Seminar "Ausgewählte Beiträge zum Software Engineering"

Sommersemester 2006

Qualitätssicherung bei Open Source Projekten

Stephanie Wilke
swilke@inf.fu-berlin.de
Betreuer: Christopher Oezbek

27.07.06

Zusammenfassung

Bei der Entwicklung von Open-Source-Software ist klar, dass die Qualität des Produktes nur schwer zu beurteilen ist. Durch die vielen unabhängigen und parallel arbeitenden Entwickler und das scheinbare Fehlen von bewährten Kontrollstrukturen der klassischen Software-Entwicklung kommt es trotzdem zu erfolgreichen Produkten, wie z.B. Linux.
Diese Seminararbeit untersucht, inwieweit Open Source Projekte Qualitätssicherheits-Standards enthalten und ob diese der klassischen Qualitätssicherheit entsprechen.

Inhaltsverzeichnis

1 Einleitung

Ein entscheidender Aspekt von Open Source Software ist das kooperative Entwicklungsmodell, gestützt durch die Freiheiten und die Kommunikationsmöglichkeiten des Internets. Nicht mehr Unternehmen oder Entwickler arbeiten für einen Zweck, vielmehr findet ein aktiver Austausch innerhalb einzelner Projekte und zwischen Projekten statt. [uMB06]
Daraus folgen diverse strukturelle Vorteile gegenüber normalen Entwicklungsmodellen.
Linux Torvald machte diese Art der Software Entwicklung im Basar-Stil populär.
Die Entwickler und Anwender verhalten sich wie Händler auf einem Basar, wenn sie ein neues Open Source Projekt entwickeln und dabei auf vorhandene Softwarekomponenten zugreifen. Dabei entsteht bei einer Vielzahl von Projekten ein reger Austausch, bei dem vor allen Dingen kommuniziert wird. [Ray02]

Die Kommunikation unter den Teilnehmern eines Open Source Projektes basiert auf zwei wichtigen Säulen:
Dem Internet als schnelle und billige Datenübertragungsmöglichkeit und der englischen Sprache als international akzeptierte Sprache. [Lut04]
Die Kommunikation geschieht häufig über Mailinglisten oder E-Mails wie z.B. bei Linux Kernel, über Foren oder Datenbanksysteme. Mozilla z.B. verwaltet sein Fehlermanagement über eine so genannte Datenbank namens Bugzilla. [Bar05]
Dies hat auch den Vorteil, dass die Kommunikation sehr leicht archiviert werden kann.
Anders als bei vielen Softwareprojekten beginnt die Entwicklung der Open Source Software meist ohne jeden zusätzlichen Verwaltungsaufwand. Typische frühe Projektphasen zur Strukturierung und Koordination der noch folgenden Phasen, wie z.B. eine exakte Anforderungsanalyse, spielen oftmals überhaupt keine Rolle.
Dadurch ist es schwer, die Software Entwicklung bei Open Source Software zu untersuchen, wenn vorher keine Produkteigenschaften festgelegt wurden und nicht ganz klar ist, was am Ende für ein Produkt entstehen soll, da jeder Entwickler eine andere Vorstellung von dem Produkt hat.

In dieser Seminararbeit soll nun untersucht werden, inwieweit Open Source Software Qualitätssicherung unterstützt. Im Einzelnen wird auf Komponententests, Integrationstests, wie man Leute zur Testdurchführung anregen kann und auf zwei verschiedene Tools zur Qualitätssicherung eingegangen. Zuerst wird jedoch die Theorie von Sommerville [Som01] und Balzert [Bal98] über Qualitätssicherung vorgestellt.

2 Theorie

Als Erstes wird die Frage geklärt, was Qualitätssicherheit überhaupt beinhaltet.
Qualitätssicherung ist der unternehmensinterne Prozess, der sicherstellen soll, dass ein hergestelltes Produkt ein Qualitätsniveau erreicht und behält, wobei das Erreichen des Qualitätsniveaus über Testen erzielt wird und das Behalten des Niveaus über die Wartung des Produktes.
Eingeordnet im Softwareentwicklungszyklus bedeutet dies, dass während und nach dem Implementieren des Codes der Entwickler seinen eigenen Code testet. Nach der Auslieferung an den Kunden beginnt die Wartung, wobei das Produkt verbessert und erweitert werden kann.

2.1 Sommervilles Definition vom Testen

Sommerville macht zwei wesentliche Unterscheidungen, die beide möglichst in allen Testphasen erfüllt werden sollen. Hierbei geht es um Verifikation und Validierung. Die Verifikation soll die Frage beantworten, ob das Produkt richtig erstellt wird, d.h., ob die Software ihre Spezifikation erfüllt.
Validierung dagegen klärt, ob das richtige Produkt erstellt wird, d.h., ob die Software die Kundenerwartung erfüllt.

Weiterhin macht Sommerville eine Unterscheidung der Phasen des Testens in Komponententests und Integrationstests. Beim Komponententest wird eine klar abgegrenzte Komponentenfunktion vom Programmierer getestet. Beim Integrationstests werden die Komponenten von Integrationsteams im System integriert und zusammen getestet.

Die Komponententests können mit verschiedenen Verfahren durchgeführt werden.
Unter anderem mit dem Black-Box-Test. Dieser Test ist ein funktionaler Test, bei dem die Tests von der Programm- oder Komponentenspezifikation abgeleitet werden. Das System ist eine "Black-Box", deren Verhalten nur durch die Untersuchung ihrer Eingaben und der dazugehörigen Ausgaben festgestellt werden kann.
Der Tester nimmt Eingaben in die Komponente oder in das System vor und untersucht die entsprechenden Ausgaben. Wenn die Ausgabe nicht den Vorhersagen entspricht, hat der Test erfolgreich ein Problem mit der Software aufgedeckt.
Ein anderer Komponententest ist der White-Box-Test. Dies ist ein struktureller Test, bei dem die Tests aus der Kenntnis der Softwarestruktur und der Implementierung abgeleitet werden. Wie der Name des Tests schon besagt, kann der Tester den Code analysieren und seine Kenntnisse der Ablaufstruktur zur Ableitung der Testdaten nutzen. Die Codeanalyse dient dazu, herauszufinden, wie viele Testfälle erforderlich sind, um alle Anweisungen im Programm oder in der Komponente wenigstens ein Mal im Verlauf des Testprozesses auszuführen.

Die Integrationstests können auch mit verschiedenen Verfahren durchgeführt werden.
Ein Verfahren ist der Top-Down-Test, bei dem der Entwicklungsprozess mit den übergeordneten Komponenten beginnt und sich durch die Komponentenhierarchie nach unten hin durcharbeitet. Das Programm wird als eine einzige abstrakte Komponente dargestellt, deren Teilkomponenten als Dummys dargeboten sind. Nachdem die übergeordnete Komponente programmiert und getestet worden ist, werden auf dieselbe Weise ihre Teilkomponenten implementiert und getestet. Dieser Prozess setzt sich bis zur Implementierung der Komponenten der untersten Ebene fort.
Ein weiteres Verfahren ist der Bottom-Up-Test, bei dem der Test mit der Integration und den Tests der Module auf den untersten Ebenen der Hierarchie beginnt und sich durch die Hierarchie der Module aufwärtsarbeitet, bis das letzte Modul getestet ist. Dieser Ansatz erfordert anfangs keinen vollständigen Entwurf der Systemarchitektur und kann daher bereits in einem frühen Stadium des Entwicklungsprozesses beginnen.

2.2 Balzerts Definition vom Testen

Balzert erklärt, dass man drei Klassen des analytischen Verfahrens zum Testen durchführen muss, um einen vollständigen Test des Programms zu erhalten.
Diese drei Klassen sind das Testende Verfahren, das Verifizierende Verfahren und das analysierende Verfahren.
Testende Verfahren lassen sich in dynamische und statische Tests aufteilen. Hierbei geht es um die Erkennung von Fehlern. Wobei dynamische Tests mit konkreten Eingaben getestet werden, wie z.B. mit dem Black-Box-Test, dem White-Box-Test oder dem Diversifizierenden Test. Back-Box- und White-Box-Tests werden hierbei wie bei Sommerville definiert. Der Diversifizierende Test vergleicht die Ergebnisse von verschiedenen Versionen, falls solche vorhanden sind. Bei statischen Tests dagegen werden die Komponenten nicht ausgeführt, sondern analysiert, wofür es mehrere Methoden gibt wie z.B. die Inspektion, bei der eine Person den Code durchsucht, um Anomalien und Fehler zu finden, ohne das Programm auszuführen.
Die zweite Klasse ist das Verifizierende Verfahren, bei dem unter anderem die Verifikation oder symbolische Ausführung durchgeführt wird, um die Korrektheit zu beweisen. Dies wird mit mathematischen Beweisen gemacht.
Das Analysierende Verfahren stellt die Eigenschaften der Komponenten mit Grafiken und Tabellen dar.
Die beiden letzten Verfahren werden jedoch nicht in Open Source Projekten umgesetzt und aus diesem Grund nicht weiter erläutert.

2.3 Sommervilles Definition der Wartung

Nach Sommerville lässt sich die Softwarewartung in drei Arten untergliedern.
Die Wartung zur Reparatur von Softwarefehlern macht 17 Prozent der Wartung aus, die Wartung zur Anpassung der Software an eine andere Betriebsumgebung (wenn z.B. das Betriebssystem geändert wird) entspricht 18 Prozent der Wartung. Die Wartung zum Hinzufügen oder Ändern von Systemfunktionen hat den größten Anteil der Wartung mit 65 Prozent, wenn der Kunde beispielsweise neue oder erweiterte Funktionen haben möchte.

Diesen Zahlen können wir entnehmen, dass die Reparatur von Systemfehlern nicht die kostspieligste Wartungsaktivität darstellt. Stattdessen erfordert die Weiterentwicklung des Systems zu dem Zweck neue Umgebungen und neue oder geänderte Anforderungen bewältigen zu können, den größten Aufwand.
Sommerville sieht jedoch auch Probleme in der Wartung, wie z.B. das die Wartung viel Zeit kostet. Zwischen 50 Prozent und 75 Prozent der Zeit des gesamten Programmieraufwandes wird für die Weiterentwicklung vorhandener Systeme benötigt.
Die Wartung ist ebenfalls zeitaufwändig, weil sie einen relativ langen Prozess durchlaufen muss. Der Kunde muss zuerst Änderungen vorschlagen bzw. monieren. Die Softwarefirma muss eine neue Version planen, in der wieder eine Anforderungsanalyse durchgeführt wird. Es wird entschieden, ob sich die Veränderung lohnt, oder der Aufwand zu komplex oder zu teuer wird. Entscheidet man sich für die Änderung, muss diese implementiert werden. Idealerweise sollte die Spezifikation angepasst werden.
Viele Organisationen machen immer noch Unterschiede zwischen Systementwicklung und Wartung. Die Wartung wird als zweitklassige Aufgabe angesehen und nicht sehr gerne gemacht.
Auch besitzen die Entwickler keine sehr große Motivation, alle Fehler beim Entwickeln zu finden, da sie meistens unter Zeitdruck arbeiten und Abgabefristen einhalten müssen. So soll das Produkt schon möglichst fehlerfrei zum Kunden geliefert werden. Das führt dann wiederum sehr schnell zu den ersten Wartungsarbeiten.
Trotz des großen Aufwandes der Wartung sagt Sommerville, dass Systeme, die nicht geändert werden müssen, nicht existieren. So ist der Prozess der Wartung immer zu vollziehen.

2.4 Balzerts Definition der Wartung

Balzert trifft bei der Wartung zwei grundlegende Unterscheidungen. Er unterteilt die Wartung in Wartung und Pflege, wobei die Wartung sich mit der Lokalisierung und Behebung von Fehlerursachen beschäftigt, wenn Fehler bekannt sind. Pflege dagegen beschäftigt sich mit der Lokalisierung und Durchführung von Änderungen und Erweiterungen. Betrachtet man beide Teile, stimmen sie mit der Definition der Wartung von Sommerville überein.
Weiter macht Balzert die gleiche Einteilung der Wartung und Pflege wie Sommerville, bezeichnet sie nur anders. Balzert teilt die Wartung und Pflege in drei Kategorien ein.
In korrigierende Aktivitäten, die der Fehlerbehebung dienen, in anpassende Aktivität, wenn z.B. das Unternehmen ihr Betriebsystem ändert und in perfektionierende Aktivität, wenn Funktionen geändert werden sollen oder neue dazukommen.
So kann man abschließend sagen, dass Sommervilles und Balzerts Definitionen der Qualitätssicherung sich ähneln, jedoch nicht identisch sind.

Es wird nun die Qualitätssicherung von Open Source Projekten betrachtet und inwieweit man die Definitionen von Sommerville und Balzert wiederfinden kann.

3 Qualitätssicherung in Open Soucre Projekten

Bei der klassischen Software Entwicklung, spielt das Testen eine wichtige Rolle, weil man dem Kunden ein hochwertiges Produkt zu einem hohen Preis liefern möchte. Dagegen spielt das formale Testen bei Open Source Projekten keine so große Rolle. Validierung wird eher auf den Benutzer übertragen, bei dem der Tester zum Benutzer wird und der Benutzer erst durch die Anwendung der Software das Produkt testet.

3.1 Komponententests

Zuerst wird betrachtet, inwieweit Open Source Projekte die Komponententests durchführen.
In einer Studie von Zhao und Elbaum [uSE02] wurde herausgefunden, dass 68 Prozent der Entwickler Eingabe-Versuche machten, um Benutzer zu simulieren, was einem Black-Box-Test entspricht. 25 Prozent der Entwickler testeten mit Extremwerten als Eingabe, was ebenfalls zum Black-Box-Test gehört. 26 Prozent der Entwickler führten andere Validierungsmethoden aus.
Die Prozentzahlen ergeben in der Summe mehr als 100 Prozent, weil die Entwickler auch mehrere Testarten durchführen konnten.
Zu dieser Studie ist jedoch noch anzumerken, dass es weder Daten darüber gab, ob die Befragten an Open Source Projekten teilnahmen, noch wie oft sie diese Tests durchführten. Es war jedoch eine Studie über Qualitätssicherung in Open Source Projekten, so dass man davon ausgehen kann, dass die Befragten Open Source Teilnehmer waren.

Weiter gibt es für Komponententests das Peer Review, welches eine wichtige Säule von Open Source Projekten und Qualitätssicherung bildet. Beim Peer Review wird der Code über Mailinglisten veröffentlicht und kann so von anderen Entwicklern bewertet werden, hierbei dürfen Vetos eingelegt werden. Daraus können informelle Diskussionen entstehen oder bei manchen Projekten auch formalisierte Vetos.
So ein formalisiertes Veto hat der Apache Webserver. [Moc02]
In Apache Webserver Projekten testet der Entwickler seinen Code zuerst selber mit Komponententests, am häufigsten mit ad-hoc Tests. Dann veröffentlicht der Entwickler den Code über Mailinglisten mit Peer Review.
Dabei kann der Entwickler um eine positive Bewertung bitten, d.h. es müssen mindestes drei Personen seinen Code befürworten und keiner darf ein Veto einlegen, damit der Code übernommen wird. Der Entwickler muss nicht um diese Bewertung bitten, jedoch kann der Code ansonsten, wenn er zu viele Gegenstimmen bekommt, wieder herausgenommen werden.

Ein weiteres Verfahren für Komponententests ist das Verteilte Debugging.
Die Voraussetzung für das Verteilte Debugging entspricht genau der Definition der Open Source Projekte, mit einer Vielzahl von Benutzer und einem frei verfügbaren Quellcode.
So tragen die Benutzer zur Fehlersuche, Fehlerbehebung sowie zur Programmverbesserung bei. Das hat den Vorteil, dass Probleme schneller als in der klassischen Software-Entwicklung behoben werden können. Der Benutzer ist in der Lage, das Problem selber zu lösen oder es gibt meistens mindestens einen anderen Entwickler/Benutzer, den das Problem interessiert und der es dann lösen kann. Dies ist die Stärke von Open Source Projekten.
Das Interesse ist bei den Entwicklern höher, da sie fast immer die Software auch selber benutzen.
Verteiltes Debugging hat jedoch auch die Nachteile, dass es keine Daten darüber gibt, wie viele den Code schon bearbeitet haben. Der Benutzer muss sich den Code runterladen und somit Vertrauen gegenüber dem Projekt haben. Um das Problem nicht schlimmer zu machen oder ein neues Problem einzubauen, muss der Benutzer den Code richtig verstehen. Aus diesem Grund hat der Benutzer das Problem auch richtig zu lösen.
Der Benutzer sollte ebenfalls vertrauenswürdig erscheinen, sodass seine Lösung übernommen wird.

So kann man feststellen, dass Komponententests bei Open Source Projekten durchgeführt werden, wenn auch nicht so formal wie in der Theorie definiert.

3.2 Integrationstests

Weiter wird betrachtet, ob in Open Source Projekten Integrationstests durchgeführt werden.

In den meisten Projekten gibt es Power-User, die Systemtests durchführen. [Cha05] Systemtests gehören zu den Integrationstests. Power-User sind Benutzer, die immer die neueste Version haben und gerne Feedback geben.
Dieses Verfahren wird z.B. bei Linux Kernel und Mozilla Webbrowser durchgeführt.
Bei Linux Kernel wird meistens nach der Implementierung der Code in einer Mailingliste für die Besprechung veröffentlicht. Meinungen zu dem Code werden ebenfalls über die Mailingliste verschickt. Dadurch werden Verifikation und Validierung von der Gemeinschaft durchgeführt.
Mozilla Webbrowser dagegen hat ein Test-Team, das Integrationstests durchführt. Dieses Team stammt noch aus der Zeit von Netscape. Wie und ob dieses Team bezahlt wird, ist nicht bekannt.

3.3 Wie man Leute zum Testen animiert

Eine wichtige Frage ist, wie man die Leute zum Testen animiert, da man gesehen hat, dass das Testen auf den Benutzer übertragen wird. Testet der Benutzer die Software, indem er sie benutzt, muss er die Probleme, die er findet, auch melden oder selber beheben, damit die Probleme beseitigt werden können.
Bei Open Source Projekten entsteht ein Gefühl der Zusammengehörigkeit, mit dem die Leute angeregt werden, nach Fehlern zu suchen.
Linux Kernel z.B. belohnt seine Benutzer mit der Gewissheit, Teil einer lohnenden Sache zu sein und in regelmäßigen Abständen mit Verbesserungen versorgt zu werden, bei denen sie ihren eigenen programmierten Teil sehen können. [Ray02]
So findet auch ein Geben und Nehmen von Informationen statt, bei dem der Benutzer seine Fähigkeiten zur Verfügung stellt und er wiederum ein Programm ohne zusätzliche Kosten erhält.

Da vermutlich nur 10 Prozent bis 20 Prozent der Benutzer Feedback geben, müssen die Projekte eine möglicht große Gemeinschaft haben, um viele Fehler aufzudecken und beheben zu können.
Linux verhält sich aus diesem Aspekt nach folgendem Grundsatz:
"Kann man auf genügend große Anzahl von Mitentwicklern zählen, lässt sich praktisch jedes Problem schnell charakterisieren und die Lösung ist einem der Beteiligten offensichtlich."(Linus Law) [Ray02]

3.4 Tools

Um Qualitätssicherung in Open Source Projekten umsetzen zu können, gibt es verschiedene Werkzeuge (Tools). Zwei dieser Tools sind der Bug-Tracker und der Standford Checker.

Mit dem Bug-Tracker können Fehler behoben werden, indem ein Benutzer sich ein interessantes Problem sucht, den Code dazu findet und den Code so bearbeitet, dass das Problem behoben wird. Anschließend veröffentlicht der Benutzer den Code und wartet auf die Feedbacks.
Dieses Verfahren wird unter anderem bei Mozilla Webbrowser mit Bugzilla angewendet.
Bugzilla ist eine Datenbank, in der alle noch vorhandenen Probleme veröffentlicht werden.
Jedes Problem bekommt eine Nummer (ID), eine Kurzbeschriftung des Fehlers und ein Fehlerszenario, wobei die Kurzbeschriftung und die Szenarien nicht geschrieben werden müssen. Dies bleibt dem Benutzer überlassen, nur die ID wird automatisch vergeben. Jedes Problem hat noch einen Status der Bearbeitung, wo vermerkt wird, ob schon angefangen wurde, das Problem zu bearbeiten, oder ob es sogar schon gelöst wurde.
Zusätzlich hat Bugzilla die Möglichkeit, Wünsche für zukünftige Funktionen anzubringen, falls ein Benutzer nicht in der Lage ist, diesen Wunsch selber umsetzen zu können.

Abbildung 1 zeigt eine Webseite von Bugzilla, in der in die erste Spalte die ID steht, in der zweiten Spalte der Schweregrad, dann die Dringlichkeit, danach auf welcher Plattform der Fehler aufgetreten ist, in der nächsten Spalte wer ihn bearbeitet, darauf folgt die Spalte mit dem Status der Bearbeitung, dann die Lösung, wenn schon vorhanden und in der letzten Spalte die Kurzbeschriftung des Fehlers.

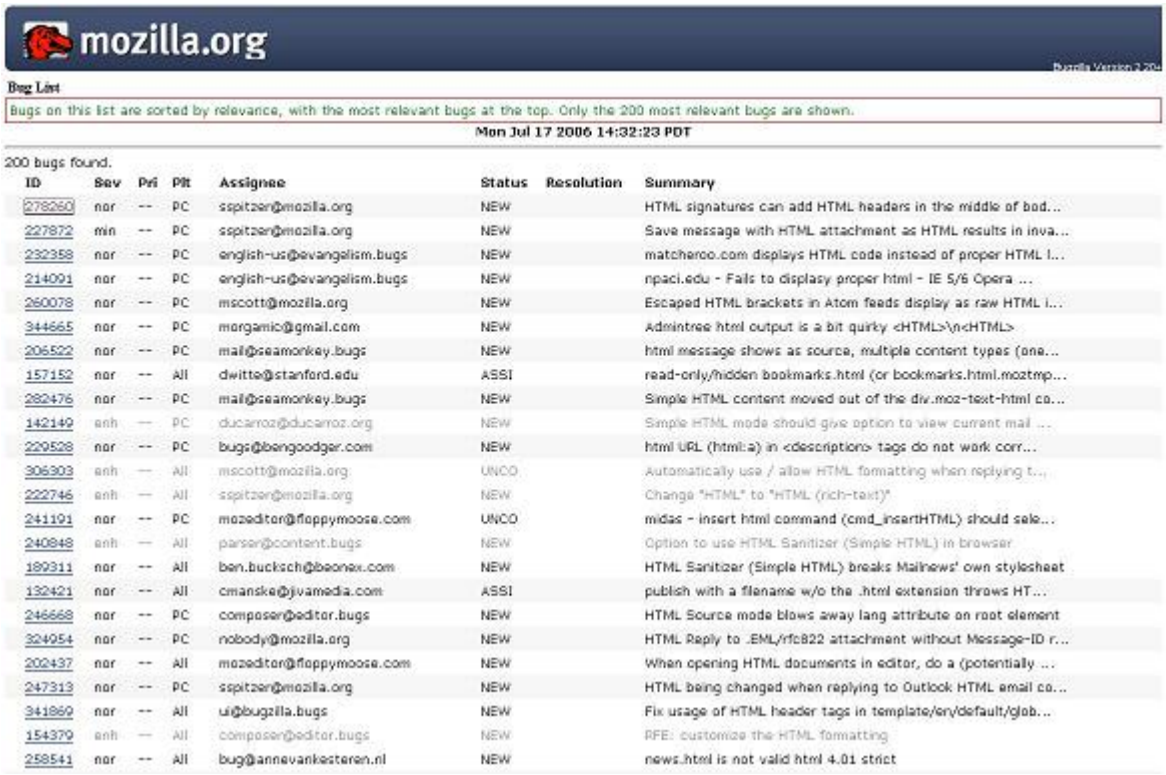

Abbildung 1: Webseite von Bugzilla

Der Stanford Checker ist dagegen ein systematischer Fehlerscanner für Open Source Quellcode. Diese Software soll alle Fehler, wie z.B. Pufferüberläufe und andere kritische Bugs finden. Die gefundenen Fehler werden dann in einer Datenbank aufgelistet.
Auftraggeber zur Entwicklung dieser Software ist das Departement for Homeland Security (DHS).
Umgesetzt wurde dieses Projekt in Zusammenarbeit der Standford Universität mit der Sicherheitsfirma Coverity. Die Software ist eine frei verfügbare Software, jedoch nicht Open Source, das bedeutet man muss nachfragen, ob sie den Standford Checker über das Projekt laufen lassen und man erhält die Liste der gefundenen Fehler. Da Coverity eine Firma ist und somit nicht umsonst arbeitet, hat das DHS 1,24 Mio USD für die Arbeit der nächsten drei Jahre zur Verfügung gestellt.
Der Standford Checker wird bis jetzt unter anderem von Linux Kernel, Mozilla Webbrowser, MySQL und Apache Webserver benutzt. [Tho06]

Zur Verdeutlichung, wie gut die Qualitätssicherung bei Open Source Projekten im Gegensatz zu klassischen Software Projekten ist, hier noch ein Beispiel: Der Standford Checker wurde über Linux Kernel laufen gelassen. Bei 5,7 Millionen Programmzeilen wurden 985 Bugs gefunden. Im Gegensatz zu kommerziellen Programmen mit der gleichen Anzahl von Programmzeilen werden durchschnittlich 5.000 Bugs gefunden. [Kol04]

4 Zusammenfassung

Zusammenfassend kann man sagen, dass Validierung überwiegend auf den Benutzer übertragen wird.
Es gibt jedoch trotzdem Komponententests wie z.B. durch den Black-Box-Test, den der Entwickler durchführt, durch das Peer Review, bei dem man Vetos einlegen kann, oder das Verteiltes Debugging, bei dem Benutzer Fehler finden und die Fehler selbst lösen können.
Auch Integrationstests werden indirekt mit Systemtests durchgeführt durch die Power-User, die oft und gerne Feedback geben. Jedoch gibt es auch noch Open Source Projekte, die aus ihrer kommerziellen Zeit Test-Teams haben, die Integrationstests durchführen.
Durch das Gemeinschaftsgefühl bekommt man die Leute zum Testen, was sehr wichtig ist, da die meisten Tests von den Benutzern übernommen werden. Wenn die Benutzer kein Feedback geben, können die meisten Probleme nicht gefunden werden, da die Fehler in Open Source Projekten ja sonst nicht gesucht werden.

Es gibt jedoch auch noch Tools zur Qualitätssicherung in Open Source Projekten wie den Bug-Tracker, bei dem der Benutzer sich ein für ihn interessantes Problem sucht und löst oder der Standford Checker, welches eine Software ist, die Fehler findet und auflistet.

Abschließend ist zu bemerken, dass die Qualitätssicherung von Open Source Projekten dem Ansatz von Sommervilles Qualitätssicherung am nächsten kommt. Obwohl die Theorie nicht ganz mit der Praxis übereinstimmt, ist die Qualitätssicherung von Open Source Projekten dennoch sehr erfolgreich.

Literatur

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik*. Pro-Linux, 1998.

[Bar05] Istvan Bartkowiak. Qualitätssicherung bei open source projekten. April 2005.

[Cha05] Sebastian Charlet. Prozesse in open source projekten. März 2005.

[Kol04] Panagiotis Kolokythas. Studie: Linux hat 985 bugs in 5,7 mio. code-zeilen. *PC-Welt*, Dez 2004.

[Lut04] Robert A. Gehring; Bernd Lutterbeck. *Open Source Jahrbuch 2004*. Pearson Studium-Verlag München, 2004.

[Moc02] Audris Mockus. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, pages 309–346, July 2002.

[Ray02] Eric Steven Raymond. The cathedral and the bazaar. *The Journal of Systems and Software*, pages 65–75, April 2002.

[Som01] Ian Sommerville. *Software Engineering*. Pearson Studium-Verlag München, 2001.

[Tho06] Thomas. Homeland security will sicherheit von open source erhöhen. *Pro-Linux*, Jan 2006.

[uMB06] Bernel Lutterbeck; Robert A. Gehring und Mattias Bärwoltt. *Open Source Jahrbuch 2006*. 2006.

[uSE02] Luyin Zhao und Sebastian Elbaum. Quality assurance under the open source development model. *The Journal of Systems and Software*, pages 65–75, April 2002.