße, dass Qualität und Ergonomie der Produkte in den Vordergrund rücken. Eine solche Anforderung gewinnt mit der Größe des Projektes an Priorität. Besonders wenn man das eigene Produkt von der Masse abheben möchte, sind ausgefeilte Produkttests ein gutes Mittel und ein hilfreicher Qualitätsindikator. Steigt die Anzahl der gefundenen Fehler oder nimmt die durchschnittliche Fehlerschwere zu, sollte die Weiterentwicklung eingefroren werden und eventuell über die Abspaltung eines neuen Versionszweiges nachgedacht werden [Wie04].

Die Entwicklung neuer Funktionen in einem OSS-Produkt beginnt meist mit einem gesteigerten Bedarf. So kann man durchaus zu dem Schluss kommen, dass sich das Produkt durch Benutzung weiterentwickelt [Jek04]. Auf Grundlage dieses Gedankens lässt sich von einem *Krieg der Projekte* sprechen, die sich gegenseitig überbieten. Wie in der freien Wildbahn wird nur das geeignetste bestehen und weiterentwickelt werden. Da die Eignung eines OSS-Produktes direkt aus dessen vierteiliger Qualität (s. Seite 2) resultiert, stellt die Qualitätssicherung einen überlebenswichtigen Faktor dar. Eine ungewohnte aber häufiger anzutreffende Eigenschaft von OSS-Projekten ist die Möglichkeit der Wiederaufnahme der Entwicklung nach dem diese eingestellt wurde. Dies ist stets möglich und im Sinne einer ersetzenden Übernahme durch einen neuen Betreuer zu verstehen [Koc01].

Durch die intensive Weiterentwicklung und Rückkopplung durch die Benutzer nähern sich OSS-Projekte erheblich schneller dem *Qualitätsoptimum*. Dieses wird erreicht, wenn keine wahrnehmbaren Korrekturen mehr vorgenommen werden müssen [Bac95]. Zu beachten ist aber auch hier, dass dies keinesfalls mit der Mängelfreiheit des Produktes gleichzusetzen ist. Im Gegensatz zu CSS gibt es hier den Zustand *„gut genug“* nicht, da OSS solange verbessert wird, wie sich Defekte im Programmquellcode finden

Abbildung 7: Die Grundpfeiler eines OSS-Projektes

lassen – auch wenn diese keine Praxisrelevanz besitzen. Neben der Gewichtung von Defekten nach reinen Qualitätsgesichtspunkten sollte auch eine Risikobewertung der Defekte erfolgen *(risk management)*. Da OSS häufigen Änderungen unterliegt und ihre Einsatzfelder sich ändern können, stellt die Risikoabschätzung eine Art Prognose dar, um Fehlverhalten der Software in der Zukunft unwahrscheinlicher zu machen.

8.1 Versionsverwaltung

Mag es ein großer Vorteil sein, dass viele Entwickler in OSS-Projekte gleichzeitig tätig sind, so bedeutet dies jedoch einigen Aufwand bei der Versionspflege. Die Lösung dieses Problems besteht darin, mit Quellcodearchiven *(repository)* zu arbeiten, die automatisch von Open-Source-Werkzeugen verwaltet werden. Als Beispiele seien hier CVS, Subversion und BitKeeper (Linux) genannt [KK00]. Um die Gemeinschaft über lokale Änderungen auf dem Laufenden zu halten, werden die bereits bekannten Nachrichtenverteiler genutzt (s. Abb. 7).

Zusätzlich muss jede der Änderungen mit einem Kommentar versehen werden, der den Grund und die Änderung selbst beschreibt. Da die Nachrichtenverteiler mit nur einer einzigen Eingangs-E-Mail alle registrierten Benutzer informieren, genügt es, bei der Hinterlegung einer Änderung im Quellcodearchiv, eine E-Mail mit dem Änderungskommentar an den Verteiler zu schicken. Wollen neue Entwickler in ein Projekt einsteigen, so müssen diese sich erst bewähren. Sie schicken dazu Verbesserungsvorschläge als Korrekturen an die zuständigen Entwickler. Die Vorschläge dienen als Beleg für die Fachkompetenz, so dass ein Neueinsteiger bei ausreichenden Fachkenntnissen einen Zugang mit Schreibrechten zum Quellcodearchiv erhält.

Da OSS-Projekte ohne Terminvorgaben auskommen, wird das Produkt erst nach Erreichen des gewünschten Reifegrades freigegeben. Die Versionsnummern implizieren dabei den Grad der Änderungen. OSS ist nicht als ein Produkt im herkömmlichen Sinn zu verstehen, sondern vielmehr als ein kontinuierlicher Prozess, dem kein Ende gesetzt ist [Gra02]. Futuristen verstehen ein OSS-Produkt vielmehr als künstliche Lebensform, die sich selbstständig entwickelt und kontinuierlich Nachkommen hervorbringt. Die bereits erwähnte kollektive Intelligenz (s. Seite 13) wird dabei als der Schöpfer oder Züchter angesehen. Erreicht das Produkt die Vorgaben eines Meilensteines wird der

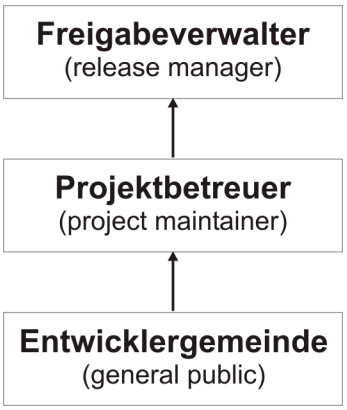

Abbildung 8: Entscheidungshierarchie bei der Produktfreigabe

aktuelle Versionszweig eingefroren und die abschließenden Tests eingeleitet. Wurden diese erfolgreich absolviert, erfolgt die offizielle Veröffentlichung des Produkte als sog. Freigabe *(release)*.

8.1.1 Freigabeautorität

Für die Freigaben sind sog. Freigabeverwalter *(release manager)* verantwortlich, die die Freigabe koordinieren und verantworten. Bei Fragen der Integration der Quellcodes zum Endprodukt sind sie die letzte Instanz, die darüber entscheidet, ob Korrekturen noch vorgenommen werden können oder nicht [Ere03] (s. Abb. 8).

Natürlich stellt sich in einem Projekt die Frage, wer die verantwortungsvolle Aufgabe des Freigabeverwalters übernehmen soll. Die Vorgehensweise zu dessen Bestimmung ist von Projekt zu Projekt unterschiedlich. Sowohl Ernennung, Wahl und Rotation sind gängige Methoden.

8.1.2 Entwicklungsstadien

Wie bereits weiter oben skizziert, spiegeln die Produktversionsnummern das Entwicklungsstadium wieder. Zusätzlich können aber auch eine eindeutige Identifizierung des Produktes erfolgen und Aussagen über die konkreten Produkteigenschaften wie Stabilität, Funktionsumfang, Alter etc. gewonnen werden [Ere03]. Die massiv-parallele Entwicklung bei OSS-Projekten erfordert oft zeitlich parallel laufende Versionszweige *(branches)*, so dass es oft schwierig ist, Abhängigkeiten und Verwandtschaften einzelner Versionen zueinander allein aus den Versionsnummern abzuleiten. Um die baumförmige Struktur der Versionszweige besser zu gliedern und handhabbar zu machen, werden Meilensteine bezüglich der Stabilität festgelegt. Diese werden als *Alpha-Freigaben*, *Beta-Freigaben* und *Freigabekandidaten (release candidates)* bezeichnet, wobei die Leistungsbeschreibungen der einzelnen Meilensteine auch hier vom Projekt abhängen. Freigabe-

kandidaten dienen dazu, den Benutzer in die Abschlusstests einzubeziehen, mit dem Ziel, die Qualität der letztlichen Freigaben zu erhöhen. Dabei übernehmen die Benutzer die Validierung und Teams, die aus der Entwicklergemeinde gestellt werden, die Verifikation. Eine weitere Maßnahme zur Sicherstellung der Freigabenqualität können Kontrollgremien, Freigabegremien und Kontrollausschüsse sein, die sich bei mittleren bis großen Projekten als hilfreich erwiesen haben.

8.1.3 Verteilung von Freigaben

Wichtig bei der Freigabe einer neuen Produktversion ist Werbung bei der Benutzergemeinschaft *(user community)*. Sie muss über die Verfügbarkeit und die Vorteile einer neuen Freigabe informiert werden [Ere03]. Je besser dies gelingt, desto größer ist das Interesse an nachfolgenden Freigaben. Keinesfalls darf es passieren, dass die Integrität einer Freigabe auf dem Weg zum realen Einsatz beim Benutzer verloren geht. Es muss stets die Unverfälschtheit auf dem gesamten Verbreitungsweg *(delivery path)* sichergestellt sein, z.B. durch den Einsatz von Prüfsummen und geeigneten Zertifikaten. Die Verteilung der Freigaben *(distribution)* kann auch über Dritte erfolgen und zu kompletten Metaprodukten gebündelt werden wie dies bei Linux-Distributionen oder *MikTex* geschieht. Aufgabe des Distributors ist es, die Bündelung von OSS *(packaging)* in verschiedenen Formaten vorzunehmen, z.B. RPM-Pakete für Linux oder Installationsprogramme für Windows.

8.1.4 Versionsnummernvergabe am Beispiel des Linux-Kernel-Projektes

Das Linux-Kernel-Projekt verwendet für Versionsnummern das Format *x.y.z*, wobei die Platzhalter folgende Funktion haben:

- *x* entspricht Freigaben mit vielen und tiefgreifenden Änderungen,
- *y* entspricht Freigaben mit einigen kleinen Änderungen und
- *z* kennzeichnet den Korrekturzustand *(patch level)* der Freigabe.

Stabil lauffende Freigaben weisen als Besonderheit ein gerades *y* auf, instabile dagegen ein ungerades. Inoffizielle Freigaben erhalten zusätzlich die Initialen des Autors der Freigabe und Freigabekandidaten das Suffix `-pre`. Die instabilen Freigaben sind nicht für den Produktiveinsatz gedacht, sondern dienen Herstellern von Software und anderen OSS-Projekten als Testplattform und Technologiestudie.

8.2 Sicherheitsaspekte

Die Quellcodetransparenz bei OSS ermöglicht stark beschleunigte Revisionszyklen und Korrekturbereitstellungen im Vergleich zu CSS. Im Idealfall steht eine Korrektur zeitgleich mit dem entsprechenden Fehlerbericht zur Verfügung [KK00]. Gerade die Ana-

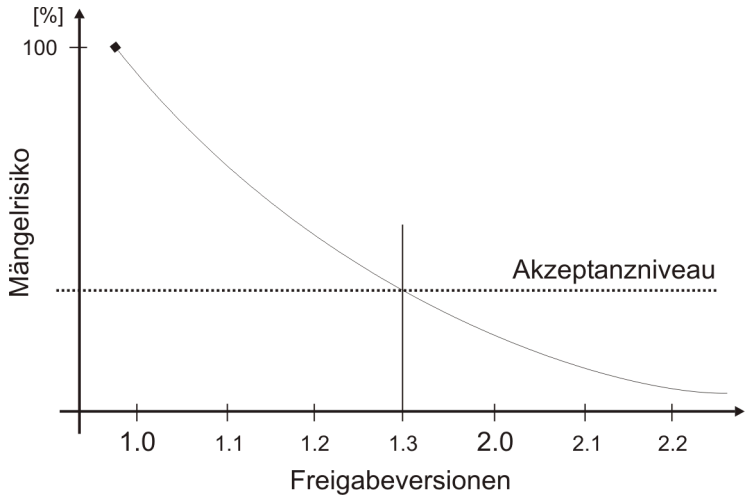

Abbildung 9: Mängelrisikoabschätzung

logie der Philosophien von OSS und Kryptographie bezüglich der Sichtbarkeit der Algorithmik, erweckt den Eindruck, dass die Kryptographie der OSD Pate stand. Vertrauen entsteht meist durch Überzeugung. So kann die Wiederverwendung von (zertifiziertem) Programmcode vertrauensbildend wirken, doch sollte stets angestrebt werden, das eigene Produkt im Ganzen zertifizieren zu lassen (z.B. Zertifizierung des *SuSE Linux Enterprise Server V8* durch das BSI [fSidI03]), da eine Übertragbarkeit oder Vererbung von Zertifikaten nicht möglich ist (s. Abschnitt 7). Die OSD schafft die besten Voraussetzungen für sicheren Programmquellcode, liefert jedoch keine garantierte Sicherheit. Es existiert im Sicherheitskontext kein Qualitätsoptimum, sondern es gilt vielmehr ein Dualismus der sicherheitsbezogenen Qualitätseigenschaften eines OSS-Produktes – entweder es ist sicher oder nicht [Bac95].

Bei sicherheitsrelevanten Produkten muss empirisch eine Gewichtungsfunktion der Risiken ermittelt werden, die einem Aussagen über den *tolerierbaren* Reifegrad des Produktes erlaubt (s. Abb. 9), d.h. ab welcher Revision das Produkt in Betrieb genommen werden darf. Dabei gibt es zwei Extrema zwischen denen sich das zu bewertende Risiko quantitativ bewegt: 0% entspricht der formal bewiesenen Korrektheit und Belegbarkeit der Fehlerfreiheit, 100% der Sicherheit, dass der untersuchte Produktabschnitt mängelbehaftet ist. Die Zertifizierung des Gesamtproduktes sollte daher erst erfolgen, wenn alle kritischen Teile das *Akzeptanzniveau* erreicht haben. Darunter ist die Bereitschaft des Benutzers oder Kunden zuverstehen, Risiken bis zu einem bestimmten Grad zu akzeptieren. Das Niveau kann über eine Benutzer- bzw. Kundenbefragung quantitativ ermittelt werden.

Zum Thema Vertrauenswürdigkeit von Software bleibt daher festzuhalten: Vertrauen in ein Softwareprodukt erfordert die Einsehbarkeit des Programmquellcodes, substanzielle und effektive Sicherheit die Anpassbarkeit des Quellcodes an die konkreten Bedingungen. Da CSS beides nicht bieten kann, ist sie grundsätzlich als vertrauen-

sunwürdig abzulehnen [Gra02]. Das Problem mit dem OSS zu kämpfen hat, ist dagegen ganz anderer Natur. Sie ist neu und somit für viele Benutzer befremdlich, Vertrauen in sie muss langsam durch eine Eingewöhnungsphase wachsen. Dagegen genießt die etablierte CSS ungerechtfertigtes Vertrauen allein auf der Grundlage, dass sie bekannt und akzeptiert ist. Diese Einstellung ist nicht nur riskant, sondern kann auch ernsthafte Schäden verursachen.

9 Zusammenfassung und Ausblick

Diese Arbeit unternahm den Versuch, herauszustellen, wie schwer der Begriff *Qualität* zu erfassen und zu bewerten ist. Auch OSS-Projekte müssen sich der Problematik der Qualitätssicherung stellen. Nicht zuletzt deshalb, weil OSS im professionellen Sektor nicht mehr wegzudenken ist. Eine effektive Qualitätssicherung setzt eine durchdachte Prioritätensetzung voraus [Bac95]. So kann man Qualitätssicherung als die Kunst verstehen, gute Kompromisse zu finden. Dass dies in der Politik genauso ist, kann hier als Beleg verstanden werden. Auch dort müssen Prioritäten gesetzt werden.

Absolute Qualität existiert nicht. Das einzige bekannte Mittel sie zu erreichen, ist die formale Verifikation von Software, die bereits bei kleinen OSS-Projekten unpraktikable Ausmaße annimmt. Es bleibt nur der Versuch, hoher Qualität möglichst nahe zu kommen. In den meisten Fällen zeigt die Praxis, ob ein Produkt bereits eine akzeptable Qualität erreicht hat. Dies sollte bei keinem OSS-Projekt aus den Augen verloren werden.

Der Trend bei Open Source geht in Richtung Professionalisierung. Die enormen Einsparpotentiale bei dieser Art der Softwareentwicklung sind aus kommerzieller Sicht verlockend. Mittelfristig bis langfristig steht die Ausweitung des Open-Source-Gedankens auf die Hardwareentwicklung an, um eine lückenlose Zertifizierung von kompletten *Open Systems* zu ermöglichen (erste Ansätze für ein FreeBIOS s. [Sta05]). Kurzfristig die Verbesserung der Dokumentation und Ergonomie von OSS.

Literatur

[Bac95] James Bach. The challenge of good enough software. *American Programmer magazine*, 1995.

[Dau03] Berthold Daum. *Java-Entwicklung mit Eclipse 2*. dpunkt.verlag GmbH, Heidelberg, 2003.

[Ere03] Justin R. Erenkrantz. Release management within open source projects. In *3rd Workshop on Open Source Software Egineering*, pages 51–55. International Conference on Software Egineering, 2003.

[fSidI03] Bundesamt für Sicherheit in der Informationstechnik. *Certification Report BSI-DSZ-CC-0216-2003*. Bundesamt für Sicherheit in der Informationstechnik, July 2003.

[Gan03] David Ganster. Sicherheit up to date. *IT-Security-Special 4/2003*, 4:39–40, 2003.

[Gra02] Volker Grassmuck. *Freie Software - Zwischen Privat- und Gemeineigentum*. Bundeszentrale für politische Bildung (bpb), 2002.

[Gru02] Dr. Ernst-Hermann Gruschwitz. *Zertifitierungstag 2002: Aktuelles aus der TÜViT-Zertifizierungsstelle*. TÜV Informationstechnik GmbH, June 2002.

[Hen01] Elisabeth Hendrickson. *Better Testing – Worse Quality?* Aveo Inc., 2001.

[Jek04] Sebastian Jekutsch. *Diverses zum Thema Open-Source-Entwicklungsmodell*, December 2004.

[Kad00] Prof. Dr.-Ing. Firoz Kaderali. *Ansätze zur Qualitätssicherung und Rückkopplung der Campus-Source-Nutzer*. FernUniversität Hagen - Fachbereich Elektrotechnik, June 2000.

[KK00] Marit Köhntopp und Andreas Pfitzmann Kristian Köhntopp. *Sicherheit durch Open Source? - Chancen und Grenzen*. Secure Electronic Marketplace for Europe (SEMPER), July 2000.

[Koc01] Dr. Stefan Koch. *Entwicklung von Open Source und kommerzieller Software: Unterschiede und Gemeinsamkeiten*. Wirtschaftsuniversität Wien, 2001.

[Kru04] Stefan Kruber. *Die Rolle des Internets für Open Source Projekte*. FH Regensburg, November 2004.

[Mat03] Martin Matuska. *Kategorisierung von Open Source Projekten*. Institut für Informationsverarbeitung und -wirtschaft, Wirtschaftsuniversität Wien, 2003.

[Mie02] Caspar Clemens Mierau. *Motivation und Organisation von Open Source Projekten*. Bauhaus-Universität Weimar, 2002.

[Neu00] Peter G. Neumann. Robust nonproprietary software. May 2000.

[Org04] The Mozilla Organization. *Mozilla Quality Assurance*, November 2004.

[Sau04] Heinz Sauerburger. *Open-Source-Software*. dpunkt.verlag GmbH, Heidelberg, August 2004.

[Som04] Ian Sommerville. *Software Egineering*. Addison-Wesley, Longham, Amsterdam, May 2004.

[Sta05] Richard Stallman. *The Free Software Foundation's Campaign for Free BIOS*. Free Software Foundation, February 2005.

[Tho03] Craig Thomas. Improving verification, validation, and test of the linux kernel: the linux stabilization project. In *3rd Workshop on Open Source Software Egineering*, pages 133–136. International Conference on Software Egineering, 2003.

[uoD00] Arbeitskreis Technische und organisatorische Datenschutzfragen. *Transparente Software - eine Voraussetzung für datenschutzfreundliche Technologien*. Der Bayerische Landesbeauftragte für den Datenschutz, November 2000.

[uSE03] Luyin Zhao und Sebastian Elbaum. Quality assurance under the open source development model. *The Journal of Systems and Software*, 66:65–75, 2003.

[uSvE04] Markus Pasche und Sebastian von Engelhardt. *Volkswirtschaftliche Aspekte der Open-Source-Softwareentwicklung*. Friedrich-Schiller-Universität Jena, 2004.

[Wie04] Stephan Wiesner. *Qualitätssicherung von J2EE Anwendungen V1.0*, November 2004.

[www05] www.opensource.org. *The Open Source Definition*, 2005.