Freie Universität Berlin

Arbeitsgruppe

Software Engineering

Die Kontaktaufnahme mit Open Source Software-Projekten. Eine Fallstudie

Luis Quintela García
quintela@inf.fu-berlin.de
Matrikel-Nr. 3891724

Erstgutachter: Prof. Dr. Lutz Prechelt
Zweitgutachterin: Prof. Dr. Elfriede Fehr

15. September 2006

Zusammenfassung: Mit dieser Fallstudie sollte der Einstieg in zwei Open Source Software (OSS)-Projekte (KDE und Gnome) untersucht werden. Dazu wurde eine teilnehmende Beobachtung in Form eines simulierten Kontaktversuchs mit den beiden OSS-Projekten durchgeführt und ausführlich qualitativ-deskriptiv ausgewertet. Die Ergebnisse zeigen, dass sich die realen Schwierigkeiten und Hindernisse für einen Einstiegswilligen anders gestalten, als das der aktuelle Stand der Literatur vermuten lässt.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Bachelorarbeit bis auf die offizielle Betreuung selbst und ohne fremde Hilfe angefertigt habe und die benutzten Quellen und Hilfsmittel vollständig angegeben sind.

Berlin, den 15. September 2006 ....

(Luis Quintela García)

Inhaltsverzeichnis

1 Einführung....................4
2 Theoretischer Bezugsrahmen....................6
2.1 Open Source Software....................6
2.2 Die Struktur von OSS-Projekten....................6
2.2.1 Technisch-organisatorische Struktur von OSS-Projekten....................6
2.2.2 Die soziale Struktur von OSS-Projekten....................7
2.2.3 Einfluss in OSS-Projekten....................8
2.2.4 Der Einstieg in OSS-Projekte....................10
3 Begründung der eigenen Fragestellung....................11
4 Methodik der Studie....................12
4.1 Wahl des Untersuchungsdesigns....................12
4.2 Datenerhebungsmethode....................12
4.3 Eigenes Vorgehen bei der Datenerhebung....................12
4.3.1 Das Software-Projekt....................13
4.3.2 Die Vorstellung meines Vorschlags in den Mailinglisten....................14
4.4 Datenauswertungsmethode....................15
4.4.1 Eigenes Vorgehen bei der Auswertung....................15
4.5 Verallgemeinerbarkeit der Ergebnisse....................16
5 Darstellung und Auswertung der Ergebnisse....................17
5.1 Interaktion mit den Teilnehmern der Mailinglisten....................17
5.2 Überblick über die Ergebnisse....................17
5.3 Darstellung und erste Interpretation der Kategorien....................19
5.3.1 Die Kategorien der technischen Ablehnung....................20
5.3.1.1 Die Kategorien „es gibt schon so etwas“ (egw) und „alternative-Vorschläge zur Idee“ (avi)....................20
5.3.1.2 Die Kategorien „dadurch erreicht man nicht das angestrebte Ziel“ (naz) und „falsche Zielsetzung“ (fz)....................21
5.3.1.3 Die Kategorie „keine technische Betrachtung des vorgeschlagenen Codes“ (nbc) .22
5.3.2 Die Kategorien der sozialen Ablehnung....................22
5.3.2.1 Die Kategorie „keine Zugehörigkeit zur Gesprächsgruppe“ (kzzg)....................22
5.3.2.2 Die Kategorie „falsches Forum“ (ff)....................23
5.3.2.3 Die Kategorie „power-users versus newbies“ (puvn)....................23
5.3.3 Die Kategorien der technischen Akzeptanz....................24
5.3.3.1 Die Kategorie „gute Idee“ (gi)....................24
5.3.3.2 Die Kategorie „Vorschläge für die Weiterentwicklung“ (vfw)....................24
5.3.4 Die Kategorien der Umfeld-bezogenen Akzeptanz und Ablehnung....................24
6 Zusammenfassung und Diskussion der Ergebnisse....................25
6.1 Diskussion ausgewählter Ergebnisse....................25
6.2 Reflexion....................29
7 Ausblick....................30
Literaturverzeichnis....................31
Anhang A....................32
Anhang B....................40

1 Einführung

Open Source Software (OSS)-Projekte finden in jüngster Zeit immer mehr wissenschaftliches Interesse (vgl. Brand, 2003; Ducheneaut, 2005). Das liegt u.a. daran, dass verschiedene OSS-Projekte und deren Produkte einen großen Einfluss in der Software-Branche gewonnen haben (vgl. Maass, 2004) sowie daran, dass die spezifische Art der Zusammenarbeit in OSS-Projekten als innovatives und äußerst effizientes Modell der Softwareentwicklung entdeckt worden ist (vgl. Ducheneaut, 2005; Spaeth, 2005).

Dabei hängt der Erfolg eines OSS-Projektes eng mit der Größe der Entwicklergruppe zusammen (vgl. von Krogh et al., 2003; Ducheneaut, 2005): Nur wenn die Gruppe von technisch hoch versierten, erfahrenen und engagierten Software-Entwicklern ausreichend groß ist, können die Vorteile der kooperativen, netzwerkartig organisierten Zusammenarbeit in OSS-Projekten gegenüber der kommerziellen, hierarchisch organisierten Software-Entwicklung voll zum Tragen kommen. Insofern ist es existenziell wichtig für OSS-*Communities*, genügend kompetenten Nachwuchs für die Mitarbeit am Projekt zu gewinnen: „the key to its success is the ability to attract new volunteers" (KDE, 2006). Obwohl sich die OSS-Projekte der Bedeutung der Gewinnung und Einarbeitung von Einsteigern völlig bewusst sind und auf ihren Webseiten versuchen, Einstiegswilligen alle möglichen Tipps und Hilfen zukommen zu lassen (vgl. z.B. KDE, 2006; Python, 2006), ist es den Teilnehmern selbst, als den involvierten Akteuren, prinzipiell nicht möglich, alle tatsächlichen Mechanismen eines erfolgreichen oder missglückten Einstiegs in eine OSS-Entwicklungs-*Community* zu durchschauen. Dazu bedarf es empirischer wissenschaftlicher Forschung. Von Krogh et al. (2003) betonen, dass es entscheidend für eine Theorie des OSS-Entwicklungsprozesses ist, zu verstehen, wie der Einstieg von Neumitgliedern vonstatten geht, „a theory of the open source software innovation process needs to explain how people sign up for the production of the public software good" (von Krogh et al., 2003, S. 1218).

Mittlerweile liegen einige wenige Übersichtsarbeiten zum Einstiegsprozess in OSS-Großprojekte vor[1] (z.B. von Krogh et al., 2003; Ducheneaut, 2005). Aus diesen empirischen Arbeiten wurden z.T. unterschiedliche Schlussfolgerungen in Bezug auf die entscheidenden Vorgänge beim Einstieg in OSS-Projekte gezogen, so verweist Ducheneaut (2005) auf die zentrale Rolle von Netzwerkbildung (vgl. Kap. 2.2.3), von Krogh et al. (2003) fokussieren dagegen vor allem auf den ersten Kontakt mit einer OSS-*Community* über die entsprechenden Mailinglisten und untersuchen u.a. den Einfluss von „Geschenken"[2] an die *Community* in Form von Code-Beilagen.

Edwards (2001) betont, dass zwar viele OSS-Projekte auf ihren Webseiten Hinweise und Tipps für Einstiegswillige bereithalten, wie man sich am Projekt beteiligen kann, dass aber entscheidende Hinweise in Bezug auf „empfehlenswertes" Vorgehen und soziales Verhalten fehlen. Gerade solche differenzierten Vorgehensweisen sind nach Edwards (2001) jedoch entscheidend dafür, ob ein Einstieg in das OSS-Projekt schließlich gelingt oder nicht.

In der vorliegenden Einzelfallstudie soll dieser Frage exemplarisch nachgegangen werden. Dazu wird in zwei verschiedenen OSS-Projekten (KDE und Gnome) ein versuchter Einstieg inszeniert. Im ersten Szenario bietet ein Außenstehender dem KDE-Projekt einen Prototypen seiner eigenen Software-Entwicklung an und äußert den Wunsch, dass seine Entwicklung als Erweiterung in die Software des OSS-Projekts aufgenommen wird. Im zweiten Fall präsentiert ein Außenstehender dem Gnome-Projekt lediglich einen inhaltlichen Vorschlag für eine Erweiterung der OSS-Software, hat aber bisher keine eigene Software entwickelt. Diese beiden Versionen der Kontaktaufnahme mit einem OSS-Projekt werden als teilnehmende Beobachtung durchgeführt. Anschließend werden die

1 Tjøstheim und Tokle (2003) haben andere Einstiegsmodalitäten in kleinen OSS-Projekten mit weniger als zehn Teilnehmern gefunden.

2 Vgl. die Diskussion zur so genannten „gift economy", wie sie in Kap. 2.2.3 kurz angerissen wird.

Reaktionen der Mailinglistenteilnehmer detailliert ausgewertet.

In der schriftlichen Ausarbeitung werden in Kapitel 2 zunächst die theoretischen Bezüge der Arbeit dargestellt und ein kurzer Überblick über die relevante Literatur zur technisch-organisatorischen und sozialen Struktur in OSS-Projekten, zur Frage des Einflusses und schließlich zum Einstieg in OSS-Projekte gegeben. Daraus wird in Kapitel 3 meine eigene Fragestellung für diese Arbeit abgeleitet. In Kapitel 4 werden die angewandten Methoden ausführlich dargestellt. Dabei wird zunächst die teilnehmende Beobachtung als Datenerhebungsmethode vorgestellt und dann auf das Softwareprojekt eingegangen. Mein Softwareprojekt war der Prototyp eines Zusatz-*Features* für den Terminal. Der Code dieses Prototypen wurde als „Code-Geschenk“ im Sinne von von Krogh et al. (2003) bei der Kontaktaufnahme mit der KDE-*Community* beigelegt. Anschließend stelle ich meine Methodik der Datenauswertung dar und mache die Grenzen der Verallgemeinerbarkeit der Ergebnisse deutlich. In Kapitel 5 stelle ich meine Ergebnisse, die Reaktionen der Teilnehmer der Mailinglisten auf meinen Vorschlag, ausführlich dar und lasse erste Interpretationen einfließen. Anschließend werden in Kapitel 6 die Ergebnisse noch einmal kurz zusammengefasst und dann unter Bezug auf die dargestellte Literatur diskutiert. Abschließend wird in Kapitel 7 ein kurzer Ausblick gegeben.

2 Theoretischer Bezugsrahmen

Im folgenden Kapitel werden die theoretischen Bezüge der vorliegenden Arbeit kurz dargestellt. Dazu soll zunächst geklärt werden, was überhaupt unter *Open Source* (OS) bzw. *Open Source Software* (OSS) zu verstehen ist. Anschließend wird überblicksartig Literatur zur sozialen Struktur von *Open Source Software*-Projekten referiert und dargestellt, wie der Einstieg von Neulingen in OSS-Projekte in der einschlägigen Literatur diskutiert wird.

2.1 Open Source Software

Der Begriff *Open Source Software* bezieht sich zunächst auf die Art von Lizenzen, unter denen Software vermarktet oder verteilt werden darf. Dafür wurde von der Open Source Initiative (OSI, 2006) eine explizite *Open Source*-Definition (OSD) entwickelt, welche die folgenden vier zentralen Punkte enthält: Die Lizenz garantiert die freie Weiterverteilung der Software und aller ihrer Teile, die Software muss den Quellcode enthalten, die Lizenz erlaubt das Ableiten und das Modifizieren von Software, die Unversehrtheit des Quellcodes eines Autors darf dabei gewährleistet werden.

Das *Open Source*-Entwicklungsmodell basiert auf der freiwilligen Arbeit einer geographisch breit verteilten Teilnehmerschaft (siehe Brand, 2003), der so genannten *Community*. Zentrale Merkmale dieses Entwicklungsmodells sind u.a. schnelle Releasezyklen und fortwährende gegenseitige Durchsichten des Codes zur Qualitätskontrolle, die als *peer-review* bezeichnet werden: „Making the source code open for inspection by peers is regarded as a guarantee of the high quality of source code“ (Bergquist & Ljungber, 2001, S. 306).

Open Source-Projekte haben die Herstellung von *Open Source Software* zum Ziel und arbeiten nach dem OS-Entwicklungsmodell.

Spaeth (2005) konstatiert, dass *Open Source* als Produktionsmodell, als Art der Projektorganisation und Zusammenarbeit, aber auch als Philosophie oder Paradigma gesehen wird.

2.2 Die Struktur von OSS-Projekten

Die Struktur eines OSS-Projekts kann auf technisch-organisatorischer und auf sozialer Ebene beschrieben werden. Bei einer technisch-organisatorischen Beschreibung wird der Blick auf die verwendeten Technologien und Organisationsstrukturen gelenkt, bei einer sozialen Betrachtung stehen eher die sozialen Interaktionen zwischen den Projektteilnehmern im Mittelpunkt.

2.2.1 Technisch-organisatorische Struktur von OSS-Projekten

Eine Besonderheit von OSS-Projekten im Vergleich zu anderen Herstellungsprozessen ist die Tatsache, dass es sich bei der Erstellung von OS-Software um einen Arbeitsprozess handelt, der von einer virtuellen *Community* erledigt wird, die fast ausschließlich über elektronische Kanäle, in der Regel Mailinglisten, kommuniziert. „Die alltägliche Produktion und Kommunikation läuft nur über die Informations- und Kommunikationstechnik ab. ... Die Informations- und Kommunikationstechnik ist somit die Basis für das Entstehen und Bestehen des *Open Source*-Projekts“ (Brand, 2003, S. 10).

OSS-Projekte verwenden eine Infrastruktur von Servern und Projekt-Verwaltungssoftware, die die Arbeit des Projektes trotz weiter geographischer Verteilung der *Community* möglich macht. Die Verwaltungssoftware besteht im Wesentlichen aus E-Mail-Verteilern für die alltägliche Kommunikation, Webservern für die Verbreitung von Informationen oder Produkten und dem *Concurrent Versioning System* (CVS). Das CVS verwaltet den Quellcode des Projektes. Der Lesezugriff

auf den Quellcode über das CVS steht in der Regel allen frei, aber es ist üblich, dass der Schreibzugriff auf den im CVS gespeicherten Quellcode beschränkt ist.

> In many OSS projects developers work in geographically distributed locations, rarely or never meet face-to-face, and coordinate their activities almost exclusively over the Internet. They successfully collaborate using simple, already existing text-based communication media such as electronic mail ... as well as revision tracking systems (e.g. CVS – Concurrent Versioning System) to store the software code they produce.(Ducheneaut, 2005, S. 323 f.)

Die Produktion von Software in OSS-Projekten erfolgt mit Hilfe einer losen Zusammenarbeit von Projektteilnehmern. Die Beteiligung als Softwareentwickler an OSS-Projekten ist in der Regel eine ehrenamtliche Tätigkeit. Motivation und selbständiges Arbeiten werden vorausgesetzt.

> Any developer of OSS communities maintains a set of collective, social, norm-oriented and reward motives. However, they also have the liability to deliver contributions of a given quality for which rewards are granted in terms of gaining reputation within the community. Contributions depend on the expertise and skills that a developer obtains. (Maass, 2004, S. 1f.)

Das gemeinsame Interesse einer *Community* kann als Summe von individuellen, gegebenenfalls auch divergierenden, Interessen betrachtet werden.

> OSS development represents a “private-collective” model of innovation where developers obtain private rewards from writing code for their own use, sharing their code, and collectively contributing to the development and improvement of software. (von Krogh, 2003, S. 1217)

Auf die Tatsache, dass in OSS-Projekten jeder Entwickler (auch) seinen eigenen Anliegen oder Interessen nachgeht, spielt u.a. das bekannte Basar-Modell von OSS-Projekten an (vgl. Raymond, 1997). Nach dem Basar-Modell bietet jeder Entwickler seine eigene Software in Rahmen eines OSS-Projekts feil, so wie das ein Verkäufer auf dem Basar tut. D.h. jeder ist frei, seine eigenen Produkte anzubieten und es gibt keine planende und kontrollierende Instanz, sondern nur den Markthallen-Aufseher, der auf die Einhaltung der notwendigen Regeln achtet, im Falle eines OSS-Projekts also den *Maintainer*.

Allerdings gelingt es nicht jedem Entwickler, seine Software zum Teil einer OSS zu machen. Von welchen Faktoren es abhängt, ob es einem Entwickler gelingt, die eigene Software in das OSS-Projekt einzubinden, wird unterschiedlich diskutiert. Einige Autoren nehmen an, dass das zentral von der technischen Kompetenz eines Entwicklers abhänge: „Contributions depend on the expertise and skills that a developer obtains“ (Maass, 2004, S. 2). Die meisten Autoren betonen jedoch, dass es entscheidend für einen erfolgreichen Einstieg sei, die sozialen Strukturen und Prozesse in der entsprechenden OSS-*Community* zu verstehen.

2.2.2 Die soziale Struktur von OSS-Projekten

Die soziale Struktur von OSS-Projekten kann sehr komplex sein. Um diese Sozialstruktur theoretisch zu beschreiben, verwenden einige Autoren das soziologische Rollenmodell. Verallgemeinernd könnte man in OSS-Projekten folgende wichtige soziale Rollen definieren: Die beiden wichtigsten Rollen sind die des *Core Developers* als anerkanntem Softwareentwickler und die des *Maintainers* als dem Verantwortlichen für die Wartung von Softwaremodulen oder *Community*-Ressourcen. Weiterhin lassen sich die Rolle des *Patchers* und des *Bug Reporters* abgrenzen. Dabei ist ein *Patcher* ein Entwickler von Lösungen für erkannte Fehlfunktionen der Software und ein *Bug Reporter* ein engagierter Benutzer, der Berichte über Fehlfunktion der Software beim Betrieb an das Projekt abliefert (vgl. Ducheneaut, 2005).

Damit die Teilnehmer der *Community* diese differenzierten Rollen und die dazugehörigen Normen besser verstehen und übernehmen können, fordern z.B. Moon und Sproull (2000), die verschiedenen Aufgaben in Form von Rollenbeschreibungen zu explizieren: „These roles, with their corresponding obligations and responsibilities, should be explicitly designated and understood by all”

(Moon & Sproull, 2000).

Brand (2003) konstatiert im Gegensatz dazu, dass es sich eher um Arbeitsaufgaben oder Tätigkeitsbereiche als um feststehende soziale Rollen handelt und weist auf Überschneidungen hin: „Die Unterscheidung in die jeweiligen Tätigkeitsbereiche, wie Softwareentwickler, Dokumentierer, Übersetzer, etc., ist schwierig, da es Überschneidungen gibt“ (Brand, 2003, S. 5f.).

Andere Autoren, wie z.B. Ducheneaut (2005), betonen, dass die Interaktion in OSS-*Communities* so vielfältig und komplex ist, dass traditionelle soziologische Beschreibungen wie Rollenmodelle nicht ausreichend sind, um die Spezifika der Interaktionen in OSS-*Communities* abzubilden. Sie fassen die soziale Struktur von OSS-Projekten mithilfe von Netzwerkmodellen. Die Besonderheit virtueller Kommunikation ist es, dass die Interaktion mithilfe technischer Hilfsmittel geschieht und es somit möglich ist, alle Kommunikationswege aufzuzeichnen “using aggregate statistics to summarize, for instance, the position of a particular participant inside a project based on the frequency of his message postings ... or the distribution of CVS commits and downloads” (Ducheneaut, 2005, S. 324 f.).

Ein weiteres Spezifikum dieser virtuellen Kommunikation ist, dass ein Mitglied der virtuellen *Community* für die anderen nur über seine elektronischen Beiträge sichtbar ist. Das impliziert, dass die Identität eines Entwicklers in der *Community* durch seine Beiträge bestimmt ist. Im Rahmen der psychologischen Internet-Forschung definiert Döring(2003) bezüglich dieses Phänomens den Begriff der virtuellen Identität:

> Mit virtueller Identität ist eine dienst- oder anwendungsspezifische, mehrfach in konsistenter und für andere Menschen wiedererkennbarer Weise verwendete, subjektiv relevante Repräsentation einer Person im Netz gemeint (Döring, 2003, S. 341).

So kann man zum Beispiel Diskussionsbeiträge in Mailinglisten von OSS-Projekten als solche Identitätsrepräsentationen bezüglich der eigenen technischen Ansichten, der bevorzugten Arbeitsweisen oder der ethischen Prinzipien in Entwicklungsaktivitäten verstehen.

Interessante Textbeiträge oder guten Code zu schreiben, machen jedoch noch keine Identität aus. Dafür bedarf es der Interaktion mit anderen. In dieser Interaktion sind technisches Wissen und soziale Kompetenzen verflochten. So braucht man z.B. gute Kenntnisse über die technischen Gegebenheiten des Projektes, um einschätzen zu können, ob ein eigener Entwicklungsvorschlag den technischen Ansichten[3] eines Mitentwicklers entsprechen könnte.

Ducheneaut (2005) betont, dass es nicht sinnvoll ist, die technischen Prozesse und die soziale Struktur von OSS-Projekten unabhängig voneinander zu untersuchen. Er plädiert dafür, diese beiden Perspektiven durch eine ganzheitliche Betrachtung von OSS-Projekten zu verbinden. Deshalb veranlasste er die Entwicklung eines OSS-Browsers, der Diskussionsbeiträge und Code-Beiträge in Hinblick auf Teilnehmer und Entstehungszeit graphisch darstellte. Mithilfe dieses Browsers untersuchte er das Zusammenwirken zwischen Entwicklern und Code-/ Diskussionsbeiträgen und zeichnete dies in komplexen Netzwerkgraphiken auf. Anhand seiner Ergebnisse schlägt er das Modell eines „hybriden“, das heißt aus technischen und sozialen Elementen bestehenden, Netzwerks vor, „the hybrid, socio-technical networks emblematic of OSS projects“ (Ducheneaut, 2005, S.332).

2.2.3 Einfluss in OSS-Projekten

Der Einfluss eines Projektteilnehmers auf Entscheidungen bezüglich der Software-/ Projekt-Entwicklung hängt zentral von der Verteilung von Zugriffsrechten auf die virtuellen Ressourcen (CVS, Mailinglisten, Web-Portale usw.) ab. So kann zum Beispiel ein Teilnehmer ohne CVS-Zugriff keinen direkten Einfluss auf das Endprodukt nehmen, weil seine Beiträge lediglich über einen

3 Mit „technische Ansichten“ sind hier nicht nur die expliziten Äußerungen eines Entwicklers gemeint, sondern auch seine Codebeiträge und inwiefern diese Codebeiträge durch den neuen Vorschlag betroffen sein könnten.

Dritten auf das CVS gespeichert werden können. Dabei korrelieren die Einflussmöglichkeiten und Zugriffsrechte eines *Community*-Mitglieds mit seiner sozialen Rolle und Reputation im Projekt. So haben Mitglieder, die viele und inhaltlich wichtige Beiträge für das Projekt beigesteuert haben, einen großen Einfluss auf Entwicklungsentscheidungen und werden in der *Community* besonders anerkannt. So berichten beispielsweise Moon und Sproull (2000), wie diese Anerkennung bei *Linux* besonders deutlich gemacht wurde: "The kernel [of linux] were accompanied by a 'credits file', which publicly acknowledges people who have contributed substantial code to the kernel" (Moon & Sproull, 2000).

Verschiedene Autoren betonen in diesem Zusammenhang die Rolle von sozialer Anerkennung oder Reputation. „In open source software development, the motivating factor is also influence, recognition from ones peers, and reputation" (Edwards, 2001, S. 9). Diese Reputation mündet in sozialen Status im Projekt und der wiederum in in erhöhten technischen Einfluss. „Dabei ist die Reputation das Selektionskriterium, nach dem die Projektbeteiligten ihren sozialen Status in dem Großprojekt festlegen" (Brand, 2003, S. 25). Brand (2003) geht davon aus, dass der Erwerb von Reputation überwiegend durch Leistungen für das Projekt erfolgt. „Reputation oder besser soziale Anerkennung erhält man Schritt für Schritt durch Leistung, kooperative Kommunikation, Seniorität/ Erfahrung und Sichtbarkeit/ Involviertheit" (Brand, 2003, S. 25). Unter Leistung versteht der Autor einen hohen, kontinuierlichen Arbeitsaufwand für das Projekt, der von den anderen Teilnehmern durch Einsichten in die Dateiablagerung identifiziert werden kann (vgl. Brand, 2003). In dieser Sichtweise hängt der soziale Status bzw. die Reputation eines Projektteilnehmers vorrangig von seinen sichtbaren konkreten Leistungen für das Projekt ab.

Die Übernahme interessanter Tätigkeitsbereiche bzw. einflussreicher Rollen durch Mitglieder der Community ist natürlich ein dynamischer Prozess. Dabei stellt sich die Frage, wie es einem Teilnehmer gelingen kann, eine Funktion mit hohem Ansehen im OSS-Projekt zu übernehmen. Manche Autoren gehen davon aus, dass es sich dabei im Grunde um einen Prozess des Erlernens von technischen Fertigkeiten handelt und dass ein Teilnehmer, der sich wie ein traditioneller Lehrling ausreichend spezialisiertes Wissen und technische Fertigkeiten angeeignet hat, irgendwann aufgrund seiner Kompetenz im Projekt aufsteigen und eine wichtige Rolle übernehmen kann (vgl. Edwards, 2001; Brand, 2003).

Andere Autoren (z.B. Ducheneaut, 2005) halten die Annahme, dass die Rollenverteilung, und somit die Verteilung von Einfluss und Status im Projekt, lediglich vom Erwerb entsprechender technischer Fertigkeiten abhänge und somit ein linearer „Bewährungsaufstieg" sei, für eine Verkürzung. Ducheneaut (2005) versteht den Prozess des Erlangens von sozialem Status in OSS-Projekten, analog zu anderen sozialen Gruppenprozessen, als mikropolitischen Vorgang. Eine Besonderheit in Bezug auf OSS-Projekte wäre dann lediglich, dass es sich um eine virtuelle Gruppe handelt, d.h. dass in der Regel alle Interaktionen über schriftliche Internet-Kommunikation vermittelt sind. Wenn also ein Projektteilnehmer Reputation und damit Einfluss in der *Community* gewinnen möchte, dann geschieht das, indem er bewusst oder unbewusst seine virtuelle Präsenz und damit Identität ausbaut. „In online communities, control over artifacts is a central source of power and influence ... and textual resources are manipulated to project a participant's 'virtual' identity" (Ducheneaut, 2005, S. 327).

Das Phänomen, dass es einer gut entwickelten virtuellen Identität bedarf, um als Vollmitglied akzeptiert zu werden, wird z.T. in OSS-*Communities* ganz deutlich angesprochen. So ist z.B. auf der Web-Site des Python-Projekts vermerkt: „If the python-dev team knows who you are, whether through mailing list discussion, having submitted patches, or some other interaction, then you can ask for full CVS access" (Python, 2006).

2.2.4 Der Einstieg in OSS-Projekte

Viele Autoren betonen, dass der Einstieg in ein OSS-Projekt ein längerfristiger, stufenweiser Prozess ist (vgl. z.B. Brand, 2003; Edwards, 2001; von Krogh, 2003). Zugleich merkt Edwards (2001) an, dass es für Einstiegswillige oft gar nicht einfach ist herauszufinden, wie sie sich aktiv an der Software-Entwicklung in einem OSS-Projekt beteiligen können, da zwar die Webseiten von OSS-Projekten in der Regel Instruktionen für die Mitarbeit enthielten, oft aber entscheidende Hinweise fehlten.

Von Krogh et al. (2003) betonen, dass Neulinge, die in ein OSS-Projekt einsteigen wollen, bereits eine gewisse technische Expertise vorweisen können müssen und zugleich verstehen müssen, welches Sozialverhalten von ihnen erwartet wird. Dazu ist es in der Regel erforderlich, dass der Einstiegswillige einige Zeit lang die Diskussionen in der Entwickler-Mailingliste verfolgt. In dieser Zeit kann sich der Einsteiger mit Sprache, Code usw. der *Community* vertraut machen und diese schließlich selbst übernehmen (Edwards, 2001). Erst dann ist es ratsam, eigene Diskussionsbeiträge an die Mailingliste zu schicken.

Von Krogh et al. (2003) weisen in diesem Zusammenhang auf die soziale Austauschtheorie von Thibaut, und Kelley (1959)[4] hin und betonen die entscheidende Rolle von Code-Geschenken bei der Kontaktaufnahme[5]. Die Autoren analysierten in ihrer Studie die Funktion von „Code-Geschenken" von Einstiegswilligen in Form von vollständigen Modulen oder eigenständigen Funktionalitäten (von Krogh et al., 2003). Bergquist und Ljungber (2001) betonen die Rolle von Code-Geschenken für die Gestaltung von sozialen Beziehungen und technischen Innovationen in OSS-Projekten und konstatieren, dass derjenige, der Code-Geschenke macht, dadurch Ansehen in der *Community* gewinnt:

> The gift economy is important, not only because it creates openness, but also because it organizes relationships between people in a certain way. Open source software development relies on gift giving as a way of getting new ideas and prototypes out into circulation. This also implies that the giver gets power from giving away (Bergquist & Ljungber, 2001, S. 305).

Um schließlich Entwickler mit CVS-Schreibberechtigung zu werden, muss ein Teilnehmer in der Regel schon längere Zeit aktives Mitglied der Entwickler-Mailingliste gewesen sein (vgl. von Krogh, 2003).

Ducheneaut (2005) stellt bezüglich der technischen Anforderungen folgenden Aufgabenkatalog für Einsteiger in OSS-Projekte auf:

(1) Beobachtung der Entwicklungsaktivitäten von außen

(2) Berichten von Fehlern in der Software und Vorschläge für ihre Berichtigung machen

(3) CVS-Zugriff erlangen und anschließend Software-Fehler selbst berichtigen

(4) Verantwortung für ein Teilprojekt (in Modul-Größe) übernehmen

(5) Teilprojekt weiterentwickeln, Unterstützung organisieren und für Projekt werben

(6) Zustimmung der „core-developer" für die Integration des Projektes in die Zielprojekt-Architektur holen und das Modul integrieren (Ducheneaut, 2005, S. 349, eigene Übersetzung).

Zugleich betrachtet Ducheneaut (2005) die Entscheidungsprozesse in OSS-Projekten als politische Vorgänge, bei denen es zentral darauf ankommt, über ein gut ausgebautes Netzwerk und Reputation innerhalb der *Community* zu verfügen. Deshalb schlägt er folgende Vorgehensweise bezüglich der sozialen Interaktionen während des Einstiegs vor. Zuerst solle man das Projekt eine Weile möglichst

4 Thibaut, J.W. & Kelley, H. H. (1959). The Social Psychology of Groups. New York: Wiley.

5 Siehe auch Raymond (1997), der diese Idee mit seinem Basar-Modell in die OSS-Welt eingeführt hat (vgl. Kap. 2.2).

unauffällig beobachten (im Original: *to lurk)*, um sich der Projektkultur besser anpassen zu können. Dabei solle man Bereiche identifizieren, in denen Mitarbeit gebraucht wird. Erst dann solle man sich Schlüssel-Verbündete suchen, um Unterstützung für die künftige eigene Arbeit zu haben (Ducheneaut, 2005, S. 358).

3 Begründung der eigenen Fragestellung

Eine große Anzahl von Projektmitgliedern erhöht die Erfolgsmöglichkeiten eines OSS-Projektes (vgl. Ducheneaut, 2005). Deshalb ist es sinnvoll, mögliche Erleichterungen des Einstiegs zu erforschen. So beobachteten von Krogh et al. (2003) im Rahmen einer Studie zum Einstieg in OSS-*Communities* am Beispiel des Freenet-Projekts den Erfolg bzw. Misserfolg von Einsteigern, die bei Kontaktaufnahme einen technischen Vorschlag gemacht hatten:

> No joiner started out by unsolicited 'new' technical suggestions, perhaps indicating that it might be wise to start out humbly and not to boldly announce "great ideas" for solving problems. In fact none of the 12.3% who suggested technical solutions without accompanying software code in their first post were joiners. In 16.7% of the cases, joiners started by offering source code for a bug fix, or an additional feature, in the form of actual software code submission, in the evolving software architecture, whereas only 4.6% of all non-developers did the same (mostly the announcement of external client projects) (von Krogh et al., 2003, S. 1227f.).[6]

Zwei Aussagen fallen bezüglich der Einstiegsmöglichkeiten auf. Erstens, dass keine neue Idee ohne Code-Beilage erfolgreich war. Zweitens, dass 16,7% der erfolgreichen Einsteiger Code bei der Kontaktaufnahme beigelegt haben. Wohlgemerkt machen von Krogh et al. im zweiten Fall keinen Unterschied zwischen Codebeiträgen zu Bugs und Code zu neuen Entwicklungsvorschlägen.

Diese Beobachtungen lassen vermuten, dass man mehr Chancen für den Einstieg hat, wenn man Code zu der vorgeschlagenen Idee beilegt, als wenn man lediglich eine neue Idee vorschlägt.

Ferner betont Edwards (2001) die Rolle der nicht explizierten sozialen Anforderungen beim Einstieg wie folgt:

> Many 'wanabe' developers and newcomers to OSS development have difficulties understanding how to contribute to the development. Often project websites contain instructions of how to contribute, but these instructions miss some of the finer points on proper conduct and behaviour. It seems that these finer points are important and the distinguishing point between becoming a part of the community and staying on the outside. (Edwards, 2001, S. 20)

Deshalb ist es eine weitere interessierende Frage für diese Arbeit, worin diese *finer points* bei einem konkreten Einstiegsversuch bestehen. Anhand des Einzelfalls soll untersucht werden, ob sich aus den Begründungen der Mailinglisten-Teilnehmer für die Akzeptanz oder Ablehnung eines Vorschlages oder aus ihren konkreten Verhaltensweisen in der Interaktion mit einem Einstiegswilligen weitere Hinweise darauf ergeben, welche Faktoren für den Erfolg eines Einstiegsversuches eine Rolle spielen.

6 Beachte, dass joiner hier definiert ist als Einsteiger mit gerade erlangtem CVS-Zugriff.

4 Methodik der Studie

In diesem Kapitel beschreibe ich kurz die verwendeten Forschungsmethoden, mache aber auch meinen eigenen Umgang mit diesen Methoden deutlich und benenne aufgetretene Probleme und Schwierigkeiten. Damit soll der Prozess der Datengewinnung und -auswertung transparent und nachvollziehbar gemacht werden.

4.1 Wahl des Untersuchungsdesigns

Für die vorliegende empirische Untersuchung wurde ein deskriptiver qualitativer Forschungsansatz gewählt, da das Ziel der Arbeit darin bestand, in einem noch nicht ausführlich beforschten Feld explorativ neue Hypothesen zu generieren. Dafür sollte anhand einer konkreten Fallstudie der Einstieg in ein OSS-Projekt exemplarisch untersucht werden. In der qualitativen empirischen Sozialforschung meint 'Fallstudie' ein Untersuchungsdesign, das

> zur Erforschung von Einzelpersonen oder Gruppen genutzt wird. Durch die Fallstudie versucht der Forscher explorativ und beschreibend Aussagen über den Untersuchungsgegenstand zu erlangen und keine generalisierbaren Aussagen. Dabei versucht die Fallstudie durch die Methode der Dichten Beschreibung ein holistisches Verständnis des Untersuchungsgegenstandes unter der Einbeziehung von so vielen als relevant erkannten Variablen wie möglich zu erreichen. Fallstudien werden mit Methoden der Ethnographie, Feldstudie und teilnehmenden Beobachtung in Verbindung gebracht. (Wikipedia, 2006)

4.2 Datenerhebungsmethode

Als Methode der Datenerhebung habe ich die teilnehmende Beobachtung gewählt, bei der es darum geht, Informationen über den interessierenden Sachverhalt direkt durch die Teilnahme an den sozialen Interaktionen zu erhalten, anstatt bloß Meinungen über Erfahrungen zu erfragen (siehe Lüders, 2000). Ziel dieses Vorgehens war es, einen möglichst direkten Einblick in den Prozess des Einstiegs in OSS-Projekte zu bekommen, ohne dass die untersuchten Mitglieder der Mailinglisten bemerkten, dass es sich um eine Untersuchung handelte.

4.3 Eigenes Vorgehen bei der Datenerhebung

In der vorliegenden Fallstudie sollte u.a. die Vermutung exemplarisch überprüft werden, dass man mehr Chancen für den Einstieg hat, wenn man Code zu einem neuen Vorschlag beilegt, als wenn man lediglich den neuen Vorschlag (ohne Code) einbringt. Dazu habe ich zwei verschiedene Bedingungen hergestellt: Ich nahm mit den OSS-Projekten KDE und Gnome auf nahezu identische Weise Kontakt auf, der einzige Unterschied bestand darin, dass ich bei KDE zu meinem Erweiterungsvorschlag Code beilegte, während ich Gnome nur meinen Vorschlag unterbreitete.

Die erste Voraussetzung für eine solche Untersuchung war, dass ich einen passenden Vorschlag bzw. ein Code-Geschenk entwickelte, wie es in der Literatur beschrieben wird (vgl. von Krogh et al., 2003). Der Vorschlag sollte für eine Software-Erweiterung des OSS-Produktes plädieren. Dabei musste die Grundidee weitere Anforderungen erfüllen: Sie sollte innovativ sein, offensichtlich nützlich sein und eine klare Abhängigkeit von der bestehenden Software des OSS-Projektes aufweisen. Innovation und Nützlichkeit würden das Werben für den Vorschlag erleichtern. Abhängigkeit von der bestehenden Software sollte für eine ausgeprägte Diskussion bzw. breite Interaktion mit den OSS-Projektteilnehmern sorgen.

Ich hatte eine Vorgabe für den Vorschlag. Das Projekt „*Skinning the Command Line*“ (siehe Anhang A) stand mir als Grundidee zur Verfügung. Die Grundidee dieses Projekts war es, die Erlernbarkeit des Terminals mittels graphischer Steuerelemente zu erhöhen.

Somit war die Auswahl an möglichen zu untersuchenden OSS-Projekten beschränkt. Es kamen nur Projekte in Frage, die eine Simulation des Terminals anbieten. Ich brauchte also eine vorhandene Terminal-Simulation, an die sich mein Vorschlag anlehnen konnte. Andererseits war es im Sinne des Erkenntnisgewinns der Studie sinnvoll, sich an bekannte bzw. große OSS-Projekte zu wenden, die eine wichtige Rolle in der Branche spielen.

So waren das KDE-Projekt und das Gnome-Projekt die passendsten Kandidaten dafür, weil sie die oben erwähnten Bedingungen erfüllten. Das KDE-Projekt und das Gnome-Projekt bieten graphische Schnittstellen, so genannte Desktops, für die Bedienung von Betriebssystemen an.

Das Innenleben eines jeden OSS-Projektes spielt sich im Rahmen von Mailinglisten ab. Deshalb musste ich eine Mailingliste pro OSS-Projekt für mein Vorhaben aussuchen. Prinzipiell standen die Mailinglisten *kde-quality* und *gnome-usability* zur Auswahl. Beide Mailinglisten widmen sich der Verbesserung der Benutzbarkeit des jeweiligen Desktops und sind als Referenzpunkt für Neulinge gedacht. Jedoch erschien es mir sinnvoll, den Vorschlag mit Code-Beilage in der Mailingliste eines Terminal-Simulation-Unterprojekts zu machen. Nach dieser Überlegung und aufgrund der ausgewählten Technologie für die eigene Entwicklung (vgl. Anhang A) kam auch die Mailingliste *konsole-devel* in Frage. *Konsole-devel* ist auf die Weiterentwicklung der KDE-Terminal-Simulation namens Konsole ausgerichtet. Wenn ich mich jedoch für die Terminal-Simulation des Gnome-Projekts entschieden hätte, hätte ich meinen Vorschlag mit Code-Beilage trotzdem auf *gnome-usability* präsentieren müssen, weil es keine eigene Mailingliste für die Weiterentwicklung des Gnome-Terminals gibt.

So wählte ich zunächst die Mailinglisten *konsole-devel* vom KDE-Projekt und *gnome-usability* vom Gnome-Projekt aus. Da im Laufe der Fallstudie jedoch keine richtige Interaktion mit *konsole-devel* zustande kam, beschloss ich, mich zusätzlich an *kde-quality* zu wenden. Dadurch würde das Grundkonzept des Vergleichs nicht verletzt werden und ich konnte auf mehr Ergebnisse hoffen.

4.3.1 Das Software-Projekt

Mit meinem Vorschlag wollte ich dem KDE-Projekt den Code eines neuen *Features* als Erweiterung für den Terminal (die Konsole) anbieten. Ziel dieses Software-Projekts war es, den Benutzer beim Erlernen und bei der Benutzung des Terminals zu unterstützen. Mittels graphischer Steuerelemente sollten neue Eingabe-Möglichkeiten in einer Terminal-Simulation angeboten werden. Die Betonung lag dabei jedoch nicht auf neuen Eingabemöglichkeiten, sondern darauf, dass dem Benutzer anhand neu eingeführter graphischer Steuerelemente Informationen zu einem Befehl näher gebracht werden sollten, d.h. dass die graphischen Steuerelemente eine Lehrfunktion haben sollten. Die aktuelle Dominanz von graphischen Desktops und das weit verbreitete Konzept der intuitiven Bedienung per GUIs ließen vermuten, dass der Lernprozess eines Benutzers durch graphisch orientierte Eingabemöglichkeiten unterstützt werden könnte.

Ich entwickelte also eine graphische Benutzerschnittstelle, die die oben genannten Möglichkeiten anbietet. Ich habe meine Entwicklung *SkKonsole* genannt. Eine genaue Beschreibung der Entwicklung des Software-Projekts findet sich in Anhang A. Um einen Eindruck von meinem Projekt zu vermitteln, ist hier ein Screenshot der *SkKonsole* abgebildet, die gerade aus einer Konsole gestartet wurde:

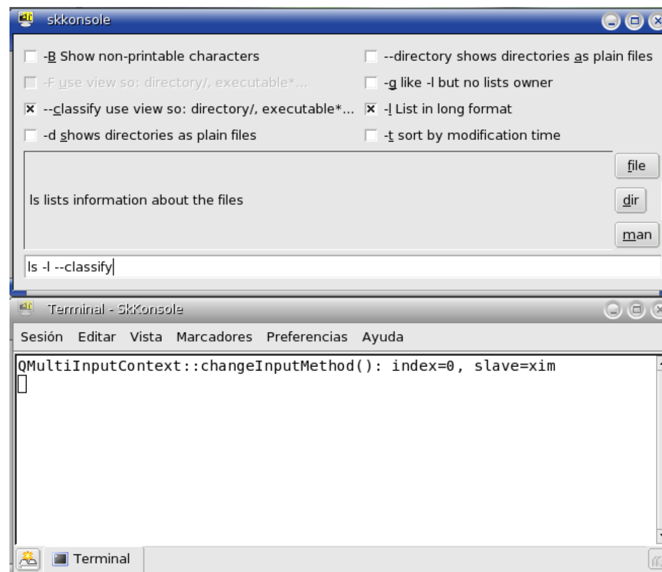

Abbildung 4.1: Die SkKonsole (oben) ist vom Terminal aus (unten) gestartet. Der Terminal zeigt am Ende die Ausgabe.

Nach der Beobachtung von von Krogh et al. (2003) (vgl. Kap. 3) sollte der zu einem Vorschlag beigelegte Code in Form und Architektur mit der zu erweiternden Software übereinstimmen. Im optimalen Fall müsste also mein beigelegter Erweiterungscode problemlos in die Konsole-Implementierung integrierbar sein. Jedoch zeigte sich, dass es mir im Rahmen meiner Bachelorarbeit aus Kapazitätsgründen nicht möglich war, einen solchen passenden Code zu erzeugen. Zunächst hatte ich mich mit den KDE-Technologien vertraut machen müssen. Die Komplexität der Konsole-Implementierung war sehr viel größer, als ich vermutet hatte. Zugleich waren meine Versuche, den Autor der Konsole zu kontaktieren und von ihm Tipps zum Vorgehen zu bekommen, nicht erfolgreich. Deshalb entwickelte ich schließlich die *SkKonsole* in C++ als Prototypen zur Demonstration der Grundidee (vgl. Anhang A). Schon das war zeitlich sehr aufwändig, allein für die Implementierung der *SkKonsole* brauchte ich etwa 150 Stunden.

Allerdings lässt sich eine Prototyp-Implementierung nicht direkt in bestehenden Code einsetzen. Trotz allem war die Möglichkeit der Einbettung meines Codes in die KDE-Architektur unter Verwendung von KDE-Technologien gegeben. Die Frage der Integrierbarkeit meines Codes erwies sich jedoch im Laufe meiner Studie als irrelevant (vgl. Kap. 5.1).

4.3.2 Die Vorstellung meines Vorschlags in den Mailinglisten

Ich stellte meinen Vorschlag jeweils mit einer E-Mail an die entsprechende Mailingliste und einer Webseite vor. Zuvor hatte ich die Diskussionen in den Mailinglisten schon einige Zeit verfolgt. Der Inhalt meiner E-Mails und die Webseiten (vgl. Anhang D) waren absichtlich fast gleich. Natürlich enthielt die Vorstellung bei den KDE-Mailinglisten Kommentare zur Code-Beilage, währenddessen die E-Mail und die Webseite für *gnome-usability* lediglich eine Entwicklungsidee vorstellten.

Ich stellte mich bei der Kontaktaufnahme mit meinen echten Personalien vor. Damit wollte ich den üblichen Sitten von OSS-Projekten in Bezug auf Selbstdarstellung folgen (vgl. Mahendran, 2002). Jedoch präsentierte ich mich nicht als Studenten, der eine Studie durchführt, sondern trat als motivierter Programmierer auf. Dieses Vorgehen sollte bewirken, dass die Mitglieder der Mailinglisten möglichst authentisch auf meinen Vorschlag reagierten[7]. Erst nach Abschluss meiner Datenerhebung informierte ich die Mitglieder der Mailinglisten über mein eigentliches Interesse und bat um

7 Wenn es jedoch jemand darauf angelegt hätte, hätte er anhand meines Namens über eine einfache Internet-Recherche relativ leicht herausfinden können, dass ich Student war und meine Bachelorarbeit schrieb.

die Erlaubnis zur Veröffentlichung der Interaktionen im Rahmen meiner Bachelorarbeit. Diese Erlaubnis wurde mir von allen Interaktionspartnern erteilt.

4.4 Datenauswertungsmethode

Die Diskussionen meines Vorschlags in den Mailinglisten bildeten mein Datenmaterial (siehe Anhang C). Zur Auswertung der E-Mails entschied ich mich für ein pragmatisches Vorgehen in Anlehnung an die qualitative Inhaltsanalyse nach Mayring (2002), bei dem im Vorhinein ein Kategoriensystem entwickelt wird:

> Inhaltsanalyse ist eine der klassischen Vorgehensweisen zur Analyse von Textmaterial gleich welcher Herkunft ... Ein wesentliches Kennzeichen ist die Verwendung von Kategorien, die häufig aus theoretischen Modellen abgeleitet sind: Kategorien werden an das Material herangetragen ... immer wieder daran überprüft und gegebenenfalls daran modifiziert (Flick, 2000, S. 212).

Mayring (2002) schlägt drei methodische Grundformen innerhalb der qualitativen Inhaltsanalyse vor:

> - Zusammenfassung: Ziel der Analyse ist es, das Material so zu reduzieren, dass die wesentlichen Inhalte erhalten bleiben, durch Abstraktion ein überschaubares Korpus zu schaffen, das immer noch ein Abbild des Grundmaterials ist.
> - Explikation: Ziel der Analyse ist es, zu einzelnen fraglichen Textteilen ... zusätzliches Material heranzutragen, das das Verständnis erweitert, das die Textstelle erläutert, erklärt, ausdeutet.
> - Strukturierung: Ziel der Analyse ist es, bestimmte Aspekte aus dem Material herauszufiltern, unter vorher festgelegten Ordnungskriterien einen Querschnitt durch das Material zu legen oder das Material aufgrund bestimmter Kriterien einzuschätzen. (Mayring, 2002, S. 115)

Für die Strukturierung des Materials hatte ich im Vorhinein anhand der Literatur ein ausführliches Kategoriensystem entwickelt (siehe Anhang B).

Meine grundlegende Fragestellung nach erfolgreicher oder nicht erfolgreicher Kontaktaufnahme mit der Mailingliste habe ich dabei als Akzeptanz bzw. Ablehnung meiner Idee operationalisiert. Als Akzeptanz wird eine Aussage gewertet, wenn sie beinhaltet, dass die vorgeschlagene Idee eine mögliche künftige oder auch nur wünschenswerte[8] Softwareerweiterung für das Projekt darstellen könnte. Als Ablehnung wird entweder eine Aussage gewertet, in der explizit die vorgeschlagene Idee als mögliche künftige Softwareerweiterung fürs Projekt verworfen wird oder eine Aussage bzw. Handlung, die implizit diese Bedeutung hat.

Inhaltlich untergliederten sich die Kategorien grob in die Bereiche technische Akzeptanz/ Ablehnung (Sinnhaftigkeit des Ziels/ Entwurfs, Code-Qualität, Passung der Implementierung zum bestehenden Code), soziale Akzeptanz/ Ablehnung (Zugehörigkeit, Beachtung von *Community*-Konventionen) und Umfeld-bezogene Akzeptanz/ Ablehnung (Passung zum Zielprojekt, Akzeptanz durch Benutzer).

4.4.1 Eigenes Vorgehen bei der Auswertung

Um einen ersten Verständnisrahmen für eine bessere Einordnung der Ergebnisse zu schaffen, habe ich zunächst eine zusammenfassende Nacherzählung meiner Interaktion mit den Teilnehmern der Mailinglisten geschrieben (vgl. Kap. 5.1).

8 Das kann z.B. vorkommen, wenn einer Diskussionspartner die Idee positiv bewertet, ohne in seinen Aussagen explizit Bezug auf die Eingliederung des Vorschlages ins Projekt zu nehmen. Hierbei ist die Tatsache, dass die Aussage im Kontext einer der Diskussionen des OSS-Projekts stattfindet, ausschlaggebend.

Im nächsten Schritt habe ich dann die einzelnen Aussagen in den E-Mail-Antworten meinen zuvor entwickelten Kategorien zugeordnet. Ein Überblick über die zutreffenden Kategorien findet sich in den Tabellen 1 bis 3 (vgl. Kap. 5.2).

Im dritten Teil meiner Auswertung habe ich die erhaltenen Antworten ausführlich dargestellt und eine erste Einordnung und Interpretation der Ergebnisse vorgenommen (vgl. Kap. 5.3). Zur Explikation im Sinne Mayrings (2002) habe ich in die Interpretation der Ergebnisse verschiedene Kontext-Informationen mit einbezogen, zum Beispiel wie sich mir die sozialen Verhältnisse in der Mailingliste darstellten oder wenn ich die Funktion bzw. Rolle eines Antwortenden recherchieren konnte.

4.5 Verallgemeinerbarkeit der Ergebnisse

Ziel dieser Arbeit ist es, die in der Literatur diskutierten Möglichkeiten des Einstiegs in OSS-Projekte anhand einer Fallstudie exemplarisch zu überprüfen. In diesem Sinne sind meine Ergebnisse als deskriptiv zu verstehen und die aus den Ergebnissen hervorgehenden Interpretationen haben hypothetischen Charakter. Für eine weitere Verallgemeinerbarkeit der Ergebnisse wären viele weitere Einzelfallstudien erforderlich: „Verallgemeinerung bei qualitativer Forschung liegt in der schrittweisen Übertragung von Erkenntnissen aus Fallstudien und ihrem Kontext in allgemeinere und abstraktere Zusammenhänge“ (Flick, 2000, S. 256).

5 Darstellung und Auswertung der Ergebnisse

In diesem Kapitel werden die Ergebnisse meiner Fallstudie dargestellt. Damit die Ergebnisse besser eingeordnet werden können, gebe ich zunächst kurz den Verlauf der Interaktionen mit den Mailinglisten-Teilnehmern wieder. Danach stelle ich tabellarisch dar, wie sich die Grundtendenz der Antworten gestaltete (Ablehnung vs. Akzeptanz meines Vorschlags) und welche meiner Kategorien zutreffend waren, d.h. wie viele Antworten semantische Inhalte enthielten, die meinen Kategorien (vgl. Kap. 4.3) entsprachen.

Die Zuordnung der Antworten zu den Kategorien erfolgte so, dass ich jede einzelne inhaltlich relevante Aussage mindestens einer Kategorie zuordnete. Die Zuordnung ist also mehrdeutig. Diese Mehrdeutigkeit der Zuordnung ist sinnvoll wegen der semantischen Vielfältigkeit von Textinhalten. So erlaubt diese Methode die Betrachtung von verschiedenen Aspekten und Implikationen eines Textabschnitts. Die einzelnen Aussagen entsprechen selten einem ganzen Diskussionsbeitrag, gehen aber meistens über einige Sätze. Dabei dient die Zuordnung zu Kategorien als strukturierter Rahmen für die tiefergehende qualitative Analyse der erhaltenen Antworten. Diese ist in Kapitel 4.3 ausführlich beschrieben.

5.1 Interaktion mit den Teilnehmern der Mailinglisten

Zuerst wendete ich mich mit meinem Vorschlag (inklusive Code-Beilage) an die Mailingliste konsole devel. In dieser Mailingliste fand aber keine Diskussion über meinen Vorschlag statt. Nur der Konsole-*Maintainer* schickte mir eine ablehnende Antwort, verfasst als private E-Mail. Auf diese Nachricht erwiderte ich und schickte auch eine Kopie an die Mailingliste. Jedoch gab es keine weitere Reaktion. Gleichzeitig wendete ich mich an *gnome-usability* mit meinem Vorschlag ohne Code-Beilage. Auf diesen Vorschlag hin bekam ich mehrere Antworten, die meist einen ablehnenden Charakter hatten. Ich antwortete darauf und nur ein Teilenhmer der Mailingliste meldete sich zurück, allerdings war er nicht mehr an der Diskussion über meinen Vorschlag interessiert, sondern wollte lediglich über einen Fehler meiner Vorschlagspräsentation kommunizieren. Darauf antwortete ich nicht mehr. Angesichts der mangelhaften Interaktion mit *konsole-devel* entschied ich mich dafür, den Vorschlag mit Code-Beilage zusätzlich an *kde-quality* zu schicken. Ich bekam mehrere, überwiegend positive Antworten. Unter diesen Nachrichten befand sich auch ein ziemlich positives Feedback vom Konsole-*Maintainer*, der diesmal als *kde-quality*-Teilnehmer auftrat. Allerdings wiesen fast alle diese Meldungen darauf hin, dass ich meine Anwendung im Rahmen von *kde-apps.org* (einem KDE-Portal, auf dem jeder die eigene Software frei anbieten kann) veröffentlichen sollte, statt die Integration meines Codes in das KDE-Projekt in Erwägung zu ziehen. Bei der Interaktion mit *kde-quality* und *konsole-devel* wurde deutlich, dass kein Diskussionsteilnehmer mein Code las.

5.2 Überblick über die Ergebnisse

Für einen ersten Überblick über die Ergebnisse sind in Tabelle 1 und 2 Akzeptanz und Ablehnung meines Vorschlags dargestellt. Tabelle 3 zeigt die zutreffenden Kategorien. Es gab insgesamt 8 Diskussionsteilnehmer und 11 Diskussionsbeiträge, diese enthielten 42 Aussagen[9]. Drei Diskussionsteilnehmer trugen jeweils 2 Diskussionsbeiträge bei. Die restlichen Teilnehmer beteiligten sich an der Diskussion mit einem einzigen Beitrag.

9 Das Datenmaterial befindet sich im Anhang C.

Wenn man die Ergebnisse in Hinblick darauf betrachtet, ob mein Vorschlag mit und ohne Codebeilage vorlag, erhält man folgende Übersicht:

	Mit Code-Beilage	***Ohne Code-Beilage***
Akzeptanz	10 Aussagen (45,4%)	2 Aussagen 10,0%
Ablehnung	12 Aussagen (54,6%)	18 Aussagen 90,0%
Anzahl	22 Aussagen insgesamt	20 Aussagen insgesamt

Tabelle 1: Aussagen bezüglich Akzeptanz/ Ablehnung nach Vorschlag mit oder ohne Code-Beilage

Wenn man die Ergebnisse in Hinblick darauf betrachtet, in welches Forum ich meinen Vorschlag einbrachte, erhält man folgende Übersicht:

	Kde-quality *(mit Codebeilage)*	***Kde-konsole-devel*** *(mit Codebeilage)*	***Gnome-usability*** *(ohne Codebeilage)*
Akzeptanz	10 Aussagen 58,9%	0 Aussage 0,0%	2 Aussagen 10,0%
Ablehnung	7 Aussagen 41,1%	5 Aussagen 100,0%	18 Aussagen 90,0%
Anzahl	17 Aussagen insgesamt	5 Aussagen insgesamt	20 Aussagen insgesamt

Tabelle 2: Aussagen bezüglich Akzeptanz/ Ablehnung nach Mailingliste

Wenn man die Ergebnisse in Hinblick darauf betrachtet, welche meiner Antwort-Kategorien wie häufig zutreffend waren, erhält man folgende Übersicht:

Ablehnung Gesamt: 30 (71,4%)	Technische Ablehnung	nicht sinnvolles Ziel oder Entwurf	schlechte Idee	1 (2,4%)
			es gibt schon so etwas	5 (11,9%)
			dadurch erreicht man nicht das angestrebte Ziel	3 (7,1%)
			falsche Zielsetzung	2 (4,8%)
			Alternativ-Vorschläge zur Idee	5 (11,9%)
		unpassende Implementierung	andere Strategie	1 (2,4%)
		keine technische Betrachtung des vorgeschl. Codes		1 (2,4%)
	Soziale Ablehnung	keine Zugehörigkeit	zur Gesprächsgruppe	4 (9,5%)
		sozial-hierarchische Einwände	falsches Forum	3 (7,1%)
			power-users versus newbies	3 (7,1%)
	Umfeld-bezogene Ablehnung	Vorschlag unpassend für Projektzustand		1 (2,4%)
		Benutzer/ Programmierer mögen das nicht		1 (2,4%)
Akzeptanz Gesamt: 12 (28,6%)	Technische Akzeptanz	sinnvolles Ziel oder Entwurf	gute Idee	4 (9,5%)
			Vorschläge für Weiterentwicklung	6 (14,3%)
	Umfeld-bezogene Akzeptanz	die Benutzer/ Programmierer mögen das		2 (4,8%)

Tabelle 3: Aussagen nach Kategorien-Zuordnung. Diese Kategorien trafen für Aussagen zu.

5.3 Darstellung und erste Interpretation der Kategorien

Im Folgenden werden die zutreffenden Kategorien ausführlich dargestellt und es findet eine erste Einordnung und Interpretation der Ergebnisse statt.

Zur Anonymisierung bezeichne ich die Akteure der untersuchten Diskussion mit Spitznamen, die ihren Rollen entsprechen. Zum Beispiel trägt der erste Diskussionsteilnehmer aus *gnome-usability*

den Namen „*gnome*-Teilnehmer1“, wobei die Zahl sein Erkennungszeichen zur Überprüfung der Aussagen im Datenmaterial ist. Auf diese Weise kann man am Spitznamen auch den Kontext der Aussage besser nachvollziehen. Aus der Mailingliste *konsole-devel* hat nur ein Teilnehmer geantwortet, der Konsole-*Maintainer*. Da der Konsole-Maintaner eine besondere Rolle in der Diskussion spielt, tritt er auch als solcher in meiner Beschreibung auf. Eine weitere besondere Rolle spielt der externe Angestellte von KDE aus der Mailingliste *kde-quality*.

5.3.1 Die Kategorien der technischen Ablehnung

Unter der Rubrik „technische Ablehnung“ sind alle Antworten in Kategorien zusammengefasst, in denen als Grund für das Zurückweisen meines Vorschlags explizit oder implizit technische Gründe angeführt werden.

5.3.1.1 Die Kategorien „es gibt schon so etwas“ (egw) und „alternative-Vorschläge zur Idee“ (avi)

Bei der Ablehnung meines Vorschlags aus technischen Gründen dominieren die Aussagen der Kategorien „es gibt schon so etwas“(*egw*) und „Alternativ-Vorschläge zur Idee“ (avi) (vgl. Tabelle 3). Dabei stammen die Aussagen über die Existenz von ähnlicher Software aus allen drei Mailinglisten. Allerdings lieferte *kde-quality* keine Aussage über mögliche alternative Vorschläge um das postulierte Ziel zu erreichen. Die Aussagen in diesen beiden Kategorien kommen oft vom selben Diskussionspartner, stammen aber nicht zwingend aus der selben Nachricht (vgl. Anhang C Z. 26-30, 180-182, 244-249). Es fällt auf, dass diese Aussagen vorwiegend bei der Interaktion mit *gnome-usability* und *konsole-devel* formuliert worden sind (vgl. Tabelle 2).

Alle *egw*-Aussagen erwähnen Software, die entweder für explizites Erlernen des Terminals gedacht ist oder sie beziehen sich auf graphische Schnittstellen zur Verwendung von Konsole-Funktionalitäten ohne Lerneffekt. Nur *kde*-Teilnehmer1 erwähnt neben einer *egw*-Aussage, dass er das Ziel meines Vorschlags, die Erlernbarkeit des Terminals zu erhöhen, bemerkt hat:

> I'm reminded strongly of Segusoland ... which allowed users to select actions on a fi e based on its mimetype. I know that there has been research done on learning the UNIX command line with automated tools (kde-Teilnehmer1, Z. 79-82).

Dieser Umstand bringt mich zu der Annahme, dass meine Werbestrategie „*SkKonsole* auch für *power-user*“ das Hauptziel der Erlernbarkeit des Terminals verdeckte (vgl. dazu Anhang D). Jedoch unterscheiden sich die *egw*-Aussagen vor allem durch ihren Kontext. Die Antwort des Konsole-*Maintainers* auf meinen Vorschlag folgt sehr eng dem Muster *egw*- und *avi*-Aussagen. Er vermied jedoch die Diskussion im Rahmen der Mailingliste, indem er mir mit einer privaten E-Mail antwortete. Dadurch wirkten seine Aussagen so, als wolle er dringend meine Aufmerksamkeit von der Mailingliste ablenken. Ich schickte ihm eine Antwort auf seine private E-Mail mit Kopie an die Mailingliste. Auf diese letzte Mail reagierten keiner der Mailinglisten-Teilnehmer. Interessanterweise antwortete der Konsole-*Maintainer* später im Rahmen der Mailingliste *kde-quality* auf meinen Vorschlag. Seine Aussagen (vgl. Konsole-*Maintainer*, Z. 88-92) waren diesmal deutlich positiver gegenüber meinem Vorschlag. Nun spielte er nicht mehr die Rolle des Konsole-*Maintainers*, sondern die eines *kde-quality*-Teilnehmers.

Wie diese widersprüchlichen Reaktionen des Konsole-*Maintainers* auf meinen Vorschlag zustande kam, ist sicherlich allein anhand der E-Mail-Antworten nicht zu beantworten. Jedoch könnte ein Blick auf einige Vorgänge der *konsole-devel* hilfreich sein, um den Kontext dieser Reaktion zu verstehen. Als ich *konsole-devel* abonnierte, gab es noch einen anderen *Maintainer*. Die Diskussion in der Mailingliste drehte sich fast ausschließlich um Fehlerbehebung. Dabei war der *Maintainer* der aktivste Mailinglisten-Teilnehmer und viele von seinen Nachrichten befassten sich mit organisato-

rischen Fragen. Einmal tauchte ein Fehler bei der Darstellung von Fettschriften bei gewissen Konfigurationen der Konsole auf. Dieser Fehler war aus früheren Versionen der Konsole nicht bekannt. Einige Teilnehmer der Mailingliste bestätigten das Vorhandensein des Fehlers. Der *Maintainer* versuchte zunächst zu klären, ob es sich um einen echten Fehler handelte. Binnen weniger Stunden häuften sich die affirmativen Meldungen. So war der *Maintainer* endlich davon überzeugt und kurz danach teilte er der Mailingliste mit, dass er einen *Patch* dafür auf die Dateiablagerung gespielt hätte. Anschließend merkten einige Teilnehmer an, dass die Konsole anders als sonst mit Schriften umgehe. Zu dieser Zeit meldete sich der Autor der Konsole zur Wort um zu erklären, dass der *Patch* falsch sei. Der *Patch* breche ein Grundkonzept der Konsole-Implementierung im Umgang mit Schriften. Deshalb übernahm die Verantwortung für weitere Untersuchungen über das Vorhandensein des Fehlers. Zwei Wochen später bestätigte der Autor die Existenz von solchen Fehlern. In der Zeit danach wurden die Meldungen des *Maintainers* bei *konsole-devel* immer seltener. Eines Tages kam eine E-Mail von ihm mit der Ankündigung, dass es einen neuen *Maintainer* gebe. Dies ist der jetzige Konsole-*Maintainer*, der auf meinen Vorschlag geantwortet hat.

Ob beide Geschehnisse zusammenhängen ist hier nicht ersichtlich, allerdings kommt der soziale Druck zur Vorschein, unter dem der *Maintainer* steht. Einerseits muss er den Teilnehmern den Eindruck vermitteln, dass er für sie da ist und ihre Anstrengungen würdigt. Andererseits muss er konservativ damit umgehen, um den technischen Vorstellungen des Autors nicht zu widersprechen. Dies könnte der Grund dafür sein, dass der neue *Maintainer* der Konsole meinen Vorschlag sofort sehr klar zurückwies und dies persönlich an mich tat, also die Interaktion gezielt aus der *konsole-devel*-Mailingliste heraus zu halten versuchte, als einfacher Teilnehmer in *kde-quality* jedoch positiv auf meinen Vorschlag reagierte.

Bei den anderen beiden Mailinglisten schien es keinen besonderen sozialen Druckfaktor zu geben. So formulierte *gnome*-Teilnehmer3 erst *egw*-Aussagen (gnome-Teilnehmer3, Z. 233-234) und dann eine *avi*-Aussage (gnome-Teilnehmer3, Z. 247-249). Dabei schlossen weder seine Aussagen noch deren Kontext weitere Diskussionen aus. Hierbei wird die unterschiedliche Natur der Mailinglisten deutlich. Meiner Meinung nach folgt *konsole-devel* dem typischen Muster von OSS-Projekten, während *kde-quality* und *gnome-usability* eher offene Diskussionsforen sind.

5.3.1.2 Die Kategorien „dadurch erreicht man nicht das angestrebte Ziel“ (naz) und „falsche Zielsetzung“ (fz)

Diese beiden Kategorien sind interessant wegen des Forums, in dem die Aussagen gemacht wurden. Auffällig ist, dass *naz*- und *fz*-Aussagen nur aus *gnome-usability* kamen. Ich stellte bei dieser Mailingliste nur eine Idee vor. So mussten sich die Teilnehmer eine Vorstellung von der Idee machen, ohne auf geschaffene Tatsachen zurückgreifen zu können. Wenn man jedoch von der Annahme ausgeht, dass auch die *kde*-Teilnehmer weder die Codebeilage lasen noch das Programm installierten und ausführten, dann besteht kein großer Unterschied zwischen den Ausgangspositionen aller Beteiligten. Dennoch scheint es naheliegender zu sein, bei einer bloßen Idee ihren Grundgedanken in Frage zu stellen, als wenn man weiß, dass ein entsprechendes Objekt bereits existiert.

Auch bei den *naz*-Aussagen war offenbar der Kontext der Interaktion entscheidend. Zum Beispiel wandte sich *gnome*-Teilnehmer2 an mich und begründete seine Argumentation sorgfältig (gnome-Teilnehmer2, Z. 163-167). Seiner Ansicht nach sei solch ein Programm umständlich zu bedienen, wo ich doch eine bessere Interaktion Benutzer-Terminal postulierte. Jedoch machte *gnome*-Teilnehmer2 diese Aussage nicht explizit. Im Gegensatz dazu formulierte *gnome*-Teilnehmer4 die Aussage explizit aus (gnome-Teilnehmer4, Z. 194-195). Er tat das aber nebenbei im Rahmen einer erregten Diskussion mit *gnome*-Teilnehmer3 über andere Themen. Es wäre also denkbar, dass mein Vorschlag für diese Teilnehmer lediglich einen Vorwand für eine andere, tendenziell Konkurrenz-

orientierte Auseinandersetzung lieferte und Ablehnung oder Zustimmung eher von den aktuellen sozialen Vorgängen in der Mailingliste abhing, statt von einer intensiven Reflexion meines Vorschlages zu zeugen.

Die *fz*-Aussagen wurden überwiegend in Form von Fragen oder bedingten Aussagen formuliert. So schreibt zum Beispiel *gnome*-Teilnehmer3:

> Are you suggesting it should be a goal to introduce users to the command line and encourage them to use it more often? (gnome-Teilnehmer3, Z. 236-237)

Die Tatsache, dass die Antwortenden in Form von Fragen oder bedingten Aussagen formuliert sind, liegt vermutlich daran, dass explizite *fz*-Aussagen die vorgeschlagene Idee grundsätzlich verwerfen würden. Es ist vorstellbar, dass die Diskussionsteilnehmer meine Kompetenz nicht so offensichtlich in Frage stellen wollten.

5.3.1.3 Die Kategorie „keine technische Betrachtung des vorgeschlagenen Codes“ (nbc)

Der Konsole-*Maintainer* gab in seiner ersten E-Mail explizit zu, dass er meine Codebeilage keines Blickes gewürdigt hatte „I will try out the code when I get a chance“(Konsole-*Maintainer*, Z.35). Damit waren weitere technische Diskussionen über den Code ausgeschlossen. Diese Tatsache macht deutlich, dass meine technischen Fertigkeiten als Einsteiger überhaupt keine Rolle für die Ablehnung spielten. Es gab keine weiteren expliziten Belege dafür, dass die anderen Teilnehmer von *kde-quality* oder *konsole-devel* die Codebeilage nicht gelesen hatten. Jedoch ist es schwer vorstellbar, dass mehrere Teilnehmer den Code gelesen hätten und dann keiner von ihnen darauf Bezug genommen hätte. Außerdem bezogen sich die Kommentare aus *kde-quality* ausschließlich auf die von mir veröffentlichte HTML-Seite. Eigentlich hätte ich Kommentare über eventuelle Mängel bzw. Vorzüge der *SkKonsole* erwartet.

5.3.2 Die Kategorien der sozialen Ablehnung

Unter der Rubrik „soziale Ablehnung“ sind alle Antworten in Kategorien zusammengefasst, die mit sozialen Verhältnissen in der *Community* argumentieren oder in anderer Weise meinen Vorschlag unter Bezug auf soziale Prozesse zurückweisen.

5.3.2.1 Die Kategorie „keine Zugehörigkeit zur Gesprächsgruppe“ (kzzg)

Diese Kategorie basiert im Gegensatz zu den restlichen Kategorien nicht auf Aussagen in den Antworten der Teilnehmer. Die Relevanz dieser Kategorie liegt in ihrer beispielhaften Illustration von Brands Postulat (Brand, 2003), dass „nur Personen, die sich am Projekt beteiligen, ernst genommen werden.... Von diesen Personen werden eher Vorschläge akzeptiert“ (Brand, 2003, S.21).

Es gab zwei verschiedene Ausprägungen des Auftretens der *kzzg*-Kategorie, einmal bei *konsole-devel* und einmal bei *gnome-usability*. Bei *konsole-devel* gibt es eine sehr explizite Zurückweisung. Allein die Tatsache, dass es im Rahmen der Mailingliste keine Antwort auf meinen Vorschlag gab, betrachte ich als totalen Ausschluss der Gesprächsgruppe. Das wurde durch die Antwort des Konsole-*Maintainers* noch deutlicher, indem er mir die Antwort als private Nachricht anstatt im Rahmen der Mailingliste zuschickte.

Bei *gnome-usability* diskutierten *gnome*-Teilnehmer3 und *gnome*-Teilnehmer4 über andere Themen und nahmen meinen Vorschlag nur zum Anlass ihrer Kommunikation. Sie diskutierten über andere Erweiterungsmöglichkeiten des Terminals „if we were serious about making the terminal more usable we should first encourage more distributions to enable spelling correction by default“ (gnome-Teinehmer3, Z. 180-182) und nur *gnome*-Teilnehmer4 bescherte nebenbei meinem Vorschlag einen

ablehnenden Satz „I don't think that the proposal here, or to add spell checking, will make [the terminal] more usable“(gnome-Teinehmer4, Z. 194-197). Anschließend antwortete *gnome*-Teilnehmer3 seinem Gesprächspartner und nachdem er seine Position ausführlich geschildert hatte, wendete er sich meinem Vorschlag zu (gnome-Teinehmer3, Z. 236-249). Dabei bekommt man beim Lesen seines Diskussionsbeitrags den Eindruck, dass *gnome*-Teilnehmer3 meinen Vorschlag nur deshalb anspricht, weil er in der Diskussion mit *gnome*-Teilnehmer4 die Rolle des GUI-Befürworters spielte.

5.3.2.2 Die Kategorie „falsches Forum“ (ff)

Diese Kategorie sollte Antworten erfassen, wie sie in der Literatur über den Einstieg in OSS-Projekte oft als Hürden für den Einsteiger beschrieben werden. So zitiert Ducheneaut (2005) zum Beispiel eine Webseite des Python-Projekts, auf der folgende Empfehlung für Einsteiger steht: „To work with this large and dispersed group, you'll have to learn who's the right person to answer a question“ (Python, 2006). Die Tatsache, dass nicht jeder mit jedem spricht, hatte ich schon zeitig erkannt, als ich am Anfang meiner Softwareentwicklung auf die Idee kam, eine E-Mail-Anfrage an den Konsole-Autor zu machen. Es kam keine Reaktion.

Mehrere Teilnehmer von *kde-quality* meinten, dass ihre Mailingliste das falsche Forum für meinen Vorschlag wäre, und verwiesen mich wieder an den Konsole-*Maintainer*, so z.B. der Angestellte: „as for making it into konsole itself, you should ask the current *maintainer* about that“ (Angestellter, Z. 63-64).

5.3.2.3 Die Kategorie „power-users versus newbies“ (puvn)

In der *Linux*-Welt ist es üblich über *power-users* zu sprechen. Dies soll erfahrene Anwender bezeichnen. Den Gegensatz dazu bildet der Begriff der *newbies*, der in etwa die restlichen Anwender meint. Die *puvn*-Kategorie erfasst ablehnende Aussagen auf meinen Vorschlag, die sich auf diesen Diskurs beziehen.

Die *puvn*-Aussagen beziehen sich einmal auf die Zielgruppe des vorgeschlagenen Programms und ein anderes Mal auf mich als Einsteiger. Zur Zielgruppe des Programms äußerte *gnome*-Teilnehmer4: „It's an application meant for a certain type of user. Just like Jokosher is meant for certain types of users“[10] (gnome-Teilnehmer4, Z. 196-197). *Gnome*-Teilnehmer4 meinte mit dem Ausdruck *“a certain type of user“* offenbar Benutzer, die keine ausführlichen technischen Kenntnisse haben. Also bezüglich des Diskurses *power-users* versus *newbies* meinte er die *newbies*. Aus der Antwort des *gnome*-Teilnehmer4 wird deutlich, dass er sich selbst als *power-user* definiert. Für meinen Vorschlag interessiert er sich unter anderem deshalb nicht, weil er meine Idee als für *newbies* gemacht betrachtet.

Gnome-Teilnehmer1 merkte in ironischem Tonfall zu meiner Präsentation an, dass das Bild auf der Webseite falsche Beschriftungen zum tar-Befehlsschalter enthielt. Tatsächlich hatte ich das in Abbildung 7.1 (siehe Anhang A und D) gezeigte Bild eingefügt, ohne die fehlerhaften Beschriftungen zu bemerken. Nach dieser Anmerkung verwendete *gnome*-Teilnehmer1 zur Kommentierung meiner Idee eine Bezeichnung, die unmittelbar an Kinder erinnert: „I will agree, it sounds interesting. But it seems to me to be the equivalent of training wheels“ (gnome-Teilnehmer1, Z. 156). Der Vergleich mit Stützrädern weist auf das Anfänger-Sein hin. Obwohl diese Aussage vordergründig sehr positiv bezüglich des Vorschlages formuliert ist, fällt der ironische Ton der gesamten E-Mail auf (vgl. *gnome*-Teilnehmer1, Z. 150-156).

---

10 Jokosher ist ein Musik-Programm, mit dem man Musik bearbeiten kann, ohne dafür besondere technische Kenntnisse haben zu müssen.

5.3.3 Die Kategorien der technischen Akzeptanz

Unter der Rubrik technische Akzeptanz sind alle Antworten in Kategorien zusammengefasst, die in technischer Hinsicht positiv oder anerkennend auf meinen Vorschlag eingehen.

5.3.3.1 Die Kategorie „gute Idee“ (gi)

Die Aussagen in dieser Kategorie können in zwei Gruppen eingeteilt werden. Die erste Gruppe besteht aus sehr eindeutigen Aussagen mit explizitem Ausdruck der Akzeptanz bezüglich meiner Idee. Aussagen in dieser Gruppe stammen nur aus *kde-quality*. Ein schlichter Satz wie: „I like your idea“ (kde-Teilnehmer2, Z. 98) gehört dazu. Der externe Angestellte äußerte sich auf ähnliche Weise dazu (Angestellter, Z. 51-52).

Die zweite Aussagengruppe besteht aus solchen, die zwar die Idee explizit akzeptieren, jedoch nicht eindeutig sind. Dazu gehören Aussagen, die eine Bedingung für die Akzeptanz stellen oder solche, die durch sprachliche Relativierungen eine unklare Semantik haben, wie etwa „The proposal might be a good idea for a project but if you really feel it is necessary to use the command line then...“ (gnome-Teilnehmer3, Z. 239-240). Es gibt auch positive Aussagen, die so allgemein und unverbindlich klingen, dass es fast unklar ist, ob sie zur Akzeptanz zu zählen sind oder eher eine freundlich verpackte Zurückweisung darstellen, wie z. B. die Aussage „I will agree, it sounds interesting“ (gnome-Teilnehmer1, Z. 156).

5.3.3.2 Die Kategorie „Vorschläge für die Weiterentwicklung“ (vfw)

Die Aussagen in dieser Kategorie stammen ausschließlich aus *kde-quality*. Sie sind bezüglich ihrer Semantik sehr eindeutig. Die technischen Vorschläge von *kde*-Teilnehmer2 (kde-Teilnehmer2, Z. 100-104) und von dem Angestellten gehören dazu „you may be able to expand the content in the commands.xml file by processing output for $COMMAND –help“ (Angestellter, Z. 54-58). Jedoch beinhaltet die Aussage des Angestellten neben einem Vorschlag (Angestellter, Z. 54-55) auch eine Kritik bezüglich meiner Strategie (Angestellter, Z. 57-58).

5.3.4 Die Kategorien der Umfeld-bezogenen Akzeptanz und Ablehnung

Nur wenige Aussagen ließen sich den Kategorien *Vorschlag unpassend für Projektzustand (vup)* und *Benutzer/ Programmierer mögen das nicht (bmdn)* in der Rubrik Umfeld-bezogene Ablehnung bzw. *Benutzer/ Programmierer mögen das (bmd)* in der Rubrik Umfeld-bezogene Akzeptanz zuordnen. Deshalb stelle ich diese Kategorien hier zusammenfassend vor.

In der *bmd*-Kategorie kommen zwei Aussagen vor. In einer formuliert der Angestellte lediglich aus, dass er der Meinung ist, dass die Benutzer diese Idee sicherlich interessant fänden „i think many of our users would really appreciate such an addition“ (Angestellter, Z. 51-52). Ansonsten gibt es eine Aussage des Konsole-*Maintainers*, die ich sowohl der Kategorie bmdn als auch der Kategorie bmd zugeordnet habe, weil es eine bedingte Aussage war „If [the *SkKonsole*] proves popular it could be added to Konsole in future “(Konsole-*Maintainer*, Z. 89-90). Die Aussage des Konsole-*Maintainers* ist unter anderem deshalb interessant, weil sie Meinungsumschwung des *Maintainers* deutlich macht. Dieser Vorgang wird in Kapitel 6.1 noch einmal ausführlich beschrieben und diskutiert.

6 Zusammenfassung und Diskussion der Ergebnisse

Ziel dieser Fallstudie war es, anhand der Beobachtung einer Interaktion zwischen zwei OSS-Projekten und einem Einsteiger exemplarisch zu überprüfen, ob das Beilegen von Code von Vorteil für die Annäherung an bzw. den Einstieg in ein OSS-Projekt ist. Weiterhin sollte an dem Einzelfall herausgearbeitet werden, welche Argumente und Fakten in der Diskussion in den Mailinglisten eine Rolle bei der Ablehnung bzw. Akzeptanz des Vorschlags spielten. Damit sollten die zu beachtenden Feinheiten beim Einstieg in ein OSS-Projekt deutlich beleuchtet werden.

Als Hauptergebnis der empirischen Untersuchung lässt sich festhalten, dass das Beilegen von Code zu der vorgeschlagenen Idee günstiger war, um die Aufmerksamkeit der Projektteilnehmer zu gewinnen, als das Vorstellen der bloßen Idee ohne Code.

Zugleich wurde deutlich, dass die technischen Argumente, selbst beim Vorhandensein von Code-Beilage, ausschließlich auf eigenen allgemeinen technischen Ansichten der Gesprächspartner beruhten. Das zeigte sich unter anderem daran, dass die Teilnehmer der Mailinglisten die Code-Beilage gar nicht angeguckt hatten. Dabei waren die geäußerten technischen Ansichten oft an sozial-hierarchische Faktoren gebunden. So prägte der *Power-users*-versus-*newbies*-Diskurs einige ablehnende Argumente bzw. der Konsole-*Maintainer* äußerte sich erst positiv zum Vorschlag, nachdem ein anderer wichtiger Projekt-Teilnehmer sich dazu positiv geäußert hatte.

Die Antworten waren oft unreflektiert und deshalb nicht wirklich hilfreich für mich als Einsteiger. Die ablehnenden Aussagen der Kategorie *es gibt schon so etwas* bezogen sich auf Software, die sehr wenig mit meinen Vorschlag zu tun hatte. Ob diese Antworten lediglich als Strategie für eine höfliche Ablehnung gedacht waren, ist schwierig zu beurteilen, Tatsache ist, dass ein Informationsbedürftiger Einsteiger damit schlecht bedient wäre.

Im Folgenden sollen noch einmal einzelne Ergebnisse meiner Studie diskutiert werden, die für die weitere Untersuchung dieses Themas besonders interessant sein könnten. Dazu beziehe ich mich auch immer wieder auf den in Kapitel 2 dargestellten Stand der Fachliteratur zum Thema Einstieg in OSS-Projekte.

6.1 Diskussion ausgewählter Ergebnisse

Eine erste Auffälligkeit an den Ergebnissen war, dass in beiden Mailinglisten die Aussagen der Kategorie *es gibt schon so etwas (egsw)* stark vertreten waren. Wenn man zwei Programme im Detail vergleicht, findet man es selten, dass beide Programme genau die selben Funktionalitäten anbieten. Gerade das spezifische Zusammenspiel der Funktionalitäten eines Programms macht seinen Zweck aus. Trotz allem trat diese Antwort sehr oft auf. Allerdings erfüllte keines der Programme, die meine Diskussionspartner mit meiner *SkKonsole* bzw. dem Vorschlag verglichen, den selben Zweck wie die *SkKonsole*. Einige von den genannten Programmen boten die Funktionalitäten des Terminals über graphische Schnittstellen an. Zum Beispiel nannte der Konsole-*Maintainer* als alternative Entwicklung *WorKflow*. Dabei strebt *WorKflow* an, die selben Funktionalitäten anzubieten wie der Automator von Mac[11]. Andere Programme, die in den Antwortmails als Alternativen zur *SkKonsole* benannt wurden, waren interaktive Tutorials für explizites Lernen wie *vilearn*. Im Gegensatz dazu bietet die *SkKonsole* die Möglichkeit des impliziten Lernens beim Arbeiten an (*learning by doing*). Diese Unterschiede zwischen *SkKonsole* und den angeführten Programmen schienen mir sehr augenfällig, sobald ich die Informationen zu den erwähnten Programmen las. Es ist jedoch denkbar, dass ich diese Unterschiede so deutlich wahrnahm, weil ich mit meiner Implementierung und deren Zielen vertraut war. Sicherlich hätten sich auch meine Gesprächspartner in

11 „The goal of this project is to implement a user-friendly application, called WorKflow, similar to Automator for the KDE desktop“. (URL: http://code.google.com/soc/kde/appinfo.html?csaid=BD9ED2FC13E07E4C)

kurzer Zeit mit dem *SkKonsole*-Projekt vertraut machen können. Das taten sie jedoch nicht. Offenbar hatten sie lediglich meine Webseite mit der Vorstellung des Projekts gelesen, sich eine grobe, verallgemeinernde Vorstellung davon gemacht und dann assoziativ nach Ähnlichem gesucht. Damit war dann die Beschäftigung mit meinem Vorschlag abgeschlossen.

Vielleicht wäre das anders gewesen, wenn sich mein Vorschlag auf ein Problem bezogen hätte, das zuvor in der Mailingliste diskutiert worden war. Am Beispiel der Mailingliste *konsole-devel* hätte das etwa heißen können, dass ich ein neues Modul angeboten hätte, das die Darstellung von Schriftarten ohne das erwähnte Problem[12] für die Konsole regelt (siehe Kap. 5.3.1.1). Der Konsole-*Maintainer* war mit dieser Problematik vertraut. Deshalb es in diesem Fall sehr wahrscheinlich gewesen, dass er, anstatt eine vorschnelle egsw-Aussage zu machen, die Codebeilage inspiziert und weitere akzeptierende oder ablehnende technische Kommentare dazu gemacht hätte.

Offensichtlich illustriert dieser Punkt den Unterschied zwischen dem, was von Krogh et al. (2003) „unsolicited 'new' technical suggestions“ (S. 1228) nennen und einem aktuell gebrauchten oder erwünschten Vorschlag mit entsprechenden Erfolgsaussichten. Die Notwendigkeit, die Diskussionen in der Mailingliste eine Weile zu beobachten, bevor man sich das erste Mal selbst mit einem Kommentar oder Vorschlag an die Liste wendet, wird in der Literatur mehrfach erwähnt (vgl. Ducheneaut, 2005, von Krogh et al., 2003). Dennoch erfordert eine sinnvolle Beobachtung der Diskussionen neben allgemeinen Kenntnissen und dem Beherrschen der Sprache auch konkrete Kenntnisse über die Implementierung. Kenntnisse über Grundkonzept, Strategien, umgebende Architektur und Kodierungs-Konventionen sind dabei entscheidend (Ducheneaut, 2005). So konnte ich während der ersten Wochen meiner Beobachtung der Mailinglisten zwar die Architektur und das Grundkonzept erkennen, meine konkreten Kenntnisse über die Implementierung beschränkten sich jedoch auf die Code-Abschnitte, die ich aus Anlass der aktuellen Diskussionen las. Dabei wird noch einmal deutlich, was für einen zeitlichen und inhaltlichen Aufwand bereits solche basalen, beinahe nebensächlich klingenden, Voraussetzungen bedeuten. Jedenfalls verfügte ich im Rahmen meiner Bachelorarbeit nicht über genügend Zeit, um mich in dieses Thema ausführlich genug einzuarbeiten.

In auffälligem Gegensatz zu der genannten Anforderung an den Einsteiger, die Mailingliste zunächst zu beobachten, würdigten die Diskussionsteilnehmer von *konsole-devel,* und vermutlich auch die von *kde-quality,* meine Code-Beilage keines Blickes „I will try out the code when I get a chance“ (Konsole-*Maintainer*, Z.35) (siehe Kap. 5.3.1.3). Diese Tatsache widerspricht dem in der Literatur angeführten Lehrlingsprinzip (siehe Kap. 2.2.2), bei dem postuliert wird, dass der Einsteiger dann in das OSS-Projekt aufgenommen wird, wenn seine Arbeit von den Teilnehmern als ausreichend gut bewertet wird. Denn das würde ja eben voraussetzen, das jemand (der Meister) ein Lehrlingsstück überhaupt erst einmal anguckt und auf Richtigkeit überprüft und danach entscheidet.

Im Gegensatz dazu richteten sich einige Teilnehmer meiner Untersuchung offenbar nach sozialen Faktoren, um ihr Urteil zu fällen. Das lässt sich zum Beispiel am Meinungsumschwung (siehe Kap. 5.3.4) des *Maintainers* illustrieren. Der *Maintainer* reagierte zunächst auf meinen Vorschlag (mit beigelegtem Code) an *konsole-devel* in einer privaten E-Mail an mich und äußerte explizit, dass es schon viele Initiativen mit diesem Ziel gebe und implizit, dass keine von denen eine denkbare Erweiterung der Konsole sei „I think however there are a couple of projects which are addressing the need to bring the power of UNIX tools to newcomers more comprehensively“ (Konsole-*Maintainer*, Z. 23-24). Als ich mich später jedoch noch an *kde-quality* wendete, erklärte der Konsole-*Maintainer*, dass mein Vorschlag doch eine mögliche Erweiterung für die Konsole sein könnte, allerdings nur unter der Bedingung, dass die *SkKonsole* unter den KDE-Benutzern populär würde. Dabei ist es interessant, die Umstände dieses Meinungsumschwungs zu ein wenig näher zu betrachten:

12 Das Problem bestand darin, dass die Konsole bei einigen Schriftarten die Fettschriften nicht darstellen konnte.

Nach meiner Antwort (siehe Anhang D) an den Konsole-*Maintainer* auf *konsole-devel* meldete er sich nicht mehr zurück. Doch dann schickte der Angestellte auf *kde-quality* eine meinem Vorschlag zustimmende E-Mail: „i do think it's a good idea“ (Angestellter, Z. 64). Da schloss sich der Konsole-*Maintainer* plötzlich an und vertrat nun also das Gegenteil seiner früheren Meinung „I think Angestellter's idea of publishing the prototype on kdeapps.org and getting feedback is the best way to go. If it proves popular it could be added to Konsole in future“ (Konsole-*Maintainer*; Z. 88-89). Es lässt sich vermuten, dass der Konsole-*Maintainer* die Meinung des Angestellten hoch schätzte. Dieser Art sozial-hierarchische Interaktions-Phänomene sind offensichtlich relevant für Entscheidungsprozesse in OSS-Projekten und ein Einsteiger sollte im optimalen Fall solche Zusammenhänge kennen. Den Einfluss der Angestellten auf das KDE-Projekt kommentiert Brand (2003) wie folgt:

> So läßt sich ... sagen, daß der innere Kreis [des KDE-Projektes] im Moment einer „gewachsenen Oligarchie“ bzw. informellen Hierarchie durch sozialen Status der älteren und anfänglichen Projekt-Beteiligten gleicht, die durch bestimmte Mailinglisten zusammengehalten werden. Ihre Einflußmöglichkeit wird durch ihre Reputation und die Übernahme von gemeinschaftserhaltenden Posten erhöht. Dabei spielen die angestellten Großprojekt-Programmierer eine wichtige Rolle in dieser Kerngruppe. (Brand, 2003, S. 29)

Ein weiterer sozialer Faktor, der an den Antworten deutlich wurde, war der Einfluss des *Power-users*-versus-*newbies-(puvn)*-Diskurses auf meine Interaktion mit der *gnome-usability* Mailingliste. *Gnome*-Teilnehmer1 merkte in seiner E-Mail einen inhaltlichen Fehler auf meiner Webseite mit dem Vorschlag an und formulierte seine E-Mail auf eine Weise, dass ich die Unterstellung heraushörte, ich sei ein *newby* (siehe Kap. 5.3.2.3). Interessanterweise bezog ich mich als Einsteiger in meiner Antwort an *gnome*-Teilnehmer1 ohne groß darüber nachzudenken auch auf den *puvn*-Diskurs (siehe Anhang D). Wo ich schrieb „the interaction between beginner and terminal is quite hard for beginners“, hätte ich eine differenziertere Ausdrucksweise verwenden können. Denn insbesondere in der Informatik ist es möglich, dass ein Software-Ingenieur zwar eine spezielle Sache nicht kennt, jedoch im selben Bereich über eine große Anzahl von Kompetenzen verfügt.

Auch auf meiner Webseite machte ich explizit Gebrauch vom *puvn*-Diskurs: „As power users we believe that the console has it's strengths and allows for fast and powerful interaction“ (Anhang D). Dabei wird deutlich, dass diese Verwendung des Diskurses hier implizit auf Gruppenbildung abzielt und insbesondere dafür wirbt, mich als Einsteiger auch als Gruppenangehörigen zu betrachten.

Der *puvn*-Diskurs tauchte noch mehrfach in der Diskussion auf *gnome-usability* auf, meistens als indirekt formulierte Unterstellung:

> >It's an application meant for a certain type of user. [(Teilnehmer4 zu Teilnehmer3)]
>
> Duh. Thank you captain obvious. [(Teilnehmer3-Antwort)](gnome-Teilnehmer3, Z.223-224)

In diesem Fall handelte es sich um eine abwertende *puvn*-Aussage mit ausschließendem Charakter. Jedoch zielt die Aussage hier eher auf eine hierarchische Einordnung der Teilnehmer ab als auf den Ausschluss von *gnome*-Teilnehmer3. Das wird ersichtlich an der ironischen Formulierung „captain“, mit der *gnome*-Teilnehmer3 die hierarchische Einordnung als unangemessen zurückweist. Offensichtlich lässt sich der *puvn*-Diskurs für Einschluss in bzw. Ausschluss aus Gruppen sowie für hierarchische Einordnung in OSS-Projekten benutzen. Dabei versuchen die Diskussionsteilnehmer unter Anspielung auf den Diskurs ihre eigenen Interessen durchzusetzen. Es ist naheliegend, dass das eine wichtige soziale Hürde für den Einstieg sein kann.

Die Unterschiede in der Interaktion mit den beiden Mailinglisten können nicht nur auf die Code-Beilage zurückgeführt werden. Das macht schon der Vergleich zwischen den Interaktionen mit *konsole-devel* und *kde-quality* deutlich (siehe Kap. 5.2). Bei *konsole-devel* habe ich nur eine private und ablehnende Antwort erhalten, während bei *kde-quality* durchaus einige positive und ermutigende Feedbacks kamen.

Dabei fällt zunächst der unterschiedliche Zweck der beiden Mailinglisten auf. Während *konsole-devel* sich der Fehlerbehebung und Weiterentwicklung des Konsole-Projekts widmet, verfolgt *kde-quality* allgemeinere Ziele[13]. Die Mailingliste *kde-quality* strebt die Optimierung der Benutzbarkeit von KDE an und die Unterstützung von Teilnahmewilligen[14]. Auf der Webseite der Mailingliste findet man konkrete Empfehlungen für Einsteiger, zum Beispiel wird dort davon abgeraten, sich mit neuen Vorschlägen an Entwicklungs-Mailinglisten zu wenden (kde-quality Pr, 2006). Offensichtlich wird im Rahmen von *kde-quality* die Frage des Einstiegs von Neulingen aktiv wahrgenommen und positiv begleitet. Jedoch ist es dort zum Beispiel nicht möglich, CVS-Zugriff zu bekommen.

Die Möglichkeiten zur Kontaktaufnahme mit KDE entsprechen somit nicht denen, die in der Literatur benannt werden. Solche Mailinglisten wie *kde-quality* tauchten in der von mir durchforsteten Literatur gar nicht auf, sondern es ging immer nur um die Entwicklungs-Mailinglisten. Das erschwerte die Durchführung meiner Studie, weil die Untersuchung von zentralen Themen der Einstiegs-Literatur, wie das Erlangen von CVS-Zugriff, in Rahmen von *kde-quality* gar nicht möglich[15] war. Dabei hieß es auf der *kde-quality*-Webseite von KDE, *kde-quality* sei das richtige Forum für Vorschläge (kde-quality, 2006).

Jedoch bekam ich auch bei *kde-quality* für meinen Vorschlag lediglich eine Empfehlung. Ich sollte mein Programm veröffentlichen und dabei versuchen, das Programm populär zu machen (Angestellter, Z. 60-61). Die Veröffentlichung sollte ich auf einer KDE-Webseite machen, (http://kde-apps.org), die allen zugänglich ist und für die Veröffentlichung von Projekt-externen Entwicklungen bestimmt ist. Damit war ich als Einsteiger der gewünschten Projektteilnahme keinen Schritt näher gekommen. Diese Tatsache relativiert das erste Ergebnis dieser Fallstudie, dass das Beilegen von Code günstiger war. So müsste man präzisieren, dass die Code-Beilage die Einstellung der Diskussionsteilnehmer gegenüber meinem Vorschlags positiv beeinflusste, jedoch für eine Einstiegsperspektive praktisch keine Bedeutung hatte.

Die Tatsache, dass im KDE-Projekt die Schwierigkeit des Einstiegs ernst genommen wird, lässt sich an der *kde-quality*-Website (kde-quality Pr, 2006) erkennen. Unter dem Titel „Communication And Promotion Guide" informiert *kde-quality* die Einstiegswilligen und die Teilnehmer, dass es die Problematik des Einstiegs gibt und welchen Umgang die Teilnehmer mit den Einstiegswilligen haben sollten. Jedoch zeichnet *kde-quality* dabei ein recht stereotypes Bild des Szenariums. So seien die Vorschläge eines Einsteigers meist naiv, weil es in der Regel schon so etwas gebe und selbst wenn das nicht der Fall sei, seien die Entwickler zu beschäftigt, um solche Vorschläge wahrzunehmen. Ansonsten gebe es genau für diese Fälle eben die Mailingliste *kde-quality* und das Portal *kde-apps.org*.

Sicherlich ist an solch einer Lagebeschreibung einiges dran. Allerdings wird damit offenbar ein einschlägiger Diskurs über den Einstieg von Neulingen wiedergegeben, nach dessen Muster auch der Konsole-*Maintainer* auf meinen Vorschlag reagierte. Interessanterweise antworteten die Teilnehmer von *gnome-usability* auf eine ganz ähnliche Weise. Das legt die Vermutung nahe, dass es sich hierbei eher um einen typischen Diskurs in OSS-Projekten handelt.

Aus all dem lässt sich schließen, dass die Einstiegsstrategie, Code bei Kontaktaufnahme beizulegen in weit entwickelten Großprojekten wie KDE keine Erfolgsaussichten hat. Nach den Erfahrungen meiner Studie würde ich vermuten, dass die erfolgreichen Einsteiger, die Code beigelegt hatten, von denen von Krogh et al. (2003) berichten (vgl. Kap. 3), keine völligen Neulinge wie z.B. ich waren.

Nach meinen Ergebnissen gehe ich davon aus, dass ein langsamer Einstieg, wie ihn z.B.

13 *gnome-usability* verfolgt ähnliche Ziele wie *kde-quality.*

14 Teilnahmewillige, gleichgültig ob sie Projekt-Entwickler werden wollen oder nicht.

15 *kde-quality* unterhält keine Dateiablagerung für reine Entwicklungsarbeit. Die ankommenden Vorschläge beziehen sich meistens auf KDE-Unterprojekte und es gibt keinen klaren Rahmen für die Weitergabe von Vorschlägen an Unterprojekte.

Ducheneaut (2005) beschreibt (siehe Kap. 2.2.4), eher möglich sein würde. Das hieße z.B. zunächst zu beobachten (*to lurk*), an Diskussionen teilzunehmen, in *kde-apps* eigene Softwareentwicklungen zu veröffentlichen und nebenbei *patches* in einem Unterprojekt anzubieten, schließlich CVS-Zugriff zu erlangen und dann eigene technische Interessen im Projekt zu verfolgen.

Die Forschung zum Einstieg in OSS-Projekte steckt immer noch in den Kinderschuhen. Empirische Untersuchungen, wie diese Fallstudie, finden eine Landschaft mit vielfältigen und komplexen sozial-technischen Faktoren vor, die wichtige Entscheidungen, wie zum Beispiel die Auswahl des Zielprojekts, schwierig machen. Die Anlehnung an die vorhandene Literatur hilft hierbei wenig. Eine Kategorisierung von OSS-Projekten bezüglich verschiedener Aspekte, wie zum Beispiel Projektzustand (frühes, mittleres oder ausgereiftes Stadium), sozial-politische Verhältnisse oder Schwierigkeitsgrad der technischen Anforderungen, wäre dafür nötig. Man findet zwar implizit solche Kategorisierungen in der Literatur (siehe Tjøstheim & Tokle, 2003), jedoch betrachten diese Autoren nur einzelne Aspekte von wenigen OSS-Projekten. Die erzielten Erkenntnisse und die beobachtbaren Fakten einer solchen Fallstudie variieren, abhängig vom einzelnen Zielprojekt, stark. So gibt es zum Beispiel Studien über kleine OSS-Projekte (weniger als 10 Entwickler) im Frühstadium, wo die Forscher mit einer einfachen Anfrage CVS-Zugriff erlangten und daraufhin Vorschläge für die Weiterentwicklung machen durften (siehe Tjøstheim & Tokle, 2003). Meiner Meinung nach wäre beispielsweise ein Vergleich zwischen meiner Studie und einer solchen Studie ohne Bezug auf die Unterschiede der Zielprojekte nicht aussagekräftig.

6.2 Reflexion

Im Laufe dieser Fallstudie zeigte sich, dass meine Fragestellung theoretisch zu schwach untermauert war. Einige der anfänglichen Entscheidungen (z.B. bezüglich des Zielprojekts und der Technologien) über das Vorgehen im Zuge der Studie entpuppten sich als Fehleinschätzungen, u.a. weil sie meine Interaktionsmöglichkeiten als simulierter Einsteiger zu sehr eingeengten. Diese Tatsache implizierte, dass es keinen Raum für interessante Interaktionen gab, wie zum Beispiel CVS-Zugriff zu verlangen.

Die anfängliche Festlegung auf den Inhalt des Vorschlags war zwar grundlegend für die Entwicklung der Fragestellung dieser Studie. Nach der Literatur hätte man aber zuerst das OSS-Projekt genauer beobachten sollen, um zu bestimmen, welchen Bedarf an Softwareerweiterungen es aktuell im Projekt gibt (vgl. Krogh, 2003). Außerdem sollte der Vorschlag, so weit es geht, wenig Abhängigkeiten von der bestehenden Software haben, um die Idee besser verkaufen zu können.

7 Ausblick

In meiner Fallstudie habe ich die Interaktionen mit den Teilnehmern der Mailinglisten *kde-devel*, *kde-quality* und *gnome-usability* detailliert nachgezeichnet. Dabei hat sich ein Bild der Kontaktaufnahme ergeben, das eher den recht Hierarchie-bezogenen Hinweisen auf der *kde-quality*-Website (kde-quality Pr, 2006) entspricht, als dem in der Fachliteratur beschriebenen Einstiegs-Szenario, das von einem möglichen Erstkontakt mit Entwickler-Mailinglisten ausgeht.

Weitere Forschung in Bezug auf sinnvolle Strategien für den Einstieg in OSS-Projekte sind dringend erforderlich. Dabei sind in jedem Fall die z.T. gravierenden Unterschiede in Bezug auf Projektzustand, Projektgliederung und sozial-politische Verhältnisse im Projekt zu beachten.

Die Großprojekte KDE und Gnome haben eine Strategie für den Umgang mit Neuankömmlingen. Sie betreiben dafür Mailinglisten und apps-Seiten (http://gnome-apps.berlios.de, http://kde-apps.org). Eine interessante Frage für weitere Untersuchen wäre, inwiefern solche apps-Seiten Einsteigern ins Projekt verhelfen. Sind solche Foren Orte für die Rekrutierung von Projektmitgliedern oder eher eine Sackgasse für Einstiegswillige?

Literaturverzeichnis

Bergquist, M. & Ljungber, J. (2001). The power of gifts: organizing social relationships in open source communities. URL: http://www.blackwell-synergy.com/doi/abs/10.1046/j.1365-2575.2001.00111.x [August 2006].

Brand, A. (2003). Fallstudie horizontales elektronisches Arbeitsnetz/ Open Source-Projekt. URL:http://www.soz.uni-frankfurt.de/arbeitslehre/pelm/docs/FALLSTUDIE%20Open%20Source%20V1.pdf#search=%22Fallstudie%20horizontales%20elektronisches%20Arbeitsnetz%2FOpen%20Source-%20%0AProjekt%22 [August 2006].

Döring, N. (2003). Sozialpsychologie des Internet. Göttingen: Hogrefe-Verlag GmbH & Co.

Ducheneaut, N. (2005). Socialization in an Open Source Software Community: A Socio-Technical Analysis. Computer Supported Cooperative Work (CSCW), 14, 4, S. 323-368. URL: http://dx.doi.org/10.1007/s10606-005-9000-1 [August 2006].

Edwards, K. (2001). Epistemic Communities, Situated Learning and Open Source Software Development. URL: http://citeseer.ist.psu.edu/edwards01epistemic.html [August 2006].

Fallstudie. (2006). In Wikipedia. Die freie Enzyklopädie. URL: http://de.wikipedia.org/wiki/Fallstudie [August 2006].

Flick, U. (2000). Qualitative Forschung – Theorie, Methoden, Anwendung in Psychologie und Sozialwissenschaften. Reinbek bei Hamburg: Rowohlt.

Gnome. (2006). Gnome. URL: http://www.gnome.org [August 2006].

kde-quality Promotion. (2006). Communication And Promotion Guide. URL: http://quality.kde.org/develop/howto/howtopromo.php [September 2006].

kde-quality. (2006). KDE Quality Team. URL: http://quality.kde.org/ [August 2006].

KDE. (2006). K Desktop Environment. URL: http://kde.org [August 2006].

Lüders, C. (2000). Beobachten im Feld und Ethnogaphie. In: U. Flick, E. v. Kardorff & I. Steinke (Hrsg.). Qualitative Forschung. Ein Handbuch. (S. 384-401). Reinbek bei Hamburg: Rowohlt.

Maass, W. (2004). Inside an Open Source Software Community: Empirical Analysis on Individual and Group Level. In: 4th Workshop on Open Source Software Engineering at 26th Int. Conf. on Software Engineering (ICSE 2004). Edinburgh, UK. URL: http://www.alexandria.unisg.ch/Publikationen/12837 [August 2006].

Mahendran, D. (2002). Serpents and Primitives: An Ethnographic Excursion into an Open Source Community. Unveröffentlichte Master Thesis, Berkeley, CA: University of California, Berkeley.

Mayring, P. (2002). Qualitative Sozialforschung. Weinheim: Beltz.

Moon, J. Y. & Sproull, L. (2000). Essence of Distributed Work: The Case of the Linux Kernel. First Monday, 5, 11, URL: http://www.firstmonday.org/issues/issue5_11/moon/ [August 2006].

Open Source Initiative (OSI). (2006). The Open Source Definition. URL: http://www.opensource.org/docs/definition_plain.php [August 2006].

Python. (2006). The Python Project's Web Site. URL: http://www.python.org [August 2006].

Raymond, E. S. (1997). The Cathedral & the Bazaar. URL: http://www.catb.org/~esr/writings/cathedral-bazaar/ [August 2006].

Spaeth, S. (2005). Coordination in Open Source Projects – A Social Network Analysis using CVS data. URL: http://www.biblio.unisg.ch/www/edis.nsf/wwwDisplayIdentifier/3110 [August 2006].

Tjøstheim, S. & Tokle, M. (2003). Acceptance of new Developers in OSS projects. www.idi.ntnu.no/grupper/su/su-diploma-2003/Tjostheim_FTokleaFOSS_acceptance.pdf [August 2006].

Trolltech. (2006). Technical Documentation: Technical documentation and other resources for using Trolltech's products. URL: http://www.trolltech.com/developer/documentation/index [August 2006].

von Krogh, G., Spaeth, S. & Lakhani, K. (2003). Community, Joining, and Specialization in Open Source Software Innovation: A Case Study. Research Policy, Bd. 32, Nr. 7, S. 1217– 1241.

Anhang A
Das Software-Projekt

Ziel des Software-Projekts war es, den Benutzer bei der Benutzung und beim Erlernen des Terminals zu unterstützen.

Geschichte des Projektes

Mein Projekt lehnte sich an ein älteres Projekt namens „*Skinning the Command Line*“ an, von dem ich Grundidee und Grunddokument für mein Projekt erbte. Im Zuge dieses Projektes war ein Prototyp namens *SkTerm* auf Basis der Terminal-Simulation *xterm* entstanden. Dieser Prototyp beherrschte nur eine kleine Anzahl von Terminal-Kommandos.

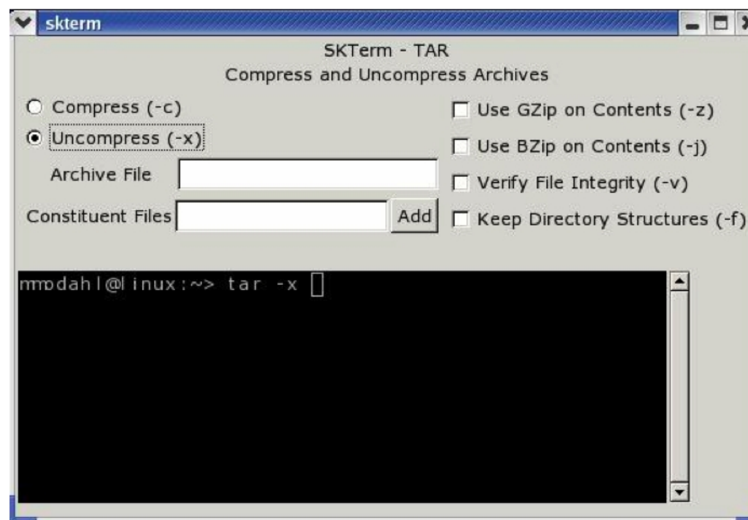

Abbildung 7.1: Das Programm SKTerm bestand aus einem Terminal und den graphischen Steuerelementen, die den gerade eingetippten Terminal-Befehl beschrieben. (Die Beschriftung der tar-Optionen in der Abbildung ist falsch. Das hatte Konsequenzen für meine Untersuchung (vgl. Kap. 5.3.2.3)).

Der Prototyp *SkTerm* war zu Beginn meines Projektes nicht mehr vorhanden. Aber das Grunddokument beinhaltete unter anderem einen Anforderungskatalog für den Prototypen, nach dem sich auch meine Entwicklung richten sollte:

1. Die GUI erscheint, wenn der Benutzer ein dem Programm bekanntes Kommando in die Kommandozeile eingibt.
2. Jedes graphische Steuerelement stellt einen Schalter des aktuellen Kommandos dar und zeigt kurze und erklärende Informationen zur Funktion des Schalters an.
3. Die Betätigung eines Steuerelements bewirkt eine semantisch sinnvolle Zustandsänderung der Kommandozeile. Beispiel: Wird die `Checkbox` des Schalters `-a` aktiviert, so erscheint der Schalter `-a` an der richtigen Stelle in der Kommandozeile. Wenn die `Checkbox` wieder deaktiviert wird, verschwindet das `-a`.
4. Analog dazu: Wenn der Benutzer einen auf der GUI dargestellten Kommando-Schalter in die Kommandozeile eingibt, soll das graphische Steuerelement aktualisiert werden.
5. Diese Anwendung soll Konfigurationsmöglichkeiten anbieten. (vgl. Anhang D)

Die Entscheidung über die Technologien

Die erste Entscheidung über Technologien war das Auswählen einer Terminal-Simulation als Basis für meine Entwicklung. Zur Auswahl standen Konsole von KDE (2006) und Terminal von Gnome (2006). Ich habe mich für Konsole entschieden.

Die KDE Entwicklung basiert auf *Qt*-Biblioteken (Trolltech, 2006). *Qt*-Bibliotheken sind prinzipiell für Entwicklung von graphischen Oberflächen in C++ auf verschiedenen Plattformen angelegt. KDE selbst bietet auch eine eigene API mit Klassen an, die oft Spezialisierungen von *Qt*-Bibliotheken für Entwicklungen im Rahmen des KDE-Desktops anbieten. Zwei nennenswerte eigene Konstrukte von KDE sind `Dcop` und `KParts`. Dcop ist eine *Corba*-ähnliche Grundkomponente von KDE und dient der Kommunikation zwischen KDE Anwendungen zur Laufzeit. `KParts` ist ein Rahmenwerk für die Arbeit mit graphischen Komponenten von KDE. `KParts` erlaubt die Integration und weitere Verwendung von graphischen Komponenten des KDE-Desktops in selbst entwickelte Programme.

Als Entwicklungsprogramm bot sich *Kdevelop* an, weil *Kdevelop* hauptsächlich für KDE-Entwicklung gedacht ist.

Die ersten Schritte

Ich hatte meine Untersuchung als teilnehmende Beobachtung geplant. Das bedeutete, ich führte meine Studie durch, indem ich die Rolle eines Einstiegswilligen in ein OSS-Projekt spielte. Zuvor hatte ich die Diskussionen in den entsprechenden Mailinglisten schon einige Zeit verfolgt.

Als OSS-Einsteiger war ich (wie in der Realität) ein Neuling auf diesem Gebiet. Ich hatte keine Kenntnisse über KDE- und *Qt*-Technologien. Selbst die Entwicklungssprache (C++) beherrschte ich nur auf einem Beginner-Niveau. Andererseits war mein Interesse am KDE-Projekt groß und damit meine Motivation sehr hoch. Deshalb bereitete ich mich bereits einige Monate vor Beginn meines Projektes auf die Technologien von KDE vor.

In dieser Zeit erforschte ich unter anderem die Struktur der Konsole-Implementierung. Ich versuchte vergebens, ein *Interface* mit den gewünschten Funktionalitäten zu finden. Eine weitere Möglichkeit war das `KParts`-Rahmenwerk. Allerdings hätte die Verwendung von `KonsolePart` die gewünschte Einbettung meiner GUI in die Konsole nicht gewährleistet. `KonsolePart` bietet lediglich eine fensterlose Kommandozeile an, ohne die zusätzlichen graphischen Steuerelemente von Konsole. Eine Erweiterung der Konsole per Vererbung kam wegen ihrer Komplexität nicht in Frage.

So habe ich mich dafür entschieden, einen Prototypen zu bauen, der ein Edit-Feld als Ersatz für die Kommandozeile hat. In Anlehnung an *SkTerm* sollte mein Programm *SkKonsole* heißen. Als Erläuterung der Funktionsweise des Prototyps habe ich folgende Anwendungsfälle geschrieben:

Anwendungsfall 1: Konstruktion

Hauptakteur:
- *Linux*-Anwender mit Anfängerkenntnissen

Anwendungsbereich:
- Arbeit auf Desktop-Computer unter KDE

Niveau:
- Aufbau einfacher Kommandos

Beteiligte und Interessen:
- *Linux*-Anwender
  - möchte ein Video abspielen und das Video soll gleich und im Vollbild-Modus mit dem *Kaffeine Media Player* abgespielt werden
- *SkKonsole*
  - Ziel dieser Entwicklung ist es, eine bessere Bedienbarkeit der Konsole über eine graphische Ober-

fläche anzubieten

Voraussetzungen:

- KDE-Version 3.x oder höher
- Der *Linux*-Anwender hat bereits den Konsolenbefehl `kaffeine` in die Befehlszeile korrekt eingegeben
- Der *Linux*-Anwender betätigt nur einmal die graphischen Steuerelemente der Befehlsschalter `-p` und `-f`
- Der *Linux*-Anwender wählt eine Videodatei mit passendem Format über einen Datei-Dialog aus, der über dem Knopf mit der Beschriftung *File* erreichbar ist
- Der *Linux*-Anwender betätigt die Eingabe-Taste

Mindestzusicherung:

- Die *SkKonsole* fügt die Befehlsschalter `-p` und `-f` mindestens ein Leerzeichen entfernt nach dem Konsolenbefehl in die Befehlszeile ein
- Die *SkKonsole* fügt den Pfad der Videodatei an der aktuellen Stelle des Cursors in die Befehlszeile ein
- Nach Betätigung der Eingabe-Taste durch den Anwender reicht die *SkKonsole* den Inhalt der Befehlszeile an das Betriebssystem weiter

Zusicherung im Erfolgsfall:

- Die *SkKonsole* fügt die Befehlsschalter `-p` und `-f` in den Befehlsschalter-Abschnitt der Befehlszeile ein
- Die *SkKonsole* erzeugt die Befehlsschalter `-p` und `-f`
- Der Schalter `-play` (der die selbe Semantik wie `-p` hat) wird von der *SkKonsole* aus unerreichbar
- Die *SkKonsole* fügt den Pfad der Videodatei mindestens ein Leerzeichen entfernt nach den Befehlsschaltern ein
- Nach Betätigung der Eingabe-Taste durch den Anwender reicht die *SkKonsole* den Inhalt der Befehlszeile an das Betriebssystem weiter und *Kaffeine* spielt anschließend die ausgewählte Videodatei im Vollbild-Modus ab

Haupt-Erfolgsszenario:

Der *Linux*-Anwender hat bereits den Konsolenbefehl `kaffeine` in die Befehlszeile korrekt eingegeben. Die *SkKonsole* hat den Befehl erkannt und daraufhin die graphischen Steuerelemente mit den Befehlsschaltern von `kaffeine` sichtbar und *unchecked* gemacht. Der *Linux*-Anwender betätigt die Steuerelemente der Schalter `-p` und `-f`. Die *SkKonsole* fügt beide Befehlsschalter an der richtigen Stelle in der Befehlszeile ein. Anschließend betätigt der Anwender den Knopf mit der Beschriftung *File* und wählt über einen Datei-Dialog eine Videodatei aus. Die *SkKonsole* fügt den Pfad der Videodatei am Ende der Befehlszeile ein. Der Anwender drückt die Eingabe-Taste. Daraufhin erscheint *Kaffeine* auf dem Bildschirm und spielt das gewünschte Video im Vollbild-Modus ab.

Anwendungsfall 2: Analyse

Hauptakteur:

- *Linux*-Anwender mit Anfängerkenntnissen

Anwendungsbereich:

- Arbeit auf Desktop-Computer unter KDE

Niveau:

- Erkennung der Texteingabe

Beteiligte und Interessen:

- *Linux*-Anwender
  - möchte ein Beispiel verstehen, das er auf einer Webseite gelesen hat. Das Beispiel ist `tar -tvf your_file.tar`
  - erwartet diesen Befehl per *copy & paste* in die Befehlszeile der *SkKonsole* eingeben zu können und dadurch die Bedeutung des Befehls zu verstehen
- *SkKonsole*
  - Ziel dieser Entwicklung ist die bessere Bedienbarkeit der Konsole über eine graphische Oberfläche anzubieten

Voraussetzungen:

- KDE-Version 3.x oder höher
- Der Anwender fügt die Zeichenkette ein

Mindestzusicherung:

- die *SkKonsole* erkennt das Kommando `tar` und baut die GUI auf
- die *SkKonsole* erkennt die Befehlsschalter `-t, -v` und `-f` und schaltet daraufhin die entsprechenden graphischen Elemente in den checked-Modus
- die *SkKonsole* zeigt Kommentare unter anderem zu den Befehlsschaltern `-t, -v` und `-f` an

Zusicherung im Erfolgsfall:

- der Anwender versteht die Bedeutung des Befehls und der Befehlsschalter anhand der Kommentare auf der GUI
- der Anwender drückt die Eingabe-Taste. Die Ausgabe bestätigt die gerade gewonnenen Erkenntnisse

Haupt-Erfolgsszenario:

Der *Linux*-Anwender hatte die tar-Datei `backup_documents2005.tar` per E-Mail erhalten. Er wollte zuerst den Dateiinhalt überprüfen und machte eine online-Recherche, um Information über `tar` zu finden. Auf einer Webseite fand er folgenden Hinweis:
„Um den Dateiinhalt zu sehen `tar -tvf your_file.tar` in die Konsole eingeben“. Der Anwender kopiert das und startet die *SkKonsole*. Anschließend fügt er den Befehl in die Befehlszeile der SkWidget ein und ändert die Pfad-Angabe. Die *SkKonsole* erkennt den Befehl und macht die GUI von `tar` mit den dazu passenden Kommentaren sichtbar. *SkKonsole* versetzt die `Checkboxes -t, -v` und `-f` in den *checked*-Modus. Der Anwender versteht die Bedeutung jedes Befehlsschalters anhand der Kommentare und drückt die Eingabe-Taste. Die Ausgabe bestätigt seine gerade gewonnenen Erkenntnisse.

Anwendungsfall 3: Exploration

Hauptakteur:

- *Linux*-Anwender mit Anfängerkenntnissen

Anwendungsbereich:

- Arbeit auf Desktop-Computer unter KDE

Niveau:

- experimentelles Aufbauen von Befehlen

Beteiligte und Interessen:

- *Linux*-Anwender
  - möchte probieren, welche Effekte er mit seinem geringen Wissen über die Unix-Kommandozeile erzielen kann. Er kennt lediglich das Kommando `kaffeine`
- *SkKonsole*
  - Ziel dieser Entwicklung ist es, eine bessere Bedienbarkeit der Konsole über eine graphische Oberfläche anzubieten

Voraussetzungen:

- KDE-Version 3.x oder höher
- Der *Linux*-Anwender hat bereits das Kommando `kaffeine` in die Befehlszeile korrekt eingegeben

Mindestzusicherung:

- Die *SkKonsole* erkennt das Kommando `kaffeine` und baut die GUI mit graphischen Steuerelementen für einige Befehlsschalter von `kaffeine` auf

Zusicherung im Erfolgsfall:

- Die *SkKonsole* zeigt die Optionen von `kaffeine` an
- Die *SkKonsole* gibt web-basierten Zugriff auf *man pages* über den Browser *Konqueror*
- Die *SkKonsole* zeigt an, dass es weitere Optionen gibt

Haupt-Erfolgsszenario:

Der *Linux*-Anwender gibt das Kommando `kaffeine` in die Befehlszeile korrekt ein. Die *SkKonsole* baut die GUI auf. Jetzt kann der Anwender an den Kommentaren der graphischen Befehlsschalter erkennen, dass der Befehlsschalter `-p` der unmittelbaren Darstellung einer Videodatei unter *Kaffeine* dient und schreibt `-p` in die Befehlszeile. Die *SkKonsole* setzt das dazugehörige graphische Steuerelement in den checked-Zustand. Der Anwender möchte wissen, welche anderen Möglichkeiten das Kommando anbietet. Er drückt den Knopf mit der Beschriftung *man*. Daraufhin startet die *SkKonsole* einen Konqueror mit einer Web-Darstellung der `man`-Datei von `kaffeine`. Über dieses Dokument entdeckt der Anwender weitere Befehlsschalter, die nicht auf der *SkKonsole* vorhanden sind. Der Anwender trägt

einige dieser Befehlsschalter in die Befehlszeile ein. Anschließend wählt der Anwender eine Videodatei über den Datei-Dialog aus und drückt die Eingabe-Taste. Die *SkKonsole* führt `kaffeine` aus.

Die SkKonsole

In der Abbildung 7.2 sieht man die Struktur von den Klassen der *SkKonsole* in UML. Die *SkKonsole* besteht im Wesentlichen aus zwei Klassen und einem Datum. Das Datum ist eine aus Text bestehende Ressource im XML-Format namens `CommandList.xml`. `CommandList.xml` enthält die Informationen, die die *SkKonsole* zur Startzeit benötigt. Dabei handelt es sich um Informationen über die Syntax mehrerer Terminal-Kommandos, die die *SkKonsole* zur Laufzeit auswerten kann. Das Lesen solcher Informationen kommt über ein Objekt der Klasse `SkXMLLoader` zustande. Die Klasse `SkXMLLoader` ist eine Spezialisierung der Klasse `QxmlDefaultHandler`[16] und ist in der Header-Datei `skxmlloader.h` deklariert. Ansonsten wird in `skxmlloader.h` eine Datenstruktur deklariert, deren Form einem Mehrweg-Baum ähnelt, und die zwei verschiedene Arten von inneren Knoten (`SkCommand` und `SkOption`) hat. Die Blätter bestehen entweder aus primitiven Typen oder aus dem Typen `QRegExp`. Die Kanten sind durch Objekte der Klasse `QMap` realisiert. Diese Datenstruktur speichert zur Laufzeit die Informationen aus `CommandList.xml`. Die *Header*-Datei `skwidget.h` beinhaltet die Klassendeklarationen der Klassen, die für die graphische Darstellung, die Auswertung der Eingabe und die Ausgabe zuständig sind.

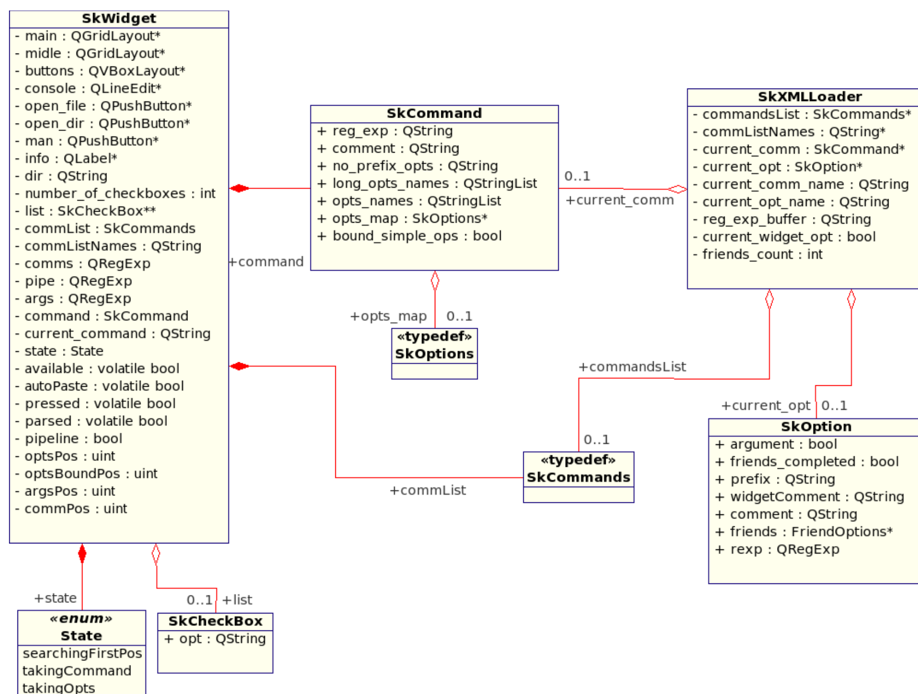

Abbildung 7.2: Die SkKonsole. Die Klassen sind ohne Methoden dargestellt.

Die Hauptklasse dieser *Header*-Datei ist `SkWidget`. Die *Header*-Datei `skwidget.h` beinhaltet auch eine weitere Hilfsklasse `SkCheckBox`. `SkCheckBox` ist eine Adapter-Klasse für `QCheckBox`. In der Abbildung 7.2 ist die Klasse `QMap` nicht zu finden. Das ist so wegen des Generizitätsmodells von C++. Die Objekte der Klasse `QMap` werden zur Laufzeit jeweils unter den Typdefinitionen `SkOptions` und `SkCommands` verwendet.

`SkWidget` ist eine Spezialisierung der Klasse `QWidget` und beinhaltet ein `Panel` mit einer konfigurierbaren Anzahl von `Checkboxes`, drei Knöpfe und ein *Edit*-Feld als Simulation der Befehlszeile.

16 Alle Klassen, deren Namen mit dem Buchstaben „Q“ beginnen, gehören zu den Qt-Bibliotheken Version 3.

Die Knöpfe sind jeweils mit *file*, *dir* und *man* beschriftet. Die *file*- und *dir*-Knöpfe öffnen bei Betätigung jeweils einen Datei-Dialog um Dateipfade und Ordnerpfade auswählen zu können. Der *man*-Knopf bietet die Manual-Seite des aktuellen Kommandos in HTML über den Konqueror-Browser an.

Wenn *SkKonsole* startet, sind die `Checkboxes` zunächst verborgen und ungecheckt, die Befehlszeile ist im Zustand `searchingFirstPos` (vgl. Abb. 7.2). Die *SkKonsole* kann jetzt genau die Kommandos erkennen, deren Beschreibung ein `SkXMLLoader`-Exemplar aus `CommandList.xml` gelesen hat (vgl. Abb. 7.3).

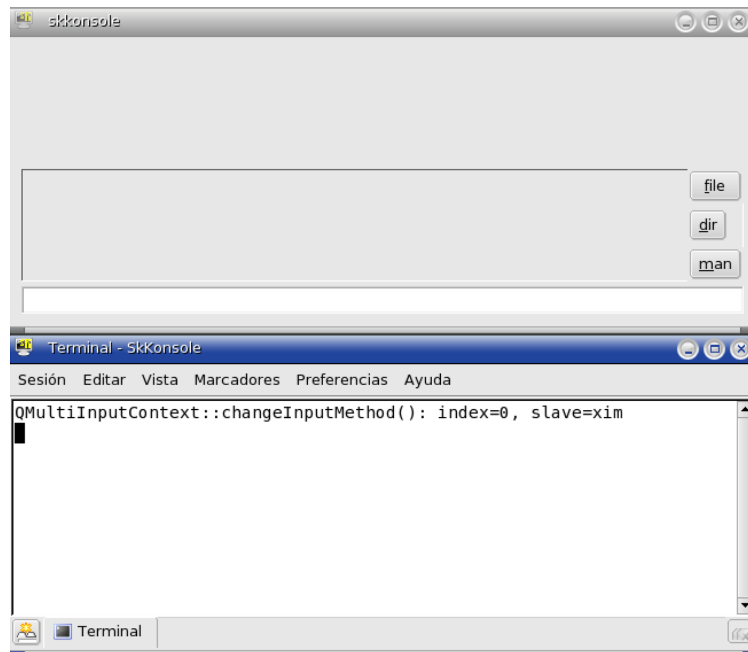

Abbildung 7.3: Die Skkonsole ist gerade gestarted

Tippt man ein Zeichen ein, geht die Befehlszeile in den Zustand `takingCommand` über. Wenn ein gültiges Kommando in der Befehlszeile steht, werden die `Checkboxes` und die Kommando-Informationen sichtbar. Ab diesem Zeitpunkt ist die Befehlszeile im Zustand `takingOpts` (vgl. Abb. 7.4).

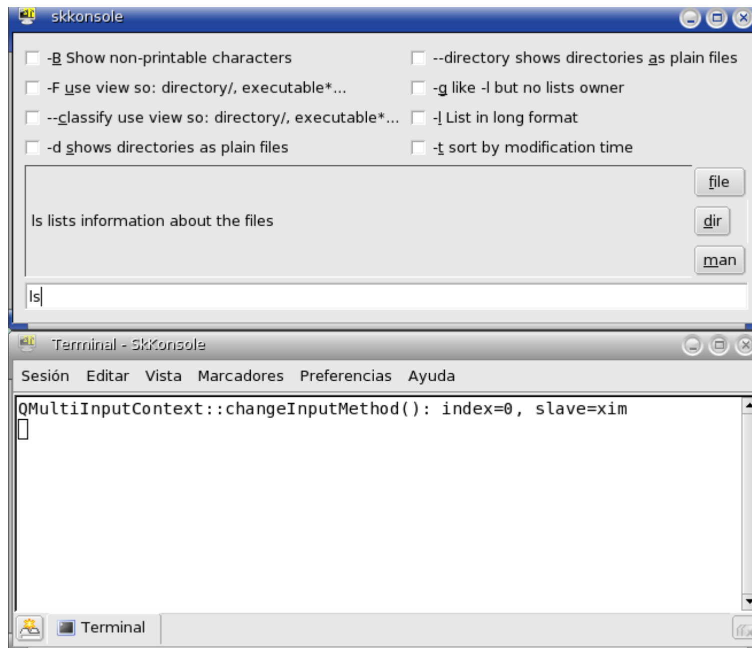

Abbildung 7.4: Die SkKonsole erkennt den Befehl „ls“ und baut die dazu gehörige checkboxes mit den Kommentaren auf.

So lange die Befehlszeile im Zustand `takingOpts` ist, kann der Anwender sowohl eine graphische als auch eine Zeichen-orientierte Eingabe verwenden. Beide Eingabe-Mechanismen aktualisieren gegenseitig ihre jeweiligen Zustände (vgl. Abb. 7.5).

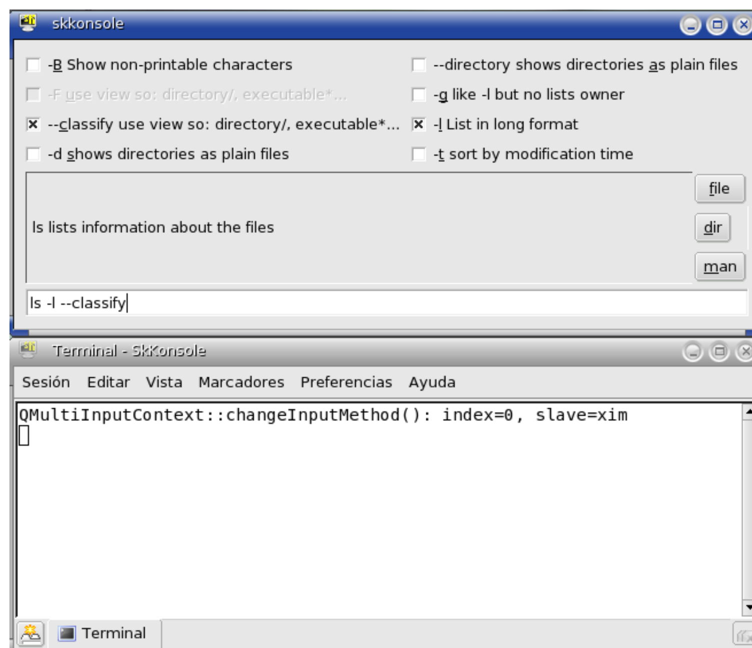

Abbildung 7.5: Tippt man „-l“ so wird das dazu gehörige checkbox belegt. Klickt man das checkbox „--classify“ so erscheint den Schalter in den Editfeld. Dabei ist das checkbox „-F“ nicht mehr erreichbar, weil „-F“ die selbe Bedeutung wie „--classify“ hat.

Bei Betätigung der Befehlstaste gehen die Befehlszeile und das `Widget` in den Anfangszustand über. Gleichzeitig übergibt die *SkKonsole* den Befehl an die Standard-Eingabe. Da die *SkKonsole* nicht auf die Ausgabe wartet, ist es sinnvoll, das Programm per Konsole zu starten um die Ausgabe sehen zu können (vgl. Abb. 7.6).

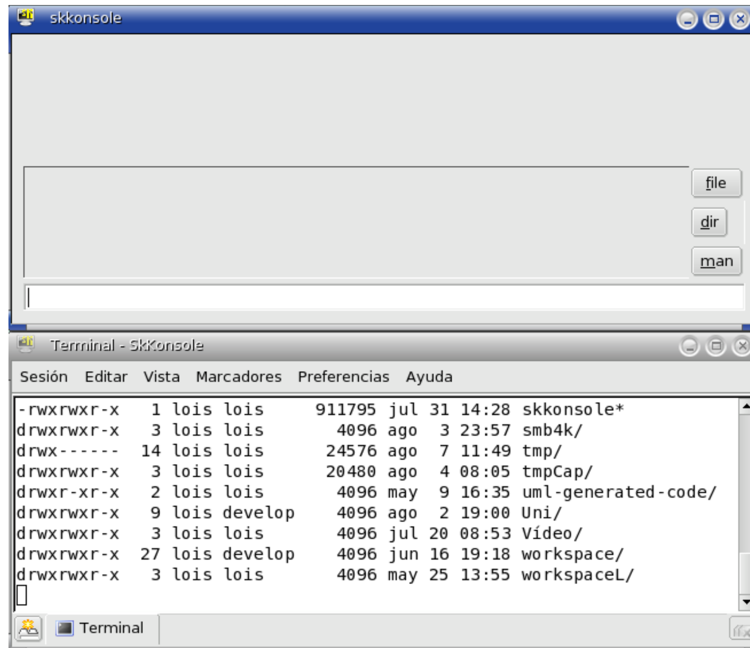

Abbildung 7.6: Druckt man die Eingabetaste, so erscheint die Ausgabe in der Konsole.

Bekannte Bugs

SkKonsole reagiert fehlerhaft auf einige nicht sinnvolle Eingaben. Folgendes Problem tritt auf, wenn beispielsweise der Befehlsschalter `-a` in der Befehlszeile steht und anschließend der Befehlsschalter `-b` eingetippt wird: Falls `-a` und `-b` sich logisch ausschließen, dann geraten die graphischen Steuerelemente von beiden Schaltern in einen undefinierten Zustand. Das Problem liegt daran, dass die Kommandozeile ihren vorherigen Zustand nicht kennt.

SkKonsole gerät nicht deterministisch in einen undefinierte Zustand, wenn man schnell und wahllos die `Checkboxes` aktiviert und deaktiviert. Das liegt am nebenläufigen Zugriff von den `Checkbox`-Methoden auf die Kommandozeile. Synchronisierungsmaßnahmen sind erforderlich.

Anhang B

Das Kategoriensystem für die Inhaltsanalyse

a) Ablehnung
   1. Technische Ablehnung
      i. nicht sinnvolles Ziel oder Entwurf
         - ✗ schlechte Idee
         - ✗ es gibt schon so etwas
         - ✗ damit erreicht man nicht das angestrebte Ziel
         - ✗ falsche Zielsetzung
         - ✗ Alternativ-Vorschläge zur Idee
      ii. mangelhafte Code-Qualität
         - ✗ Performance
         - ✗ Lesbarkeit (Dokumentation wie Code-Selbstdokumentation, ...)
         - ✗ Erweiterbarkeit
         - ✗ Ressourcen-Nutzung (Speicher, Rechenzeit, ...)
      iii. zum bestehenden Code unpassende Implementierung
         - ✗ andere Architektur (Inkompatibilität mit Code-Beständen)
         - ✗ andere Strategie (Herangehensweise an bestimmte Themen, Konventionen, ...)
         - ✗ andere Technik (Thread-safe oder nicht, Verwendung bestimmter Bibliotheken,...)
      iv. keine technische Betrachtung des vorgeschlagenen Codes
   2. Soziale Ablehnung
      i. bezüglich fehlender Zugehörigkeit
         - ✗ zur Gesprächsgruppe (das heißt, sie wollen kein Gespräch mit mir führen oder nicht über meinen Vorschlag reden)
         - ✗ zur Entwicklergruppe
         - ✗ zur Linux-Distribution
         - ✗ zu akademischen Kreisen
      ii. bezüglich der Nicht-Beachtung von Sitten (Community-Konventionen)
         - ✗ in Bezug auf die Nutzung von Kommunikationskanälen
         - ✗ in Bezug auf die Dokumentations-Ethik (z.B. „von bekannten Bugs soll man in der Code-Doku berichten“)
         - ✗ in Bezug auf die Klarheit meiner Ziele (Trittbrett-Fahrer-Diskurs, ...)
         - ✗ sonstige persönliche Anmerkungen
      iii. sozial-hierarchische Einwände

    - ✗ Vorstellungen über den Beförderungsprozess (Lehrling-Meister-Prinzip)
    - ✗ falsches Forum („an wen soll man sich wenden“)
    - ✗ Entscheidung auf höherer Ebene (Guru-Prinzip)
    - ✗ negativ bezüglich des Power-users-versus-newbies- Diskurses

3. Umfeld-bezogene Ablehnung
   i. unpassender Projektzustand oder unpassend fürs Projekt
   ii. die Benutzer/ Programmierer mögen das nicht

b) Akzeptanz

1. Technische Akzeptanz
   i. sinnvolles Ziel oder sinnvoller Entwurf
      - ✗ gute Idee
      - ✗ es gibt so etwas noch nicht
      - ✗ damit erreicht man das verfolgte Ziel
      - ✗ gute Zielsetzung
      - ✗ Vorschläge für die Weiterentwicklung der Idee/ des Programms
   ii. ausreichende Code-Qualität
      - ✗ Performance
      - ✗ Lesbarkeit (Dokumentation wie Code-Selbstdokumentation)
      - ✗ Erweiterbarkeit
      - ✗ Ressourcen-Nutzung (Speicher, Rechenzeit)
   iii. zum bestehenden Code passende Implementierung
      - ✗ selbe/ ähnliche Architektur (Kompatibilität mit Code-Beständen)
      - ✗ selbe/ ähnliche Strategie (Herangehensweise an bestimmte Themen, Konventionen, ...)
      - ✗ selbe/ ähnliche Technik (Thread-safe oder nicht, Verwendung bestimmter Bibliotheken,...)
2. Soziale Akzeptanz
   i. Zuschreibungen in Bezug auf Zugehörigkeit:
      - ✗ guter Gesprächspartner (das heißt, sie wollen ein Gespräch mit mir führen)
      - ✗ er könnte ein gutes Teammitglied werden
      - ✗ er könnte ein guter Programmierer werden
      - ✗ positiv bezüglich des Power-users-versus-newbies- Diskurses
   ii. Zuschreibungen in Bezug auf die Beachtung von Sitten:
      - ✗ Beachtung der Kommunikationsregeln
      - ✗ Transparenz-Regeln im Handeln
      - ✗ Dokumentation von eigenen Bugs

3. Umfeld-bezogene Akzeptanz
    i. passender Projektzustand
    ii. die Benutzer/ Programmierer mögen das